FRONT MATTER

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations

Reading time

4 min

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations

Copyright © 2026 Sam Mozaffari.

Second edition · August 2026 · Version 2.1.0

License

Copyright (c) 2026 Sam Mozaffari. Personal and noncommercial use: the prose of this work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0) — https://creativecommons.org/licenses/by-nc-nd/4.0/. You may share the unadapted work with attribution for noncommercial purposes. You may not sell this book, redistribute adapted editions, or use it primarily for commercial advantage without a separate commercial license. Commercial and organizational use (including company or team distribution, paid training, and resale on any storefront) requires a commercial license from the copyright holder — contact the author via the book repository or website. The copyright holder retains the exclusive right to sell editions via Amazon Kindle Direct Publishing (normal KDP, non-exclusive) and other retailers. Code samples in this work are licensed under the Apache License 2.0 or MIT License, as noted in context. Third-party trademarks, brand systems, templates, and assets retain their original licenses and rights.

About the Author

Sam Mozaffari

Experience Designer · Sydney, Australia

Sam Mozaffari is an Experience Designer at KFC South Pacific in Sydney. This book is a personal project, written and coded with AI assistance; its worked example is a fictional ferry operator.

Sam's work focuses on the pragmatic transition from static design deliverables (post-it syntheses, PDF journey maps, and presentation decks) to executable, living service systems. His methodologies establish rigorous human-in-the-loop governance across the full autonomy spectrum,from *Suggest* to *Confirm* to *Act*,grounded in cryptographic SHA-256 evidence registers, machine-checked role contracts, and adversarial Critic validation gates.

Core Capabilities & Signature Contributions:

- **Full-Spectrum Service Design:** End-to-end journey mapping, 5-lane canonical service blueprinting, 6th-lane autonomous agent decision blueprinting, mental model alignment, and rapid multi-fidelity prototyping.
- **Agentic AI & Systems Architecture:** DeepSeek Harness Cordis plugin suite, Model Context Protocol (MCP) servers, 10-role Eve-style agent teams, SOPs-as-code pipelines, and deterministic SVG compilation engines.
- **Large-Scale Research Ops:** Transcript mining algorithms, contextual inquiry protocols, telemetry fusion, and evidence-backed behavioral persona segmentation.

Connect with the Author:

GitHub: github.com/sammozaffari
LinkedIn: linkedin.com/in/sam-mozaffari-210588a7
Project Repository: github.com/sammozaffari/the-agentic-service-designer

Written with AI Assistance & Agentic Verification

This manuscript was authored through an advanced multi-agent publishing harness. The author directed, curated, reviewed, and approved all architectural concepts, operational schemas, and analytical narratives. Independent adversarial agents conducted tripartite evaluations across theoretical canon alignment, toolchain integrity, and schema compliance.

Trademarks & Brand Disclaimers

Product names, logos, and frameworks mentioned in this work (including Claude Code, Anthropic, OpenAI, DeepSeek, Google Antigravity, Figma, Miro, Cordis, Node.js, Bun) are the property of their respective owners. Their inclusion is strictly educational and analytical and does not imply official endorsement.

All organizational case studies (Lighthouse Transit, Calder Community Bank, Wattle Grove Community Care, Northgate Health, Fernwood Council, Aurora Vento, Forge Industrial, Sable & Slate) are fictional composites written for this book to illustrate technical and organizational mechanics; none of them describes a real client or employer.

Professional & Regulatory Disclaimer

The information, schemas, scripts, and frameworks contained in this book are provided for educational and organizational development purposes. Service operations involving high-stakes regulatory compliance (healthcare clinical pathways, financial lending, public safety, and critical infrastructure) require human sovereign oversight and certified professional verification prior to production execution.

This Edition

Second edition · August 2026 · Version 2.1.0

Version: 2.1.0

Published: August 2026

Book repository: imehr/the-agentic-service-designer

Release page: Releases (v2.1.0)

What's new in this release

No changelog entries recorded for this edition yet.

CHAPTER 01

The AI-First Service Design Operating System

NEXT CHAPTER

The AI-First Service Design Operating System

From sticky notes to living, autonomous service systems

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09

CHAPTER 10

Living Journey Maps: From Transcripts to Sentiment Curves

The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 01

The AI-First Service Design Operating System

From sticky notes to living, autonomous service systems

Reading time

14 min

The Death of Workshop Theater

I still believe in fieldwork. I still believe in synthesis walls, service journeys, and the disciplined act of reframing a bad brief into a better problem. What I do not believe in anymore is the theater that often surrounds those methods. The familiar pattern goes like this: three weeks of interviews, a room covered in sticky notes, a heroic synthesis sprint, a polished playback deck, and then a slow decay into screenshots, half-remembered rationale, and political reinterpretation. The team calls that rigor because the workshop felt intense. I call it an unversioned loss of evidence.

The problem is rarely a lack of intelligence. It is a lack of executable structure. A quote becomes a theme because somebody remembers why. A theme becomes an insight

because the room nodded. An insight becomes a concept because it sounded plausible in a workshop. By the time engineering asks, “What exact evidence supports this service change?”, the answer lives in Slack fragments, slide speaker notes, or a designer’s memory. That is not a practice. That is oral tradition with brand typography.

Static deliverables make the problem worse. A journey map can be beautiful and still die on impact. A blueprint can look comprehensive and still hide invented dependencies. Traditional service design artifacts are usually final-form documents pretending to be living systems. They are snapshots of a team’s temporary conviction, frozen before the organization has the chance to test, challenge, or operationalize them.

> **My take:** The biggest practical failure in modern service design is not shallow empathy. It is broken traceability. When a sponsor pushes back on cost, risk, or feasibility, “users told us this mattered” is too weak. I need to show the exact evidence row, the exact insight node, the exact blueprint consequence, and the exact decision that accepted it.

That broken traceability destroys iteration economics. If the first loop required manual coding of transcripts, manual clustering, manual artifact cleanup, and manual review, the team quietly avoids a second loop. The language of iteration stays in the deck, but the operating reality becomes waterfall. Research compresses. Synthesis becomes decorative. Prototypes get staged late, when political sunk cost is already high. The service design process remains conceptually sound and operationally brittle.

ASD starts from a harsher premise: if a service design claim matters, it must survive machine scrutiny. That does not make the practice less human. It makes human judgment more valuable because the administrative sludge is forced into explicit contracts. Evidence rows are typed. Methods are executable. Artifact acceptance rules are machine-checkable. Critic review is explicit. Phase advance requires a human approval receipt. Once those rules exist, service design stops depending on personal heroics and starts behaving like a real operating system.

> **CORE PRINCIPLE:** In an AI-first service practice, the artifact is never the source of truth. The source of truth is the evidence-backed operating loop that produced it.

This is why I use the phrase operating system deliberately. An operating system allocates work, enforces boundaries, manages permissions, and keeps state coherent across interfaces. That is exactly the job here. Service design is no longer a stack of one-off workshop outputs. It is a governed runtime for continuous research, synthesis, ideation, prototyping, measurement, and decision support.

The ASD Operating System Architecture

Earlier drafts called this platform SDAP, short for Service Design Agent Platform. The name is retired because it was too narrow. ASD is wider than a platform and more disciplined than a loose agent bundle. It is an operating system for the practice itself.

At its center is one canonical practice and knowledge contract. The source tree holds ten role instructions in `platform/agents`, eighty-one skill sources in `platform/skills` that ship as eighty usable skills, seven authored templates in `platform/templates`, the compiler in `platform/compiler`, emitted targets in `platform/packs`, suite packages under `suite/packages`, and the book corpus under `books/`. The compiler projects that single source into twelve emitted targets: `agy`, `buzz`, `claude-code`, `cli`, `codex`, `dsh`, `grok`, `kimi`, `mcp`, `omp`, `opencode`, and `pi`. Different host. Same practice contract.

canonical-tree.txt

```
platform/
  agents/        # 10 canonical role instructions
  skills/        # 81 sources -> 80 shipped skills
  templates/     # 6 authored templates
  compiler/      # compile.mjs, generate-skills.mjs, emit-packs.mjs
  packs/         # 12 emitted targets
suite/packages/
```

```
    core/
    knowledge/
    engine/
    agents/
    render/
    runtime/
    ui/
  books/           # 20-book corpus
  articles/        # case studies and method articles
```

The first architectural shift is simple and profound: the system stores state in portable project files, not in a host chat transcript. A project can be opened from the CLI, an MCP client, a coding-agent host, BUZZ-linked workflows, or DeepSeek Harness Studio because the runtime state does not belong to the host. It belongs to the project workspace. That is why the same rules hold across surfaces.

The second shift is evidence discipline. ASD recognizes exactly three evidence kinds: `quote`, `telemetry`, and `document`. Each captured row is content-addressed with SHA-256. That sounds technical because it is technical. Service design claims become durable when every downstream artifact can point back to an exact evidence row instead of waving at a workshop memory.

The third shift is that method execution is explicit. Agents do not improvise a service design method from vibes. They call `method-run` and receive the executable SOP: inputs, role-attributed steps, output kind, acceptance rules, and review gate. The runtime tells the team what the method is, what it expects, and how it will judge success.

method-run-response.json

```
{
  "method": {
    "id": "journey-from-transcripts",
    "phase": "synthesis",
    "outputKind": "journey-map"
  },
  "steps": [
    {"role": "researcher", "instruction": "Select evidence rows for the journey scop
    {"role": "synthesizer", "instruction": "Cluster moments, tensions, and transitio
```

```
      {"role": "mapper", "instruction": "Write the journey nodes and evidence referenc
      {"role": "critic", "instruction": "Review evidence sufficiency and structural co
    ],
    "acceptance": [
      {"rule": "min-nodes", "min": 3},
      {"rule": "min-evidence-per-node", "min": 1},
      {"rule": "field-non-empty", "field": "title"}
    ]
  }
```

The fourth shift is a hard split between generation and enforcement. An agent may draft. It may not silently bless its own work. Artifact writes must satisfy structural integrity and acceptance checks. Failed acceptance does not disappear. It persists as a draft with machine-readable failure rows. If an artifact cites evidence that does not exist, the write fails closed. If an artifact passes structural checks, it still does not become final by default. The Critic reviews it, records evidence-cited rows, and either passes or vetoes it.

The fifth shift is governance. Phase progression is human authority, full stop. Agents can prepare recommendations, summarize evidence, and even draft the approval object, but only a human can authorize the transition. This boundary matters because service design is more than analysis. It is organizational commitment under uncertainty.

> **CAUTION:** Autonomy in ASD is deliberately asymmetric. Agents may accelerate evidence handling, drafting, and review. They may not silently authorize a phase, rewrite a boundary condition, or upgrade a risky inference into a decision.

The shared loop is therefore exact, not aspirational. It is the same loop every surface must honor:

Create or open a project, then capture evidence as `quote`, `telemetry`, or `document`.

Run `method-run` to obtain the executable SOP with role-attributed steps.

Write a typed artifact and let acceptance rules check it.

Submit the draft to Critic review for a pass or veto.

Replay integrity, acceptance, and review through `factcheck`.

Advance the phase only when a human signs with an approval receipt.

That sequence is the heart of the book. The rest of the platform exists to make it portable, auditable, and cheap enough to run continuously.

Mapping the 4D Process to the TISDD Loop and Kumar's 7 Modes

The operating system is new. The underlying discipline is not. I still use Discover, Define, Develop, and Deliver because organizations understand those phase boundaries. I still value the TISDD loop because service design is a composite practice of research, ideation, prototyping, and implementation. I still use Kumar's seven modes because they tell me what kind of thinking the team should perform, not merely what artifact it should output. ASD does not replace these models. It operationalizes them.

The main correction ASD makes is temporal. In traditional engagements, teams often treat research, synthesis, ideation, prototyping, and delivery as macro-sequential blocks. ASD makes them recur inside every phase. Discover includes evidence capture and early framing prototypes. Define includes research backfill when the evidence is thin. Develop includes measurement design before anything ships. Deliver feeds live telemetry back into the next research loop. The practice becomes recursive instead of ceremonial.

CLIENT-FACING PHASE	TISDD EMPHASIS INSIDE THE PHASE	KUMAR MODES DOING THE WORK	ASD LOOP EMPHASIS
Discover	Research dominates, with early framing probes	Sense Intent, Know Context, Know People	Evidence capture and method selection at full depth

CLIENT-FACING PHASE	TISDD EMPHASIS INSIDE THE PHASE	KUMAR MODES DOING THE WORK	ASD LOOP EMPHASIS
Define	Synthesis, reframing, and map construction	Frame Insights	Typed artifacts, acceptance rules, Critic review
Develop	Ideation plus prototype planning and test design	Explore Concepts, Frame Solutions	Concept generation, blueprinting, review-veto cycles
Deliver	Implementation support, measurement, learning	Realize Offerings	Factcheck, human gate, live telemetry returning as evidence

Mode 1, Sense Intent, deserves special treatment. In most organizations the brief arrives as if it were settled truth. ASD treats it as a provisional hypothesis. Because evidence handling and synthesis are faster, I can afford to challenge the client's framing earlier and with more precision. That is not an AI novelty. It is a service design advantage finally backed by runtime economics.

The loop also gives Kumar's modes an enforcement surface. "Know People" is not a slogan if the evidence register proves whose voices shaped the map. "Frame Insights" is not a performative workshop move if each insight node must meet minimum evidence density. "Realize Offerings" is not hand-wavy if blueprint consequences, review logs, and phase-gate receipts remain linked in one project store.

> **My take:** The real gift of ASD is not faster output. It is faster contradiction. Weak research claims, shallow concepts, and politically convenient myths get challenged earlier because the runtime makes the supporting evidence legible.

That makes the 4D process more honest. Discover is less romantic and more empirical. Define is less mystical and more accountable. Develop is less brainstorm theater and more option engineering. Deliver is less handoff and more operational

learning. The phases remain human-readable, but the loop underneath is now machine-enforced.

The Ten-Role Agent Topology

ASD ships ten canonical roles. These are not ten personalities for style. They are ten bounded responsibilities in the operating system: Orchestrator, Researcher, Synthesizer, Mapper, Ideator, Prototyper, Measurer, Critic, Presenter, and Librarian. Together they form a topology of labor that mirrors what a serious service design team already does, except the handoffs are explicit and the boundaries are testable.

ROLE	WHAT IT OWNS	WHAT IT MUST NOT FAKE
Orchestrator	Phase planning, dispatch, human handoffs, decision preparation	It does not silently approve a phase
Researcher	Evidence capture plans, corpus assembly, context gathering	It does not invent participant evidence
Synthesizer	Themes, insight structures, narrative tensions	It does not detach claims from evidence rows
Mapper	Journey maps, stakeholder maps, ecosystem maps, blueprints	It does not add unsupported nodes or lanes
Ideator	Option generation, reframing prompts, concept variation	It does not resolve strategic trade-offs alone
Prototyper	Experiment design, walkthrough scaffolds, pilot mechanics	It does not confuse a mock with live proof
Measurer	KPI trees, baselines, metric integrity, evaluation logic	It does not launder weak numbers into certainty
Critic	Evidence-cited review rows, pass or veto, quality pressure	It does not pass work without stated evidence

ROLE	WHAT IT OWNS	WHAT IT MUST NOT FAKE
Presenter	Executive summaries, report exports, decision packaging	It does not add new evidence while presenting
Librarian	Method provenance, corpus lookup, contract fidelity	It does not let the practice drift from source

The reason this topology works is that it matches the real shape of service design labor. Research and synthesis are different kinds of work. Mapping and ideation are different kinds of work. Measurement and presentation are different kinds of work. When one general-purpose assistant tries to do all of them in one prompt, it collapses the practice into a single opaque reasoning blob. ASD resists that collapse. It externalizes the roles so the project can inspect where judgment happened and where it still needs a human.

The Critic is the load-bearing role here. It sits on the write path. That means the operating system has an internal adversary by default. The Critic's veto writes a `review-veto-log` instead of just scolding the draft in chat. Now the defect becomes part of the project's memory.

review-veto-log.json

```
{
  "kind": "review-veto-log",
  "reviewOf": "artifact-7f3c",
  "verdict": "veto",
  "rows": [
    {
      "rowKind": "defect",
      "severity": "high",
      "rule": "min-evidence-per-node",
      "message": "The wait-state node cites no evidence rows.",
      "fix": "Add at least one resolvable evidence reference or remove the claim."
    }
  ]
}
```

That is the difference between agent theater and an operating system. The system does not merely say a draft is weak. It records how the draft failed the contract. In practice this changes team behavior quickly. People stop treating AI output as a clever first draft and start treating it as a submission that must earn promotion through evidence and review.

The Librarian matters more than many teams expect. Service design breaks when method language drifts. A blueprint becomes a process map. A persona becomes a stereotype card. A KPI tree becomes a random metric dump. The Librarian forces the operating system back to canonical method intent, which is essential when eighty skills are meant to remain selectable and distinct across many hosts.

The Autonomy Spectrum: When Agents Lead and When Humans Decide

AI-first service design fails when teams ask the wrong question. The question is not, “Can the agent do this task?” The question is, “What kind of authority should this task carry?” ASD answers by distributing work across an autonomy spectrum rather than pretending every step deserves the same freedom.

At the most autonomous end, I want agents doing volume work that benefits from consistency: parsing transcripts, checking evidence references, generating option sets, scaffolding pilot materials, building KPI trees from approved structures, and replaying fact checks. These are high-friction, rule-shaped tasks where machine stamina beats human patience.

In the middle sit recommend-only tasks. Here the agents can propose insight framings, compare operating models, draft service blueprints, or suggest KPI structures, but they should not treat those proposals as decisions. This is where human design judgment earns its keep. Naming the real tension, deciding what political trade-off matters, and choosing which risk is acceptable are not formatting problems.

At the hard-gated end sit explicit human authorities: participant ethics, scope reframing with sponsors, approval of phase progression, commitment to pilots, and any decision that changes rights, responsibilities, or organizational exposure. The operating system can prepare the packet. It cannot own the consequence.

AUTONOMY BAND	TYPICAL ASD WORK	WHY IT BELONGS THERE
Autonomous execution	Evidence normalization, method lookup, draft artifact writing, Critic checks, factcheck replay	Rules are explicit, failure is cheap, and outputs remain reviewable
Recommend only	Insight framing, concept comparison, future-state blueprint options, KPI-tree proposals	Interpretation matters, but the runtime can still make the option space legible
Human authorization	Ethics calls, sponsor reframing, pilot commitment, phase advance	These decisions allocate risk, authority, and accountability

phase-advance.sh

```
asd phase-advance \
  --projectId <project-id> \
  --toPhase define \
  --idempotencyKey gate-discover-1 \
  --approvalReceipt '{"receiptId":"gate-discover-1","decision":"approve","rationale"
```

That command tells the whole story. The system expects a human approval receipt because human gatekeeping is not implied by a polite sentence in chat. It is an object the runtime can inspect. Agents prepare decisions; humans authorize them. That one boundary protects the practice from a thousand small acts of automation theatre.

Once teams internalize this spectrum, the fear around agentic service design usually drops. People realize the operating system is not trying to replace the designer's hardest work. It is trying to clear everything around that work so the designer can spend more time on the parts that should remain stubbornly human: building trust, reading institutional power, reframing the brief, and deciding when a service promise is ethically or operationally unsound.

My take: The mature question is never “human or AI?” It is “which authority belongs where, and what receipt proves we respected that boundary?” When I ask that question rigorously, the platform becomes far more ambitious and far less reckless.

This is the chapter’s core claim. Service design becomes stronger, not weaker, when its repetitive labor is delegated into an evidence-gated operating system and its consequential judgment is made more visible. ASD does not flatten the craft into automation. It sharpens the craft by forcing clarity about what must be explicit, what may be automated, and what still requires a human to stand behind the decision.

CHAPTER 0

The Agentic Service Designer

CHAPTER 02

The ASD Platform Architecture

NEXT CHAPTER

The ASD Platform Architecture

One canonical practice, twelve compiled surfaces

Continue Reading →

ASD ' 26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 02

The ASD Platform Architecture

One canonical practice, twelve compiled surfaces

Reading time

13 min

One Engine, Several Surfaces

The cleanest way to read the architecture is to start with the invariant and work outward. The invariant is the practice contract: ten role prompts, eighty-one skill sources that ship as eighty skills, thirty runtime tools, thirty-eight registered artifact-kind contracts, thirty-nine catalog rows, seven authored templates, and a corpus of two hundred twenty-eight resources. The compiler then projects that contract into surfaces without changing the contract itself.

That distinction matters because a host can radically change the user experience while leaving the runtime semantics untouched. The CLI gives me local operator flow over `ASD_WORKSPACE`. The MCP bundle exposes newline-delimited JSON-RPC. Claude Code and other coding-agent packs add role prompts, skill files, and pack-local MCP wiring. BUZZ contributes persona-pack definitions and a collaboration layer. DSH

adds a host adapter, Studio UI, and loopback RPC. None of those surfaces are allowed to change what counts as evidence, what `method-run` returns, how artifact acceptance works, or who may advance a phase.

That is why the book keeps returning to the phrase one engine, twelve surfaces. I am describing a projection model, not a federation of loosely related tools. If a host pack says something the CLI would reject, the host pack is wrong.

EMITTED TARGET	PRIMARY JOB	WHAT IT ADDS BEYOND THE ENGINE
`cli`	Local operator workflow and CI surface	Pack-local launcher, shell ergonomics, deterministic exit codes
`mcp`	Generic stdio integration	JSON-RPC transport, resources, prompts, client configuration snippets
`claude-code`, `codex`, `kimi`, `opencode`, `omp`, `pi`, `grok`, `agy`	Coding-agent host packs	Host layout, prompt packaging, skills, vendored MCP wiring, commands
`buzz`	Persona Pack for open agent collaboration	Ten keyless persona definitions, relative MCP binding, Canvas linkage conventions
`dsh`	DeepSeek Harness Studio adapter	`@asd/ui` bundle, Studio client, loopback `/asd/app` and `/asd/rpc` routes

The distribution repositories make the same point at release time. `imehr/asd-plugin` is not an independently authored product line. It is generated from emitted host packs by `scripts/publish-distribution-repos.mjs`, and it carries a `provenance.json` file so the release can point back to the exact source commit. The same is true of `imehr/asd-mcp`. This is less glamorous than hand-maintained downstream repos, and that is precisely why I trust it. Drift is boring until it corrupts a live integration.

> **CORE PRINCIPLE:** A surface may repackage the runtime. It may not redefine the runtime.

Once I accept that rule, a lot of architectural choices become obvious. The emitted packs are generated, not hand-curated. Host READMEs interpolate inventory counts instead of hardcoding them. MCP entrypoints are always pack-local. Release evidence distinguishes emitted-pack verification from host execution. Even the documentation is written under receipt discipline because the docs are part of the contract, not marketing afterthoughts.

Seen this way, the architecture is almost anti-charismatic. There is no special host that counts as the "real" ASD. The CLI is real. The MCP bundle is real. The plugin packs are real. BUZZ and DSH are real within their stated boundaries. That lack of hierarchy is a design choice. It prevents the organization from treating one favored interface as the product and everything else as a second-class afterthought. The product is the contract. The surfaces are different ways to enter it.

The Compiler and the Pack Tree

The compiler is the hinge between canonical source and distributable surfaces. Three files do the heavy conceptual work: `compile.mjs`, `generate-skills.mjs`, and `emit-packs.mjs`. They are not interchangeable. Each one narrows a different kind of ambiguity.

`compile.mjs` scans the canonical trees under `platform/agents`, `platform/skills`, and `platform/templates`, derives inventory using the capability matrix, and emits deterministic manifests. No timestamps. No absolute paths. Sorted traversal. Byte-identical regeneration is part of the contract. In other words, `compile.mjs` answers the question: what exactly exists in the canonical practice tree right now?

`generate-skills.mjs` takes the method manifests in `suite/packages/knowledge/data/methods` and turns them into real `SKILL.md` files. That

matters because eighty skills are only useful if they are discoverable and distinct. The generator builds frontmatter, descriptions, trigger language, provenance references, and review-gate wording from method metadata so the skills remain synchronized with the canonical method contract.

`emit-packs.mjs` is where the projection happens. It vendors the runtime, builds the standalone CLI and MCP bundles, synthesizes host-specific READMEs and configs, syncs emitted trees without recursively trashing unrelated files, and writes twelve distributable targets. It is also where pack-local server rules get enforced. The emitted MCP server is always launched relative to the installed pack root. No ambient executable assumptions. No “works on my PATH” shortcuts.

emit-sequence.sh

```
node platform/compiler/compile.mjs
node platform/compiler/generate-skills.mjs
node platform/compiler/emit-packs.mjs --output /tmp/asd-packs

PACK=/tmp/asd-packs/cli
mkdir -p "$HOME/.local/bin"
ln -sf "$PACK/asd" "$HOME/.local/bin/asd"
export PATH="$HOME/.local/bin:$PATH"
asd --version
asd help
```

I like this pipeline because each step leaves evidence. The first step defines the inventory. The second produces the skill surface. The third projects that inventory into installed form. If something drifts, I can ask which stage lied. That is a much better debugging question than “why is the plugin weird?”

pack-tree.txt

```
platform/packs/
  agy/
  buzz/
  claude-code/
  cli/
```

```
    codex/
    dsh/
    grok/
    kimi/
    mcp/
    omp/
    opencode/
    pi/
    .claude-plugin/
```

The hidden elegance here is that the pack tree is both broad and conservative. Broad, because it targets twelve distinct surfaces. Conservative, because every target is synthesized from the same source-of-truth inventory. Even README prose is generated from `deriveInventory()` so a corpus or method change cannot leave a stale literal lying around in a shipped pack. That is the sort of move people call pedantic right up until a customer installs from docs that have drifted six counts away from reality.

> **My take:** Compilers also matter outside programming languages. In ASD the compiler is the thing that turns service design knowledge into stable distribution surfaces without asking human maintainers to keep twelve host trees mentally synchronized.

There is also an architectural humility in the emitter. It does not claim more than it proves. Pack emission is evidence of deterministic packaging. It is not evidence that every host executed natively. That is why the release story keeps separate receipts for emitted bundles, DSH integration, BUZZ collaboration, and downstream publication repos.

Suite Packages and the Runtime Registry

The monorepo's suite packages tell you where responsibility lives. `@asd/core` carries the foundational types and shared contracts. `@asd/knowledge` packages the corpus

data and method metadata. `@asd/engine` owns the runtime tool logic. `@asd/agents` packages the role-facing layer. `@asd/render` handles deterministic output transformation. `@asd/runtime` binds the engine into the CLI and MCP entrypoints. `@asd/ui` is the DSH-facing host adapter and Studio client.

If I had built all of that in one package, the emitted surfaces would be fragile. By splitting the concerns, the compiler can vendor what a target needs without making the whole system depend on the quirks of one host. The runtime package is especially important here: its `bin` exports define `asd` and `asd-mcp`, which means the CLI and MCP server are twins over the same registry rather than cousins that happen to share branding.

PACKAGE	ARCHITECTURAL ROLE	WHY IT EXISTS SEPARATELY
`@asd/core`	Shared types, status vocabulary, contract primitives	Keeps every surface aligned on the same domain model
`@asd/knowledge`	Method metadata, corpus data, templates, bundle patches	Lets the corpus travel with emitted packs without folding it into runtime logic
`@asd/engine`	Thirty `sd_*` tools and execution rules	Protects runtime semantics from host packaging noise
`@asd/agents`	Role-facing layer and agent contracts	Separates role logic from tool implementation
`@asd/render`	Deterministic rendering and export helpers	Keeps presentation deterministic and testable
`@asd/runtime`	CLI and MCP binding layer	Provides stable entrypoints for emitted bundles
`@asd/ui`	DSH Studio plugin and capability adapter	Contains the host-owned UI seam without contaminating the core runtime

This package split also explains why the host stories differ. The CLI needs `@asd/runtime` and knowledge data. The MCP bundle needs the same plus the stdio server surface. DSH needs `@asd/ui` because it has to bind into host slots and

services. BUZZ does not need the Studio client at all; it needs pack-local MCP wiring and persona-pack definitions. One engine. Different shells.

The runtime registry is where that unity becomes concrete. Thirty tool names are derived from source rather than curated by hand. Tool groups cover project operations, evidence handling, method lookup and execution, artifact reads and writes, review, export, knowledge access, fact checking, orchestration, and the human phase gate. That registry is the mechanical spine of the system. The surfaces are merely access patterns.

pack-manifest.json

```
{
  "target": "cli",
  "inventory": {
    "tools": 30,
    "methods": 80,
    "schemas": 38,
    "authoredTemplates": 7,
    "agents": 10,
    "skillSources": 81,
    "catalogRows": 39,
    "corpusResources": 228
  },
  "liveHostExecution": "unverified"
}
```

Notice the last line. Even the manifest speaks with boundary discipline. It advertises what is emitted and refuses to lie about host proof.

Verified Counts and Receipt Discipline

Architectural integrity lives or dies on count discipline. A small mismatch is never small when a host pack, website page, and article each claim a different inventory. ASD therefore treats counts as source-derived facts, not prose flourishes. The current

manifests for emitted packs carry the canonical release counts: ten roles, eighty shipped skills from eighty-one sources, thirty tools, thirty-eight schemas, thirty-nine catalog rows, seven authored templates, and two hundred twenty-eight corpus resources. The capability-matrix tests assert that the committed corpus really derives to 228. The site index and top-level README repeat the same number.

This matters because historical material in the repository still contains superseded inventory snapshots in older receipts and article boundaries. Those records are worth keeping because they document what a prior release actually proved. They are not allowed to override the current manifests, tests, and documentation contract. Receipt history is part of the architecture, but current counts still need a single canonical answer.

RELEASE FACT	CURRENT CANONICAL VALUE	GROUNDING SURFACE
Roles	10	pack manifests, source tree, site index
Skills shipped / skill sources	80 / 81	pack manifests, generated skills tree
Runtime tools	30	runtime registry, pack manifests
Artifact-kind contracts	38	pack manifests, capability matrix
Catalog rows	39	pack manifests, practice metadata
Authored templates	7	pack manifests, templates tree
MCP corpus resources	228	pack manifests, site index, emitter tests
Role prompts	10	MCP prompt surface, docs, facts sheet
Emitted targets	12	emitter receipt, pack tree

The receipt discipline is just as important as the counts themselves. `docs/reports/shared-pack-emission.receipt.json` is the emitted CLI and MCP matrix

receipt. It captures the twelve-target emission run, isolation details, a representative CLI workflow, and disposable MCP bundle checks. `docs/reports/shared-pack-release-ledger.json` records what the release scope covers and what exceptions remain bounded instead of being hand-waved away. `docs/reports/dsh-live-integration.receipt.json` is the DSH-specific proof surface, with its own bundle, UI, and capability claims. `docs/reports/buzz-case-study-ledger.json` holds the BUZZ evidence boundary.

Those receipts do more than prove success. They define honesty. A shared-pack receipt does not magically prove native Claude Code execution. A DSH receipt does not prove BUZZ persistence. A BUZZ screenshot does not prove ACP transport. By separating receipts, ASD keeps distribution evidence from mutating into mythology.

> **CAUTION:** Do not collapse “emitted”, “disposable verified”, and “host executed” into one sentence. They are different proof classes, and confusing them is how architecture docs become fiction.

The downstream repos reinforce this discipline. `imehr/asd-plugin` and `imehr/asd-mcp` each carry a `provenance.json` pointer back to the source commit used for publication. That sounds mundane. Good. Provenance should feel mundane. If release lineage feels magical, it is probably broken.

I treat that distinction as operational hygiene. The shared-pack receipt tells me what the emitter and disposable bundles proved. The release ledger tells me what exceptions were consciously bounded rather than quietly ignored. The DSH receipt tells me what the Studio adapter actually exercised in a live host. The BUZZ ledger tells me exactly how far the collaboration evidence goes before claims must stop. Together they form a proof stack. Separately they prevent the worst architectural habit in AI tooling: letting one successful demo imply ten untested capabilities.

A Tour of the Common Loop

All of that structure would be academic if the everyday workflow were messy. It is not. The common loop is extremely plain, which is why it travels well. First I create or open a project and capture evidence. Second I ask `sd_method_run` for the executable SOP. Third I write a typed artifact. Fourth the Critic records a pass or veto with evidence-cited rows. Fifth I run `sd_factcheck`. Sixth the human phase gate decides whether the project advances. Same loop in the CLI. Same loop through MCP. Same loop behind host packs. Same loop under DSH. BUZZ can collaborate around it, but BUZZ does not redefine it.

pack-local-mcp.json

```
{
  "mcpServers": {
    "asd": {
      "command": "node",
      "args": ["mcp/bin/server.mjs"]
    }
  }
}
```

That pack-local entrypoint rule is one of those unglamorous design decisions that saves weeks of pain. It means the installed pack carries the exact server it expects. I do not need to hope the right binary is somewhere on the machine. The same philosophy applies to the CLI pack, which links to the emitted `asd` shim rather than an ambient executable.

The same loop also explains why the platform can support both strict automation and careful judgment. The early steps are highly automatable because the contracts are explicit. The final step remains human because no receipt can make an organizational commitment on somebody else's behalf. In architecture terms, ASD is opinionated about state transitions and conservative about authority. That combination is what keeps the common loop portable instead of brittle.

Once the server is live, the loop becomes operationally simple. I can call `sd_project_create`, add evidence, inspect the method contract, write an artifact, ask for review, and run `sd_factcheck`. A failed artifact remains a draft with check rows; the system does not erase the attempt. A Critic veto writes a `review-veto-log`. A pass may promote the artifact to `final`. Then, and only then, does the human gate decide whether to move forward.

This is where the architecture stops feeling abstract. The common loop is the place where one engine proves it deserves many surfaces. If the CLI, MCP bundle, plugin pack, BUZZ-linked workflow, and DSH Studio all express the same loop without semantic drift, the architecture is working. If one surface invents extra states, hides the human gate, or weakens the evidence rules, the architecture is leaking.

> **My take:** Good platform architecture makes the right thing feel repetitive in the best possible way. When I switch surfaces, I do not want new metaphysics. I want the same contract wearing different clothes.

That is the real lesson of the ASD platform architecture. The interesting part is not that it emits twelve targets. The interesting part is that twelve targets are allowed to exist without splintering the practice. The compiler, the package boundaries, the manifests, the receipts, and the distribution repos all serve that one outcome. Service design becomes portable because the contract stays singular.

A skeptic will ask what stops the loop from decaying as it travels. Three mechanisms do the work. The compiler refuses semantic drift at the source: a surface that wants different rules has to change the canonical manifest, in review, where the change is visible. The release receipts pin each surface to a named run, so "verified" always names the exact bundle and environment that ran it. And the fail-closed defaults mean a surface that loses its corpus roots or its workspace degrades loudly rather than quietly improvising. Together these three turn a twelve-surface distribution from a maintenance nightmare into a boring, repeatable release.

CHAPTER 01

The AI-First Service Design Operating System

CHAPTER 03

Service-Dominant Logic & Value Co-Creation in the Age of AI

NEXT CHAPTER

Service-Dominant Logic & Value Co-Creation in the Age of AI

Why agent teams fit service-dominant practice

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 03

Service-Dominant Logic & Value Co-Creation in the Age of AI

Why agent teams fit service-dominant practice

Reading time

12 min

Goods-Dominant vs Service-Dominant Thinking

Service-dominant logic begins with a blunt correction. Service, singular, is the application of specialized competences for the benefit of another actor. Goods are only distribution mechanisms for service provision. That sounds academic until you look at how AI systems are currently sold. Most vendors promise faster answers, more summaries, more generated screens, more automated output. Those are goods-dominant promises. They assume value lives inside the object produced.

In practice, that assumption fails constantly. A generated journey map is worthless if the operations lead rejects its framing. An executive summary has no value if it cannot change a steering decision. A blueprint creates no value if it sits in a repo without changing the operating cadence, staffing logic, or service promise. The artifact is not the value. The artifact is a distribution mechanism through which value might later be co-created.

ASD aligns with SDL because it never treats an artifact as self-justifying. The system keeps dragging every claim back to its use context. Who is the beneficiary? What evidence shows the value proposition matters? What role does the human play in realizing it? What boundary conditions shape the outcome? When an architecture forces those questions, it becomes much harder to confuse production volume with service value.

GOODS-DOMINANT FRAME	SERVICE-DOMINANT FRAME	WHAT ASD DOES DIFFERENTLY
Value is embedded in the output	Value is realized in use and in context	Artifacts remain in review until evidence, review verdicts, and human decision align
The provider delivers value	The provider offers a value proposition and participates in co-creation	Methods, reviews, and phase gates keep authority and consequence visible
The user consumes a product	The beneficiary integrates resources	Human context, field evidence, and operational constraints remain part of the contract
More output means more value	Only useful change in a concrete situation counts	Measurement ties artifacts to KPI trees and learning loops instead of vanity counts

This is why I resist describing ASD as a content generator for service design. It is a co-creation system. The role prompts, the skill library, the deterministic renderer, and the evidence loop are all there to improve the quality of actor-to-actor work between designers, stakeholders, frontline staff, and decision-makers. The artifact matters. The relationship between actors matters more.

> **CORE PRINCIPLE:** In ASD, an artifact is a value proposition under evidence and review. It becomes valuable only when a beneficiary integrates it into a real decision or service change.

That perspective also clarifies why the human phase gate is so important. If value is phenomenologically determined by the beneficiary, then the system must preserve explicit moments where institutional actors decide what they are willing to adopt, fund, or operationalize. A platform that tries to skip those moments may feel "more autonomous." It is actually less service-aware.

Operant Resources: Agents Join the Workforce

SDL's distinction between operand and operant resources is where agentic design starts to get interesting. Operand resources are acted upon. Operant resources do the acting. In a conventional software setup, data files, transcripts, dashboards, and PDFs are operand resources. Human expertise is the operant resource. In an agentic system, the picture changes: method manifests, role prompts, executable runtimes, and evidence structures all begin to act on the situation.

Agents are clearly operant resources. The Researcher can parse a large corpus. The Synthesizer can pattern-match across coded evidence. The Critic can challenge unsupported claims. The Measurer can structure a KPI tree. Each role applies competence rather than simply storing information. But the surprising shift in ASD is that evidence itself begins to behave like an operant resource once it becomes typed, queryable, and compulsory.

Raw notes in a folder are inert. Evidence rows in ASD are not. Because every row carries a kind, source, body, and content-addressed identity, the evidence register actively shapes what the rest of the system may claim. It constrains maps. It powers review. It blocks fabricated certainty. It redirects the work. In practice, that means

evidence is no longer passive material waiting for a workshop. It becomes an active force in the service system.

evidence-row.json


```
{
  "evidence": {
    "id": "5ddb444f18bc3d27b30b5234346e90e05206c19af05c89506d7651625f2fd20f",
    "kind": "quote",
    "source": "maint-supervisor-04",
    "body": "If the tracker goes quiet for three days, the whole plant assumes the j
    "capturedAt": "2026-08-23T09:14:00Z"
  }
}
```


Once that row exists, it can be cited by a journey map, pulled into a service blueprint, challenged by the Critic, linked into a KPI rationale, and surfaced in a report. That is evidence acting through the system. It is still human evidence. ASD does not magically convert people into data exhaust. But it does convert fragile research memory into a structured operant resource the practice can keep applying.

> **My take:** The moment a team treats evidence as an active resource instead of workshop wallpaper, the quality of disagreement improves. People stop arguing in slogans. They argue with the actual corpus.

Method execution deepens this shift. The system does not ask an agent to “do some synthesis.” It returns an SOP with role-attributed steps. That means the method itself becomes an operant resource too: a callable, inspectable pattern of action that organizes how capability gets applied.

sd-method-run.json


```
{
  "method": {
    "id": "kpi-tree-design",
```

```
      "phase": "measurement",
      "outputKind": "insight-set"
    },
    "steps": [
      {"role": "measurer", "instruction": "Name the north-star outcome in beneficiary
      {"role": "measurer", "instruction": "Decompose the outcome into driver branches
      {"role": "critic", "instruction": "Reject branches with no evidence, baseline, o
    ],
    "acceptance": [
      {"rule": "min-nodes", "min": 3},
      {"rule": "min-evidence-per-node", "min": 1},
      {"rule": "field-non-empty", "field": "metric"}
    ]
  }
```

This is the part many AI teams miss. A model alone is not a service system. An agent without an evidence discipline is not an operant resource I should trust in a serious design practice. ASD earns that trust by giving the operant side structure: explicit methods, bounded roles, review gates, and receipts.

Value Co-Creation and Value-in-Context

SDL's most useful warning is that firms do not deliver value unilaterally. They participate in value co-creation, and the beneficiary determines value in context. That point lands with force in servitization work because outcome promises collapse the distance between the provider's internal logic and the customer's lived reality.

Aurora Vento is the clearest example in the ASD corpus. The company is a fictional composite industrial vendor trying to move from capital-equipment sales into uptime and performance partnerships. A goods-dominant response would focus on more telemetry, better dashboards, and more premium service bundles. An SDL response starts elsewhere: what does the customer plant actually experience as value, what resources must both sides contribute, and which interactions are true co-creation moments rather than backstage facilitation?

The Aurora Vento case makes the distinction concrete. The customer did not want “insights.” The customer wanted stable production windows, lower emergency freight, faster mean time to repair, and credible planning confidence. Aurora could propose a value architecture, but it could not honestly promise to deliver value in isolation because part of the outcome depended on shutdown windows, operator behavior, data freshness, and shared review cadence inside the plant.

AURORA QUESTION	GOODS-DOMINANT ANSWER	SDL ANSWER
What is being sold?	More intelligent equipment plus service add-ons	A better-maintained production system and outcome partnership
Who creates value?	The provider by shipping a superior offer	Provider and plant together through resource integration
What proves success?	Alerts sent, dashboards installed, contracts signed	Uptime, MTTR, first-time fix, renewal behavior, confidence in use
What can be guaranteed?	The outcome itself	Only controllable commitments, shared metrics, and governance rules

ASD supports this logic because its methods and artifacts keep asking what the beneficiary must do, what the provider must do, and what evidence shows the relationship is working. In the Aurora story, the operating model becomes a shared gameboard rather than a vendor dashboard. The Measurer does not merely chart internal service metrics. The Measurer structures a value-in-context system that both sides can use to judge whether the partnership is actually helping the plant.

That is why the service blueprint matters in SDL terms. A blueprint is more than a provider process map. It is a co-creation map that makes frontstage interactions, backstage work, support systems, customer actions, and agent decision logic visible together. If the customer action lane is thin, the blueprint is probably hiding a value fantasy. If the review cadence is missing, the partnership is probably wishful thinking dressed as transformation.

> **CAUTION:** A service team drifts back into goods-dominant logic the moment it confuses internal activity with beneficiary value. More notifications, more dashboards, or more generated content do not matter if the customer's lived situation stays unchanged.

SDL also clarifies why ASD preserves bounded host claims. Host execution itself is not the value. The service value appears when the host lets the team perform better co-creation work. Claude Code, MCP, BUZZ, and DSH are distribution mechanisms. The value lies in the improved design and operational decisions they help people make together.

The Service Profit Chain, Instrumented

The Service Profit Chain becomes much more concrete once I stop treating it as a motivational poster and start treating it as a measurement problem. Internal service quality shapes employee satisfaction and capability. That in turn shapes external service value, customer loyalty, and financial performance. In agentic work, internal service quality includes the quality of the decision-support system itself: evidence hygiene, method reliability, review fidelity, and whether frontline people receive better cognitive leverage or just more automated noise.

This is exactly where `kpi-tree-design` matters. A KPI tree forces the team to move from grand claims to causal structure. What is the beneficiary-defined outcome? Which drivers influence it? Which leading and lagging indicators signal movement? Which branches are controllable? Which are shared? Which metrics are vanity? The method turns the Service Profit Chain from theory into an inspectable artifact.

kpi-tree.yaml

```
north_star:
  metric: uptime_in_agreed_window_pct
  beneficiary: plant_maintenance_team
```

```
    baseline: 92.1
    target: 98.0
  branches:
    - metric: mean_time_to_repair_days
      driver_of: uptime_in_agreed_window_pct
      baseline: 5.2
      target: 3.0
      evidence: [ops-log-12, telemetry-07]
    - metric: first_time_fix_pct
      driver_of: mean_time_to_repair_days
      baseline: 61
      target: 75
      evidence: [service-orders-q2]
    - metric: emergency_freight_incidents
      driver_of: mean_time_to_repair_days
      baseline: 14
      target: 5
      evidence: [parts-audit-02]
  review_gate: critic
```

Once a team has that artifact, the Service Profit Chain stops being vague. Internal quality can be expressed as things like evidence freshness, method completion, field readiness, or tooling coherence for frontline teams. Employee capability can be linked to faster diagnosis, fewer handoff failures, lower rework, or clearer decision rights. External value can be observed through customer effort, uptime, resolution trust, or renewal behavior. Profit outcomes then become the downstream consequence of a designed and measured service system rather than a heroic hope.

Aurora Vento is useful again here because the company's servitization move only works if internal quality improves first. If dispatch planning remains chaotic, if technicians lack the right part logic, if shared review rituals do not exist, then no amount of contractual language will create customer value. The Service Profit Chain reminds me that the backstage experience for staff and partners is part of the product. In ASD terms, that means the Measurer and the Critic become essential business roles, not reporting accessories.

My take: Most teams jump straight to customer metrics because they are easier to sell upstairs. That is usually too late. If the internal service system is incoherent, the customer

measures are already downstream casualties.

This is why AI deployed only as a cost-cutting device often backfires. If the agent layer strips discretion, floods people with low-trust output, or hides accountability, internal quality deteriorates. The Service Profit Chain predicts the rest: frontline capability falls, customer value thins out, loyalty erodes, and the business calls the transformation disappointing. ASD tries to avoid that trap by using agents to strengthen the internal service system rather than to bypass it.

SDL Commitments in the Platform Contract

The final question is whether the theory actually shows up in the platform contract. I think it does, and I can say exactly where.

First, SDL's insistence on value-in-context appears in the evidence discipline. Every consequential claim in ASD must connect to evidence, because the system refuses to treat internal conviction as beneficiary truth. Second, operant-resource logic appears in the role model and method surface. Roles are specialized competences. Methods are executable routines for applying those competences. Third, value co-creation appears in the artifact model itself: journeys, stakeholder maps, ecosystem maps, and blueprints all force the team to represent multiple actors and their relationships, rather than provider activity alone.

Fourth, the Service Profit Chain appears in the measurement surface. KPI trees, audits, and baselines force the system to make internal quality and external value legible. Fifth, institutional arrangements show up in the human gate. A phase-advance receipt is not bureaucracy for its own sake. It is the explicit institutional act that says a sponsor, service lead, or governance body is prepared to own the next commitment.

Even the Critic supports SDL. A veto is not merely a software safeguard. It is a protection against the provider-centric fantasy that a team can announce value

without properly grounding it. The `review-veto-log` records where the system overreached. That is operational humility written into the platform.

phase-advance.json

```
{
  "projectId": "aurora-vento-808ed8c3",
  "toPhase": "develop",
  "idempotencyKey": "gate-define-1",
  "approvalReceipt": {
    "receiptId": "gate-define-1",
    "decision": "approve",
    "rationale": "Value proposition, KPI tree, and co-creation boundaries accepted."
    "approvedBy": "service-lead",
    "approvedAt": "2026-08-23T15:30:00Z"
  }
}
```

That object is more than a control artifact. It formalizes what SDL has always implied: service systems move when actors commit resources under real institutional conditions. An LLM cannot will that commitment into existence. It can only help the organization prepare for it honestly.

This is where ASD earns its theoretical seriousness. The platform does not bolt service design language onto generic AI tooling. It encodes service-dominant commitments into the execution path: evidence before assertion, roles as competences, artifacts as provisional value propositions, review before promotion, measurement tied to value-in-use, and human authority at consequential boundaries.

So when I say agent teams fit service-dominant practice, I do not mean they are fashionable companions for workshops. I mean they can act as operant resources inside a rigorously designed co-creation system. They help me research faster, synthesize deeper, instrument the Service Profit Chain, and challenge weak claims earlier. They do not abolish the beneficiary. They do not replace the institution. They do not nullify the designer. They sharpen all three.

> **CORE PRINCIPLE:** The point of agentic service design is not to automate away co-creation. It is to build a runtime that makes co-creation more inspectable, more disciplined, and easier to sustain at scale.

That is the practical force of SDL inside ASD. It keeps the platform from becoming a very fast artifact factory. It keeps the work oriented toward lived service outcomes, shared operating routines, and measured value in context. Without that lens, agentic service design would degenerate into output theater. With it, the system becomes a serious instrument for designing and governing services that people can actually use, trust, and improve.

CHAPTER 02

The ASD Platform Architecture

CHAPTER 04

The Role Contract, Evidence Registers & Deterministic Fact-Checking

NEXT CHAPTER

The Role Contract, Evidence Registers & Deterministic Fact-Checking

Handoffs are artifacts, never chat

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 04

The Role Contract, Evidence Registers & Deterministic Fact-Checking

Handoffs are artifacts, never chat

Reading time

13 min

The Law of Artifacts: Handoffs Are Never Chat

The first mistake most teams make with multiple agents is social. They assume more conversation means more intelligence. In practice, it usually means more compression. The Researcher paraphrases an interview. The Mapper paraphrases the paraphrase. The Critic then reviews a derivative story rather than the thing that actually happened. After three hops, the original evidence is gone. ASD forbids that pattern at the contract level. A role may explain itself in a host interface, but the durable handoff is always a typed artifact written through the store.

This matters because ASD is not a chat choreography layer sitting above a hidden database. It is one canonical practice contract and one runtime registry projected into twelve targets. The CLI, MCP server, coding-agent packs, BUZZ Persona Pack, and DeepSeek Harness Studio all touch the same underlying ideas: 10 roles, 80 shipped skills from 81 sources, 30 `sd_*` tools, 228 resources, 10 role prompts, 38 registered artifact-kind contracts, 39 vocabulary rows, 7 authored templates, and 12 emitted targets. The human-friendly artifact vocabulary lives in Appendix A. The command and tool surfaces live in Appendix B. In this chapter I stay on the write path itself.

The role contract is deliberately boring. That is a feature. A role runs a registered method, writes the method's declared output kind, and only then can another role consume it. If a handoff needs a condition, ASD records one. If a handoff names an artifact kind, that pointer must resolve inside the same project. No invisible context. No magical side channel. No "trust me, the previous agent covered it."

platform/_shared/role-contract.md

```
## 1. Handoffs are artifacts, not chat

1. Role A completes a registered method and writes its declared typed artifact.
2. The Orchestrator presents the gate and records the human decision through asd pha
3. Role B consumes the artifact; it never reconstructs the work from conversation.
4. Everything shown to a model must be reconstructable from the store.
5. Parallel work is explicit; writes still serialize through the project store.
```

Once I adopt that rule, the rest of the platform gets sharper. The Researcher does not hand the Mapper a speech about what riders felt. The Researcher hands over an `insight-set` or a `journey-map` draft with evidence pointers that can be inspected, reviewed, vetoed, replayed, and exported. The Orchestrator does not quietly decide that the team is ready for the next phase. The Orchestrator prepares the gate record, then a human authorizes it with an approval receipt. Even the Critic does not shout into the void. It writes rows.

My take: The artifact rule feels heavy for about three days. Then a stakeholder asks why a recommendation exists, I open the store, and the answer is sitting there as evidence IDs, artifact JSON, review rows, and a gate receipt. That is when the weight starts to feel like relief.

QUESTION	CHAT-NATIVE ANSWER	ASD ANSWER
What does one role hand to the next?	A summary in prose.	A typed artifact plus resolvable evidence references.
How do I know what the model actually saw?	I usually cannot prove it.	The store, log, artifact, and evidence must tell the same story.
How does parallel work stay coherent?	Thread memory and goodwill.	Explicit orchestration plus a single serialized write path.
How is a gate crossed?	Someone says we are done.	A human records an approval receipt through `asd phase-advance`.

Evidence Registers and Content Addressing

The artifact rule only works because artifacts do not float free. They stand on a register, and the register is closed. ASD allows exactly three evidence kinds: `quote`, `telemetry`, and `document`. That small vocabulary does more work than a bloated ontology ever could. Quotes preserve verbatim human voice. Telemetry preserves measured operational facts. Documents preserve written institutional context: field notes, rosters, policies, notices, screenshots captured into prose, and the dull paperwork that usually kills overconfident ideas.

Each evidence row is content-addressed. The id is the SHA-256 of `(source, kind, body)`. Same source, same kind, same body, same id. Different body, different id. Re-import a transcript twice and you do not get duplicate truth; you get the same row.

That property is what lets me cite an evidence id in a journey node, in a Critic defect row, and later in a traceability report without inventing a second naming scheme.

terminal / asd evidence-add

```
asd evidence-add --projectId lighthouse-transit-f7e5f8a2   --source call-821004   --

asd evidence-add --projectId lighthouse-transit-f7e5f8a2   --source avl-extract-2025

asd evidence-add --projectId lighthouse-transit-f7e5f8a2   --source rosters-07   --k
```

If I try to smuggle in an `observation` kind, the CLI rejects it. Good. An observation is not a fourth primitive truth type in ASD. It becomes either a quote or a document, depending on what it actually is. That distinction sounds fussy until the first time somebody tries to promote a designer's interpretation into ground truth. The platform says no before the meeting gets confused.

> **CORE PRINCIPLE:** Evidence is allowed to be incomplete. It is not allowed to be vague about what kind of thing it is.

KIND	WHAT IT CAPTURES	TYPICAL SERVICE-DESIGN USE
`quote`	Verbatim human words.	Interview lines, complaint excerpts, diary snippets, frontline speech.
`telemetry`	Measured operational or behavioral data.	Queue times, no-show rates, completion rates, usage extracts, ticket volumes.
`document`	Written context and institutional constraints.	Policies, rosters, field notes, consent forms, service standards, notices.

One more boundary matters here. Appendix A lists the platform's artifact vocabulary and its aliases; the register is not that catalog. Evidence rows are smaller and blunter than artifacts on purpose. I want the raw facts to stay primitive so the design layer above them can be checked mechanically.

Acceptance Rules, Failed Writes, and the Draft Lane

Most systems do something unhelpful at write time. They either accept too much and hope review catches it later, or they reject too hard and leave no record of what almost got in. ASD takes a better path. Every method manifest names an acceptance contract, and failed writes persist as `draft` with machine-readable failures attached. That means the platform tells the truth twice: it tells the truth that the artifact exists, and it tells the truth that the artifact is not yet good enough.

The acceptance vocabulary is intentionally tiny: `min-nodes`, `min-evidence-per-node`, and `field-non-empty`. Some methods add a narrow extra rule such as `forbid-field-values`, but the platform's core posture is still compact. A chapter like this one can explain the whole thing because there are not thirty hidden validators. For example, `contextual-inquiry` demands at least 3 nodes, at least 1 evidence row per node, and a non-empty `theme` field. `transcript-mining` tightens the evidence floor to 2 and the node floor to 4. `behavioural-personas` requires 2 evidence rows per persona, a non-empty `goal`, and rejects demographic description as a defining basis. The manifest is the contract. The write path enforces it.

artifact-write.failure.json


```
{
  "accepted": false,
  "status": "draft",
  "artifact": {
    "id": "2f537689386e439abd3ae43769ab710c",
    "kind": "journey-map",
    "methodId": "journey-from-transcripts",
    "status": "draft",
    "check": {
      "pass": false,
      "failures": [
        {
          "rule": "min-nodes",
          "message": "expected at least 3 nodes, found 1"
        },
        {
```

```
            "rule": "min-evidence-per-node",
            "nodeId": "transfer",
            "message": "node \"transfer\" cites 0 evidence refs, needs 1"
          },
          {
            "rule": "field-non-empty",
            "nodeId": "transfer",
            "message": "node \"transfer\" field \"touchpoint\" must be non-empty"
          }
        ]
      }
    },
    "failures": [
      "expected at least 3 nodes, found 1",
      "node \"transfer\" cites 0 evidence refs, needs 1",
      "node \"transfer\" field \"touchpoint\" must be non-empty"
    ]
  }
```


The subtle point is the status line. Even a structurally accepted artifact remains `draft` until the review gate is cleared. So ASD really has three observable moments even though it exposes two statuses: a failed draft, an accepted draft, and a final artifact. That middle state matters. It means the method contract passed, but the Critic has not yet granted promotion.

> **CAUTION:** Draft does not mean disposable. If you treat drafts like scratchpads, you lose the audit trail that explains why the eventual final looked the way it did.

RULE	WHAT IT PROTECTS	TYPICAL FAILURE
`min-nodes`	Prevents tiny artifacts from pretending to cover a system.	A journey with one stage, a research plan with one activity, a persona set with one persona.
`min-evidence-per-node`	Prevents unsupported claims inside otherwise well-formed artifacts.	A pain point with zero quotes, a persona defined by one anecdote, a gap row citing nothing.

RULE	WHAT IT PROTECTS	TYPICAL FAILURE
`field-non-empty`	Prevents placeholder semantics from slipping into delivery.	`touchpoint: ""`, `theme: "tbd"`, `lane: "n/a"`.

Because the failure report stays attached, remediation gets precise. I do not ask an agent to "improve the journey." I tell it to rewrite artifact `2f537689386e439abd3ae43769ab710c`, add evidence to node `transfer`, fill `touchpoint`, and bring the node count above the floor. Specificity is the reward for preserving the failure instead of hiding it.

The Critic on the Write Path: Pass, Veto, and `review-veto-log`

Acceptance is necessary. It is not sufficient. A journey can meet its node floor and still propose a fantasy. A blueprint can cite real evidence and still violate an operational constraint hidden in a document row. That is why ASD makes review a data structure, not an opinion. If a method declares `reviewGate: critic`, the artifact is only final after `asd review` returns a valid verdict.

A pass is stricter than people expect. It still needs evidence-cited rows. You do not get to pass with an empty array because "everything looked fine." The minimum honest pass is an evidence-cited proposal row. A veto is stricter still. It needs at least one defect row, and every defect row needs a fix. That is what turns review from hand-waving into an executable remediation loop.

terminal / asd review

```
asd review --projectId lighthouse-transit-f7e5f8a2   --artifactId 2f537689386e439abd
    {
      "rowKind": "defect",
      "severity": "high",
      "rule": "evidence-sufficiency",
      "message": "The missed-connection node needs the roster-constraint document ci
      "fix": "Cite rosters-07 on the transfer node.",
```

```
        "evidenceId": "ebc691cacc31cf5fb2db2f3c8347e5e085690a0971b0d3ca4d104a6e150fdc6
      }
    ]'
```

review-veto-log.json

```
{
  "id": "a3c09c90fdb24c49be6efd1c1a58ce43",
  "kind": "review-veto-log",
  "title": "Review: Current-state journey",
  "status": "final",
  "reviewOf": "2f537689386e439abd3ae43769ab710c",
  "nodes": [
    {
      "id": "r1",
      "label": "The missed-connection node needs the roster-constraint document cite
      "evidence": [
        {
          "evidenceId": "ebc691cacc31cf5fb2db2f3c8347e5e085690a0971b0d3ca4d104a6e150
        }
      ],
      "fields": {
        "rowKind": "defect",
        "severity": "high",
        "rule": "evidence-sufficiency",
        "message": "The missed-connection node needs the roster-constraint document
        "fix": "Cite rosters-07 on the transfer node."
      }
    }
  ]
}
```

This artifact is administrative, and that is an important distinction. A `review-veto-log` is part of the runtime registry, but it is not one of the flagship fillable templates. It exists because governance itself needs evidence. The Critic's row is not a Slack comment. It is a final artifact linked by `reviewOf` to the artifact it judged. On a later pass, ASD reuses that same log artifact and appends the new rows so the story stays continuous.

Promotion therefore becomes exact. Failed acceptance yields `draft`. Accepted write still yields `draft`. Critic veto yields `draft` plus a final `review-veto-log`. Critic pass may promote the artifact to `final`. Nothing else does.

> **My take:** The Critic becomes politically useful the moment its veto stops sounding like preference. If the row says which evidence id blocks the claim and which fix would satisfy the gate, the conversation changes from argument to workmanship.

The Fact-Check Triad and Why CI Trusts Exit Codes

The platform's final mechanical witness is `asd factcheck`. It replays the project as it exists now, not as it was when a role first wrote an artifact. That distinction matters. A manifest may tighten later. A bad ref may have slipped into an older file. A human may have copied something around the edges. `factcheck` does not care about excuses. It re-runs three tests for every artifact: integrity, acceptance, and review.

Integrity asks whether every cited evidence id resolves. **Acceptance** re-runs the method's rules against the live registry. **Review** checks that any artifact claiming `final` under a gated method actually carries a pass verdict. Drafts can remain drafts honestly. Failed finals cannot sneak through.

terminal / asd factcheck

```
asd factcheck --projectId lighthouse-transit-f7e5f8a2
```

factcheck-report.json

```
{
  "pass": false,
  "rows": [
```

```
    {
      "artifactId": "2f537689386e439abd3ae43769ab710c",
      "kind": "journey-map",
      "status": "draft",
      "integrity": {
        "pass": true,
        "dangling": []
      },
      "acceptance": {
        "pass": false,
        "failures": [
          "node \"transfer\" cites 0 evidence refs, needs 1"
        ]
      },
      "review": {
        "pass": true,
        "gate": "critic",
        "verdict": null
      },
      "methodId": "journey-from-transcripts",
      "methodResolution": "recorded"
    }
  ]
}
```


The row tells me exactly why the build is red, but CI does not even need to read the JSON. It reads the exit code. When `pass` is false, `asd factcheck` exits 1. That single design choice turns service-design rigor into an enforceable delivery gate. A project cannot talk its way into green. It must satisfy the replay.

> **CORE PRINCIPLE:** A green-looking report is theater if the command still exits 1. CI reads the exit code, not the meeting energy.

This is also where the platform's honesty shows. A draft is allowed to exist. A veto log is allowed to exist. Open work is allowed to exist. What is not allowed is pretending that open work is ready for export, phase movement, or executive reporting. The six report kinds render deterministically, but only from what the store can defend.

Human Phase Gates, Approval Receipts, and the Boundary of Authority

After all that machinery, the most important rule is still human. ASD does not infer approval from a passing artifact. It does not promote a project because the Critic was satisfied. It does not let ACP pretend to authorize work. Only a human phase gate moves the project from discover to define, define to develop, or develop to deliver, and that move must carry an approval receipt.

The receipt is the antidote to fuzzy governance. Somebody approved. Somebody stated a rationale. Somebody put a timestamp on it. The command is idempotent because gates are too serious to duplicate casually.

approval-receipt.json

```
{
  "receiptId": "gate-discover-1",
  "decision": "approve",
  "rationale": "Discover evidence and synthesis are complete; the open risks are doc
  "approvedBy": "service-lead",
  "approvedAt": "2026-08-22T12:00:00Z"
}
```

terminal / asd phase-advance

```
asd phase-advance --projectId lighthouse-transit-f7e5f8a2   --toPhase define   --ide
    "receiptId": "gate-discover-1",
    "decision": "approve",
    "rationale": "Discover evidence and synthesis are complete; the open risks are d
    "approvedBy": "service-lead",
    "approvedAt": "2026-08-22T12:00:00Z"
  }'
```

In practice the tool also protects the shape of the gate. The transition must be legal. The receipt must be non-empty and well formed. The project must have at least one design artifact, passing `factcheck`, and no unresolved Critic veto standing in the way. That is why phase change belongs to the Orchestrator plus a human authority, not to whichever role happens to be loudest. The system can prepare the case; it cannot sign its own release.

This is also the point where ACP's boundary matters. `asd-acp/1` is record-only. It can log a human gate, a handoff, an unsupported dispatch, or a Critic veto. It cannot promote a phase by narrative implication. BUZZ messages are not ACP transport. DSH UI events are not ACP transport. A host may present the gate beautifully, but the durable project truth is still the gate record.

> **My take:** I do not want a platform that sounds decisive. I want one that knows exactly where its authority ends. ASD earns trust because it is willing to stop at the human gate and wait for a named person to own the call.

Put all of this together and the chapter's thesis becomes simple. Agents can discover, cluster, map, critique, and render. They cannot improvise the evidence, waive the acceptance floor, erase a veto, fake a fact-check, or authorize the next phase. That is why ASD stays useful under pressure. The platform is opinionated about what must be true before design becomes durable, and it keeps those truths inspectable all the way down to the row.

Service-Dominant Logic & Value Co-Creation in the Age of AI

Research Ops at Scale: Mining Transcripts & Field Inquiries

NEXT CHAPTER

Research Ops at Scale: Mining Transcripts & Field Inquiries

Two hundred interviews, one agent team

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03

CHAPTER 04

Service-Dominant Logic & Value Co-Creation in the Age of AI

The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 05

Research Ops at Scale: Mining Transcripts & Field Inquiries

Two hundred interviews, one agent team

Reading time

13 min

The Real Bottleneck Is Not Interviewing. It Is Traceable Synthesis.

Teams love to talk about "doing the research." What they usually mean is recruiting, interviewing, shadowing, and maybe running a survey. Hard work, yes. But the operational cliff appears one week later, when the corpus is large enough that nobody can hold it all in working memory. At that moment people start trimming the data to fit the wall rather than expanding the wall to fit the data. The corpus shrinks. Quotes get paraphrased. Saturation gets declared because the calendar is tired.

ASD treats that failure as a systems problem. Research operations is a pipeline, not a heroic workshop. The human designer still owns rapport, consent, framing, and

participant-facing judgment. The platform owns the boring rigor around storage, method contracts, acceptance rules, review, and replay. Once I separate those responsibilities, 200 interviews stops sounding reckless and starts sounding like throughput.

That separation is why the CLI matters. The command surface is plain: `asd project-create`, `asd evidence-add`, `asd method-run`, `asd artifact-write`, `asd review`, `asd factcheck`. The engine is telling me that research is not a floating memo. It is a series of durable records with gates between them.

> **CORE PRINCIPLE:** The research corpus is a database before it is a deck. If your analysis cannot survive as a store of evidence-backed artifacts, it will not survive executive challenge either.

Calder Community Bank is the cleanest illustration. The board wanted whole-service discovery across SME onboarding, branch service, digital, and complaints. A traditional agency bid assumed sixteen weeks and quietly expected the corpus to collapse to forty interviews. The ASD posture was the opposite. Keep the corpus. Industrialize the processing. Protect the evidence boundary so the volume stays believable.

> **My take:** I stopped trusting declarations of saturation years ago. I trust a visible quota burn-down, a codebook version, a reviewable evidence register, and a method contract that says what "enough" means for this artifact. Everything else is mood.

Research Plans Are Typed Artifacts, Not Prelude Text

The first move in a scaled study is to stop treating the research plan as throat-clearing. In ASD the plan itself is a registered artifact kind: `research-plan`. That

changes the behavior of the whole engagement. If the plan is an artifact, it must declare sample, method, timing, and provenance in a form that can be reviewed, failed, rewritten, and handed off. A vague PDF becomes impossible.

The `research-plan-design` method makes that explicit. Its output kind is `research-plan`. Its acceptance rules are small but unforgiving: minimum 2 nodes, non-empty `sample`, non-empty `method`. Its Critic instruction is equally specific: reject any plan whose sample size or recruitment channel is left unspecified. That is the right level of rigor for planning. I do not need fake certainty. I do need named activities with declared coverage.

terminal / asd project-create and method-run

```
asd project-create --name "Calder Research Ops"   --client "Fictional composite — Ca

asd method-run --methodId research-plan-design   --projectId calder-research-ops-4e2
```

research-plan.method-contract.json

```
{
  "method": {
    "id": "research-plan-design",
    "name": "Research Plan Design",
    "phase": "research",
    "reviewGate": "critic"
  },
  "acceptance": [
    {
      "rule": "min-nodes",
      "min": 2
    },
    {
      "rule": "field-non-empty",
      "field": "sample"
    },
    {
      "rule": "field-non-empty",
      "field": "method"
    }
```

```
    ],
    "outputKind": "research-plan"
  }
```

Once Calder's plan is an artifact, the bank's hardest questions become explicit research nodes instead of hidden assumptions. One node can cover abandoned SME onboarding cases. Another can cover branch-loyal seniors at closure-risk branches. Another can cover complaint handlers. The plan carries the research architecture in durable form, and that means the rest of the pipeline can attach itself to something real.

PLAN ACTIVITY	SAMPLE FIELD	METHOD FIELD	WHY IT EXISTS
SME onboarding interviews	50 interviews across owners, finance managers, and relationship managers	Semi-structured interviewing	Expose handoff friction and document rework.
Branch-shadowing tranche	45 days across rural, commuter, and high-complaint branches	Contextual inquiry	Capture the service as performed, not as described.
Complaint-case review	40 recent complainants plus complaint handlers	Transcript mining	Surface recurrent breakdown language at volume.

A plan artifact also improves handoffs. After the Critic passes it, the plan can be handed to the next role with open gaps visible. That matters more than people think. Research operations breaks when the fieldwork coordinator, analyst, and designer each believe they are working from the same plan while quietly editing different mental versions of it.

Contextual Inquiry Turns Fieldwork into Evidence Instead of Folklore

Large-scale transcript work gets most of the attention, but service design still begins in the messy world. I need to watch the banker re-enter a document because the branch scanner jammed. I need to see the rider miss the connection. I need to hear the parent explain the workaround while doing it. Contextual inquiry keeps research grounded in performed reality rather than post-rationalized explanation.

ASD's `contextual-inquiry` contract is compact and useful. It produces an `insight-set`. It requires at least 3 nodes, at least 1 evidence row per node, and a non-empty `theme` field. The Critic rejects any insight not traceable to a specific observed moment. That sentence is doing heavy lifting. It stops the designer, and the model, from upgrading mood into fact.

In Calder the field discipline looked like this: humans ran the branch, onboarding, and complaints shadowing; every note became either a `document` row or a `quote` row, depending on whether it was observed context or verbatim speech; the resulting evidence could then feed `contextual-inquiry` or `transcript-mining` without inventing a second evidence system. One research database. Multiple methods. Clean boundaries.

contextual-inquiry.nodes.json


```
[
  {
    "id": "branch-queue-opacity",
    "label": "Branch visitors cannot tell whether the line is moving or stalled",
    "fields": {
      "theme": "status visibility"
    },
    "evidence": [
      {
        "evidenceId": "6e3150d6830b02c2f0cdb2b99bfa417b9f9e9dc9a17113e71d2d9913cd9d1
      }
    ]
  },
  {
    "id": "sme-document-rework",
    "label": "Owners resubmit the same document because acknowledgement is missing",
    "fields": {
      "theme": "confirmation failure"
```

```
      },
      "evidence": [
        {
          "evidenceId": "a6a0d6ee6bf6f3ba13fd8fa5f739b04db769e2c26c5b51a0ec9a246fd0e33
        }
      ]
    },
    {
      "id": "complaint-looping",
      "label": "Customers retell the complaint because ownership disappears between te
      "fields": {
        "theme": "handoff opacity"
      },
      "evidence": [
        {
          "evidenceId": "f44379f1b00aa782c212d2a9c63b09189d7e688ef1d02c2428461d52f383f
        }
      ]
    }
  ]
```


That JSON is boring on purpose. Three nodes. Three themes. Three evidence ids. No mysticism. The platform is asking a simple question: can you point to the moment? If yes, write it. If not, keep working.

> **CAUTION:** Do not confuse optional human interpretation with optional provenance. You may still debate what an observed moment means. You do not get to skip recording the moment itself.

Transcript Mining, Coding Frames, and Honest Saturation

Transcript mining is where scaled research either becomes trustworthy or becomes a hallucination factory. The common failure mode is obvious: somebody pastes twenty transcripts into a chat window and asks for themes. The model obliges. The quotes

are smooth. The themes feel plausible. The offsets do not exist. The extraction logic is unrecoverable. The corpus becomes a style exercise.

ASD's answer is not one magical tool. It is a contract plus a discipline. The `transcript-mining` method requires at least 5 evidence records before it will even run. Once available evidence exists, `asd method-run --methodId transcript-mining` returns the contract: phase research, owner researcher, output `insight-set`, acceptance `min-nodes: 4`, `min-evidence-per-node: 2`, and non-empty `theme`. The Critic rejects any insight that merely paraphrases a flagged line without adding a distinct theme. That last clause is essential. It stops the model from copying the corpus into a prettier voice and calling it synthesis.

terminal / asd method-run --methodId transcript-mining

```
asd method-run --methodId transcript-mining   --projectId calder-research-ops-4e2f9a
```

transcript-mining.method-contract.json

```
{
  "method": {
    "id": "transcript-mining",
    "name": "Transcript Mining",
    "phase": "research",
    "reviewGate": "critic"
  },
  "inputs": [
    {
      "kind": "evidence-set",
      "min": 5,
      "note": "call, chat or interview transcripts"
    }
  ],
  "acceptance": [
    {
      "rule": "min-nodes",
      "min": 4
    },
    {
      "rule": "min-evidence-per-node",
```

```
        "min": 2
      },
      {
        "rule": "field-non-empty",
        "field": "theme"
      }
    ],
    "outputKind": "insight-set"
  }
```

Calder used this contract against a rolling corpus. Interviews came in waves. So did evidence. The bank never waited for interview 200 before learning something. That would have been wasteful. Instead the transcript-mining pass re-ran on the growing evidence base, while the codebook evolved at declared checkpoints rather than by casual drift. Week 3 might still be immature. Week 7 might genuinely stabilize a theme. The point is that the platform preserves both the finding and the conditions under which the finding was produced.

Coding frames matter here. Calder did not let every analyst invent categories on the fly. The codebook version moved in declared steps: seed codes for access, effort, trust, status visibility, fees, handoffs, staff capability, and life events; then add or merge codes only at checkpoint reviews. That gave the team two benefits. First, disagreement became inspectable. If one agent tagged a line as `status visibility` and another tagged it as `fees`, the dispute had a bounded vocabulary and could be resolved against the codebook rather than by charisma. Second, rolling synthesis stayed comparable week to week. The cluster named in week 4 still meant the same thing in week 7 unless the codebook change log said otherwise.

The coding frame is also where good research operations resists false precision. A code is not a verdict. It is a lens. I can tag a line as `handoff opacity` without claiming I have already solved the bank's organizational design. The mistake is to let coding drift straight into solutioning. ASD prevents that by making the method output an `insight-set`, not a roadmap. The next role still has to do its own work.

Now saturation. I treat saturation as a reporting discipline, not a mystical threshold. If rural branch seniors have only 14 completed interviews against a planned 18, I say so. If decline cases in SME onboarding came in below target because the bank refused

another contact wave, I say so. That honesty is not a weakness. It tells the next phase which claims are strong, which are moderate, and which need another pass.

STRATUM	TARGET	COMPLETED	STATUS
SME owners who abandoned onboarding	8	6	Gap remains explicit; no false saturation claim.
Branch-loyal seniors in at-risk rural areas	18	18	Target met; branch-closure evidence is decision-grade.
Recent complainants by unresolved outcome	12	12	Target met; complaint handoff themes triangulated.
Complaint handlers and branch staff	22	24	Over target; operational counter-evidence available.

My take: Honest gaps strengthen the readout because they tell the sponsor where uncertainty still lives. Fake completeness does the opposite. The first hard question blows it up.

Case Spine: Calder's 200 Interviews and Aster Grove's Synthetic Setup

Calder is the heavy-duty spine. Two hundred interviews. Forty-five shadow days. Eight hundred diary entries. A ten-week research program with a board ready to challenge every conclusion. The agent team handled the transcript-facing work: ingestion discipline, evidence extraction, rolling clustering, review gates, and factcheck. Humans handled interviewing, consent, and the judgment calls around quota trade-offs. The result was not "AI did the research." The result was that the human research team stopped drowning in post-fieldwork admin and could keep the full corpus alive long enough to learn from it.

Calder's weekly operating rhythm is worth naming because it is where scaled research usually breaks. Monday and Tuesday were field-heavy. Wednesday night the new transcripts were ingested and verified. Thursday the rolling synthesis pass refreshed cluster counts and highlighted contradictions. Friday the designer and client leads reviewed only the red items: quota gaps, PII incidents, weak clusters, and any Critic veto. That cadence kept the corpus alive without forcing a giant end-of-program synthesis panic. It also meant the board saw evidence maturity rising over time rather than being asked to believe a single dramatic reveal.

Seen this way, the agent team is less like a replacement researcher and more like a disciplined research-ops layer. It makes the infrastructure of inquiry explicit. That is what lets a 200-interview program remain legible enough to challenge, defend, and hand over.

Aster Grove is the complementary lesson. Article 39 begins with a fictional service and a live tutorial workflow. The service case is synthetic. The BUZZ roster and Canvas round trip are real. That makes it the perfect example of how to start safely when I need to teach the loop, rehearse a method, or demonstrate a host integration boundary. I create an isolated project, add synthetic evidence rows with clear labels, then run the method and review loop honestly as synthetic practice. Nothing in that move is lesser. It is only dangerous if I later pretend the seeded evidence came from live participants.

terminal / Aster Grove synthetic evidence setup

```
asd project-create --name "Aster Grove Makerspace"   --client "Fictional composite –

asd evidence-add --projectId aster-grove-makerspace-6a8498ef   --source librarian-01

asd evidence-add --projectId aster-grove-makerspace-6a8498ef   --source usage-2026-0

asd evidence-add --projectId aster-grove-makerspace-6a8498ef   --source policy-03 --

asd method-run --methodId transcript-mining   --projectId aster-grove-makerspace-6a8
```

That shell block is the right way to teach or rehearse. It uses real commands. It uses the real evidence vocabulary. It keeps the fictional boundary visible in the project name, client field, and source rows. It does not invent a parallel demo mode.

Between Calder and Aster Grove, the operational lesson becomes clear. Scale is not the enemy. Ambiguity is. Calder proves that a 200-interview corpus can stay tractable when the plan, evidence, method contract, review, and factcheck all stay explicit. Aster Grove proves that synthetic evidence can be legitimate when it is labelled, isolated, and never upgraded into fake field proof. Same engine. Different evidence boundary. Same honesty rule.

> **CORE PRINCIPLE:** Synthetic evidence is useful for workflow proof, host proof, and teaching. It is useless for claiming customer truth. Keep those lanes separate.

So here is the operational formula I keep coming back to. Start with a typed research plan. Run contextual inquiry where lived behavior matters. Mine transcripts against a declared contract instead of a vibe. Track coverage honestly. Keep synthetic setups clearly synthetic. And never let scale tempt you into summarizing away the evidence. Once I do that, the research operation stops being a bottleneck and starts being a durable production capability.

What matters is repeatability. When another designer opens the project six months later, the plan, evidence, codebook, gaps, and review rows still explain themselves.

Teams copying this operation usually want a concrete starter sequence. Week one: create the project, write the research plan artifact, and load the codebook you intend to hold yourself to. Week two: capture evidence in bursts, running `evidence-query` at the end of each burst so gaps surface while fieldwork can still fill them. Week three: run `method-run` for synthesis, write the insight register, and let the Critic reject every claim that leaned on paraphrase instead of verbatim rows. That cadence fits a normal sprint, produces an auditable register, and, most importantly, survives the departure of whichever person happened to remember the interviews.

CHAPTER 04

The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 06

Mixed Methods & Evidence-Backed Segmentation

NEXT CHAPTER

Mixed Methods & Evidence-Backed Segmentation

Quant and qual under one evidence roof

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 06

Mixed Methods & Evidence-Backed Segmentation

Quant and qual under one evidence roof

Reading time

13 min

Why Mixed Methods Usually Break in Practice

The phrase sounds sophisticated. Mixed methods. In reality the failure mode is painfully ordinary. The analytics team runs a funnel and produces a cohort split. The research team runs interviews and produces an empathy-heavy readout. Each team is internally rigorous. The organization still ends up confused because the two outputs cannot dispute, confirm, or refine each other at row level. They only meet as presentations.

That is not a methods problem. It is a storage problem. If quant and qual do not land in one evidence system, the reconciliation step becomes theater. Somebody says, "The survey proves users love self-serve." Somebody else says, "The interviews say

they hate it." Nobody can see whether those statements are about the same cohort, the same time window, the same definition of success, or even the same service moment.

ASD's answer is blunt: one register, three evidence kinds, one artifact pipeline. Survey toplines arrive as `telemetry` because they are measured distributions. Survey instruments, questionnaires, store procedures, and consent forms arrive as `document`. Human words arrive as `quote`. The boundary is not academic. It is what lets me fuse a wait-time extract, a field note, and a verbatim complaint into one evidence-backed node without inventing a fourth class called "insight" halfway through.

> **CORE PRINCIPLE:** Mixed methods works when the evidence disagrees in one visible place. If the disagreement happens only in separate slide decks, it is already too late.

Vesper Pharmacy makes the point cleanly. The chain's dashboards said click-and-collect should be the low-effort channel. The survey said customer effort was worse than walk-in. Interviews said the ready-notification could not be trusted. Store operations said the problem was customer unreliability. All of those claims were individually plausible. None were useful until the evidence entered one register and could be cited together inside the same artifact rows.

Telemetry, Survey Data, and Documents Under One Roof

The second-edition discipline matters here because the platform never added a special `survey` evidence kind. That omission is wise. A survey produces several kinds of evidence. The raw instrument is a document. The distribution of answers is telemetry. Free-text comments are quotes. Treating the whole survey as one mystical object is exactly how teams hide category errors inside a respectable method name.

In Vesper, for example, a ready-notification timing extract is telemetry. A store SOP that explains when staff are supposed to move orders onto the shelf is a document. A shopper saying, "The app said ready. It was not ready. So now I phone," is a quote. The mixed-methods job is to place those three rows next to each other and let them argue honestly. The analytics row tells me how often the failure happens. The document row tells me what the organization thinks should happen. The quote tells me what the failure costs in trust.

Evidence-strength tagging is the underrated part of this pattern. In Vesper's triangulation matrix, a vivid interview line never outranked a weak distribution on charm alone. A theme reached `strong` only when the quant row had adequate sample and effect, the qual row had at least three independent sources, and operations corroborated the same mechanism. Moderate meant two streams. Weak meant one. That sounds procedural, but it changes the politics of synthesis. The loud anecdote loses its ability to hijack the room. The rare but severe event can still stay visible, yet it stays labelled as rare until more evidence upgrades it.

Conflict handling follows the same logic. If telemetry says wait time is short and interviews say waiting feels endless, I do not average them into mush. I ask what they are each measuring. Usually one is measuring duration and the other legibility. At Vesper that distinction mattered because queue opacity, not raw minutes, explained the low CSAT in the fast elderly cohort. The conflict was the insight.

terminal / telemetry, document, and quote capture

```
asd evidence-add --projectId vesper-pharmacy-1c7293aa   --source pickup-events-2026q

asd evidence-add --projectId vesper-pharmacy-1c7293aa   --source sop-ready-shelf-v4

asd evidence-add --projectId vesper-pharmacy-1c7293aa   --source int-31   --kind quo
```

Notice what this does to interpretation. I no longer have to choose between numbers and stories. I can write an insight node that cites all three. When the Critic later reviews the artifact, it can ask a real question: does the proposed claim follow from

the telemetry, the document, and the quote together, or is the team leaning too hard on one stream?

RESEARCH INPUT	STORED AS	WHY THAT CHOICE IS CORRECT
Likert-scale survey distributions	`telemetry`	They are measured outputs with a sample size and a time window.
Survey questionnaire and coding guide	`document`	They define instrument context, not respondent truth.
Free-text survey comments	`quote`	They preserve verbatim participant language.
Operational SOP and policy constraints	`document`	They describe the intended service logic or constraint boundary.
Usage logs, queue extracts, completion data	`telemetry`	They quantify what happened and for whom.

Kestrel Energy followed the same rule at much larger scale. Daily aggregate usage distributions, portal-usage shares, switching rates, and contact volumes became telemetry. The metering mandate and the data-protection agreement became documents. Interviews and diary excerpts stayed as quotes. That separation is what let the team combine 2.4 million household records, 1.9 million contact logs, 400 interviews, 60 diary studies, and 28,400 survey verbatims without turning the segmentation into a bag of incompatible formats.

My take: When teams ask for a bespoke evidence kind for every method, they are usually trying to avoid making clean analytical distinctions. The discipline of three kinds forces better thinking.

From Coded Evidence to Segments That Can Survive

Audit

Segmentation gets sloppy when people jump straight from raw data to labels. I do not. I want an intermediate layer of coded evidence and observable behavioral dimensions. Otherwise the cluster is just a prettier demographic bucket. The whole point of mixed methods is to carry the behavioral signal from the raw evidence into the segment definition.

At Vesper, that meant joining quant and qual around explicit dimensions: trust in status messages, channel preference under uncertainty, readiness to wait, sensitivity to queue opacity, and tolerance for self-serve. Some of those dimensions came from telemetry first. Some emerged from interviews first. What mattered was that each dimension had to be backed by evidence rows rather than by organizational lore. For Kestrel, the dimensions expanded into payment posture, channel posture, contact posture, price posture, and consumption posture, but the logic stayed the same. Build the variables from what users do and say. Then cluster.

This is where `behavioural-personas` earns its keep. The method does not ask the Synthesizer to write colorful cards out of thin air. It tells the Synthesizer to cluster participants by observed behavior and goal, never by demographic attribute alone. It tells the next step to draft one persona per behavioral cluster, citing the evidence for the defining goal and the main frustration. It tells the Critic to reject any persona whose defining goal is demographic rather than behavioral.

terminal / asd method-run --methodId behavioural-personas

```
asd method-run --methodId behavioural-personas   --projectId kestrel-energy-f4180ce3
```

behavioural-personas.method-contract.json

```
{
  "method": {
    "id": "behavioural-personas",
    "name": "Behavioural Persona Construction",
```

```
      "phase": "synthesis",
      "reviewGate": "critic"
    },
    "acceptance": [
      {
        "rule": "min-nodes",
        "min": 2
      },
      {
        "rule": "min-evidence-per-node",
        "min": 2
      },
      {
        "rule": "field-non-empty",
        "field": "goal"
      },
      {
        "rule": "forbid-field-values",
        "field": "description",
        "deny": [
          "demographic"
        ]
      }
    ],
    "outputKind": "persona-set"
  }
```

That final denial rule is the quiet assassin of lazy segmentation. It does not ban age from appearing anywhere in a dataset. It bans demographic identity from masquerading as the behavioral explanation. "Over 65" is not a goal. "Needs visible certainty before leaving home" might be. That is why the elderly repeat-patient cohort at Vesper could become a meaningful design segment instead of a stereotype. Their defining issue was not age. It was the need for acknowledged, trustworthy progress through the pickup experience.

> **CAUTION:** A cluster that predicts nothing is not a segment. If it does not change a service decision, it is a spreadsheet ornament.

The Behavioural-Personas Artifact Is a Boundary Object, Not a Poster

Once the mixed-methods evidence and the clustering logic are in place, the output has to stay useful outside the analytics room. That is the job of the `persona-set` artifact. In ASD a persona is not a slide with a stock photo. It is a typed node inside a registered artifact kind, with fields that downstream methods can read and the Critic can inspect. The artifact becomes a boundary object between research, mapping, ideation, measurement, and delivery.

For that reason I care about the fields. The node needs a goal. It needs evidence. It needs a frustration. It needs a label the rest of the team can use without translating it every time. The acceptance floor does not guarantee a beautiful persona. It does guarantee that the persona has at least the minimum ingredients to be defensible.

persona-set.nodes.json


```
[
  {
    "id": "c2-paper-trail-keeper",
    "label": "Paper trail keeper",
    "fields": {
      "goal": "Get certainty that the smart-meter visit will happen exactly as promi
      "description": "Phone-first, documentation-seeking, reassurance-driven househo
      "frustration": "Mandatory-looking changes arrive without clear next steps or t
    },
    "evidence": [
      {
        "evidenceId": "e1c27ce41870f130e3a2f7e827d4d4fe5df12d3d9f35ea71ca98fcf74db2c
      },
      {
        "evidenceId": "8d8d59d615c1a9ddff0d3ca1d6482da2e6ab224f69d42f0f0852958f46dbd
      }
    ]
  },
  {
    "id": "c3-tariff-hawk",
    "label": "Tariff hawk",
```

```
      "fields": {
        "goal": "Check whether every tariff change is economically rational for this h
        "description": "Calculation-driven, channel-flexible, quick-to-switch if arith
        "frustration": "Opaque pricing messages make comparison work necessary and chu
      },
      "evidence": [
        {
          "evidenceId": "443f5dca7c070d3ea499c98e88be5aa0a654bf82f943465ce26cb79e2c994
        },
        {
          "evidenceId": "79b710828f711f5a2e4b00f0b89da2d0c840466f5c22e2d26730d88d977f6
        }
      ]
    }
  ]
```


Because this is a typed artifact, downstream work sharpens. A journey map can be regenerated through the lens of `c2-paper-trail-keeper`. A prototype plan can state which persona it is trying to help. A KPI tree can measure whether the service improved the right behavior. I am no longer waving at a persona in a workshop. I am passing a registered object through the system.

WEAK PERSONA PRACTICE	ASD PERSONA PRACTICE
Demographic summary with a name and a stock photo.	Evidence-cited persona node with behavioral goal and frustration.
One-way presentation artifact.	Registered `persona-set` consumed by later methods.
Qual-only storytelling.	Quant and qual fused into one audit trail.
Debated informally.	Review-gated, fact-checked, promotable from draft to final.

Case Spine: Vesper Pharmacy and Kestrel Energy

Vesper Pharmacy shows the mixed-methods loop at medium scale and high operational consequence. The research team carried 1.8 million pickup events, a 12,038-response survey, 35 shopper interviews, 20 pharmacist shadow days, and 15 mystery shops through one evidence discipline. The breakthrough was not merely discovering that no-shows were high. It was explaining why. The telemetry showed the premature-ready cohort. The document row showed the intended shelf-scan logic. The quote row exposed the trust wound. Once those three lived together, the fix became obvious: ready means ready, and queue visibility matters more than shaving another minute off a wait for the cohort already being served quickly.

Kestrel Energy scales the same discipline into segmentation proper. The utility inherited four demographic buckets from marketing and a mountain of contradictory data. The team rebuilt the evidence base, engineered behavioral features, ran clustering, built a separate qualitative typology, then fused the two into a segmentation that could survive an adversarial check. The most important decision was not computational. It was conceptual. The team refused to let vulnerability become a persona type and refused to let age become the story. That is why the final personas changed operations instead of merely decorating them.

The resulting interventions were precise. Kestrel's paper-trail keepers received assisted-digital booking routes, visible phone paths, and clearer access-preparation guidance. Tariff hawks got transparent arithmetic instead of generic reassurance. Vesper's first-time click-and-collect users got a guided script and a truthful status line. Elderly repeat patients got visible queue progress and acknowledgement rather than faster-but-silent processing. None of those interventions would have emerged cleanly from quant alone or qual alone.

Validation matters here too. Kestrel did not stop at plausible clusters. The household model had to clear a mean silhouette of 0.35, preserve usable cluster share, and survive an independent qualitative derivation that matched most of the resulting types. That is a much better discipline than asking whether the persona names feel right in a workshop. A segment survives because it separates behavior, predicts something the service can act on, and does not collapse when another model family attacks it.

Vesper's equivalent validation lived in the pilot. The mixed-methods logic predicted that status truthfulness and queue visibility would matter more than generic speed optimization for the elderly repeat-patient cohort. The pilot then checked the bet with telemetry, follow-up interviews, and mystery shops. In other words, the segmentation claim stayed reversible until the service proved it in the world.

> **My take:** Mixed methods is not about making peace between numbers people and research people. It is about forcing both sides to stand on the same evidence floor until the service decision stops being arguable in lazy ways.

That is why I treat evidence-backed segmentation as a service-design capability rather than a marketing deliverable. The end product is not a set of audience cards. It is a reliable way to decide which route, script, message, queue, or support pattern fits which behavioral situation. When the artifact is typed, reviewed, and replayable, the segmentation becomes part of the operating system. When it is only persuasive, it dies the first time a dashboard or a stakeholder mood shifts.

One more practical rule: do not let survey percentages float free of cohort definition. If a survey result cannot be tied back to a named slice of the service and a named segment hypothesis, it becomes decoration. Mixed methods earns its keep when every number knows which behavior it is trying to explain.

The practical lesson is simple. Put every method under one register. Store survey findings, documents, telemetry, and quotes in their right evidence forms. Build the behavioral variables explicitly. Demand persona nodes that can survive the Critic. Then let the service change. That is mixed methods with teeth.

That is the hidden advantage of ASD here: the platform gives segmentation a memory. Months later I can still inspect the rows, rerun the check, and see why a service choice belonged to this segment and not another.

One caution from practice: mixed-method projects drift when teams treat the quantitative arm as an audit of the qualitative arm instead of as a second lens on the same behavior. Telemetry that contradicts an interview theme is not an

embarrassment to reconcile away. It is usually the strongest finding in the project, because it marks exactly where stated intent and lived behavior diverge. I put those contradictions into their own evidence rows and make the personas carry them explicitly. A segment description that admits where its two evidence forms disagree is worth more than a tidy one that quietly picked a side.

CHAPTER 05

Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 07

Mental Models, Alignment & Opportunity Mapping

NEXT CHAPTER

Mental Models, Alignment & Opportunity Mapping

Closing the gap between belief and behaviour

Continue Reading →

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 07

Mental Models, Alignment & Opportunity Mapping

Closing the gap between belief and behaviour

Reading time

12 min

Mental Models Are Executable Artifacts, Not Workshop Sketches

Teams often treat mental models as inspirational wallpaper. Somebody runs a workshop, draws towers on a wall, and everybody nods at the idea that users think differently from the org chart. Then the diagram disappears because nobody can attach it to delivery work. ASD fixes that by treating mental-model outputs as artifact contracts rather than facilitation residue.

There are two closely related runtime forms in play. `mental-model-diagramming` produces a `mental-model-map`, which is the skyline: towers, spaces, and the durable cognitive structure of a task domain. `mental-model-alignment` produces a `journey-map`,

because the alignment question is inherently comparative and staged: what the customer thinks is happening, what frontline staff think is happening, where the two diverge, and which touchpoints express that divergence. Those two forms complement each other. One captures structure. The other captures the clash between structures in lived service flow.

The point of keeping both as artifacts is practical. A mental model then stops being a metaphor and starts being a dependency. An opportunity matrix can consume it. A content-model change proposal can cite it. A journey map can express its gaps explicitly. A Critic can veto it when a divergence lacks evidence from both sides.

> **My take:** I do not care whether a mental-model workshop felt profound. I care whether a product owner can point to a node, ask what evidence it stands on, and then make a build decision with it.

ARTIFACT	WHAT IT CAPTURES	WHY IT EXISTS
`mental-model-map`	Towers, mental spaces, beliefs, questions, and tasks.	Expose the durable demand-side structure before solutioning.
`journey-map` from `mental-model-alignment`	Parallel stage-by-stage views of customer and frontline interpretation.	Make divergence visible where service behavior must change.
`opportunity-matrix`	Prioritized responses to the misalignments.	Translate diagnosis into sequencing without losing evidence weight.

This is why the language of belief, question, and task matters so much. If a model says, "The platform organizes grades by class," that is system language. If it says, "I need one picture of how my child is doing," that is user logic. The whole alignment exercise depends on preserving that distinction. Otherwise I end up redrawing the product architecture in a softer font and calling it empathy.

The taxonomy discipline is load-bearing. A belief is a claim the user treats as true about how the service works. A question is an information need. A task is an intended action. If I blur those categories, the model turns mushy fast. "The spinner stopped so my work is saved" is a belief. "Did my grade actually publish" is a question. "Resume yesterday's grading" is a task. Those differences are exactly what later content-model and interaction changes must answer.

That is why the research team at Lumen extracted and reviewed mental-model items before they talked about screens. The point was not to make the workshop more poetic. The point was to stop the design team from collapsing cognition, behavior, and interface preference into one bucket called feedback.

> **CORE PRINCIPLE:** A mental model is useful only when it can survive contact with delivery. That means typed structure, evidence, review, and a downstream consumer.

The Alignment Grid Makes Divergence Concrete

Once the cognitive demand side is explicit, I need a way to compare it with the service as actually performed. That is the alignment grid. Think of it as a disciplined collision table. One column names the user belief or question. One column names the system truth or frontline interpretation. One column names the touchpoint where the clash becomes visible. One column names the consequence. Evidence sits underneath all of it.

The method does not ask for abstract empathy. It asks for paired claims. Customer quote on one side. Frontline or system quote on the other. Shared stage. If both sides are not present, the divergence is not ready. That is exactly what the `mental-model-alignment` contract enforces. The Critic's gate instruction is explicit: reject any divergence marked without a quote from both sides.

The Lumen EdTech case shows why this matters. Parents thought by child. Teachers thought by grading sitting. Students thought by due date. The platform, meanwhile, thought by course and reverse chronology. Each actor was rational on their own terms. The failure demand emerged because the service forced all three groups to think in the platform's ontology. The alignment grid made that mismatch visible as service work rather than as user confusion.

mental-model-alignment.method-contract.json


```
{
  "method": {
    "id": "mental-model-alignment",
    "name": "Mental Model Alignment",
    "phase": "synthesis",
    "reviewGate": "critic"
  },
  "steps": [
    {
      "role": "mapper",
      "instruction": "Build two parallel journeys for the same process: the customer
    },
    {
      "role": "synthesizer",
      "instruction": "Mark every stage where the two mental models diverge, citing t
    },
    {
      "role": "critic",
      "instruction": "Reject any divergence marked without a quote from both sides."
      "gate": true
    }
  ],
  "acceptance": [
    {
      "rule": "min-nodes",
      "min": 3
    },
    {
      "rule": "min-evidence-per-node",
      "min": 2
    },
    {
      "rule": "field-non-empty",
      "field": "emotion"
```

```
      },
      {
        "rule": "field-non-empty",
        "field": "touchpoint"
      }
    ],
    "outputKind": "journey-map"
  }
```

GAP	USER MENTAL MODEL	SERVICE TRUTH	OPERATIONAL COST
Parent aggregation gap	"Show me my child as one story."	Grades and updates are spread across course pages.	Parent search failure, support contacts, low trust.
Teacher draft-state gap	"A grading sitting can be resumed later exactly where I left it."	Drafts are append-only and vulnerable to concurrent-tab overwrite.	Lost work, teacher distrust, high-ticket rework.
Student priority gap	"The top of the feed is what matters next."	Feed order is reverse chronological by posting date.	Missed deadlines, hidden urgency, avoidable panic.

The grid changes the quality of debate. I am no longer hearing vague claims like "parents find it confusing." I am seeing a named divergence, a touchpoint, paired evidence, and a measurable cost. That makes the next step possible.

Opportunity Mapping Needs Evidence Weights, Not Design Charisma

Once misalignments are visible, teams get excited and immediately start ideating. That is understandable. It is also dangerous. Not every gap deserves the same sequence, and not every emotionally resonant gap has the same business gravity. Opportunity mapping is the step that stops the loudest problem from stealing the roadmap by drama alone.

I score opportunities against four questions. First, how strong is the evidence? Second, how severe is the user harm? Third, how much failure demand or delivery risk does the gap generate? Fourth, how reversible is the likely intervention? The score is not meant to pretend objectivity where none exists. It is meant to make trade-offs explicit enough that design, product, engineering, and operations can argue on the same page.

Evidence weight is the anchor. A gap supported by ticket volume, usability failure, interview evidence, and frontline corroboration should outrank a clever but weakly evidenced idea. In Lumen, the by-child parent view, grading-session continuity, and due-date ordering all cleared that bar. A handful of attractive side opportunities did not. They were kept visible but sequenced later.

opportunity-weighting-table.csv

```
id,opportunity,evidence_weight,user_harm,delivery_impact,reversibility,total
X,parent-child-aggregation,5,5,5,3,18
Y,grading-session-state,5,5,4,2,16
Z,due-date-primary-feed,4,4,4,4,16
G4,notification-bundling,3,2,3,5,13
G5,teacher-copy-cleanup,2,2,2,5,11
```

The point is not that every team must use my exact weights. The point is that ASD gives those weights somewhere durable to live. If the opportunity later turns out weaker than expected, I can see whether the original problem was in the evidence, the score, or the execution. Without that memory, prioritization becomes folklore within a quarter.

There is a governance benefit too. When an opportunity score cites evidence ids and an artifact id, leadership can challenge the weighting without denying the existence of the underlying gap. That is a healthier argument. It keeps the fight on sequencing rather than reality itself.

My take: Opportunity maps are where smart organizations quietly smuggle politics back in. Evidence weighting is how I keep the conversation honest without pretending politics

disappears.

Lumen's strongest opportunities also illustrate the difference between interface tweaks and ontology changes. A child-level summary is not a filter bolted onto a course model. A grading session is not a tooltip explaining autosave. A due-date-primary feed is not a cosmetic sort order if the entire student's planning behavior depends on it. Opportunity mapping, done properly, tells me which gaps demand structural change and which can be handled with lighter service moves.

A Valid `method-run` Walkthrough for Mental-Model Alignment

The platform contract matters most when I can actually run it. So here is the shape of the workflow using the real CLI surface. I create the project, capture evidence rows from both sides of the relationship, inspect the method contract, write the alignment artifact, send it to the Critic, and only then run the deterministic replay. The commands below are deliberately plain. No pseudo-tool names. No catch-all shortcut command. Just the shipped interface.

terminal / asd mental-model-alignment walkthrough

```
asd project-create --name "Lumen Alignment"   --client "Fictional composite — Lumen

asd evidence-add --projectId lumen-alignment-9b3f5d2a   --source parent-12 --kind qu

asd evidence-add --projectId lumen-alignment-9b3f5d2a   --source teacher-07 --kind q

asd evidence-add --projectId lumen-alignment-9b3f5d2a   --source support-queue-q1 --

asd evidence-add --projectId lumen-alignment-9b3f5d2a   --source content-model-v2 --

asd method-run --methodId mental-model-alignment   --projectId lumen-alignment-9b3f5

asd artifact-write --projectId lumen-alignment-9b3f5d2a   --methodId mental-model-al
```

```
    {
      "id":"find-grades",
      "label":"Find the latest school picture",
      "fields":{"emotion":"2","touchpoint":"parent portal"},
      "evidence":[
        {"evidenceId":"38ff95bb5223d05c6eeafae8d15819e50c4cb27ba22809da1958be2bb5a46
        {"evidenceId":"4a24df6d58ec99e275dcc98f0dd65e0d1c64133eaa7f66d89ad1263f23d77
      ]
    },
    {
      "id":"resume-grading",
      "label":"Resume a grading sitting",
      "fields":{"emotion":"2","touchpoint":"gradebook"},
      "evidence":[
        {"evidenceId":"a24b51bac317fb3baa60f8d5e5cbcde0ab3f62c0487e51321e4191855c0ea
        {"evidenceId":"4a24df6d58ec99e275dcc98f0dd65e0d1c64133eaa7f66d89ad1263f23d77
      ]
    },
    {
      "id":"see-what-matters-next",
      "label":"See what is due next",
      "fields":{"emotion":"3","touchpoint":"student feed"},
      "evidence":[
        {"evidenceId":"c872dc90f8078c90d74c7ae71fbd27f545d81ca0092d284211c208b2c9010
        {"evidenceId":"4a24df6d58ec99e275dcc98f0dd65e0d1c64133eaa7f66d89ad1263f23d77
      ]
    }
  ]'

asd review --projectId lumen-alignment-9b3f5d2a   --artifactId 7bfbea7ad1f34940bd46a
    {
      "rowKind":"proposal",
      "severity":"medium",
      "rule":"source-diversity",
      "message":"Keep the support-queue telemetry visible alongside the paired quote
      "evidenceId":"c872dc90f8078c90d74c7ae71fbd27f545d81ca0092d284211c208b2c901068f
    }
  ]'

asd factcheck --projectId lumen-alignment-9b3f5d2a
```

Read that carefully and a few truths jump out. `method-run` gives me the contract; it does not write the artifact for me. `artifact-write` still needs nodes that satisfy the method's acceptance rules. The review step still needs evidence-cited rows, even on

pass. And the final check is still project-wide, because the platform cares about the state of the store, not merely the confidence of the last agent who touched it.

The ids in the shell block are illustrative content-addressed examples, included so the walkthrough stays mechanically faithful to the real CLI shape without pretending this chapter is a live workspace receipt. That distinction matters. A tutorial can be illustrative without becoming fake.

> **CAUTION:** Do not rename these steps into friendlier pseudo-tools for teaching. The fastest way to rot a technical book is to document commands that never existed.

Case Spine: Lumen EdTech and the Cost of Ontology Mismatch

Lumen EdTech looked, at first glance, like a classic UX clean-up. Support volume was ugly. Parent trust was low. Teachers complained about lost grading drafts. Students missed deadlines. The executive reflex was to reskin the interface. The alignment work showed something harsher: the service was faithfully implementing the wrong model of the world.

Parents did not think in courses. They thought in children. Teachers did not think in assignment columns. They thought in grading sittings that start, pause, and resume. Students did not think in posting chronology. They thought in urgency. The platform's content model violated all three. That is why tickets kept coming even when individual screens looked competent.

The mental-model work created three durable actor models and then forced a comparison against the system model. That yielded the three major gaps in the previous table plus smaller secondary ones. Opportunity scoring then made the sequencing obvious. The first structural move had to be parent-by-child aggregation because it had the heaviest combination of evidence weight, ticket load, and daily

user harm. The teacher draft-state gap followed close behind because it was eroding expert trust. The feed-order gap mattered enough to sit alongside it because students were paying the price in real missed work.

From there the design work became less mystical. The team created a unified Communication Hub rather than three disconnected portals. It introduced child aggregation across courses, explicit grading-session objects with visible save and conflict states, and a due-date-primary student feed. Those are not cosmetic fixes. They are service-architecture responses to named cognitive mismatches.

Measurement closed the loop. Lumen did not declare victory because the new architecture sounded cleaner. The pilot re-ran task-success tests, monitored support-ticket categories, and checked whether first-click paths were converging on the intended actor models. That kept the alignment story empirical. If the new child-level summary still sent parents hunting across class pages, the model-match metric would expose it. If the grading-session object still leaked drafts, teachers would tell us quickly and the ticket queue would confirm it.

That outer loop matters because mental-model work is never a one-shot artifact. Schools change reporting practice. Terms change. Product sprawl creeps back in. A durable alignment capability therefore needs periodic replay, not one triumphant workshop and a framed diagram.

> **CORE PRINCIPLE:** Treat a persistent mental-model gap as an engineering requirement. If the alignment evidence points to a new domain object, the right answer may be a new domain object.

The pilot result mattered because it proved the diagnosis had teeth. Support tickets fell. Parent task success rose sharply. Teachers trusted the gradebook again because saving meant what they thought it meant. Students could see the week ahead in a structure that matched their planning behavior. The service had not simply become prettier. It had become more legible to the people using it.

That is the larger lesson of this chapter. Mental models are not soft context. They are a way of discovering where a service is asking people to think like the provider rather than like themselves. ASD makes that discovery durable by giving it artifacts, evidence, review, and replay. Once I do that, opportunity mapping stops being a brainstorm and becomes a disciplined argument for which structural changes deserve to happen first.

That durability is the real win. The model, the divergence, the score, and the resulting change proposal all stay connected instead of dissolving into meeting memory.

That is how a cognitive diagnosis becomes engineering work, service work, and eventually measurable operating performance rather than a one-off insight artifact.

That traceability is what keeps the walkthrough honest enough to be copied into real practice.

Run the alignment exercise on a cadence, not once. Services drift toward provider logic as back-office constraints change, so the mental model that was accurate at launch quietly stops being accurate two quarters later. I revisit the grid whenever telemetry shows new drop-off patterns, whenever a frontline team invents a workaround, and whenever a stakeholder cannot explain a screen without referring to the org chart. Any of those three signals means the model and the service have started diverging again, and the cheaper moment to catch it is always the earlier one.

CHAPTER 06

Mixed Methods & Evidence-Backed Segmentation

CHAPTER 08

Ecosystem Value Mapping & Stakeholder Topologies

NEXT CHAPTER

Ecosystem Value Mapping & Stakeholder Topologies

Actors, exchanges and hidden dependencies

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01

CHAPTER 02

The AI-First Service Design Operating System

The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 08

Ecosystem Value Mapping & Stakeholder Topologies

Actors, exchanges and hidden dependencies

Reading time

15 min

Beyond the Stakeholder List

I distrust stakeholder lists. They look responsible. They feel thorough. They almost always hide the real structure of the service. A list says who exists. It does not say who can block you, who absorbs uncompensated work, who holds legal authority, or which actor is quietly acting as the systems integrator because nobody else bothered. In practice that missing structure is where the service either holds together or collapses.

ASD separates two jobs that many teams blur together. First I map *stakeholder topology*: who holds influence, who holds interest, and what kind of decision rights they actually possess. Then I map the *ecosystem value network*: what each actor gives, what each actor receives, which channel carries the exchange, and which

dependencies make the link fragile. The first artifact tells me how the room behaves. The second tells me how the service behaves. I need both before I design any orchestration.

This distinction matters because power and value do not align neatly. A regulator may have very high influence and almost no operational interest. A family caregiver may have almost no formal power while carrying half the coordination burden. A distributor may sit outside the brand book while controlling the economics of the field experience. A partner may happily consume data while refusing reciprocal visibility. When I flatten those differences into a generic actor map, I design fantasies.

The platform helps by forcing concrete outputs. `stakeholder-mapping` produces a `stakeholder-map`. `ecosystem-value-mapping` produces an `ecosystem-map`. Both are gated by the Critic. Both require evidence on their nodes. Both stay draft until the review path says otherwise. That sounds bureaucratic until you have lived through the alternative: a steering committee arguing about whether a partner is "important" because nobody bothered to tie the claim to quotes, telemetry, or documents.

> **CORE PRINCIPLE:** A stakeholder map tells me where political force sits. An ecosystem map tells me where value, work, and dependency sit. If I use one artifact to do both jobs, I lose both kinds of truth.

There is also a practical reason to keep the artifacts distinct. Stakeholder positions change faster than structural exchanges. A new COO can reorder influence overnight. A regulator memo can change tolerance without changing the underlying value network. By contrast, the member still carries the referral, the technician still needs the part, the store associate still absorbs the angry conversation. Separating topology from exchange lets me update governance assumptions without redrawing the whole service universe.

ASD's authored templates are useful here because they narrow the blank page. Two of the seven authored templates are the stakeholder map and the ecosystem map. That is a deliberate choice. These are not decorative diagrams for late-stage storytelling. They are load-bearing instruments. The stakeholder map template

demands named actors, roles, and evidence references. The ecosystem map template demands value flows, channels, and dependencies. The authored form is the human-readable aid; the runtime contract is what the system checks.

stakeholder-mapping-method.json

```
{
  "method": {
    "id": "stakeholder-mapping",
    "phase": "research",
    "outputKind": "stakeholder-map"
  },
  "steps": [
    {
      "role": "researcher",
      "instruction": "Identify stakeholders with explicit evidence of influence, int
    },
    {
      "role": "mapper",
      "instruction": "Place each stakeholder in a power-interest topology and name t
    },
    {
      "role": "critic",
      "instruction": "Reject any stakeholder placement with unresolved evidence or p
      "gate": true
    }
  ],
  "acceptance": [
    { "rule": "min-nodes", "min": 3 },
    { "rule": "min-evidence-per-node", "min": 1 },
    { "rule": "field-non-empty", "field": "strategy" }
  ]
}
```

Notice what is absent. There is no room for "we all kind of know legal matters." The map has to say why legal matters here, which decision it can stop, and what evidence supports that placement. Once I work that way, the conversation gets sharper. Fewer vibes. More consequences.

My take: If a client cannot name the actor who pays the hidden coordination tax, I know the service is under-mapped. That actor might be a patient, a scheduler, a store associate, or a distributor. Find them first. They usually reveal the real architecture faster than the org chart does.

Ecosystem Value Maps as Artifacts

An ecosystem map in ASD is a typed artifact, not a clever drawing. I care about that because a real ecosystem contains claims that can be tested. Does a GP send structured referral data or a PDF? Does a distributor receive telemetry early enough to act? Does a customer provide labor without compensation? Does a partner get a visible benefit in exchange for data sharing? Those are not illustration problems. They are artifact problems.

The platform template is plain on purpose. It asks for actors, value flows, dependencies, and orchestration implications. Nothing more. Nothing mystical. That modesty is powerful because it blocks the lazy habit of labeling an arrow "data" and moving on. In the artifact I have to say what kind of data, for whose benefit, through which channel, under which constraint. The result is a map that can survive review.

ecosystem-map-template.md

```
# Ecosystem Map — <service> (mapping view)

## Actors
| Actor | Type | Role in value network | Evidence refs |
|---|---|---|---|

## Value flows
| From → To | Value exchanged | Channel | Evidence refs |
|---|---|---|---|

## Dependencies & constraints
```

```
<regulatory, technical, contractual walls — evidence cited>

## Orchestration implications
<channel-role candidates feeding later design work>
```

The authored template sits upstream of the runtime registry. The runtime itself registers 38 artifact-kind contracts; the practice-facing vocabulary has 39 rows, which includes aliases and pending concepts. I keep that distinction straight because otherwise people start speaking as if every practice term already exists as a live runtime schema. It does not. The ecosystem map does. The stakeholder map does. That lets me work with confidence on these two surfaces.

When I run `ecosystem-value-mapping`, the engine returns a machine contract: source references, role-attributed steps, acceptance rules, and the declared output kind. The model is allowed to draft. The runtime is allowed to check. The Critic is allowed to veto. That separation matters because the hardest ecosystem claims are the most politically tempting ones to overstate. A team loves to write "shared visibility" or "outcome guarantee" long before it can name the measurement window or the accountable actor.

Aurora Vento gave me a clean example. The first pass of the ecosystem map described Aurora's offer to customers as "guaranteed operational performance." That sounded impressive and useless at the same time. The Critic vetoed it under a measurability rule: no formula, no window, no exclusions, no telemetry source. Exactly right. "Guaranteed operational performance" is not a value flow. It is a slogan pretending to be a contract.

critic-veto-row.json

```
{
  "rowKind": "defect",
  "severity": "critical",
  "rule": "measurability",
  "message": "The value Aurora exchanges is stated as guaranteed operational perform
  "fix": "Name the uptime formula, the measurement window, the exclusions, the telem
}
```

That one veto does more to explain the value of the artifact model than a hundred process diagrams. The platform did not invent a better strategy. It merely refused a false one. Once the team had to name the formula, the window, the exclusions, and the remedy, the conversation moved from aspiration to architecture. That is what I want from an ecosystem artifact. It should force the hard nouns into the room.

I also use the map to make channels legible. A value flow without a channel is a fantasy. If a patient carries a referral letter physically, I record manual hand-carry. If a hotel brief moves through a shift handoff, I record the brief. If a telemetry packet reaches a distributor dashboard every four hours, I record the sync pattern. Channel detail sounds mundane until a redesign depends on a signal that never actually arrives.

ARTIFACT QUESTION	WEAK ANSWER	ANSWER THAT SURVIVES REVIEW
What flows here?	Data	Structured referral summary plus consent token
Why would the receiver participate?	Because we need them to	Less re-keying, faster booking, visible status, reversible participation
What constrains the flow?	Compliance	Special-category health data rules, partner autonomy, retained local systems
How do we know the flow is real?	Workshop agreement	Evidence rows from interviews, documents, and telemetry linked to the node

Once the map is built this way, it becomes useful far beyond diagnosis. It feeds orchestration work, requirement shaping, governance design, and pilot scoping. It tells me where shared metrics are possible and where they are fiction. It tells me which actor must be able to exit without the whole design collapsing. It tells me where a portal-first idea will fail because the real pain is re-keying, not access.

Exchanges, Dependencies and Risk

The heart of ecosystem work is the exchange itself. Services do not move only money and information. They move trust, legal authority, scheduling power, care responsibility, physical labor, exception handling, and reputational risk. If I model only the easy exchanges, I misread the operating model.

Northgate Health made this visible immediately. The formal story was about digital referrals. The real story was that the member was still carrying context between actors because the seam had no persistent state. The GP printed a referral. The specialist owned the booking. Diagnostics held results in a separate portal. Pharmacy saw prescriptions without care intent. Home care saw discharge tasks without medication changes. Northgate saw claims after the fact. Meridia saw telehealth episodes without the whole pathway. That is not one broken channel. It is a fractured exchange economy.

When OpenCode reconstructed 47,800 referral events and 11,900 multi-actor care chains, the crucial number was not glamorous: only 54% of referral outcomes made it back to the referrer within 30 days. Another brutal finding sat beneath that: 23% of referral coordination still happened by phone chains that Northgate's owned systems could not see at all. The member was still doing unpaid integration work in a supposedly digitized pathway.

Aurora Vento had a different topology and the same lesson. Hardware margin was eroding. Customers wanted availability commitments. Aurora had 41,000 machines in the field, 340 technicians, operations in 46 countries, and an MTTR of 5.2 days. The service economy around those machines depended on distributors, telemetry, parts, procurement, and technicians who each saw only part of the system. Here the hidden exchange was more than data. It was coordinated readiness: the right part, the right skill, the right timing, and a shared accounting of what uptime actually meant.

Dependency risk appears when one actor receives a burden that nobody has acknowledged. In Northgate the burden was administrative and cognitive. Members repeated their history, carried documents, and chased status across portals. In Aurora the burden was operational. Distributors and technicians absorbed the mismatch between contract ambition and messy logistics. In both cases the service looked stable until you asked who was compensating for the seams.

> **CAUTION:** One-sided exchanges are warning lights. If one actor keeps giving data, labor, or patience without a visible reciprocal benefit, the ecosystem is already unstable even if the KPI dashboard still looks green.

This is why I use dependency risk registers alongside the ecosystem map. The map shows the shape. The register shows where the shape is brittle. I want to know which flows are legally constrained, which depend on manual workarounds, which rely on one individual, and which lack a graceful fallback. Those questions are especially important when AI enters the mix, because a model will happily recommend a flow that assumes universal API adoption or universal willingness to share data.

dependency-risk-register.csv

```
flow_id,from_actor,to_actor,risk_type,symptom,current_workaround,design_move
NX-04,GP,Specialist,manual-handoff,printed referral carried by member,member hand-ca
NX-09,Specialist,Referrer,missing-loop-closure,outcome returns late or not at all,GP
AV-07,Telemetry,Distributor,partial-visibility,machine alerts arrive too late,techni
AV-11,Parts chain,Field technician,dispatch-mismatch,wrong part arrives after visit,
```

Risk work also changes design posture. Teams often want a central platform because it feels tidy. The ecosystem map usually tells me whether that instinct is arrogant. Northgate's evidence did not support a portal-first rollout. Practices wanted fewer handoffs and less re-keying, not another login. That single insight produced the principle *Never re-key* and pushed the design toward an API-first model with structured fallback. The member benefit stayed central. Northgate did not get to pretend it was the natural center of care.

The same happened at Aurora. A centralized dispatch model sounded elegant from headquarters and threatening in distributor regions whose economics still depended on emergency parts. The stakeholder topology exposed the political risk. The ecosystem map exposed the operational dependency. Together they stopped the team from calling a power struggle an implementation detail.

RISK PATTERN	NORTHGATE HEALTH	AURORA VENTO	WHAT I DO WITH IT
Invisible manual work	Member carries referral context	Technician reconciles parts mismatch by phone	Make the hidden labor explicit and redesign the seam
Over-centered focal actor	Insurer assumes all partners want its portal	Manufacturer assumes distributors will accept margin loss	Run adversarial review from the partner perspective
Undefined value promise	Status without reciprocal control	"Guaranteed operational performance" without formula	Force measurable definitions before contracting
Constraint denial	Professional autonomy treated as inconvenience	Collective agreements treated as footnote	Model constraints as design material, not blockers

My take: Ecosystem maps are where designers stop pretending the service belongs to the sponsor. Most of the time the sponsor owns a budget, a brand, and some systems. The service itself belongs to a negotiated field of actors with exit rights, pride, fear, and workloads. Design starts getting real the moment you accept that.

Case: Northgate Health and Aurora Vento

Northgate Health and Aurora Vento sit in different worlds. One coordinates care across providers, pharmacies, diagnostics, and home-care actors under heavy data sensitivity. The other is trying to turn industrial equipment uptime into a credible service promise across telemetry, distributors, technicians, and parts chains. I pair them because together they show the full range of ecosystem work: one highly relational and consent-bound, one heavily operational and contract-bound.

At Northgate, the redesign started only after the map corrected the framing. The issue was never merely slow referrals. The issue was that the system closed cost

loops faster than care loops. Once I phrased it that way, decisions changed. The target stopped being a shiny portal. The target became a care-coordination layer that respected partner autonomy while reducing invisible labor. The future-state direction became the Northgate Gateway: a care-coordination service with four bounded capabilities — a unified member record assembled from consented summaries rather than full takeover, referral orchestration, a provider API hub using open standards, and a loop-closure engine that chased missing outcomes before the patient fell through the seam.

The ecosystem map made the guardrails explicit. Data sharing had to be consensual, reversible, and non-lock-in. GPs remained autonomous. Specialists kept their systems. The design promise was narrower and stronger: orchestrate the seam, do not colonize the ecosystem. That is why the numbers improved in the pilot target logic. Median referral admin cycle was attacked at the seam. Loop closure was attacked at the missing return path. Manual channel share was attacked through structured payloads and status heartbeats, not through institutional grandstanding.

Aurora's case had more obvious commercial drama. The board wanted to sell availability instead of parts. Customers wanted 98% machine availability. Leadership's first instinct was pricing. The ecosystem work showed the real problem was service delivery capability. Remote parts availability was 81%. MTTR was 5.2 days. Dispatch mismatch was a recurring culprit. The map traced value flows across Aurora, legacy ERP, telemetry, the parts chain, distributors, procurement, and customer operations. It showed exactly where the promise would fail if the contract got ahead of the operating system.

The decisive moment was the Critic veto on the vague outcome definition. Once forced to specify the uptime formula, measurement window, exclusions, data source, accounting level, and remedy, the team could no longer hide behind rhetoric. The design then moved toward what the ecosystem could actually support: telemetry rules for earlier detection, a coupled parts-and-skills dispatcher, clearer distributor roles, and contract language built on measurable events rather than marketing fog.

orchestration-handoff.json

```
{
  "projectId": "northgate-gateway-7c2d",
  "planId": "plan-ecosystem-to-canvas",
  "from": "mapper",
  "to": "ideator",
  "condition": "Ecosystem and stakeholder maps have passed Critic review; begin orch
  "artifactId": "artifact-ecosystem-map-01",
  "artifactKind": "ecosystem-map"
}
```

I include a typed handoff like this because ecosystem work is only valuable if it carries forward without distortion. The next phase should inherit validated actors, dependencies, and evidence-backed seams. It should not restart from a workshop summary. In ASD that handoff has to resolve in the same project and the `artifactKind` must match the stored artifact. If it does not, the record fails before it is written. Good. That is the kind of failure I want early.

The two cases also show why reciprocity matters. Northgate succeeded because every actor got something visible out of participation: less re-keying, better loop closure, clearer status, reversible consent, fewer blind phone chases. Aurora succeeded when the offer stopped pretending value flowed one way. Distributors needed a viable role. Technicians needed better readiness, not more dashboard poetry. Customers needed a measurable outcome with an accountable remedy.

By the end of both programmes, the ecosystem map had done its real job. It had reduced the surface area for self-deception. The sponsor could still make trade-offs. The human designer still made framing calls. But nobody got to claim that a portal equaled coordination, or that a guarantee existed because a brochure said it did. The artifacts carried the argument, and the evidence carried the artifacts.

> **My take:** The biggest maturity jump in service design happens when a team stops asking, "What should our service do?" and starts asking, "What exchanges must become easier, safer, more reciprocal, and less dependent on heroics?" Ecosystem maps pull you toward the second question, which is why they matter.

If you remember one thing from this chapter, make it this: ecosystem work is where you discover whether your service is actually designed for the world it must survive. Stakeholder topologies tell you whose resistance matters. Ecosystem maps tell you whose work matters. When both artifacts are evidence-backed and Critic-reviewed, you get something rare in service design — a strategic picture that is finally concrete enough to build against.

CHAPTER 07

Mental Models, Alignment & Opportunity Mapping

CHAPTER 09

Living Journey Maps: From Transcripts to Sentiment Curves

NEXT CHAPTER

Living Journey Maps: From Transcripts to Sentiment Curves

Journey maps that update themselves

Continue Reading →

ASD ' 26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 09

Living Journey Maps: From Transcripts to Sentiment Curves

Journey maps that update themselves

Reading time

16 min

The Journey Map Rots; Ours Compiles

The standard journey map dies in three predictable ways. First, the spine is wrong. Teams organize it around internal departments rather than customer intent. Second, the emotions are fictional. People draw dramatic valleys because the story needs drama, not because the evidence supports it. Third, the map has no maintenance path. Once the workshop ends, nobody can tell which rows are real, which came from one anecdote, and which were smuggled in by an opinionated manager.

ASD counters all three failures by treating the journey map as an artifact under method control. The output kind is `journey-map`. The producing method is usually `journey-from-transcripts` or, when mental structure matters more than chronology,

`mental-model-alignment`. The runtime cares about the boring details that make the artifact honest: minimum node count, evidence per node, non-empty emotion field, non-empty touchpoint field, and later a Critic verdict before anything becomes final.

That means I do not begin with diagramming software. I begin with a lens, a scope decision, and evidence rows. The lens says whose journey this is. The scope says where it starts and ends. The evidence rows say what actually happened, in whose words, through which channel, at what point in the chronology. From there the map is a projection, not a performance.

The practical shape is simple. A journey map artifact carries stages. Each stage is a node. Each node cites the evidence register. The rows can be rendered as tables or diagrams later, but the map's truth is in the data structure. This is why I can update it. If a new transcript or telemetry row lands, I do not need to restage a workshop. I need to revise the artifact and rerun the gates.

> **CORE PRINCIPLE:** A journey map is an index over the evidence register. If a stage, emotion score, or pain point cannot cite the register, it is commentary, not map data.

The authored journey map template makes that discipline tangible. It asks for stages, channel and touchpoint detail, quoted emotion, pain or breakdown, and evidence references. It also keeps the emotion line as data. Unless I have a proper number ledger, the emotional line is a markdown table, not a seductive chart. That restraint matters. Charts are persuasive. Tables are accountable.

journey-map-template.md

```
# Customer Journey Map — <segment> (<current|future>)

## Stages
| Stage | Customer actions (evidence) | Channels & touchpoints | Emotion (quoted) |
|---|---|---|---|---|---|

## Emotion line (derived)
| Stage | Emotion score (1-5) | Supporting quote | Evidence ref |
```

```
|---|---|---|---|

## Moments of truth
<evidence-backed list>
```

The machine rules do not make the work mechanical. They simply shift my judgment to the right place. I decide the scope. I decide whether adjacent quotes belong to one stage or two. I decide which summary label respects the evidence best. The engine decides whether I have cited enough to make that judgment auditable. That division of labor is healthy.

It also keeps current and future states from contaminating each other. The most common journey-map fraud is a blended current-future poster where desired features are drawn into the current experience. ASD blocks that habit because current and future are separate artifacts. If I want to design a better journey, I can. I just cannot pretend the better journey already happened.

> **My take:** The emotional payoff of a journey map comes from recognition, not decoration. When a frontline team says, "Yes, that is exactly what people go through," I know the artifact is earning trust. When a map looks polished before it looks painfully true, I assume it is lying.

From Verbatim Evidence to Journey Stages

The step that matters most is stage construction. A good stage is about customer intent in time. It is not "contact centre triage" or "claims processing" because those are inside-out labels. It is "send and wait," "rebook," "return," "check in," or "ask for help." If I name stages from the organization's point of view, I have already started to misdiagnose the service.

My pipeline is narrow by design. I ingest transcripts, tickets, documents, and telemetry into the evidence register under the only three allowed kinds: `quote`, `telemetry`, and `document`. I then tag rows with provisional phase candidates, channel, touchpoint, and whether the row describes an action, a feeling, a breakdown, a workaround, or a recovery. The goal is not deep ontological elegance. The goal is enough structure to sort the chronology without flattening the voices.

Albatross Airways is a useful case because the corpus was messy enough to expose weak method. Over eight weeks the programme assembled 140 passenger interviews, 9,000 disruption tickets, 1.1 million app and site sessions, 45 staff debriefs, and 2,300 social posts. If I had tried to journey-map that corpus by manual stickies, I would have drowned in anecdotes. Instead the method treated the corpus as a searchable evidence field. Stage candidates had to emerge repeatedly across sources. Rows with insufficient support stayed parked or marked open.

The journey-from-transcripts method is explicit about the contract. It produces a `journey-map`. It assigns steps to roles. It requires acceptance rules. It does not allow a mapper to smuggle in a stage because it sounds narratively pleasing.

journey-from-transcripts.json

```
{
  "method": {
    "id": "journey-from-transcripts",
    "phase": "synthesis",
    "outputKind": "journey-map"
  },
  "steps": [
    {
      "role": "researcher",
      "instruction": "Extract distinct customer actions, touchpoints, and breakdowns
    },
    {
      "role": "mapper",
      "instruction": "Order the actions into customer-intent stages and attach evide
    },
    {
      "role": "critic",
      "instruction": "Reject any stage lacking supporting evidence or any emotion un
```

```
      "gate": true
    }
  ],
  "acceptance": [
    { "rule": "min-nodes", "min": 3 },
    { "rule": "min-evidence-per-node", "min": 1 },
    { "rule": "field-non-empty", "field": "emotion" },
    { "rule": "field-non-empty", "field": "touchpoint" }
  ]
}
```

What that contract really does is force me to defend the stage boundary. Suppose I have five passenger quotes about checking the app, then the departures board, then asking staff because none of the channels agree. I can collapse that into a single stage if the customer goal is still the same: understand what is happening. I should not split it into three stages unless the intent changes. The map becomes better when stages reflect the customer's errand rather than my urge for more granularity.

Tagging still matters. A decent tagging schema lets me check whether a stage is over-weighted toward one source or one channel. If every quote in a stage comes from social posts and none from interviews or telemetry, I read it cautiously. If a stage has quotes but no operational trace, I know I may be seeing a perception seam rather than a measured queue. Both are useful. They are just different.

tagging-schema.json

```
{
  "evidence_item": {
    "phase": ["pre_flight", "at_airport", "new_arrangements", "rebook", "recovery"],
    "channel": ["app", "sms", "email", "phone", "desk", "gate_board", "social", "web
    "type": ["verbatim_quote", "observed_behavior", "operational_fact", "workaround"
    "emotion_1_5": "integer 1-5 or null",
    "pain_code": ["INFO", "REB", "CARE", "COMP", "QUEUE", "OTHER"],
    "source_lines": "required for quote verification"
  }
}
```

The Albatross map eventually stabilized around five stages because that is where the evidence converged: Pre-flight, At-airport, New arrangements, Rebook, and Recovery. Not six. Not twelve. Five. That structure was strong precisely because it survived workshop challenge. During validation workshops, participants turned objections into issue rows against the map. Twenty-nine fixes were merged and twelve were parked pending stronger evidence. That is what I mean by a living map. It is not endlessly mutable. It is formally repairable.

STAGE-BUILDING DECISION	BAD HABIT	ASD DISCIPLINE
Name the stage	Use internal department labels	Use customer goal language
Choose boundaries	Split on every touchpoint	Split when customer intent changes
Handle gaps	Invent a plausible stage	Mark open gaps or strengthen the evidence
Move to future state	Blend proposed steps into current map	Create a separate future artifact

That last row is where most teams cheat. They cannot resist improving the story while mapping the current state. I get the temptation. But if I do that, I lose the diagnostic. The journey map's job is to tell the truth about what is happening now. The blueprint can carry the redesign later.

Sentiment Curves from Evidence

Emotion is the most abused row in service design. Teams either avoid it because it feels soft, or they overperform it because it makes the poster persuasive. ASD takes a more severe view. Emotion belongs in the map, but only as an evidence-backed field. If nobody said it, showed it, or wrote it in a way that clearly carries valence, the map should not pretend to know.

This is why the old freeform `emotion.yaml` block had to die. A YAML file is not a mood board. If I label something YAML, it should parse. More importantly, it should encode operational rules I can actually run. The sentiment configuration below is valid YAML and expresses the load-bearing constraints: explicit-evidence scoring, thresholds, lexicon path, minimum quotes per stage, source diversity, and calibration guards.

emotion.yaml

```
version: 1
scale:
  min: 1
  max: 5
labels:
  1: distressed
  2: negative
  3: neutral
  4: positive
  5: delighted
scoring:
  explicit_evidence_only: true
  null_when_absent: true
  min_quotes_per_stage: 2
  min_sources_per_stage: 2
  calibration_sample_size: 50
  stddev_floor_per_phase: 0.8
thresholds:
  negative:
    max_score: 2
  neutral:
    min_score: 3
    max_score: 3
  positive:
    min_score: 4
lexicon:
  path: research/emotion-lexicon.yml
  use_anchor_quotes: true
anchors:
  distressed:
    description: crisis language, panic, tears, giving up
    example_quote: "The children slept on our coats."
    example_source: "INT-041"
  negative:
    description: frustration, distrust, workaround forming
```

```
      example_quote: "Nobody could tell me anything, so I stopped asking."
      example_source: "INT-074"
    neutral:
      description: transactional completion with no explicit valence
      example_quote: "I filled in the form and waited."
      example_source: "INT-118"
    positive:
      description: relief, gratitude, acknowledged effort
      example_quote: "The agent actually fixed it while I watched."
      example_source: "INT-096"
    delighted:
      description: advocacy language, unprompted praise, retelling
      example_quote: "They rebooked me before I reached the terminal."
      example_source: "SOC-402"
```

The rule that matters most is `explicit_evidence_only: true`. I do not allow a model to infer that someone must have felt abandoned because the queue was long. Maybe they were annoyed. Maybe they were resigned. Maybe they were furious. Maybe they were emotionally flat and pragmatically focused. If the quote does not carry valence, the field should be `null`, and the phase should aggregate only rows that genuinely express emotion.

This is how I avoid the sentiment smoothie. Models love to smooth ambiguity into plausible middles. That is lethal here. The real value of the emotional line is contrast. At Albatross the phase-level means came out as 2.2 for Pre-flight, 1.4 for At-airport, 2.2 for New arrangements, 1.6 for Rebook, and 1.8 for Recovery. That shape challenged executive folklore. The cancellation itself was not the deepest wound. Silence and contradiction at the airport were.

I also insist on calibration. A human-coded sample still matters. If an agent's scoring drifts wildly from the calibration set, I do not trust the emotional curve, however elegant the chart looks. The YAML's `calibration_sample_size` and `stddev_floor_per_phase` encode that discipline. The point is not mathematical fetishism. The point is to protect the map from dramatic flattening and theatrical exaggeration alike.

> **CAUTION:** Never average nulls into comfort. A stage with weak emotional evidence is a research gap, not proof that the experience was emotionally mild.

The journey map template keeps the sentiment line attached to supporting quotes for exactly this reason. If a stakeholder disputes the valley, I can point to the evidence rows, not my intuition. If they still disagree, that is useful. It means we should challenge the tagging or gather more evidence. What I do not do is redraw the curve to reduce discomfort in the room.

> **My take:** Emotional curves become useful the moment they stop flattering the narrative. When a phase score embarrasses the sponsor or contradicts the team's pet theory, I start paying attention. That is usually where the map is finally telling the truth.

Emotion, Pain and Gain Registers

Emotion alone is too blunt. A strong journey map carries emotion, pain, and gain registers side by side. Emotion tells me how the service feels. Pain codes tell me what kind of breakdown is recurring. Gain codes tell me what actually restores confidence or saves effort. This layered view is what turns a map from empathy theater into a design instrument.

Pain codes are especially useful when the evidence corpus is large. At Albatross, the codes made the narrative operational. `INFO` captured notification lag and contradictory status. `REB` captured self-serve rebooking failure. `CARE` surfaced care bottlenecks such as hotel or meal support. `COMP` captured compensation friction. Because those codes sat on tagged evidence rows, the journey map could say more than "people were upset." It could show which class of failure dominated each stage.

The gain side matters too. I have seen too many maps dwell on pain until they become misery collages. The service still has bright spots. Relief can be operationally specific. A passenger gets a ranked rebooking offer before reaching the desk. A patient sees an acknowledgement instead of entering a phone chain. A retail customer watches the refund clock update instead of wondering whether money disappeared. Gains should be evidence-backed as rigorously as pains. Otherwise teams end up shipping delight fiction.

Because the artifact is typed, I can carry this layered reading into later work. Pain codes inform fail points in the blueprint. Gain codes inform moments worth protecting. Emotion tells me where a design move must be especially careful. None of these registers replace judgment. They simply keep my judgment connected to the evidence field.

STAGE	EMOTION MEAN	DOMINANT PAIN CODE	WHAT THE CODE REALLY MEANS
Pre-flight	2.2	INFO	Notification lag after the operations decision
At-airport	1.4	INFO	Contradictory signals across app, board, and staff
New arrangements	2.2	CARE	Care support depends on fragile desks and closing times
Rebook	1.6	REB	Self-serve fails and passengers default to the phone queue
Recovery	1.8	COMP	Claim friction, abandonment, and long resolution tail

The review path becomes important here. A Critic veto is about more than missing evidence. It can also challenge a misleading register assignment. If I tag a stage primarily as `REB` but the supporting evidence shows passengers are actually most distressed by contradictory information before they even reach rebooking, the map needs correction. That is why a good reviewer attacks classification as well as structure.

The runtime lifecycle helps me here. A failed write persists as draft with its acceptance failures attached. I can rewrite the same artifact in place via `artifactId`, strengthen the evidence, or remove an overreaching node. That means the map can improve without losing continuity. I keep one artifact history instead of a trail of disconnected poster versions.

journey-map-repo-layout.txt

```
maps/disruption-recovery/v1.0/
├── map.md
├── rows.csv
├── phases.yaml
├── emotion.yaml
├── traceability.csv
└── CHANGELOG.md

CI checks:
- every rows.csv row has the required evidence refs
- every quoted line grep-matches the corpus store verbatim
- emotion scores follow emotion.yaml rules
- no passenger-identifiable data appears in the artifact
```

That versioned layout matters more than it first appears. It lets workshops generate issues against rows instead of vague comments against a poster. It lets a map owner refresh the artifact quarterly instead of commissioning a re-map. It lets downstream blueprint work cite journey row IDs directly. In short, it turns the journey map into infrastructure.

Case: Albatross Airways

Albatross Airways is the cleanest argument I know for evidence-gated journey mapping. The airline already had a disruption poster from 2023. It had been built in two days, validated by nobody, and updated never. Meanwhile disruption days kept producing industrial-scale failure demand: 380 calls per 1,000 disrupted passengers

into a 340-seat contact centre, post-disruption NPS at −34, and a compensation process everyone loved to blame because it was visible, measurable, and late in the journey.

The new programme ran for eight weeks. It assembled 140 passenger interviews, 9,000 disruption tickets, 1.1 million sessions across three disruption events, 45 staff debriefs, and 2,300 social posts. The aim was modest and ruthless: build a current-state map with evidence-per-row traceability, reach at least 90% agreement in an extraction audit sample, and hand a defensible journey artifact into blueprint work without losing the passenger view.

The biggest discovery came from the emotional line. Executives assumed the cancellation itself or the compensation tail would be the nadir. The map proved otherwise. The deepest trough was At-airport at 1.4, where app, gate board, and staff contradicted each other and passengers queued for roughly 90 minutes in an information blackout. Rebook was close behind at 1.6 because the self-serve engine failed on 66% of attempts and 41% of users defaulted to the phone queue. Recovery was miserable, but it was not the first wound.

That distinction changed the design conversation immediately. Once the map showed the true order of harm, the team could stop fetishizing the compensation form as the singular problem. The critical moves shifted toward earlier, more structural seams: a faster notification layer, synchronized disruption communications, ranked rebooking offers with soft holds, and a journey-to-blueprint traceability gate so every top-tier pain had a corresponding operational response.

The map's code form also changed the workshop mechanics. Three validation workshops involved thirty-four participants across gate operations, OCC, ground handling, contact centre, claims, digital, legal, and passenger advisers. Participants did not merely comment on the map. They opened row-level issues against it. Twenty-nine were merged. Twelve were parked because the evidence did not yet clear the bar. That is a healthier meeting. It rewards specificity.

By the handoff stage, the map had become the passenger-facing truth for the programme. The blueprint work had to answer journey row IDs rather than reinvent

pain categories. That traceability check caught blueprint steps that solved vendor wish-list items but answered no journey pain, and it exposed at least one Tier-1 moment with no blueprint coverage. Again: the map did not design the service. It stopped the team from designing the wrong one.

The downstream impact is why this matters. In the illustrative pilot the redesigned recovery service improved self-serve rebooking from 34% to 78%, cut disruption-day calls by 62%, reduced median EU261 resolution from 63 days to 19 days, and lifted disruption NPS from −34 to −9. I would never attribute those gains to the journey map alone. That would be nonsense. But I will say the map changed the direction of effort. It revealed the seam that actually mattered.

> **My take:** The line I remember from Albatross is simple: passengers forgive the storm; they do not forgive the silence. That sentence was not poetic flair. It was a compressed reading of the evidence field, and it only became trustworthy because the map earned it.

If you carry this method into your own work, keep the bargain clear. Let transcripts and telemetry do the grounding. Let the journey spine stay outside-in. Let the emotional line earn its shape. Let the Critic be annoying. And keep the map alive in the same repository as its repair log and traceability table. A living journey map is not one that changes constantly. It is one that can be changed honestly, without pretending the evidence said more than it did.

CHAPTER 08

Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 10

The Master Blueprint: Five-Lane Architecture & Line of

Visibility

NEXT CHAPTER

The Master Blueprint: Five-Lane Architecture & Line of Visibility

The operational X-ray of a service

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 10

The Master Blueprint: Five-Lane Architecture & Line of Visibility

The operational X-ray of a service

Reading time

15 min

Why Blueprints Beat Journeys for Operations

When a service is failing, the journey map gives me the symptoms. The blueprint gives me the mechanics. That is why I reach for blueprinting when the question becomes operational: where is the wait formed, which actor actually owns the handoff, which backstage step creates the frontstage silence, which support system is making the team look incompetent, and which fail point is expensive enough to deserve redesign first.

The blueprint is powerful because it preserves chronology while revealing production. I still move left to right through the customer's errand, but now I can look vertically and see what stands behind each visible moment. A missed appointment is no longer

just a customer frustration. It becomes a chain: a notification design choice, a scheduling rule, a support-system blind spot, a contractor incentive, a parts feed that arrived late, or a manual invoice cycle that trapped the job in limbo. That vertical logic is why blueprints belong in serious service work.

ASD turns that logic into artifact discipline. The service blueprint template declares a `service-blueprint` artifact. The renderer draws the diagram deterministically from typed data. The runtime checks acceptance rules before the artifact is treated as trustworthy. The Critic then reviews it. That matters because blueprints are vulnerable to a particular kind of nonsense: they look authoritative even when half their boxes are guesses.

I have seen endless blueprints that were really compressed org charts. They named departments, not actions. They merged current and future states. They buried waits in vague backstage clouds. They treated support systems as one generic rectangle called "platform." They often looked polished. They were useless. The ASD grammar is a corrective. It keeps the blueprint close to observable service production.

> **CORE PRINCIPLE:** A blueprint is not a broader journey map. It is the operational underside of the journey, column by column, with each visible moment tied to the work that enables or obstructs it.

The authored flagship skill for `service-blueprinting` makes this plain. It exists as the only hand-authored flagship among the 81 skill sources because blueprinting needs extra judgment guidance beyond the generated manifest. The compiler ships 80 skills, not 81, because the authored flagship replaces its generated twin. That detail matters here because it tells you something about the method: the machine contract is stable, but the craft layer matters enough to author by hand.

The flagship skill also keeps the promise modest. It says blueprinting is for diagnosing a current-state operation, specifying a future-state service, or aligning a cross-functional team on who does what behind the line of visibility. It explicitly says not to use a blueprint for a single-touchpoint fix or a purely digital flow with no meaningful

organizational backstage. I agree. Blueprinting is heavy equipment. Use it where seams, handoffs, waits, and hidden work actually exist.

service-blueprinting-skill.txt

```
Canonical procedure:
1. Lay out physical evidence, customer actions, frontstage and backstage work, and s
2. Annotate the line of visibility and mark waits, failures, handoffs, and moments o
3. Reject placeholder lanes and invented nodes.
4. Keep current state and future state as separate artifacts.
5. Send the artifact through Critic review before any downstream handoff.
```

That procedure sounds stricter than many design teams are used to. Good. The cost of loose blueprinting is real. Teams build channels to cover process defects instead of fixing the defects. They fund chatbots to answer questions that the operation itself should have made unnecessary. They celebrate better scripts while leaving the root cause below the line of visibility untouched. The blueprint exists to prevent that misallocation.

> **My take:** If a service is generating high volumes of status-chasing, the answer is almost never "another contact surface." The answer is usually a blueprint that makes the silence legible and expensive.

The Five Canonical Lanes

The ASD blueprint grammar begins with five lane names and I use them exactly: **Physical Evidence**, **Customer Actions**, **Frontstage Interactions**, **Backstage Operations**, and **Support Processes**. Those names are not loose suggestions. They are the canonical structure of the method. If I rename them casually, merge them, or dump extra concerns into a sixth unnamed bucket, I weaken the diagnostic.

The sequence matters. Physical Evidence sits above the customer because customers interpret services through traces: screens, forms, letters, uniforms, spaces, trackers, receipts, QR codes, and notification banners. Customer Actions form the spine because chronology belongs to the person living the service. Frontstage Interactions show what the customer sees the organization doing. Backstage Operations show the invisible human work that makes the frontstage possible. Support Processes carry systems, policy, third parties, and other enabling structures.

CANONICAL LANE	WHAT BELONGS THERE	COMMON MISTAKE
Physical Evidence	Things the customer can sense: letters, dashboards, messages, desks, passes, receipts	Leaving the lane empty because the team is thinking only in software actions
Customer Actions	Observable customer behavior in customer language	Writing internal process labels instead of actions
Frontstage Interactions	Visible staff or service behavior above the line of visibility	Confusing customer-facing scripts with the systems that enable them
Backstage Operations	Invisible human work, triage, scheduling, exception handling, coordination	Using the lane as a dumping ground for anything vague
Support Processes	Systems, policies, third parties, integrations, infrastructure	Drawing a single giant system box with no meaningful distinction

Physical Evidence is routinely underestimated. Teams treat it as decorative output when it is often where the service makes or breaks trust. A job reference in every message, a live tracker page, a clear QR pickup pass, a slot offer, an outcome summary, or a compensation receipt can materially reduce failure demand because it changes what the customer has to ask about. Missing evidence is itself diagnostic. If nothing visible tells the customer that work has progressed, the frontstage is forced into reassurance theater.

Customer Actions must stay observable. "Feels anxious" is not an action. "Checks the app again" is. "Calls to chase status" is. "Prints the referral and carries it to the

specialist" is. When I let inference creep into the action lane, I flatten the chronology and weaken later measurement.

Frontstage Interactions are the behaviors the customer can hold the organization accountable for directly. A call-centre agent re-explains a wait. A store associate scans the return and initiates a refund. A coordinator offers slot options. A gate agent makes an announcement. These steps matter because they absorb the consequences of backstage and support design. They are visible labor, more than interface moments.

Backstage Operations are where the real diagnostic work often happens. Triage queues, scheduler trade-offs, supervisor assignments, parts ordering, manual exception handling, invoice chasing, recovery decisions, shift briefs. Customers do not see these steps, but they feel the latency, inconsistency, and rework they generate. The flagship skill is right to warn that backstage is not a dumping ground. Every backstage node should enable a frontstage moment or consume a support process. Orphaned backstage boxes usually mean invented process or missing research.

Support Processes are the base layer: legacy databases, rules engines, policy constraints, third-party portals, notification routers, audit logs, payment rails, and batch integrations. The mistake here is either oversimplification or fetishization. I do not need every database field. I do need the support structure that materially shapes the experience. The lane exists to expose where the service is leaning on systems and partners that are invisible to customers but decisive in outcomes.

service-blueprint-template.md

```
## Lanes (five, all mandatory)
| Lane | Nodes (evidence-cited) |
|---|---|
| Physical evidence | what the customer senses |
| Customer actions | observable actions |
| Frontstage interactions | visible actor behaviors |
| Backstage operations | invisible human work |
| Support processes | systems, policy, third parties |
```

Notice the method's conservatism. Five lanes, all mandatory. That is a feature. It forces the team to confront missing work instead of hiding it in a merged bucket called "operations" or "system." The blueprint becomes readable because each row has one job.

Line of Visibility, Fail Points, and Wait Semantics

The line of visibility is the most famous boundary in blueprinting and still the one teams misuse most often. It is not a decorative separator. It is a design decision about accountability. Everything above the line can be perceived directly by the customer. Everything below it is hidden production. Where I draw the line shapes how I interpret responsibility, which means it changes what redesign moves make sense.

If a contractor reschedules a housing-repair visit below the line of visibility, the resident blames the frontstage agent who answers the chase call. If the operation emits no visible status event, the resident experiences silence, not honest process. In other words, the line is not about metaphysics. It is about what the customer can reasonably perceive and therefore judge.

I still use the line of interaction above Frontstage Interactions and the line of internal interaction above Support Processes as craft aids, but the line of visibility is the essential load-bearing cut. It tells me where backstage failure becomes frontstage consequence. It also tells me where a new piece of physical evidence could relieve pressure. Silence below the line often implies a missing visible trace above it.

Fail points and wait points give the blueprint teeth. A fail point is not any unpleasant thing. It is a structural point where the service predictably breaks down or creates avoidable rework. A wait point is a delay that matters enough to the experience or cost structure to deserve explicit marking. When I annotate F and W on a blueprint, I expect them to be ledger-bound. Frequency, duration, and cost should trace back to evidence, not hunches.

STRUCTURAL MARK	WHAT I MEAN BY IT	WHAT EVIDENCE SHOULD BACK IT
Wait	A delay with experiential or operational consequence	Observed timing, telemetry, or documented SLA miss
Failure point (F)	A recurring breakdown that creates extra demand or rework	Quotes plus volume, frequency, or exception evidence
Handoff	A transfer of work, accountability, or information	Observed workflow or documented process row
Moment of truth	A visible moment that disproportionately shapes trust	Strong customer evidence, often repeated across sources

The line of visibility also helps me distinguish symptom from cause. Status-chasing happens in the customer lane. The root cause often sits below visibility: triage queue lag, overnight batch sync, invisible contractor calendars, parts ordered only after the first visit, or a manual finance cycle keeping completion uncertain. Without the blueprint, a team will treat the symptom as a communication problem. With the blueprint, I can show where communication is only the visible face of a deeper production issue.

> **CAUTION:** Do not place an experience problem below the line just because the team is uncomfortable owning it. If the customer experiences it directly, the symptom belongs above the line even if the cause sits below it.

Wait semantics deserve a special note. Not every delay is a failure. Some waits are expected and tolerable if the service explains them well and manages uncertainty honestly. The problem is unbounded, silent, or contradictory waiting. That is why a visible tracker, acknowledgement, or predictable status event can radically change the experience without changing the underlying throughput yet. The blueprint lets me show both truths at once: the service still waits, but the wait is no longer invisible or confusing.

My take: Whenever a team wants to add a chatbot to a failing service, I ask one question first: what exactly will it be able to see below the line of visibility? If the answer is "the same broken status data the human sees," the chatbot is just a prettier apology.

Blueprint Acceptance Rules

Because blueprints look convincing even when they are wrong, ASD treats acceptance rules as non-negotiable. The authored template and flagship skill define the craft. The runtime checks the structure. The current machine rules are intentionally small: minimum nodes, minimum evidence per node, non-empty lane, and a deny-list that blocks placeholders such as `unknown`, `tbd`, and `n/a`.

The small rule set is enough. It stops the worst failure modes without pretending the engine can judge operational elegance. The engine does not know whether a handoff is politically smart. It does know whether I left the support lane as a shell. It does know whether I cited nothing for a fail point. It does know whether I tried to sneak a future-state promise into a current-state artifact by leaving a placeholder for later.

service-blueprint-acceptance.json

```
{
  "acceptance": [
    { "rule": "min-nodes", "min": 4 },
    { "rule": "min-evidence-per-node", "min": 1 },
    { "rule": "field-non-empty", "field": "lane" },
    { "rule": "forbid-field-values", "field": "lane", "deny": ["unknown", "tbd", "n/
  ],
  "reviewGate": "critic"
}
```

There is a subtle but important lifecycle implication here. A failing write is still persisted as draft with its failure report attached. That makes remediation specific. I

do not lose the artifact. I repair it in place. An accepted write is still only draft until Critic review passes. That is healthy pressure. It prevents teams from treating structurally valid garbage as final just because the JSON compiled.

The Critic's job is narrower and more useful than many people assume. It does not redesign the blueprint. It attacks unsupported claims, weak attribution, missing evidence, and poor structural discipline. If a fail point lacks a meaningful recovery implication, the Critic can say so. If a number is ledger-wrong, it can say so. If a customer problem is being blamed on the customer despite evidence showing a support-system root cause, it can say so. That adversarial pressure is where the blueprint gets honest.

Current and future states must also stay separate. This is one of the flagship skill's strongest craft rules and I agree with it completely. A mixed blueprint is seductive because it lets teams narrate the desired future without fully confronting the current state. The result is a hybrid artifact that neither diagnoses nor specifies cleanly. Separate artifacts are more work and far better.

blueprint-review-checklist.txt

```
- every lane named and non-empty
- every node cites at least one resolvable evidence row
- waits carry duration where known
- fail points carry frequency or impact where known
- current-state and future-state artifacts are distinct
- no placeholder language in lane fields or node labels
- every backstage node enables a visible or support step
```

These checks are boring. That is why they work. The point of method infrastructure is to catch boring failures before they become expensive programme confusion.

Case: Fernwood Council

Fernwood Council is the textbook argument for blueprinting as operational diagnosis. The service was reactive housing repairs across roughly 19,000 council homes. It handled about 28,000 repair requests a year, which turned into 30,200 jobs once follow-on works were included. The headline pain was familiar and misleading: residents kept calling to chase status. A weaker team would have started with channel expansion. The blueprint showed why that would have been a waste.

Forty-one percent of jobs attracted at least one status-chase contact. That meant about 12,400 jobs producing roughly 37,200 status-chase contacts a year, around 3,100 a month, at a fully loaded cost of €6.40 per contact. First-time-fix sat at 62%. Nineteen percent of first visits ended in no-access or no-show. The service was burning money reassuring residents that something might eventually happen.

The current-state blueprint gave the service its proper grammar. In Physical Evidence, the traces were bleak: a web form with no reference number, an acknowledgement letter on day three, an appointment letter days before the visit, and paper job sheets. In Customer Actions, residents noticed faults, reported them, repeated details, waited, chased status, hosted missed or unprepared visits, and sometimes re-reported linked follow-on work because the job lineage broke. In Frontstage Interactions, agents re-keyed data and operatives explained follow-on delays they did not fully control. In Backstage Operations, triage queues, nominal-capacity scheduling, manual supervisor assignment, and post-visit parts ordering created delay and rework. In Support Processes, three systems coexisted without a shared job ID: RepairsDB, ContractorLink, and AssetPlan.

That last point mattered more than any script improvement. The blueprint traced the resident's visible frustration down through the line of visibility into a support-layer truth: nine percent of jobs could not be matched cleanly across systems, and the overnight sync pattern meant frontstage staff were often blind to the freshest contractor state. A chatbot would have answered from the same blind systems. The blueprint made that obvious.

The fail-point logic got sharper under review. Codex challenged the first draft's attribution of no-access to "resident not in." The logs showed that 62% of no-access visits had been preceded by a contractor-initiated reschedule within the prior 48

hours. The root cause moved from customer blame to support and scheduling design. That single correction is why blueprints need adversarial review. Teams are quick to blame the visible actor at the top of the column.

fernwood-current-state-lane-table.csv

```
lane,node,evidence,mark,overlay
Physical Evidence,Web form with no reference number,INT-04;LOG-03,F-01,duplicate adm
Customer Actions,Chases status,LOG-05,F-05,41 percent of jobs; about €238k yearly de
Frontstage Interactions,Agent re-keys into RepairsDB,INT-27;RDE-01,F-01,7.2 minute a
Backstage Operations,Triage queue,LOG-04,F-02;W-02,1.8 days; 14 percent re-triaged
Backstage Operations,Parts ordered after first visit,LOG-09;RDE-05,W-04,4.6 days; fi
Support Processes,No shared job ID across three systems,LOG-01,root cause,9 percent
```

Once the blueprint made the real structure visible, the future-state direction became more disciplined. Fernwood did not replace RepairsDB within the programme horizon. The redesign wrapped it. A unified job-record API sat across the legacy systems. A status event bus emitted six core state changes to residents through SMS and a live tracker page. Operatives received digital job cards. Completion summaries replaced opaque silence. In other words, the team moved the line of visibility deliberately by creating better physical evidence and tighter support coherence.

The illustrative pilot outcome showed why that mattered. Status-chase contacts fell materially, first-time-fix improved, and routine waits shortened because the operation itself became more coordinated and more legible. Communication improved because the process improved and because its traces improved. That sequence matters. It is the difference between cosmetic messaging and actual service redesign.

> **My take:** Fernwood's breakthrough was brutally simple: the council did not need a better story about repairs. It needed a repair system capable of emitting trustworthy status. The blueprint made the silence visible, and once the silence had a structure, the redesign finally had somewhere real to act.

That is the master blueprint in practice. Five canonical lanes. A real line of visibility. Structural marks that mean something. Acceptance rules that block placeholders. And one case after another proving the same lesson: if you want to change how a service feels, you eventually have to change how the service is made.

CHAPTER 09

Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 11

Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

NEXT CHAPTER

Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

Where agents act, and how we gate them

Continue Reading →

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 11

Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

Where agents act, and how we gate them

Reading time

14 min

The Sixth Lane: Agent Decisions

I do not add a sixth lane because agents are exciting. I add it because they make a specific class of work visible: interpretation, recommendation, tool selection, escalation, and refusal. None of that fits cleanly inside Support Processes. A database does not decide whether it has enough confidence to continue. A notification router does not weigh ambiguity and ask for help. An agent does.

The five canonical lanes remain intact: Physical Evidence, Customer Actions, Frontstage Interactions, Backstage Operations, and Support Processes. The Agent

Decision Lane extends that structure for AI-native blueprints. It does not replace the old grammar. It sits alongside it so I can show where probabilistic reasoning touches deterministic infrastructure and human judgment.

That distinction is crucial because many service teams still describe AI as if it were a smarter rules engine. It is not. Rules engines execute declared logic. Agents interpret, choose, and sometimes drift. If I hide that inside Support Processes, I erase the exact moments where the service can go wrong in new ways: overconfident automation, silent fallback, escalation with missing context, looping between tools, or a confident answer unsupported by evidence.

The sixth lane also lets me distinguish between a system action and an agent decision in the same column. A scheduler database can return capacity. An agent can decide whether the returned capacity is sufficient to auto-confirm a slot or whether a human should review the proposal. Those are different kinds of moves. They deserve different marks, different review logic, and often different safety boundaries.

> **CORE PRINCIPLE:** The Agent Decision Lane is where I model interpretation, choice, and escalation. If a step can fail through ambiguity rather than pure process breakdown, it belongs here, not hidden inside a generic systems box.

I keep the lane definition concrete. For every agentic node I want five things named: trigger, evidence basis, permitted action, escalation path, and boundary condition. Trigger says what wakes the agent. Evidence basis says what rows, artifacts, or signals it is allowed to consider. Permitted action says whether it may recommend, write, dispatch, or only classify. Escalation path names the human or downstream role. Boundary condition names the stop rule.

agent-decision-node.json

```
{
  "lane": "Agent Decision Lane",
  "nodeId": "care-routing-eval",
  "trigger": "new urgent request plus current capacity snapshot",
```

```
    "evidenceBasis": ["journey-map:urgent-request-v1", "telemetry:wait-stats-aug"],
    "mode": "recommend-only",
    "permittedActions": ["classify_acuity", "propose_route"],
    "escalationTarget": "clinical-ops-lead",
    "boundary": "stop on missing evidence or safety-stop rule"
  }
```

Notice what the node does not claim. It does not claim that the agent is the service owner. It does not claim the host can magically spawn another agent. It does not confuse a logged decision record with live transport. That restraint is important across ASD. The platform is very explicit that ACP is `asd-acp/1`, record-only, and fail-closed on the generic surface. A lane can describe a decision. It cannot fabricate a capability.

> **My take:** The sixth lane is less about adding intelligence to the diagram and more about stopping people from lying about where judgment now lives. If an agent is making a meaningful decision, I want it visible enough to argue with.

Autonomy Patterns and Gate Semantics

The biggest mistake I see in AI service work is binary thinking. Teams talk as if a step is either automated or manual. Real services need a richer autonomy spectrum. Some actions are safe to automate. Some are safe only to recommend. Some are reversible but still sensitive. Some should stop outright and demand human review. The lane is useful only if it carries that nuance.

I use four autonomy patterns repeatedly. **Classify-and-recommend** means the agent proposes a route or summary but does not act. **Auto-execute low-risk** means the agent can perform a bounded action with a clear rollback path. **Human-gated execution** means the agent prepares the action, packages the context, and waits for

approval. **Safety stop** means the agent halts because the case crossed a risk boundary, violated a policy rule, or lacked enough evidence to proceed honestly.

PATTERN	WHAT THE AGENT MAY DO	WHEN I USE IT	FAILURE I AM CONTAINING
Classify-and-recommend	Read, classify, propose next step	Ambiguous requests, early triage, narrative synthesis	Premature automation on partial context
Auto-execute low-risk	Perform bounded, reversible action	Status messaging, artifact preparation, low-stakes routing	Needless human load on routine work
Human-gated execution	Prepare action and wait for approval	Commercial offers, staffing changes, guest-facing memory use	Overconfident action with social or operational cost
Safety stop	Refuse, escalate, and preserve context	Clinical risk, consent breach, unsupported evidence, policy boundary	Silent failure or unsafe improvisation

These patterns are useless without gate semantics. A gate is not a vibe check. It is a named boundary. For service design artifacts the review gate is often the Critic. For service operations the gate might be a clinician, a privacy officer, a journey owner, or a supervisor. What matters is that the lane shows who can authorize the transition and what record the authorization attaches to.

In Halo Care, the design principle was brutally clear: clinical safety was non-negotiable. Success criteria included virtual wait under eight minutes, DNA under ten percent, first-visit resolution above seventy-five percent, and no safety-event trend increase. That last criterion changes everything. It means speed and convenience can never silently outrank safety. So the agentic blueprint cannot merely optimize throughput. It must encode where the service stops, who reviews, and which Critic rows and gate minutes persist the decision.

In Sable & Slate the safety question was different but just as real. The brand promise depended on memory and recognition without crossing into dossier behavior. The

non-negotiables were explicit: no visible tracking, no "algorithmic" service language, and human final authority. That means a memory-related agent can summarize consented facts and boundaries, but it cannot infer traits or stage a guest-visible performance without human judgment. Same lane. Different gate logic.

autonomy-policy.yaml

```
version: 1
policies:
  - id: low-risk-status
    mode: auto-execute
    allowed_actions:
      - prepare_status_message
      - attach_traceability_rows
    requires_human_approval: false
  - id: staffing-or-clinical-route
    mode: human-gated
    allowed_actions:
      - propose_route
      - assemble_context
    requires_human_approval: true
  - id: consent-or-safety-breach
    mode: safety-stop
    allowed_actions: []
    requires_human_approval: true
    stop_reasons:
      - missing_consent
      - unsupported_evidence
      - red_flag_condition
```

The nice thing about this policy style is that it keeps the conversation operational. We are no longer talking about whether the AI is "smart enough." We are talking about which actions are safe to permit, which records must exist first, and how refusal behaves. That is design language I can defend.

> **CAUTION:** An agent that never escalates is not proof of quality. It may be proof that you forgot to design refusal, dissent, or uncertainty into the service.

Typed Agent Handoffs as Durable Records

Once a service includes several roles or several agents, handoff quality becomes a first-class design problem. Chat is not enough. Memory of a conversation is not enough. A screenshot of a thread is definitely not enough. ASD's answer is durable, typed handoffs. They are records of condition plus artifact pointer, and the pointer must resolve inside the same project if one is declared.

The ACP contract is specific here. Event types include `handoff`, `orchestration`, `dispatch`, `critic-veto`, and `human-gate`. The generic CLI and MCP surface expose exactly one ACP capability: `{"protocolVersion":"asd-acp/1","capabilities":["record-only"]}`. Record-only means the platform can validate and persist a handoff record. It cannot pretend that a host process appeared, that another agent definitely replied, or that some external transport succeeded. This boundary is one of the healthiest things about the platform.

For blueprints, typed handoffs matter because the sixth lane often bridges human and agent work. A mapper hands a reviewed blueprint to an ideator. A triage agent hands a proposed route to a clinician. A concierge context service hands a guest brief to the front-desk ambassador. A care-routing agent hands a warm-handoff card to a clinician. In all these cases the receiving role needs the condition and the artifact, not vague recollection.

typed-handoff.json

```
{
  "projectId": "halo-care-8fbf1a5a",
  "planId": "plan-care-routing",
  "from": "mapper",
  "to": "prototyper",
  "condition": "Warm handoff blueprint has passed Critic review and is cleared for s
  "artifactId": "68897cff12ba43a581451187b4918bf2",
  "artifactKind": "service-blueprint"
}
```

This record has consequences. If the `artifactId` is missing while `artifactKind` is supplied, the handoff fails before recording. If the artifact exists but is the wrong kind, it fails. If the pointer is dangling, it fails. That is precisely what I want. An agentic workflow should fail early on referential sloppiness. Otherwise the system ends up pretending it handed off a real thing when it only handed off a sentence.

The same logic shows up in the service-blueprinting flagship skill itself. Its declared handoff says that after the Critic review passes, the `service-blueprint` goes to the ideator with evidence and provenance intact. The wording matters. Handoffs are about continuity of truth, more than task delegation. The receiver should inherit the exact artifact that survived review, including the provenance basis that made it trustworthy.

In operational services, typed handoffs improve human experience too. Halo Care's warm handoff pattern is a good example. The design fix was a sixty-second handoff protocol: patient plus context card in thirty seconds, open clinical question in twenty, confirmation and next contact in ten. Rule: no new assessment in the handoff; red flag means urgent class. That compactness is only safe because the handoff payload is explicit and bounded. A naked escalation where the next clinician must restart the story is a service failure and usually a safety failure too.

HANDOFF QUALITY	WEAK PATTERN	TYPED ASD PATTERN
Trigger	"Someone should look at this"	Named condition recorded in a handoff event
Payload	Chat summary or screenshot	Resolvable artifact pointer plus context fields
Capability assumption	Assume another host or agent exists	Record-only unless the host proves dispatch capability
Safety	Receiver rebuilds state from scratch	Boundary, provenance, and unresolved risks travel with the handoff

My take: Most bad handoffs are not about missing effort. They are about missing shape. People think they passed the work because they passed a message. They did not. They passed a rumor.

Failure Modes: Drift, Loops, Silent Fallbacks

The sixth lane earns its keep when I use it to model failure honestly. Three patterns show up repeatedly. The first is **drift**: the agent's internal framing diverges from the actual evidence or policy basis. The second is **looping**: the service keeps re-querying, re-escalating, or re-summarizing without resolving the condition. The third is **silent fallback**: the service quietly degrades in a way the customer or downstream worker cannot interpret.

Drift often begins with over-broad summary. An agent compresses a guest history into a trait judgment, or a care request into a false urgency class, or a staffing problem into a generic capacity note. Once the summary drifts, every downstream step looks plausible and wrong. This is why Sable & Slate's no-trait-inference veto matters so much. The represented concept brief inserted "guest seems anxious" into the solution field. The Critic vetoed it. Correctly. Memory stores facts, requests, and boundaries only; never trait judgments. That is a design rule and a dignity rule at the same time.

Looping usually appears where ownership is unclear. An agent proposes a route, a human rejects it without a typed fix, the same route gets proposed again, and the customer experiences delay disguised as diligence. Or two agents keep handing a case back and forth because the condition for completion was never formalized. Typed handoffs help here, but only if the conditions are specific enough to break the loop.

Silent fallback is the most dangerous pattern because it looks smooth. A service quietly shifts from intelligent assistance to generic script, or from evidence-backed route to queue default, without preserving the customer's context or explaining the

change. In hospitality that feels cold. In healthcare it can become unsafe. In public services it creates expensive chase demand. The antidote is visible degradation design: if the service falls back, the next actor gets the context and the customer gets a coherent explanation.

failure-mode-checklist.txt

```
- drift: does any summary introduce traits, diagnoses, or motives not present in evi
- loop: can two roles hand the same unresolved condition back and forth indefinitely
- silent fallback: if automation stops, does the next actor inherit usable context a
- unsupported dispatch: are we assuming a host capability ACP does not claim?
- fake finality: did an artifact skip Critic review or a human gate?
```

ASD's record-keeping helps because it preserves the review and veto trail. A `review-veto-log` is not glamorous, but it is the living memory of what the service was not allowed to claim. In agentic systems that negative memory is invaluable. It reminds the team which optimizations crossed consent, safety, or measurability boundaries.

> **CORE PRINCIPLE:** The sixth lane should show how the service stops as well as how it succeeds. Refusal, escalation, and recovery are part of the design, not embarrassing exceptions to hide later.

Case: Sable & Slate and Halo Care

Sable & Slate and Halo Care make a useful pair because they sit at opposite ends of tone and risk. One is luxury hospitality, where memory and recognition create value but must never become surveillance theater. The other is telehealth and virtual ward care, where any redesign has to move with clinical safety still in charge. Both cases needed the sixth lane because both involved agentic help under hard human boundaries.

Sable & Slate had nine properties, €680 ADR, and a brand promise built around remembered stays. A three-year digital programme had accidentally stripped out the memory by removing human continuity. Guests repeated preferences. Staff lost context across shifts. Turnover at thirty-four percent drained tacit expertise. The tempting move was obvious: build a guest emotion score and a magical memory engine. The platform refused the worst version of that fantasy.

The first major veto came in concept review. A context-memory concept included trait inference in the solution field. The Critic blocked it under `no-trait-inference`. The fix was sharp: memory stores facts, requests, and boundaries only; never trait judgments. That one move defined the design ethic for the whole service. The future-state blueprint then positioned the agentic work where it belonged: a backstage, consent-bound brief assembly step plus a visible human performance step in frontstage. The system cues. The staff perform. Human final authority remains intact.

A second Sable & Slate lesson came from shift handoff. The operational analysis found that a sixty-second brief degraded to three minutes under short staffing. The fix became a two-tier handoff: minimum brief and complete brief. That is classic sixth-lane work. The agentic layer can assemble the brief. It cannot decide that no handoff is needed. The boundary is the service design.

Halo Care had a harsher operating context. It needed to improve access without increasing safety-event trends. The redesign work registered 1,983 evidence rows in the synthetic scenario and used Critic vetoes plus signed human gate minutes as the governance mechanism. One review blocked a route that would have quietly returned patients to the old five-day queue. Another set of safety rules bounded the warm handoff and the context service. The point was clear: no service move that improved flow was acceptable if it undermined clinical safety or consent.

The warm handoff pattern became the emblematic sixth-lane design. The patient should not retell the whole story at each transition. The context card should follow. But the handoff rule stays clinically humble: no new assessment in the handoff, and any red flag escalates to urgent class. That is a perfect example of bounded autonomy. The agentic part assembles and carries context. The clinical judgment stays with the clinician.

safety-gate-record.json


```
{
  "projectId": "halo-care-8fbf1a5a",
  "type": "human-gate",
  "condition": "Warm handoff route passed Critic review; clinical lead approves regi
  "linkedReviewLogId": "2de447148c3b4f36b294524d813fef40",
  "approvalRole": "clinical-lead",
  "boundary": "No region rollout if safety-event trend increases or consent rules ar
}
```


Both cases underline the same point. The sixth lane is not where I celebrate autonomy. It is where I discipline it. At Sable & Slate the discipline protects dignity, brand, and consent. At Halo Care it protects safety, continuity, and professional authority. In both, typed handoffs and Critic review make the service more legible to the people who must trust it.

> **My take:** The real test of an agentic service is whether the humans inside it feel better informed and better protected after the automation arrives. If staff and customers both end up reconstructing context from scratch, the intelligence is fake no matter how elegant the demo looked.

That is why I keep the sixth lane close to the rest of the blueprint rather than treating it as a separate AI architecture diagram. The customer still sees a service. The frontline still carries the brand. The backstage still absorbs trade-offs. The support layer still constrains what is possible. Agent decisions belong in that same production reality. Once I draw them there, I can finally ask the right questions: what may the agent do, what must it never do, who can veto it, how is the handoff typed, and how does the service fail without pretending nothing went wrong.

CHAPTER 10

The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 12

Cross-Channel Orchestration & Story Mapping

NEXT CHAPTER

Cross-Channel Orchestration & Story Mapping

One service, many channels, zero drift

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 12

Cross-Channel Orchestration & Story Mapping

One service, many channels, zero drift

Reading time

13 min

Channels Drift; Artifacts Don't

Most channel strategies are really ownership strategies in disguise. The app team owns the app. Retail operations own stores. CRM owns email. Contact-centre leadership owns phone. Logistics owns courier interactions. Each group optimizes its local metrics and calls the result omnichannel. Customers then walk through the service and discover a relay race where nobody passed the baton.

The drift shows up in recognizable forms: one channel promises what another cannot fulfill, a status appears in email but not in the app, the store sees a different truth than the website, a refund is "initiated" in one system and invisible in another, or the only

recovery path is a phone number because nobody designed the degraded experience on the primary channel. This is not a copywriting issue. It is an orchestration issue.

ASD's strength here is not a prettier multi-channel diagram. It is shared artifacts with gates. A current-state journey map shows where the customer crosses channels. An ecosystem map shows which actors sit behind those channels. The Orchestration Canvas forces channel roles, signals, responses, ownership, and metrics into one surface. Story mapping then turns that design into release slices without losing the customer's chronology. The common thread is that the artifacts survive teams, tools, and planning ceremonies.

The contrast with style-guide thinking is important. Style guides help interfaces look consistent. They do almost nothing to keep operational truths consistent. I can have perfectly matched visual design while the app says ready, the email says delayed, the store cannot find the order, and the contact centre sees a batch-synced half-truth. Shared artifacts solve a deeper problem: they align what the channels are allowed to claim.

> **CORE PRINCIPLE:** Channel consistency comes from shared service artifacts, not from brand vocabulary alone. A beautiful contradiction is still a contradiction.

This is why I like saying that channels play roles. A role is contextual and relational. A territory is defensive. In a good cross-channel service, one channel may be Primary in one phase, Supporting in the next, Silent by design in another, and Recovery-only when the happy path breaks. Once teams accept role logic, duplicated features start looking less sophisticated and more wasteful.

The method also reduces the common temptation to let channels chase vanity metrics independently. If the app team wants push engagement and the CRM team wants email open rate, they can easily flood the customer with overlapping status. The canvas makes that collision visible because the same phase cannot honestly have two Primary channels saying different things. One service. One story.

My take: I have never seen a true omnichannel problem solved by asking each channel owner to "coordinate better." Coordination gets real only when there is a shared artifact precise enough to prove where the contradiction lives and who has to change.

The Orchestration Canvas

The Orchestration Canvas is the boundary object I use when a service crosses several channels or partners and the customer still expects one coherent experience. It puts the journey phases across the top. It places the customer hemisphere above and the organizational hemisphere below. That simple move matters because it stops the meeting from collapsing into channel-specific feature requests before the team has aligned on the shared chronology.

For each phase I want the customer goal, actions, emotional valence, and moment that matters. Then I want the channels with explicit roles, the ecosystem actors involved, the signals the organization can actually sense, the responses it will issue, the named owner, and the metric that tells us whether the choreography is holding. The canvas is useful because these rows are interdependent. A signal without a response is a gap. A moment that matters without an owner is a risk. Two Primary channels in the same phase are usually a political compromise masquerading as design.

orchestration-canvas-blank.txt

```
ORCHESTRATION CANVAS - [service] - v[ ]
Journey phases: [1] [2] [3] [4] [5]
CUSTOMER HEMISPHERE
  Customer goal
  Actions and behaviors
  Emotional valence (1-5)
  Moments that matter
ORGANIZATIONAL HEMISPHERE
```

```
  Channels - Primary
  Channels - Supporting
  Channels - Silent / Recovery
  Ecosystem actors
  Signals sensed
  Responses issued
  Ownership
  Metrics
GAP-FIRST AUDIT
  orphaned signals / unsupported moments / role collisions / unowned seams
```

I read the canvas gap-first. That is essential. Teams want to admire the finished rows. I want to find the empties, collisions, and fiction. A signal row entry with no response is an orphaned signal. A moment that matters with no named owner is unsupported. Two channels both claiming Primary is a collision. A transition between channels with no operational owner is an unowned seam. These are not subtle UX flaws. They are structural contradictions.

Because the artifact lives in versioned markdown, I can also audit it mechanically. A simple agent can scan columns for the four gap patterns and output a JSON gap register with cell coordinates. That is mundane and immensely useful. It gives the room something crisp to work on instead of another abstract conversation about alignment.

canvas-auditor.txt

```
You are CANVAS-AUDITOR.
For each journey-phase column report:
1. ORPHANED SIGNAL - a signal with no response
2. UNSUPPORTED MOMENT - a moment-that-matters with no owner
3. COLLISION - two channels marked Primary in one phase
4. UNOWNED SEAM - a transition with no named owner
Output: JSON gap register with canvas cell coordinates only.
```

The role system is where the canvas becomes politically useful. I assign one Primary channel per phase. Supporting channels reinforce, remind, or mirror. Silent channels are deliberately quiet, which is often the most sophisticated choice. Recovery

channels are reserved for breakdown, not normal flow. This role grammar stops teams from stuffing every channel with the same functionality and calling that customer centricity.

CHANNEL ROLE	WHAT IT DOES	WHAT IT MUST NOT DO
Primary	Carries the main interaction in that phase	Contradict another Primary channel
Supporting	Reinforces the primary with confirmation or context	Create a divergent story or require duplicate action
Silent	Stays out of the way by design	Wake up unnecessarily and increase noise
Recovery	Activates when the happy path fails	Substitute for fixing the primary path

Once I have that structure, the canvas starts feeding everything else. Requirement cards become cross-channel conditions instead of channel feature lists. The blueprint can inherit the agreed choreography. Measurement can watch seam health rather than isolated click-throughs. Governance can attach to a Journey Owner instead of leaving every contradiction to inter-team diplomacy.

> **My take:** The canvas earns its keep when a channel owner has to say out loud, "In this phase we are Silent by design." That is the moment the organization stops equating visibility with value.

Story Mapping for Incremental Delivery

The Orchestration Canvas tells me how channels should behave together. Story mapping tells me how to build that service without destroying the chronology. This is

where many delivery teams regress. They take a beautifully coherent multi-channel design and shred it into component backlogs, team epics, or platform streams until nobody can see the customer journey anymore. Story mapping repairs that by keeping the backbone of the experience visible while I slice releases underneath it.

ASD's `story-mapping` skill lives in the ideation phase and produces a `story-map` artifact. I use it after the canvas has stabilized because then the backbone is grounded. The backbone is the sequence of customer activities. Under each activity I place the tasks, capabilities, or conditions needed to make that activity work. The release slices then run horizontally, preserving a coherent path through the journey instead of delivering one isolated piece per channel.

This matters especially in cross-channel services because a release can look complete to one team while being useless to a customer. An app push without store readiness is fake progress. A return tracker without refund settlement visibility is partial truth. A graceful window-rescue email without OMS support is a new promise resting on old drift. Story mapping makes these half-builds visible before the roadmap does damage.

story-map-slices.yaml

```
backbone:
  - place_order
  - prepare
  - ready_and_collect
  - grace_window
  - return_and_refund
slices:
  - id: S1
    name: honest_order_timeline
    includes: [place_order, prepare]
    promise: one order, one story across app, web, and email
  - id: S2
    name: five_minute_collection
    includes: [ready_and_collect]
    promise: QR pass, desk guidance, and first-time-found pickup
  - id: S3
    name: life_happens_rescue
    includes: [grace_window]
```

```
      promise: extend, re-slot, or release with visible consequences
    - id: S4
      name: visible_return_money
      includes: [return_and_refund]
      promise: route-aware return choice and refund clock the customer can see
```

The slice should represent a usable story, more than technical progress. That is why I still care about the older delivery idea of a walking skeleton. The first slice must let the service tell one truthful story end to end, even if the experience is narrower than the future ambition. In cross-channel work, this usually means choosing fewer phases and keeping them coherent rather than trying to light up every surface at once.

Story mapping also helps me protect recovery and degradation work from being perpetually deferred. Teams love building the happy path first and promising to return later for rescue, exception handling, and channel outage behavior. The customer then discovers the real product during the first broken moment. If the canvas shows Grace over penalty as a principle, the story map should include a release slice that makes that grace operational. Otherwise the principle is ceremonial.

I also use story mapping to keep metrics honest. A slice is not done because one team deployed a feature. A slice is done when the service can actually support the phase promise across its relevant channels. If the app shows a ready push but the desk cannot scan the QR pass and locate the order, then the ready-and-collect slice is not done. Story mapping gives me the language to argue that clearly.

> **CORE PRINCIPLE:** Release slices should preserve customer coherence. If a slice cannot tell one truthful story from the customer's point of view, it is probably a team convenience slice, not a service slice.

DELIVERY HABIT	WHAT GOES WRONG	STORY-MAP CORRECTION
Channel-first backlog	Each team ships local wins that do not compose	Slice horizontally through the journey

DELIVERY HABIT	WHAT GOES WRONG	STORY-MAP CORRECTION
Happy-path obsession	Recovery logic arrives too late	Give degradation and rescue their own release slice
Metric siloing	Local dashboards hide cross-channel contradictions	Define slice success by end-to-end journey promise
Invisible ownership	Cross-team seams drift after launch	Carry Journey Owner and seam owner into the slice plan

Case: Juniper Retail

Juniper Retail is the case I return to when someone says omnichannel is mostly a UX consistency problem. It was a fictional EU home-and-lifestyle retailer with 140 stores, 3.2 million loyalty members, 1.6 million click-and-collect orders per year, and 820,000 returns across four routes. Each channel worked tolerably on its own. Together they formed a drift machine. Thirty-four percent of click-and-collect orders generated a pickup-related contact. Eleven percent of orders were never collected inside the window. Returns disappeared into an eight-day refund blackout. The service was organized by channel territory and experienced as one long apology.

The programme used the Orchestration Canvas as the boundary object from Define through Deliver. That continuity was the real breakthrough. The same canvas co-built by five channel-owning teams in week six later became the basis for choreography, blueprinting, pilot backlog, and governance dashboard specification. Instead of creating a workshop artifact and then abandoning it to delivery, the team kept one shared truth alive.

The gap-first audit found nineteen issues in the first canvas version. Two of them explain the power of the method immediately. First, the `prep_started` and `window_day4` signals were orphaned. The systems sensed them, but the service had no response planned. Second, the app and CRM both claimed to own the ready

notification, creating a Primary-channel collision. That is exactly the kind of contradiction customers feel as "I got three messages and still don't know what is true."

The governance answer was elegant and strict: app as Primary, email as Supporting, Journey Owner as arbitrator, and phone held back for Recovery rather than routine narration. That single choice did more for coherence than a dozen content workshops would have.

juniper-canvas-v1.csv

```
phase,primary,supporting,silent_or_recovery,signal,response,owner
place_order,web_or_app,email,phone,order_placed,confirmation_plus_timeline,ecommerce
prepare,app_push,email,phone,prep_started,progress_push,retail_ops
ready_and_collect,store_desk_plus_app_pass,email,none,ready,ready_push_plus_qr_pass,
grace_window,app_push,email,phone_recovery,window_day4,rescue_prompt_extend_reslot_r
return_and_refund,app_tracker_plus_store_till,email,phone_recovery,refund_settled,vi
```

The experience principles that emerged from the canvas were strong because they were testable. *One order, one story.* Every channel shows the same status timeline. *Tell, don't be asked.* The customer should not need to phone for a status the organization already knows. *Grace over penalty.* A missed collection window should trigger rescue, not silent lapse. *Refund at the speed of return.* Money should move when goods move and the customer should be able to watch that movement. *Consent-gated awareness.* Location-triggered push should exist only where explicitly opted in and easily revoked.

Those principles then fed the story map. The first slice was not "mobile notifications" or "store tooling." It was the honest order timeline: the ability for web, app, and email to tell one coherent story. The second slice was the five-minute collection promise: QR pass, desk guidance, and reliable ready state. The third slice was the grace-window rescue: extend, re-slot, or release with visible consequences. The fourth slice was visible return-and-refund: route-aware return choice and a refund clock the customer could see.

This release logic stopped a classic failure. One team initially wanted to ship instant settled refunds for the parcel-locker route because it sounded impressive. Adversarial review caught the lie. The service could promise instant initiation, not instant settlement, for that route. The story map forced the truth back into the slice. Customers would rather receive a narrow honest promise than a broad fake one.

The pilot ran across twelve stores plus online. The illustrative results justify the discipline: pickup-related contact rate fell from 34% to 17%, no-show dropped from 11% to 6%, refund time moved from eight days to instant for in-store returns, and pickup NPS rose from +9 to +34. Again, I would not credit the canvas alone. The outcome came from keeping the choreography, blueprint, requirements, prototype, and pilot slices all tied to the same service story.

> **My take:** Juniper did not win because it added more channels. It won because it assigned clearer roles to the channels it already had and stopped letting each one improvise the plot.

The case also shows why story mapping belongs in a methods chapter about orchestration. Without the slice logic, Juniper would have fallen back into team backlog gravity. The app team would have shipped the push layer, CRM would have optimized email timing, store ops would have updated scripts, and the customer would still be living through channel contradiction. The story map preserved the service as a service.

release-readiness-checklist.txt

```
- does the slice preserve one truthful customer story?
- is there exactly one Primary channel per relevant phase?
- do all signals in the slice have responses?
- is recovery designed where the slice can fail?
- can support systems actually sustain the promised timing?
  - does the owning metric measure journey coherence rather than local engagement al
```

If you keep one habit from this chapter, let it be this: never let a cross-channel service go straight from journey insight to team backlog. Put the canvas in the middle. Put the story map after it. Force the organization to say who owns the seam, which channel is Primary, what happens on degradation, and what the first truthful slice really is. That is how multi-channel services stop drifting apart.

CHAPTER 11

Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 13

Structured Ideation & How Might We at Machine Speed

NEXT CHAPTER

Structured Ideation & How Might We at Machine Speed

Diverge wide, converge ruthlessly

Continue Reading →

ASD ' 26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15

CHAPTER 16

Transformative Services, Behavior Change & Recovery Engineering

Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 13

Structured Ideation & How Might We at Machine Speed

Diverge wide, converge ruthlessly

Reading time

13 min

Ideation economics changed; judgment did not

I no longer treat ideation as a sticky-note ceremony. I treat it as a throughput problem with a governance problem attached. Throughput is easy. One human facilitator and a few ASD role prompts can generate more structured idea seeds in an hour than an old-school workshop could surface in two days. Governance is harder because the machine will happily produce elegant nonsense at that same speed. The discipline therefore starts before the first concept. I want a validated evidence base, an explicit challenge frame, and a kill rule that everyone agrees to upfront.

That is why agentic ideation feels different from classic workshop ideation. In a room, scarcity is the constraint. People are tired, self-conscious, protective of hierarchy, and

slow to build on each other's thoughts. In an ASD sprint, abundance is the constraint. The machine can fill the room with plausible options so fast that the team starts admiring fluency instead of asking whether a concept actually solves the right pain, respects the operating environment, and earns the right to survive. I have watched senior teams get seduced by a crisp one-pager for a terrible idea because the prose sounded strategic. That is the modern failure mode.

So I impose a sequence. Evidence first. Reframing second. Volume third. Filtering fourth. Briefing fifth. I do not let the room vote itself into fantasy before the concept faces operational gravity. Calder taught this lesson beautifully. The sponsor originally asked for "innovative onboarding ideas for SMEs." If I had accepted that brief at face value, the agents would have served up a buffet of chatbots, dashboards, and instant-account gimmicks. Instead I anchored the sprint to validated problems: requirements invisible before branch contact, document receipt uncertainty, repeated re-requests, and status blindness during KYC review.

OLD WORKSHOP CONSTRAINT	AGENTIC SPRINT CONSTRAINT	WHAT I DO ABOUT IT
Too few ideas	Too many plausible ideas	Enforce evidence lineage before divergence starts
People self-censor	Models over-generalize	Force persona-specific prompts and severe constraints
Groupthink in the room	Consensus bias in model outputs	Use cross-sector analogies and mechanism-level clustering
Politics block kill decisions	Polished nonsense survives too long	Run `nuf-triple-lens-filter` in public with explicit vetoes

> **CORE PRINCIPLE:** In agentic ideation, generation is cheap and discernment is expensive. Spend your process design on discernment.

The practical consequence is that I care less about the theatrical energy of the ideation session and more about the quality of the handoff objects. I want an insight

register that names the pain clearly. I want HMW prompts that neither collapse into solution bias nor drift into fluff. I want every concept to carry its source evidence, target actor, underlying mechanism, and value hypothesis. Once ideas become structured objects, the agents can help with volume and deduplication without becoming the authors of strategy.

The Calder sprint moved fast because the evidence bundle was already clean. The current-state journey and insight cards from the earlier research article gave us the right tensions: applicants bought certainty with their own time, accountants had unofficial workarounds, and relationship managers spent too much of their day compensating for invisible back-office queues. Those tensions became the true raw material. The agents did not invent a problem. They expanded the option set around a problem we had already earned the right to solve.

> **My take:** The worst sentence in modern design practice is “Let’s brainstorm some ideas.” I want the challenge framed so tightly that the room knows what is off-limits before the first idea lands. Constraint is mercy.

Use `hmw-formulation` to turn evidence into generative tension

The `hmw-formulation` method is where I stop research from dissolving into platitudes. Its job is simple on paper: take an `insight-set` and produce an `hmw-set`. In practice it is the hinge between synthesis and ideation. The method forces each node to carry an opportunity field, at least one evidence reference, and clear provenance. The Critic gate rejects any prompt that names a specific solution instead of a problem space. That one rule saves enormous amounts of wasted motion.

I like this method because it encodes a subtle craft judgment that junior teams often miss. A weak HMW is usually weak in one of three ways. It is too broad, which gives the model room to hallucinate ambition. It is too narrow, which reduces ideation to

surface decoration. Or it embeds the answer in the question, which means the team is pretending to ideate after making the decision already. The method cannot replace taste, but it can force the first draft into a shape where taste has something solid to work on.

hmw-run.sh

```
PID="calder-sme-sprint-2026"
asd method-run --projectId "$PID" --methodId hmw-formulation

# then write the resulting hmw-set through the gated artifact path
# once the nodes, evidence references, and opportunity fields are ready
```

For Calder, one early draft said, “How might we approve simple SMEs instantly?” It sounded energetic. It was also invalid. The prompt assumed a policy shortcut, smuggled a solution outcome into the brief, and erased the real pain, which was uncertainty before approval. The reframed version became far more useful: “How might we reduce uncertainty before approval without weakening controls?” Same domain. Same tension. Completely different design space. Now the team could explore receipt confirmation, status heartbeats, completeness checks, named ownership, or accountant-led preflight without pretending the bank could skip KYC.

I usually write HMW prompts in pairs. One line states the design opportunity in customer language. One line states the red line in operating language. That dual framing keeps the ideation space alive without letting the room wander into illegal, brittle, or theatrically impossible territory. In regulated sectors, this matters even more because the model’s favorite move is to invent delight by erasing constraints the business actually has to live with.

EVIDENCE-BACKED TENSION	POOR HMW	STRONGER HMW	WHY THE STRONGER ONE WORKS
Applicants do not know whether documents were accepted	How might we build a document chatbot?	How might we give applicants immediate certainty that submitted documents are usable?	It opens multiple mechanisms and stays anchored to the outcome

EVIDENCE-BACKED TENSION	POOR HMW	STRONGER HMW	WHY THE STRONGER ONE WORKS
Branch staff chase back-office status by email	How might we automate branch email?	How might we let frontline staff answer status questions without inventing unofficial workarounds?	It includes the human service reality and the trust problem
Accountants prepare cases off-platform anyway	How might we make accountants use our portal?	How might we turn accountant preflight work into a consented, structured part of the onboarding service?	It frames an ecosystem role, not a channel-enforcement fantasy

Once the HMW bank is clean, I enrich it with lateral sub-prompts. Calder used prompts such as: borrow a trust mechanism from parcel tracking; borrow a readiness mechanism from appointment triage; borrow a document-assurance pattern from signing software. The HMW stays stable while the mechanism shifts. That is a far better way to get novelty than telling an LLM to “be more innovative.” Models are bad at being original on command. They are better at recombining explicit mechanism prompts when the target tension is clear.

The human checkpoint in `hmw-formulation` matters here. ASD is explicit: a human approves the evidence-backed HMW set before it moves forward. I like that rule. It reflects reality. I want the machine to help me enumerate possibilities, not to decide which problem the organization is willing to own. That remains a design and leadership judgment.

Diverge hard, then let `nuf-triple-lens-filter` do the violence

With the HMW set approved, I switch from reframing to volume. Calder ran divergence in three modes: direct solutions, cross-sector analogy, and severe-constraint variations. The target was not a magical number of ideas. The target was enough variation that the team could see patterns, dead ends, and surprising

mechanism transfers. We ended up with roughly 200 raw seeds because that was the amount required to expose the concept families hiding underneath the surface wording.

I structure raw concept records the same way every time: concept id, source insight, HMW id, target persona, mechanism, channel, backstage dependency, value hypothesis. This looks bureaucratic until you need to cluster ideas at speed. Once the data is structured, OpenCode can deduplicate the set by mechanism instead of by language. That is how Calder got from 200 seeds to 24 concept families without losing the important differences. A status bot and a status heartbeat email feel similar in prose. Mechanically, they may sit in the same family. An accountant-preflight channel belongs somewhere else entirely.

concept-seed.json

```
{
  "idea_id": "HMW03-analog-07",
  "source_insight": "INS-01",
  "hmw_id": "HMW-03",
  "persona": "Malik, finance manager",
  "mechanism": "parcel-style status heartbeat",
  "channel": "email and secure web",
  "backstage_dependency": "credit workflow emits milestone events",
  "value_hypothesis": "Reduce branch status-chasing by making progress legible"
}
```

Then comes the real work. `nuf-triple-lens-filter` is the first public act of convergence. Every surviving concept is scored on New, Useful, and Feasible. The method is blunt by design. Its acceptance rules require a non-empty score field and reject any concept advanced with a failing score. The Critic gate then attacks the weak points. I do not ask the team to love this stage. I ask them to trust it. When a favorite concept dies here, it saves weeks of downstream politics and prototype theatre.

New is contextual novelty, not global novelty. A concept does not need to be unprecedented on earth. It needs to represent a meaningful shift in the service

context. Useful asks whether the concept addresses the proven pain rather than decorating adjacent friction. Feasible asks whether the concept survives the actual operating environment: regulation, systems, staffing, incentives, and timing. I sometimes add a separate viability overlay later, but I do not overload NUF itself. The power of the method comes from being easy enough to use live.

CONCEPT FAMILY	NEW (1-10)	USEFUL (1-10)	FEASIBLE (1-10)	WHAT THE ROOM LEARNED	VERDICT
LaunchPad preflight + document vault + status heartbeat	7	9	8	Strong relief on uncertainty without touching policy	Stage
Guided Start for complex SMEs with named RM ownership	6	8	9	Human-led route solves complex-case trust and stall risk	Stage
Accountant Connect pre-validation channel	8	8	7	Ecosystem leverage beat another self-service feature	Stage
WhatsApp status bot for all applicants	5	6	3	Channel risk and weak differentiation killed it	Kill
Instant limited account before full review	7	8	2	Commercially tempting, operationally indefensible	Kill

Notice what the scoring matrix does. It translates enthusiasm into comparable judgment. Dot voting is still useful because it reveals energy. I use it to surface where stakeholders see potential. I do not use it to make the decision. Popularity is a signal. It is not governance. On Calder, the room loved the idea of instant limited accounts because it felt commercially bold. The NUF matrix made the hidden cost visible in seconds. Feasibility collapsed under compliance and operational risk. That public collapse changed the tone of the sprint. From that moment, the team stopped mistaking boldness for value.

I also like how NUF disciplines the role of the adversarial reviewer. Codex, acting as a critic in the Calder case, was not asked to generate alternatives endlessly. It was asked to do a meaner, more useful job: verify traceability to evidence, attack feasibility assumptions, and flag duplicate concepts masquerading as different ones. That is where an adversarial model earns its keep. It prunes the tree.

> **CAUTION:** Do not hide low feasibility scores inside a follow-up workstream. If a concept only works after three policy exceptions, two new integrations, and a heroic branch manager, it did not "need more detail." It failed the gate.

Calder SME Banking: a three-week concept sprint with teeth

The fictional Calder case is useful because it shows the whole pipeline under real pressure. The sponsor wanted answers before the quarterly investment gate. Relationship managers had limited workshop time. KYC operations had no patience for innovation theatre. The evidence bundle from the earlier research sprint made the challenge crisp: median operational account opening took nineteen calendar days; 68% of applicants experienced at least one document re-request; branch staff could not answer status questions confidently because they had no joined-up view of the process.

Week one was all about challenge framing. I reviewed the evidence handoff, wrote the sprint brief, and used `hmw-formulation` to create the HMW bank with explicit red lines. No relaxed AML. No bypassing UBO checks. No magical core replacement. No concepts that assumed applicants wanted to learn banking jargon. Those rules sounded restrictive to one commercial lead. They were liberating in practice because they stopped the room from wandering into fantasy solutions that the sponsor would have had to kill later anyway.

Week two was divergence. Claude Code fanned out idea generation across personas and HMWs. We used direct mode, analogy mode, and severe-constraint mode. OpenCode then normalized and clustered the output. GitHub issues held the 24 concept families so stakeholders could comment asynchronously on concrete idea clusters instead of debating abstract directions in a meeting. That workflow matters. When the concept families are visible as versioned objects, people critique substance rather than charisma.

Week three was convergence and staging. We ran a mixed workshop with customers, relationship managers, an accountant partner, and the SME sponsor. Humans added roughly twenty ideas that the agents had missed or softened. That human contribution was decisive. The machine kept drifting toward portal improvements. The accountants and RMs reframed the opportunity around preflight work and named ownership. After dot voting, we took eleven shortlisted families into the NUF matrix and then into concept briefs. Codex tried to kill the survivors. Two died. Three held.

The final staged set covered the service ecology properly. **Calder LaunchPad** handled standard-volume simplification: a preflight checker, reusable checklist, document vault, and status heartbeat. **Guided Start** handled complexity with named RM ownership and a coached path for multi-director or credit-intent firms. **Accountant Connect** extended the service boundary outward, turning a shadow workflow into a consented part of the operating model. None of these concepts were flashy. All three were evidence-bound, operationally legible, and prototype-worthy.

That last phrase matters. Ideation ends when a concept becomes a prototype proposal with a clear test question. For LaunchPad, the question was whether receipt confirmation and visible progress would materially reduce branch status-chasing. For Guided Start, the question was whether named human ownership improved confidence enough to justify the extra RM time for complex cases. For Accountant Connect, the question was whether accountants would actually do pre-validation if it saved client time and reduced document loops. Those are prototype questions. Everything before them is still concept work.

I left Calder with a small portfolio, not a bloated roadmap. That is the point. The machine-speed part of the title refers to how fast I can expand the option space. The

service-design part refers to how disciplined I am in shrinking it back down. Good ideation is not infinite creativity. Good ideation is evidence-bound possibility, followed by unapologetic elimination.

When teams tell me AI makes ideation messy, I usually agree with the diagnosis and reject the implied conclusion. AI makes undisciplined ideation messy. The cure is not to slow the machine down. The cure is to give it a tighter contract: validated evidence in, bounded HMWs, structured concept records, public NUF scoring, and human approval before prototyping. Once you do that, the speed becomes an advantage rather than a liability.

That is why I keep returning to ASD's method discipline. The manifests do not remove judgment. They create a lane where judgment can be applied at the right moments. `hmw-formulation` prevents pre-solved prompts. `nuf-triple-lens-filter` prevents sentimental attachment to weak concepts. The rest is craft: reading the evidence honestly, choosing the right tensions, and keeping the team brave enough to kill what should die.

CHAPTER 12

Cross-Channel Orchestration & Story Mapping

CHAPTER 14

Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

NEXT CHAPTER

Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

Test the service before it exists

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09

CHAPTER 10

Living Journey Maps: From Transcripts to Sentiment Curves

The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 14

Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

Test the service before it exists

Reading time

14 min

Think in a prototyping spectrum, not a single mockup

When a team says, “We need a prototype,” I immediately ask, “Prototype of what risk?” That question changes everything. If the uncertainty is basic comprehension, I want a storyboard or paper flow. If the uncertainty is spatial sequencing, I want a desktop walkthrough. If the uncertainty is emotional tone and staff ergonomics, I want roleplay. If the uncertainty is whether the service promise can survive real customer behavior without full automation, I want Wizard-of-Oz. If the uncertainty is unit economics, staffing load, or operational repeatability, I want a bounded concierge pilot. Different risks require different fidelities.

This is why I talk about a spectrum instead of a ladder you climb mindlessly. Higher fidelity is not inherently better. It is more expensive, more politically visible, and easier to over-believe. I have seen teams rush to a polished front-end because they wanted the prototype to look impressive. That move often hides the service question under interface gloss. A paper storyboard can answer a crucial learning question in two hours. A medium-fidelity roleplay can expose a broken backstage dependency before a single developer opens a branch. Cheap prototypes are not amateur work. They are sharp tools when aimed at the right uncertainty.

ASD helps because the methods make the intended learning explicit. `storyboard-authoring` returns a storyboard artifact. `desktop-walkthrough` returns a walkthrough script. `wizard-of-oz` returns a prototype plan. `concierge-pilot` does the same at higher fidelity. The artifacts force me to state the hypothesis, the touchpoints to simulate, the fallback rules, and the pass or fail criteria. That keeps a prototype from mutating into a demo whose only real job is to reassure the sponsor.

PROTOTYPE MODE	BEST USED FOR	MAIN ARTIFACT	WHAT I EXPECT TO LEARN
Storyboard	Comprehension, promise clarity, emotional arc	`storyboard`	Do people grasp the value and sequence at all?
Desktop walkthrough	Logic, timing, routing, spatial handoffs	`walkthrough-script`	Where does the service break when moved through real steps?
Service roleplay	Scripts, tone, staff ergonomics, exception handling	`storyboard` plus props	What compensating behavior appears when people enact it?
Wizard-of-Oz	Frontstage believability and backstage burden	`prototype-plan`	Can humans fake the automation long enough to learn the right thing?
Concierge pilot	Unit economics, staffing, service standards, adoption	`prototype-plan` plus `test-report`	Does the service work in bounded live conditions?

> **CORE PRINCIPLE:** Choose the cheapest fidelity that can falsify the hypothesis. A prototype that cannot prove you wrong is stage dressing.

Ember Hospitality illustrates the value of this framing. Leadership originally wanted a substantial software build for “occasion management” inside the booking flow. That would have answered the wrong question first. Before software mattered, the team needed to learn whether guests wanted to disclose special-occasion intent in a structured way, whether floor staff could trust a shared celebration brief, whether dietary and deposit promises could travel intact, and whether a human concierge layer was economically viable. Those were service questions. The prototype spectrum let us ask them in the right order.

> **My take:** I do not prototype to make stakeholders feel momentum. I prototype to buy down uncertainty with the least possible spend. If a prototype makes everybody feel confident too quickly, I assume it is hiding the hard part.

Start with `storyboard-authoring` and desktop walkthroughs

The `storyboard-authoring` method looks simple. It is not. The method requires a minimum of four nodes and rejects any frame with no action in the `scene` field. I love that gate because it prevents one of the oldest service-design sins: making mood boards and calling them experience prototypes. A service storyboard needs a protagonist, a sequence, an observable move, and a consequence. Otherwise the team is only projecting its intentions.

For Ember, the first storyboard had six panels: booking intent, occasion disclosure, dietary capture, arrival cue, table moment, and payment follow-through. Every frame had two layers: what the guest believes is happening and what the staff need backstage for the next moment to hold. That second layer is where the service

prototype becomes serious. The guest sees a simple prompt asking whether the dinner is for a special occasion. The backstage layer asks harder questions. Who needs to know this? When? In what level of detail? Does the floor runner need the emotional context, or only the timing cue?

prototype-plan.json

```
{
  "id": "ember-spectrum-01",
  "method": "storyboard-authoring",
  "hypothesis": "Guests will disclose special-occasion intent when the prompt is spe
  "fidelity": "low",
  "targetSubjects": ["repeat diners", "group bookers", "floor-staff champions"],
  "successCriteria": [
    "at least 80 percent of participants understand what the prompt is asking",
    "staff can state which role sees which occasion detail"
  ],
  "failureCriteria": [
    "guests perceive the prompt as intrusive",
    "staff say the service would expose sensitive context too broadly"
  ]
}
```

Once the storyboard is coherent, I move the service onto a table. Desktop walkthroughs sound almost childish until you see how brutally they expose broken logic. I map the environment with paper, printed interface states, table markers, shift notes, kitchen tickets, or route cards. Then I move tokens through the sequence and narrate every state change aloud. In Ember’s case, a simple walkthrough exposed the service fracture instantly. Occasion information sat in one place, dietary nuance in another, and group-deposit promises somewhere else. On the screen, that looked manageable. On the table, it became obvious that the host would be forced to reconstruct the service from fragments under real dinner-service pressure.

Desktop walkthroughs are also where I start hunting for hidden backstage work. If somebody has to invent a note, remember an exception, text a manager, or run to the kitchen, I want to see that labor immediately. The future bug often appears first as a compensating behavior in a fake environment. A person leans over and whispers, “I

would probably tell the manager here just in case." That is gold. It means the designed process is not trustworthy enough on its own yet.

WALKTHROUGH CHECKPOINT	QUESTION I ASK	COMMON FAILURE SIGNAL	WHAT IT USUALLY MEANS
Information capture	What exactly did the customer provide?	Staff rephrase or reinterpret the note manually	The input model is too vague
Handoff	Who needs this next and by when?	People invent side channels	The designed route lacks trust or timeliness
Exception point	What happens if the customer changes the plan?	The room pauses and improvises	No fallback script exists yet
Promise delivery	How does the customer know the promise held?	Staff assume the guest will "just notice"	The service lacks an explicit confirmation moment

At Ember, this stage killed one early idea elegantly. A first concept assumed any special occasion should be broadly visible to staff to ensure enthusiasm. The bodystorming and walkthrough sessions proved the opposite. Some occasions needed celebration. Others needed discretion. A proposal dinner, a fertility-anniversary meal, or a quiet reunion after bereavement should not be broadcast on the floor. The prototype taught us to separate emotional context from operational cueing. That is the kind of learning a software build would have obscured until it became an embarrassing live-service problem.

Use `wizard-of-oz` when the frontstage matters more than the code

The `wizard-of-oz` method is where service prototyping gets interesting. ASD defines it clearly: the user sees a believable frontstage while hidden humans run the manual

process behind it. The method's gate is load-bearing. A Wizard-of-Oz plan without a fallback script fails. Good. If the hidden operator has no defined response when the customer behaves unexpectedly, the prototype is testing improvisation rather than the service design.

I use Wizard-of-Oz when I need to test whether the promise and pacing of a service feel credible before building the heavy machinery underneath. Ember was a perfect fit. We could build a lightweight booking flow and a staff tablet mock quickly. The real uncertainty sat backstage. Could a hidden operator review occasion notes, dietary nuance, group size, surprise constraints, and timing windows fast enough to produce a useful celebration brief? Which bookings really needed human review? Which could route cleanly with rules alone? Those are manual-operability questions. Wizard-of-Oz is built for them.

wizard-backstage.yml

```
prototype_id: ember-woz-01
frontstage:
  - booking flow with occasion, dietary, and group prompts
  - confirmation screen promising follow-up only when coordination is required
backstage:
  - hidden operator reviews each booking in a shared queue
  - operator assigns one of: no action, staff brief only, concierge follow-up
  - operator writes celebration brief and routes it to the site tablet
fallbacks:
  unexpected_surprise_request: "A team member will confirm whether we can support th
  unsupported_allergen_complexity: "We need a manager to review this request before
  deposit_confusion: "We will send the exact payment and timing terms in writing tod
metrics:
  - guest completion rate
  - manual minutes per booking
  - staff trust score in the brief
  - promise-kept rate at table
```

The phrase I use with teams is agent backstage. The customer does not need to know which parts are manual and which parts are partially automated. The operators do. ASD is excellent here because I can let lightweight agents parse the incoming request, normalize it, and tee up a recommendation for the hidden human without

pretending the system is already ready for production autonomy. That hybrid mode matters. A pure manual Wizard-of-Oz test can become too slow for anything beyond a handful of sessions. An agent-assisted backstage lets the prototype stretch into a more realistic volume while preserving human judgment on safety, discretion, or commercial exceptions.

There is one ethical boundary I keep tight. I will fake automation. I will not fake capability in safety-critical scenarios. Ember's Wizard-of-Oz path excluded highly sensitive occasions, severe multi-allergen cases, and anything where a partial backstage script could create harm or false reassurance. The prototype must respect the domain. Hospitality feels light compared with healthcare or banking, yet it still contains safety, privacy, and consent edges that deserve caution.

> **CORE PRINCIPLE:** A Wizard-of-Oz test should expose backstage burden, not conceal it. If the hidden team is drowning, the future system will drown in the same place unless the service model changes.

That is exactly what happened at Ember. The first live Wizard-of-Oz rounds showed that a universal concierge model would be absurdly expensive. Hidden operators could manage the queue for a while, but manual effort spiked on a small cluster of complex bookings: large groups, layered dietary notes, surprise choreography, and high-value tables. The winning insight was not "build more automation." The winning insight was that only a minority of bookings needed human concierge work at all. The backstage service model changed before the software did.

> **CAUTION:** Never judge a Wizard-of-Oz prototype by guest delight alone. A frontstage that charms guests while burning out the hidden operators has not passed. It has exposed a design debt.

Instrument the pilot like a service, not a product demo

Once a concept survives storyboarding, walkthroughs, roleplay, and Wizard-of-Oz, I still do not call it validated. I call it ready for a bounded live trial. This is where teams often relapse into startup folklore. They say things like, “Let’s launch a beta and see what happens.” I refuse that language. A service pilot needs entry criteria, exit criteria, and explicit measures across both customer value and operational load. Otherwise the organization learns the wrong lesson from a lucky week.

For Ember, the bounded concierge pilot focused on one service proposition: **Celebration Concierge**. Structured capture at booking. Rules that route only risky bookings to a human concierge. One concise celebration brief visible to the right staff at the right time. That is specific enough to pilot. We then instrumented the live run around four families of evidence: guest completion and confidence, staff scramble minutes, promise-kept performance on dietary and surprise details, and attach revenue on celebratory covers. The goal was not to prove the concept was perfect. The goal was to learn whether the service economics and execution standards could hold across multiple restaurants.

pilot-test-report.json

```
{
  "prototype_id": "ember-concierge-pilot",
  "cohort_size": 84,
  "sites": 12,
  "metrics": {
    "special_occasion_nps": 54,
    "staff_scramble_minutes_per_table": 6.2,
    "attach_revenue_per_cover_eur": 10.7,
    "dietary_promise_kept_rate": 0.96
  },
  "guardrails": {
    "manual_minutes_per_routed_booking": 8.4,
    "recovery_comps_per_100_bookings": 5,
    "unsupported_request_escalation_within_hours": 4
  },
```

```
    "decision": "stage for rollout beta"
  }
```

I also instrument the decision boundary. If routed manual work exceeds a defined threshold, the model fails. If staff distrust the brief and rebuild it from scratch at the site, the model fails. If the guest experience improves but the margin story collapses, the model fails. A pilot is supposed to be demanding. Its purpose is to discover where the service cannot scale yet.

ASD's measurement stack is useful in this phase because I can keep the evidence tied to the method artifacts. The prototype plan points to the learning question. The daily notes and structured observations become evidence. The `pilot-test-report` and any follow-on decision matrix cite those inputs explicitly. That traceability prevents a familiar executive move: treating the happy anecdotes as proof while ignoring the cost curve and the hidden work that made those anecdotes possible.

Ember Hospitality: from vague occasion notes to a bounded concierge service

Ember is a fictional composite, but the operational pattern is very real. Occasion dining creates outsized emotional stakes with relatively small windows for recovery. A missed surprise, a dropped dietary note, or a broken deposit promise can erase trust faster than it does in an ordinary booking. That is why I like this case for teaching service prototyping. The difference between a screen feature and a whole service becomes visible fast.

We began with weak signal capture. Only a minority of celebratory visits were identifiable in structured form at booking. Most arrived as vague notes such as "birthday maybe cake." Floor staff carried a huge share of the recovery burden because they reconstructed the service manually at the site. During discovery, the team found five structural failures: weak signal capture, backstage fragmentation,

promise failure, frontline improvisation cost, and no testable operating model. Those failures became the hypothesis bank for the prototype spectrum.

The first two weeks were cheap and revealing. Storyboards and bodystorming showed that public celebration behavior was culturally brittle and privacy-sensitive. Desktop walkthroughs exposed the fragmentation between occasion notes, dietary details, and deposit handling. We rewrote the brief around a single service object rather than three disconnected note systems. By week three, the service was coherent enough to fake.

Weeks four and five ran the Wizard-of-Oz test across six restaurants. Guests used a believable booking flow. Hidden operators reviewed each request, triaged risk, and routed a celebration brief to the site. Every evening ended with a same-day debrief. We watched completion rates, guest confusion, staff effort, and manual minutes per routed booking. The biggest learning was economic. A universal concierge service was overkill. The high-value operational move was selective routing: only large groups, multi-part dietary notes, surprise coordination, and higher-spend bookings needed the human layer consistently.

PROTOTYPE STAGE	KEY QUESTION	WHAT FAILED	WHAT SURVIVED
Storyboard and bodystorming	Will guests and staff want explicit occasion handling?	Broad visibility of sensitive occasions	Discrete prompts and role-specific cues
Desktop walkthrough	Can the service object travel across booking, host, kitchen, and floor?	Fragmented notes and duplicate promises	One celebration brief with clear ownership
Wizard-of-Oz	How much manual work does the promise require?	Universal concierge load	Selective routing to a hidden operator
Concierge pilot	Do the economics and standards hold live?	Some sites lacked staffing cover on peak nights	Improved NPS, reduced scramble, stronger attach revenue

Week six turned that learning into a bounded live pilot across twelve restaurants and eighty-four guests. The resulting service, Celebration Concierge, proved its worth precisely because it was narrow. Structured capture became mandatory for occasion, dietary, and group contexts. Rule-based routing decided which bookings required concierge intervention. The celebration brief respected discretion: hosts, managers, and kitchen saw different slices of the same service object depending on what they needed to act on. The metrics moved in the right direction because the service design had been pressure-tested before software scale entered the conversation.

The final decision was not “ship the prototype.” It was “fund the right service model.” That difference matters. The pilot validated a rollout-beta path with explicit staffing assumptions, rule thresholds, and measurement. It also preserved several manual edges intentionally because the team now knew which parts of the experience deserved human judgment. Prototyping did not simply reduce risk. It improved strategy.

That is the deeper lesson of service prototyping. I am not trying to mimic production perfectly. I am trying to reveal where the service promise depends on human discretion, where the backstage model cannot carry the frontstage script, and where the business case only works after the scope is narrowed intelligently. The more invisible the service, the more valuable this discipline becomes.

So I keep the sequence tight. Storyboard the promise. Walk the logic. Roleplay the handoffs. Fake the automation. Measure the bounded live run. Then decide what deserves real build investment. Teams that skip these steps do not move faster. They simply postpone the moment when reality argues back.

Structured Ideation & How Might We at Machine Speed

Transformative Services, Behavior Change & Recovery Engineering

NEXT CHAPTER

Transformative Services, Behavior Change & Recovery Engineering

Designing for change and for failure

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03

CHAPTER 04

Service-Dominant Logic & Value Co-Creation in the Age of AI

The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 15

Transformative Services, Behavior Change & Recovery Engineering

Designing for change and for failure

Reading time

14 min

Transformative services start in daily life, not in the app

When a service claims to change behavior, I ask one blunt question: where does the real work happen? If the honest answer is “inside the user’s life,” then the design job changes immediately. I stop optimizing clicks and start examining routines, friction, support systems, shame triggers, energy levels, and the practical resources a person can bring to the service. A member trying to improve metabolic health at 02:00 during a warehouse break is dealing with a very different service reality from the one implied by a polished onboarding deck.

That is why transformative service design pushes me beyond narrow engagement metrics. Sprout's old programme looked active on the dashboard. Articles were opened. Reminders were sent. Meals were logged. None of that proved the service was expanding the member's actual capability to make healthier choices under the conditions of ordinary life. In practice, the old design extracted labor from members before it returned enough value. That is the clearest sign of a weak transformative service. It confuses compliance with progress.

I prefer a different framing. The member is the hero. The service is a mentor with tools, prompts, scripts, and human support. The point is not to keep the member engaged with the product forever. The point is to help the member build enough capability that the product can fade into the background when appropriate. That means my design lens shifts from content delivery to value-in-use. What does the member need in the specific context of shift work, childcare, low energy, or health anxiety? What kind of support can actually fit there?

OLD BEHAVIOR-CHANGE MISTAKE	WHAT IT SOUNDS LIKE	TRANSFORMATIVE REDESIGN MOVE
Product as hero	"Use the programme more and outcomes will follow"	Member states the reason for change in their own words
Co-production mistaken for value	"Fill in more forms so we can help you"	Strip busywork and support the moments where action is hard
One default rhythm for everyone	"Check in every morning at 07:00"	Design around the user's actual schedule and constraints
Relapse treated as user failure	"You broke your streak"	Build recovery as an expected branch of the service

> **CORE PRINCIPLE:** A transformative service should increase the user's real options in daily life. If the service only increases reporting effort, it is subtracting value.

Sprout Wellness is the clean example. The programme targeted metabolic-risk members through employers and referrals. The baseline told an uncomfortable story:

low enrolment from eligible lists, weak week-12 completion, and very poor recovery after the first missed check-in. The data did not point to a motivation deficit alone. It pointed to unstable contexts: night shifts, parenting load, post-screening anxiety, confusing handoffs between app, coach, employer, and clinician. Once I looked through that lens, the redesign problem stopped being “How do we get them to engage more?” and became “How do we help them keep moving when real life breaks the ideal plan?”

That framing also changes the language I tolerate. I do not accept segmentation labels such as “non-compliant members.” I treat them as design failures disguised as user traits. If a service punishes ordinary human disruption, then the service is brittle. Sprout improved as soon as the team named the problem honestly: relapse-prone context, not deficient users.

> **My take:** The phrase “behavior change” often smuggles in a fantasy of total control. I am designing for support, timing, and recovery. Human beings still own the decision to act. Good services respect that.

Design the behavior architecture at Sprout, including the relapse path

Once the daily ecology is clear, I need an intervention architecture. That is where behavior techniques become useful, but only if they are wired into the service rather than sprinkled as motivational flavor. At Sprout, the team mapped touchpoints to explicit behavior-change techniques, then reviewed the map with clinical guardrails in place. ASD makes that operationally manageable. Claude Code drafted the first pass. Codex challenged the labels, the claims, and any copy that sounded like diagnosis or coercion. Humans kept the boundary between coaching and medicine intact.

bct-touchpoint-map.json

```
{
  "programme": "steady-habits-v2",
  "touchpoints": [
    {
      "week": 1,
      "touchpoint": "goal-setting session",
      "bct": "goal setting (behavior)",
      "mechanism": "translate personal reason into one minimum viable habit and one
      "required_human_gate": "coach"
    },
    {
      "week": 4,
      "touchpoint": "first lapse recovery",
      "bct": "problem solving",
      "mechanism": "offer a smaller restart action within 72 hours",
      "required_human_gate": "coach or clinician if risk flag present"
    },
    {
      "week": 8,
      "touchpoint": "high-risk moment rehearsal",
      "bct": "action planning",
      "mechanism": "name the barrier before it happens and script the fallback",
      "required_human_gate": "coach"
    }
  ]
}
```

The key design move at Sprout was refusing to build the programme around a streak. Streaks are seductive because they are legible, easy to instrument, and emotionally punchy. They are also dangerous in a high-friction context. For a member already dealing with shame or unstable routines, the broken streak can become the real event. The service stops supporting change and starts narrating failure. I replaced that logic with a dual-goal model: one minimum viable habit and one stretch habit. The minimum viable habit preserved identity and continuity when life went sideways. The stretch habit preserved ambition when the member had more capacity.

Then I designed relapse as a planned branch, not an exception. The old service escalated reminders when a check-in was missed. The redesigned service changed tone and action. It detected the lapse, paused the assumption of normal cadence, and offered a smaller restart within seventy-two hours. Coaches used scripts that

asked for context before advice. If a member disclosed a symptom, concerning reading, or anything that crossed the coaching boundary, the system followed a pause-and-refer path. That rule is simple. It matters. Transformative service design becomes reckless the moment automation starts imitating clinical judgment it does not hold.

SERVICE MOMENT	OLD SPROUT PATTERN	REDESIGNED PATTERN	WHAT CHANGED OPERATIONALLY
Invite	Generic wellness challenge copy	Member-centered reason for change and plain-language invitation	Enrollment improved because the service sounded relevant
Week 1 setup	Long intake plus lesson dump	One minimum viable habit, one stretch habit, coach framing	Lower onboarding burden and clearer action
Mid-programme lapse	Louder reminders and shame cues	Smaller restart goal plus contextual problem solving	Recovery within 7 days increased sharply
Safety edge	Coach improvisation	Pause-and-refer branch with named clinical review	Claims and risk handling stayed bounded

The operational effects were substantial because the redesign respected how people actually live. Enrollment rose because the invite connected to lived reasons, not abstract health goals. Week-12 completion improved because the service reduced cognitive burden early. Recovery improved because members were offered a route back that preserved dignity. Employers finally got a better answer to their ROI question because the service was built around meaningful progression instead of vanity engagement.

I also want to stress the role of coaches here. AI helped with mapping, script drafting, simulation, and weekly reporting. It did not eliminate the need for human support. In a transformative service, human judgment often becomes more important because the automation can now do the repetitive work and surface the moments where nuance is required. That is a better use of both technology and people.

Recovery is a service in its own right

Now flip the problem. Instead of helping a person build a new capability, imagine the service has already failed. The user has been double-billed, left in outage silence, or bounced across channels while trying to complain. Most organizations still treat recovery as an administrative afterthought. I treat it as a designed experience with its own journey, standards, scripts, and economics. Once I do that, the same service-design instincts return: map the lived pain, identify the trigger conditions, write the response path, and link the recovery moment back into the system that caused it.

Aria Telecom made this unavoidable. The service estate was fragmented across billing, network operations, retail, app support, reseller care, and regulatory handoff. Subscribers experienced the seams. Complaint handling behaved like six small services pretending to be one. The baseline numbers were ugly: weak first-contact resolution, long closure times, high reopen rates, and a loyalty penalty after complaint. The recovery process itself was becoming a second failure.

I frame recovery around justice because queue metrics alone are too shallow. Procedural justice asks whether the process feels navigable and sane. Interactional justice asks whether the user is treated with respect and honest ownership. Distributive justice asks whether the outcome feels fair relative to the harm. Once those dimensions are explicit, the design work becomes clearer. A subscriber who repeats the same story across three channels is suffering a procedural failure. An apology with no action owner is an interactional failure. Inconsistent compensation for the same billing defect is a distributive failure.

> **CORE PRINCIPLE:** Recovery is complete only when the user knows what happens next, why it is fair, and whether the fix actually held.

This is where ASD's `service-recovery-design` method becomes useful. The method expects an evidence set of complaint transcripts or failure records, clusters the recurring scenarios, and outputs a `recovery-playbook`. The acceptance rules require

each node to carry a non-empty trigger, a response that is not a placeholder, and at least one evidence reference. I like this because it blocks a very common executive fantasy: writing generic apology copy and calling it recovery design. Recovery design begins with recurring failure scenarios and the conditions that should activate the response.

recovery-playbook.yml

```
playbook: aria-recovery-v1
scenarios:
  - id: duplicate-bill-shock
    trigger: "billing ledger shows duplicate charge and customer contact is confirme
    response: "freeze collection risk, acknowledge the duplicate charge plainly, iss
    evidence:
      - complaint_cluster_billing_04
  - id: outage-silence
    trigger: "network incident exceeds the status heartbeat threshold without custom
    response: "send proactive progress update, give next confirmed check-in window,
    evidence:
      - complaint_cluster_outage_02
  - id: reseller-ping-pong
    trigger: "customer has been redirected between Aria and reseller more than once
    response: "assign named owner inside Aria, own the next step, and resolve channe
    evidence:
      - complaint_cluster_reseller_03
```

Notice what is doing the work there. The trigger. Without triggers, recovery playbooks drift into nice-sounding guidance that nobody can execute consistently. With triggers, the playbook becomes operational. A certain pattern in the complaint record activates a named response. That response can still involve human judgment, especially on compensation or vulnerability. It is no longer vague.

> **My take:** An apology is not a recovery plan. I want action ownership, a due-by time, a fairness rationale, and proof that the issue held after the first fix. Anything less is public relations.

Aria Telecom: use `service-recovery-design` to engineer trust after failure

The Aria redesign began with complaint mining. The full complaint archive gave us the scale. Case listening and review gave us the texture. ASD made the handoff clean: the evidence set could support clustering, scenario definition, and playbook authoring instead of leaving the team buried in anecdotes. We identified recurring failure scenarios that mattered both operationally and emotionally: duplicate bill shock, outage silence, reseller ping-pong, vulnerable-customer dead ends, and status opacity during investigation.

From there, `service-recovery-design` imposed welcome discipline. Each recovery scenario needed a trigger, a response, and evidence. That sounds obvious. Most organizations do not do it. They have policy matrices, scripts, and compensation rules scattered across teams, then wonder why the customer experience feels random. The method forced the pieces into one executable shape. Once the clustered scenarios existed, the team could design consistent first moves, escalation conditions, and follow-through expectations.

The most important service rule at Aria was simple: fix first, then compensate, then prove the fix held. Compensation without correction produced false closure. Correction without communication felt cold. Communication without ownership felt manipulative. The redesigned playbooks therefore tied operational action to copy and to evidence. A billing case acknowledged the probable duplicate charge plainly, froze collection risk, stated the due-by time, and only then moved into compensation logic once the ledger issue was confirmed. An outage case required proactive status heartbeats before the subscriber needed to chase. A reseller ping-pong case forced Aria to internalize the channel boundary rather than making the subscriber pay for it again.

SCENARIO	JUSTICE GAP	TRIGGER	REQUIRED RECOVERY MOVE	MEASURE THAT PROVES IT IMPROVED
Duplicate bill shock	Distributive + interactional	Verified duplicate charge pattern	Freeze risk, plain-language ownership, correction before compensation	Reopen rate and fairness score
Outage silence	Procedural	Elapsed time without status update during active incident	Proactive heartbeat with next check-in window	Status-chasing contacts and CES
Reseller ping-pong	Procedural + interactional	More than one redirect between parties	Named owner inside Aria takes control of next step	Named-owner-in-1-day and escalations
Vulnerable-customer dead end	All three	Vulnerability flag plus unresolved case age threshold	Assisted path with tighter handoff limit and higher compensation authority	Handoffs per case and closure time

Aria also proved why empowerment belongs inside service design, not after it. Frontline teams were given bounded authority to apply compensation within clear bands on approved scenarios. That is an organizational design decision disguised as an operational one. Without it, the service still bottlenecks on supervisors and the interactional damage persists. The playbook, the copy system, the compensation matrix, and the root-cause loop have to align. Otherwise the agent says the right thing while the organization does the slow old thing underneath.

The root-cause loop is where recovery engineering becomes strategic rather than merely compassionate. Every resolved complaint fed back into the recurrence view. Weekly Pareto reviews sent the largest failure clusters back to billing, network, or reseller operations. That closes the loop between individual recovery and system repair. If complaint handling stops at appeasement, the service remains broken. If it routes the evidence back into the production system, the service learns.

> **CAUTION:** Do not let a recovery programme optimize closure time alone. A complaint can close quickly and still feel unfair, unresolved, or likely to recur. That is cosmetic efficiency.

Sprout and Aria are the same design challenge seen from opposite ends

Sprout and Aria look like different worlds. One is preventive health coaching. The other is telecom complaint handling. I group them together because both forced the service to deal honestly with disruption. Sprout had to support people through relapse without turning the service punitive. Aria had to support customers through service failure without turning the recovery path into a second insult. In both cases, the design quality lived in the branch conditions, the scripts, the timing, and the human authority structures.

The numbers tell the story quickly. Sprout improved enrollment, week-12 completion, and recovery after the first lapse because the service was redesigned around capability and dignity. Aria improved first-contact resolution, closure time, and repeat-complaint rates because the recovery path finally had triggers, ownership, and justice built into it. Neither outcome came from clever interface polish alone. Both came from treating the service as an operating system with explicit methods, evidence, and gates.

I take a practical lesson from that. If a service promises human change, I design the relapse path early. If a service will inevitably fail sometimes, I design the recovery path early. Waiting until after launch to discover how the system handles disruption is one of the most expensive habits in service design. ASD gives me a better pattern. Use evidence to define the scenarios. Use methods such as `service-recovery-design` to force triggers and responses into the artifact. Use critique to keep weak or unsafe branches from sneaking through. Then measure whether the service actually helps a person recover motion.

That last phrase matters to me. Recover motion. In Sprout, motion meant helping the member restart a habit after an ordinary life disruption. In Aria, motion meant helping the subscriber regain trust and progress after the service had stalled. I want my services to be good at that moment because that is where people decide whether the system is on their side.

Behavior change and recovery engineering therefore belong together. They both reject the fantasy of a frictionless human. They both assume context will bite. They both reward services that respond with clarity, proportionality, and practical next steps. When I design them well, the user feels supported rather than judged, and the organization learns where its own operating model still creates avoidable harm. That is the kind of design leverage I care about.

CHAPTER 14

Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 16

Service Quality Management & CX Maturity Governance

NEXT CHAPTER

Service Quality Management & CX Maturity Governance

From projects to permanent capability

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 16

Service Quality Management & CX Maturity Governance

From projects to permanent capability

Reading time

12 min

Build quality loops that actually close

Service quality decays for a simple reason. Most organizations launch changes as projects and manage operations as if the service will hold itself together afterward. It will not. Scripts drift. Workarounds multiply. One team optimizes handle time while another promises clarity and ownership. The customer experiences the combined effect and has no interest in your org chart. That is why I think in loops, not rollouts.

A quality loop has five moving parts. First, a clear service standard. Second, evidence showing whether the standard held. Third, a review rhythm that names root causes instead of admiring top-line averages. Fourth, a decision path that can change the

service, not merely describe it. Fifth, a check that proves the change did what it claimed. If any one of those parts is weak, the loop stays open and quality drift wins.

Kestrel Energy is the useful example because the organization had already improved some journeys in an earlier phase. The unresolved problem was operational entropy. Billing queries, connection appointments, and SME service moments still varied too much by team and month. Avoidable failure demand remained high because the organization had redesign wins without a standing machinery for defending them. That is a classic service-design transition point. The work stops being about brilliant artifact production and becomes about operating discipline.

LOOP STAGE	QUESTION I ASK	TYPICAL ASD ARTIFACT OR TOOL	NAMED OWNER
Standard	What promise are we actually making?	Service standard set, blueprint, metric definition	Journey owner or service owner
Observe	Did the promise hold in live use?	Evidence register, telemetry extracts, surveys, support logs	Measurer and operations lead
Audit	Where did quality slip and why?	`good-services-audit`, insight set, traceability matrix	Researcher plus critic gate
Decide	What changes, who funds it, and by when?	Governance board, decision matrix, report pack	Human council or named executive sponsor
Verify	Did the fix hold and reduce failure demand?	`sd_factcheck`, rerun audit, updated report	Presenter, measurer, critic

I call this a closing loop because many organizations only do the middle three steps badly. They gather evidence. They review it. They talk. Then nothing binding happens. Or a change happens and nobody checks whether it truly improved the service. A real loop closes only when the service standard or operating model changes and the next cycle confirms the effect. That is why I keep the word governance attached to quality. Without governance, the loop is a dashboard ritual.

> **CORE PRINCIPLE:** Quality management is not the act of measuring decline. It is the act of creating enough authority and evidence to stop the decline from repeating.

Kestrel made this concrete with touchpoint-level standards and a monthly operating rhythm. A bill could be technically correct and still fail functionally if the explanation was opaque. A connection appointment could be technically scheduled and still feel broken if the promise window was vague or unkept. The quality loop therefore had to combine technical and functional signals. That is why Kestrel used a touchpoint MOS model, reviewed avoidable failure demand, and tied the resulting issues back to named owners rather than generic “customer experience” goals.

> **My take:** Most quality programmes die because they confuse visibility with control. A dashboard gives visibility. Decision rights give control. I want both in the same room.

Run `good-services-audit` as a recurring factcheck job

The `good-services-audit` method is useful precisely because it refuses vibes. It expects a live or shipped blueprint or journey map plus at least two evidence inputs such as support logs, user feedback, or post-launch telemetry. It returns an `insight-set`. The researcher scores the service against the checklist. The Critic rejects any below-target score that lacks cited evidence. That rule matters. A quality audit without evidence becomes a brand exercise in disguise.

I often describe this as a factcheck job because the audit should not stand alone. In ASD the clean pattern is: run the audit method, write the resulting insight set, then run `sd_factcheck` so integrity, acceptance, and review are replayed before the insight enters the governance pack. If the evidence references do not resolve, if the artifact fails acceptance, or if the Critic verdict is missing, the pipeline fails closed. I want that. A weak audit should break the meeting packet rather than quietly poisoning it.

quality-audit-cycle.sh

```
PID="kestrel-quality-2026"

asd method-run --projectId "$PID" --methodId good-services-audit
# write the audit insight-set through asd artifact-write
# record critic review through asd review
asd factcheck --projectId "$PID"

asd export-report --projectId "$PID" --kind traceability-matrix
asd export-report --projectId "$PID" --kind method-log
```

Notice the order. I do not export the board report first and pray the underlying work is sound. I force the soundness check before the governance surface exists. That sequence is a surprisingly big cultural shift. In many organizations, reporting is a storytelling layer floating above inconsistent source material. In ASD, reporting is downstream of the typed artifacts and their gates. That reduces performative confidence because the system will happily tell you the work is still draft if the audit was weak.

Kestrel benefited from this discipline because the monthly review pack became harder to game. The team could no longer say, “billing feels better this month,” while hiding unresolved touchpoint failures underneath. The audit insight set named where standards slipped, which evidence supported the score, and which owner needed to respond. The follow-on factcheck forced the evidence chain and review state to be complete before the issue reached leadership. That is what I mean by a factcheck job. It does not replace judgment. It forces honesty before judgment happens.

I also like the side effect on organizational behavior. Once teams know the audit will surface below-target scores only when evidence exists, the conversation shifts. People stop arguing abstractly about whether the service is “good.” They argue over the root cause and the fix. That is where design and operations should meet.

> **CAUTION:** Do not run audits as theatre after the real decisions are already made. If the board cannot still change funding, standards, or owner assignments after the audit, you

> built reporting without governance.

CX maturity needs governance boards, not inspirational language

Quality loops work inside a service. Maturity governance works across the enterprise. Titan Manufacturing needed the second problem solved. The organization had five business units, multiple portals, fragmented service promises, and no forum with the authority to resolve cross-unit customer issues. Customers buying from more than one unit felt the seams immediately: multiple logins, unclear ownership, duplicate escalation paths, and wildly different standards across journeys that should have felt connected.

This is where `cx-maturity-assessment` becomes useful. The method takes process, tooling, and governance evidence and writes a `maturity-assessment` artifact. Its gate is evidence-cited scoring. No support, no score. That is exactly right for executive work because maturity claims are notoriously easy to inflate. Titan had many customer-related activities already. Activity did not equal capability. The assessment forced the enterprise to distinguish between isolated effort and repeatable operating discipline.

Titan used the six-discipline model as the frame: strategy, customer understanding, design, measurement, governance, and culture. The most important governance idea was the limiting-discipline rule. A company does not mature to the level of its most polished slide deck. It matures to the level of its weakest discipline that still blocks the customer from experiencing coherence. Titan's bottlenecks were measurement and governance. Once that became explicit, the operating-model decision was clearer.

GOVERNANCE QUESTION	WEAK ANSWER	STRONGER ANSWER IN A FEDERATED MODEL
Who owns a cross-BU journey?	"Everyone contributes"	One named journey owner with explicit escalation rights

GOVERNANCE QUESTION	WEAK ANSWER	STRONGER ANSWER IN A FEDERATED MODEL
Who sets measurement definitions?	Each unit chooses locally	Central measurement discipline defines shared metrics and evidence rules
Who can block a customer-impacting change?	No one formally	Council or standards board with review-gate authority
How do local teams keep agency?	They resist centralization informally	Delivery stays in business units while standards and review gates are shared

I almost always land on a federated model in complex enterprises. A heavy central office becomes a corporate inspector and triggers resistance. Pure decentralization preserves local autonomy and institutionalizes customer inconsistency. The federated model is a better service design move. A small central capability owns standards, definitions, review gates, and the governance rhythm. Business units keep delivery ownership and local expertise. Cross-unit journeys get named owners. Councils have authority over trade-offs that customers would otherwise feel as fragmentation.

Titan improved because the governance model gained teeth. The executive council was sponsored by the COO, not tucked under a digital transformation side programme. A journey owner forum handled cross-unit issues before they became chronic customer pain. Standards changes, owner registries, and measurement definitions lived in one reviewable repository. The maturity discussion stopped being motivational. It became operational.

My take: A governance board without the power to change funding, block releases, or reassign ownership is a support group. I am interested in governance only when it can alter reality.

Use the six report kinds to keep leadership on one truth

Governance breaks down fast when every audience gets a different version of the truth. One team has a workshop mural. Another has a PowerPoint summary. Finance has a spreadsheet. Operations has a dashboard. Nobody can trace the disagreement back to the source artifact easily. ASD fixes that by exporting deterministic reports from the same project inputs. There are six report kinds. I keep all six in play because each serves a distinct governance job.

REPORT KIND	PRIMARY AUDIENCE	WHY I USE IT
`engagement`	Full steering group	Gives the complete engagement record: summary, evidence base, artifacts, traceability, and method log
`executive-summary`	Decision-makers under time pressure	Keeps the argument short without severing it from the artifact index and next steps
`research-findings`	Research, design, and operations leads	Focuses attention on findings without burying them in delivery detail
`artifact-dossier`	Audit, design assurance, and reviewers	Resolves each artifact with its linked evidence so claims can be challenged cleanly
`traceability-matrix`	Governance, QA, and regulated domains	Shows provenance from evidence to artifact to decision
`method-log`	Process owners and practice leads	Makes the method sequence and gates inspectable instead of anecdotal

These reports are not hand-composed summaries. They are deterministic exports. That matters. If I re-export the same project inputs, I should get the same argument, apart from the controlled `generatedAt` field where applicable. This stability is a governance feature. It means the council is not debating whose deck is newest. It is debating the underlying evidence and artifact state.

governance-report-pack.sh

```
PID="titan-cx-governance-2026"

asd export-report --projectId "$PID" --kind engagement
asd export-report --projectId "$PID" --kind executive-summary
asd export-report --projectId "$PID" --kind research-findings
asd export-report --projectId "$PID" --kind artifact-dossier
asd export-report --projectId "$PID" --kind traceability-matrix
asd export-report --projectId "$PID" --kind method-log
```

At Kestrel, the `traceability-matrix` and `method-log` were especially helpful because they made the quality programme feel less like CX rhetoric and more like an auditable operating cadence. At Titan, the `executive-summary` and `artifact-dossier` did heavy lifting. The executive group needed a concise decision surface. The operating teams and reviewers needed the deeper chain from maturity evidence to governance choices. One project could support both because the exports came from the same truth set.

I also like what this does to meeting behavior. Once the board knows a dossier can be opened to inspect the resolved evidence behind a claim, casual hand-waving declines. People either bring stronger arguments or they ask for better evidence. Both outcomes improve governance.

There is another governance benefit hiding here. The report pack makes human approval boundaries clearer. Agents can assemble the evidence, render the dossier, and highlight the unresolved gaps. They do not get to declare the quality issue closed on their own. A council chair, journey owner, or named executive still has to decide whether a standard changes, whether funding moves, or whether a release pauses. I want that asymmetry preserved. Automation accelerates the preparation of judgment. It does not replace accountable judgment in a mature service organization.

That is why I pair the six reports deliberately. The `executive-summary` tells leaders what decision is needed now. The `artifact-dossier` gives reviewers the supporting body. The `traceability-matrix` lets audit-minded stakeholders inspect the chain from evidence to recommendation. The `method-log` shows whether the team actually

followed the promised process. Together they keep governance from collapsing into rhetoric or tunnel vision.

Kestrel and Titan show the two levels of permanent capability

Kestrel Energy and Titan Manufacturing taught related lessons at different altitudes. Kestrel was the service-level problem. The organization needed a standing service quality management loop for billing, connections, and SME onboarding. Titan was the enterprise-level problem. The company needed cross-unit governance so customer journeys could survive its own internal fragmentation.

Kestrel's work focused on quality as an operating rhythm. The standards library, touchpoint MOS model, avoidable failure-demand view, and recurring audits turned quality from a lagging complaint conversation into a structured management system. The best part of the case is how concrete the loop became. Below-target scores triggered audit insights. Audit insights entered factchecked report packs. Those report packs drove standards changes and owner actions. The next cycle verified whether failure demand and promise-keeping actually improved. That is a loop closed in the only way that matters: the service behavior changed.

Titan's work focused on maturity as a political and structural problem. The company already had customer-facing activity. It lacked common definitions, journey ownership, review-gate authority, and a shared measurement architecture. The maturity assessment made the weakness visible. The federated model gave the enterprise a way to respond without destroying local expertise. The governance forums then turned the model into a lived operating system. Cross-BU ownership improved because the authority became real, not advisory. Portal login improved because one lighthouse journey proved the model before scope ballooned into a full-platform fantasy.

Put differently: Kestrel answered, “How do we keep a service honest every month?” Titan answered, “How do we keep an enterprise from producing five conflicting services at once?” ASD helped in both cases because it linked methods, artifacts, evidence, reports, and gates into one contract. Agents could draft, aggregate, or challenge. Humans still made the approval decisions. That boundary is healthy. Quality governance should be evidence-rich and automation-assisted. It should never become self-authorizing.

I end this chapter with a practical stance. If your service work stops at launch, you are building decay into the system. If your governance model cannot change funding, ownership, or standards, you are documenting decline elegantly. Permanent capability comes from loops that close, audits that can fail closed, maturity models grounded in evidence, and report surfaces that keep every audience on one truth. That is harder than shipping a redesign. It is also where the real leverage lives.

CHAPTER 15

Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 17

The ASD MCP Server

NEXT CHAPTER

The ASD MCP Server

Thirty tools, 228 resources, ten prompts — over stdio

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 17

The ASD MCP Server

Thirty tools, 228 resources, ten prompts — over stdio

Reading time

20 min

Why MCP for Service Design

Service design work breaks when the method contract changes every time the host changes. I have watched teams make this mistake for years. They build one comfortable plugin for one favorite environment, then quietly fork the behavior for the next environment because the transport feels different. One tool writes evidence one way. Another invents a fourth evidence kind. A third lets an artifact bypass review because the UI is in a hurry. At that point you no longer have a platform. You have folklore.

ASD uses MCP to kill that drift. The reason is simple: the protocol is narrow, boring, and explicit. The server speaks newline-delimited JSON-RPC 2.0 over stdio, pinned to MCP revision `2025-06-18`. A client initializes once, sees the capability buckets, enumerates tools, resources, and prompts, and then calls the same registry the CLI

and host packs already rely on. The transport changes. The runtime contract does not.

That distinction matters more in service design than in ordinary code tooling because our outputs are not ephemeral chat replies. They are evidence registers, typed artifacts, review logs, exports, and orchestration records that must survive handoff. If the Researcher captures a quote in one host, the Mapper must be able to cite the same evidence id from another host. If the Critic vetoes a journey map on one surface, the Presenter must not discover a magically "passed" version somewhere else. MCP gives me the lowest common denominator that still preserves those invariants.

> **CORE PRINCIPLE:** Treat MCP as a transport contract, not as a second implementation. The ASD server is valuable precisely because it exposes the canonical runtime registry unchanged: same 30 tools, same validation rules, same fail-closed write path, same human gate boundary.

I also like MCP for a more practical reason: it is host-neutral without being host-blind. Claude Code can mount the server. OpenCode can mount the server. Codex can mount the server. A generic client can mount the server. None of those clients need to understand the whole ASD product model in advance. They only need to honor the protocol and present the server honestly. That keeps the integration layer thin and the evidence discipline centralized.

There is another benefit that senior teams feel almost immediately: observability. With MCP, discovery is inspectable. I can ask for `tools/list` and see the registry the host actually received. I can ask for `resources/list` and see whether the corpus roots are present. I can ask for `prompts/list` and confirm that the ten role contracts are mounted. That sounds trivial until you have debugged agent stacks where everyone swears the host is "wired correctly" while the mounted surface is missing half the practice.

> **My take:** I trust protocol discovery more than screenshots of a settings panel. If a host can show me `initialize`, `tools/list`, `resources/list`, and one representative workflow against a disposable workspace, I know what is really mounted. Everything else is marketing until proven otherwise.

ASD also uses MCP to keep the knowledge surface distinct from the execution surface. The 30 tools do the work. The 228 corpus resources and 10 prompts provide context. That split matters. A client can read a book chapter from an `asd://` URI, fetch a role contract with `prompts/get`, and still hit the same write gate when it calls `sd_artifact_write`. Knowledge can help the model think better. It still cannot bypass review, evidence integrity, or human phase authorization.

So when you hear "the ASD MCP server," do not picture a thin adapter around a chat product. Picture a protocol face on the same deterministic engine described throughout this book: evidence first, method contract next, typed artifact after that, Critic review, factcheck, and human phase gate at the end. MCP matters because it lets that loop travel.

Booting the Server: Sources and Standalone Repo

You have two honest ways to start the server. The first is the standalone distribution repo, `imehr/asd-mcp`. The second is an emitted pack generated from the source monorepo. Both are valid. They serve different operational needs.

The standalone repo is the cleanest path when you want a disposable MCP server with its own published provenance. The repo is regenerated from the canonical source tree by the distribution publishing script, and the resulting checkout carries a `provenance.json` file that records the source commit. That means you can hand somebody one repository URL and tell them exactly what to boot.

standalone-repo.sh

```
git clone https://github.com/imehr/asd-mcp.git
node asd-mcp/mcp/bin/server.mjs
```

The emitted-pack route is the one I use when I want to prove the server that a host pack would actually receive. Emit packs into a disposable directory, then run the pack-local server entrypoint from there. This keeps the test honest: no global install, no sibling package assumptions, no ambient fallback to some other executable on your machine.

emit-and-run.sh

```
node platform/compiler/emit-packs.mjs --output /tmp/asd-packs
node /tmp/asd-packs/mcp/mcp/bin/server.mjs
```

Those two commands are the load-bearing source story for this chapter. If you remember nothing else, remember this: the emitted pack is the product boundary. The monorepo is the source boundary. I can inspect source code in the monorepo, but the pack is what a client actually mounts.

The pack-local entrypoint rule is equally important. Inside an installed pack, the server entrypoint is always relative to the pack root. That rule keeps configs portable across hosts because the pack carries its own launcher.

asd-mcp.json

```
{
  "mcpServers": {
    "asd": {
      "command": "node",
      "args": ["mcp/bin/server.mjs"]
    }
  }
}
```

That relative path is correct when the host sets its working directory to the installed pack root. If you copy the launcher elsewhere, either preserve that working directory or rewrite the path to an absolute one. The rule is not magical. It is just deterministic.

The runtime project store still lives under `ASD_WORKSPACE`. If you do nothing, the CLI defaults to `~/.asd/projects`. For MCP work I still prefer setting `ASD_WORKSPACE` explicitly so every demo is isolated and reproducible. The corpus and prompt roots are separable too: `ASD_BOOKS_ROOT`, `ASD_ARTICLES_ROOT`, and `ASD_AGENTS_ROOT` determine what the server can enumerate for resources and prompts. That boundary matters when a client sees 30 tools but zero resources: the engine is alive, the corpus roots are not.

> **My take:** I always boot the pack in a disposable workspace the first time. If discovery counts look wrong there, I fix the pack or the roots before I blame the host. It saves hours of bad debugging.

One more practical note. The MCP server is intentionally self-contained. You do not need a global `asd` executable, and you do not need the whole monorepo mounted beside it for the emitted pack path to work. That is exactly why the distribution repo and the emitted bundles exist. They turn the runtime into a portable surface instead of a developer-only convenience.

The Tool Registry: 30 sd_* Tools by Family

The ASD MCP server exposes exactly 30 tools. I prefer to look at them twice: first as a full family inventory so I can see coverage, then as a smaller set of representative tools so I can see argument shape and boundary behavior. The full per-tool schema lives in Appendix B. What matters here is understanding the registry as a service-design operating surface.

FAMILY	TOOLS	COUNT
Project	`sd_project_list`, `sd_project_create`, `sd_project_open`, `sd_project_status_set`	4
Evidence	`sd_evidence_add`, `sd_evidence_import`, `sd_evidence_query`	3
Artifacts	`sd_artifact_list`, `sd_artifact_get`, `sd_artifact_write`	3
Methods	`sd_method_list`, `sd_method_run`	2
Review	`sd_review`, `sd_review_list`	2
Export	`sd_export_report`, `sd_export_artifact`	2
Knowledge	`sd_knowledge`	1
Factcheck	`sd_factcheck`	1
Templates and schema	`sd_template_list`, `sd_template_get`, `sd_schema_get`, `sd_practice_get`	4
Canvas	`sd_canvas_add`, `sd_canvas_list`, `sd_canvas_remove`	3
Orchestration	`sd_orchestration_plan`, `sd_orchestration_status`, `sd_orchestration_dispatch`, `sd_orchestration_handoff`	4
Phase	`sd_phase_advance`	1

That inventory already tells you something important. ASD does not model service design as a pile of clever domain-specific widgets. It models the engagement lifecycle: project state, evidence capture, method selection, artifact persistence, review, export, knowledge lookup, orchestration records, and the human gate. The newest member of the Project family, `sd_project_status_set`, sets an engagement's Kanban lifecycle status — `briefed`, `active`, `gated`, `blocked`, or `archived` — independently of the design phase, and a same-status call is a no-op unless a note accompanies it. That is why the tool surface feels operational instead of theatrical.

REPRESENTATIVE TOOL	CORE ARGUMENTS	WHAT IT RETURNS	BOUNDARY THAT MATTERS
`sd_project_create`	`name`, `client`, `sector`, `service`, optional `phase`	A new `project` object with generated `projectId`	Creates the store root. No project, no engagement state.
`sd_evidence_add`	`projectId`, `source`, `kind`, `body`	An `evidence` record with SHA-256 id	`kind` must be exactly `quote`, `telemetry`, or `document`.
`sd_method_run`	`projectId`, `methodId`	The method contract: steps, acceptance rules, output kind, evidence count	It returns the SOP. It does not do the design work for you.
`sd_artifact_write`	`projectId`, `methodId`, `title`, `nodes`, optional `artifactId`	`accepted`, `status`, `artifact`, `failures`	Failed acceptance still persists a draft. Dangling evidence refs fail closed and do not write.
`sd_review`	`projectId`, `artifactId`, `verdict`, `rows`	Updated artifact status, verdict, review log id, rows written	A pass needs at least one evidence-cited proposal row. A veto needs at least one defect row with a fix.
`sd_factcheck`	`projectId`	Top-level `pass` plus per-artifact integrity, acceptance, and review rows	Exit 1 on failure is a feature. CI is supposed to fail here.
`sd_export_report`	`projectId`, `kind`, optional `generatedAt`	Deterministic report path, kind,	Supported report kinds are exactly `engagement`, `executive-summary`,

REPRESENTATIVE TOOL	CORE ARGUMENTS	WHAT IT RETURNS	BOUNDARY THAT MATTERS
		timestamp, byte count	`research-findings`, `artifact-dossier`, `traceability-matrix`, and `method-log`.
`sd_knowledge`	Optional `query`, `phase`, `source`, `type`	Method and artifact-kind catalogue rows	It queries the knowledge registry. Raw corpus chapters still come from `resources/read`.
`sd_orchestration_handoff`	Role-to-role handoff fields plus optional `artifactId` and `artifactKind`	A durable handoff record	If `artifactKind` is supplied, the matching `artifactId` must resolve in the same project.

The most common misunderstanding I see is around `sd_method_run`. People hear "run" and imagine the server will perform the method. It will not. The tool returns the executable method contract: role-attributed steps, acceptance rules, output kind, review gate, and current evidence sufficiency. That is enough to keep agents aligned without pretending the method itself has become an opaque black box.

`sd_artifact_write` is the second misunderstanding hotspot. In many systems, a failed write vanishes. ASD refuses that pattern. If acceptance fails, the submitted artifact is still retained as a draft with machine-readable failure rows so the repair path is explicit. That gives the Critic and the next writer something concrete to work with. What never gets tolerated is silent corruption. If a node cites an evidence id that does not exist, the write path fails closed before persistence.

`sd_review` is where the platform stops flattering the operator. You cannot wave a draft artifact through with an empty "looks good." A pass must include at least one evidence-cited proposal row. A veto must include a defect row with a fix. That is not bureaucracy. It is what makes the review record reusable instead of decorative.

Finally, notice what the orchestration tools do not claim. `sd_orchestration_plan`, `sd_orchestration_dispatch`, `sd_orchestration_status`, and `sd_orchestration_handoff` persist coordination records. They do not themselves prove that a host launched a fresh process. That boundary will matter again when we get to ACP in §17.7.

Corpus Resources and Role Prompts

The MCP server exposes more than tools. It also mounts the book-and-article corpus and the role prompt registry. I think of those as the reading surface and the collaboration surface.

The reading surface is the resource tree. Current ASD inventory is 228 resources total, paginated at 100 items per page. In practice that means three `resources/list` pages: 100, 100, and 28. The URIs follow a plain address scheme rather than an invented database syntax:

`asd://books/BOOK_SLUG` for a book overview, `asd://books/BOOK_SLUG/CHAPTER_SLUG` for a chapter resource, and `asd://articles/NN` for an article resource. The page cursor is opaque, but the current server uses offset-based tokens under the hood, so you will normally see the first two cursors advance from offset 100 to offset 200. The only rule that matters to a client is that omission returns page one and the final page has no `nextCursor`.

CAPABILITY	WHAT YOU ASK FOR	WHAT YOU GET BACK	WHEN I USE IT
`resources/list`	Optional cursor	Descriptors: URI, name, MIME type, size, version, provenance, optional `nextCursor`	Discovery, pagination, inventory checks
`resources/read`	One exact URI	The UTF-8 payload plus metadata for that resource	Reading one chapter, one article, or one overview exactly

CAPABILITY	WHAT YOU ASK FOR	WHAT YOU GET BACK	WHEN I USE IT
`prompts/list`	No arguments	The ten role prompt names	Mount verification and role discovery
`prompts/get`	One prompt name	The shared role contract plus role-specific instructions	Grounding a generic client in the ASD collaboration model

The ten prompts are the canonical roles you have already seen throughout this book: Orchestrator, Researcher, Synthesizer, Mapper, Ideator, Prototyper, Measurer, Critic, Presenter, and Librarian. A generic MCP client does not need a native subagent feature to benefit from them. It can still fetch the role contract, display it, or pass it into its own orchestration layer. That is why prompt exposure belongs in the MCP server instead of being buried only inside a host plugin.

The neat part is that discovery is honest about absence. If the corpus roots are missing, `resources/list` reflects that. If the agent root is missing, `prompts/list` reflects that. The server does not fabricate a synthetic corpus just to look healthy. I prefer that bluntness. It means a missing resource page is a real configuration problem, not a ghost in the model.

You should also keep the knowledge tool separate in your head. `sd_knowledge` is fantastic for querying the method catalogue or the artifact-kind registry by phase, source, or keyword. It is not the same thing as reading a chapter from the corpus. When I want the method metadata for `journey-from-transcripts` or the provenance for `service-blueprint`, I ask `sd_knowledge`. When I want the actual book chapter text, I ask `resources/read`.

One last boundary is worth stating plainly because old receipts still float around in real repositories: some older receipts still show smaller resource counts. They are not current. The second-edition platform count is 228. If you see an older number, treat it as receipt history, not as the live inventory to print in new material.

Client Configurations: Claude Code, OpenCode, Codex

The good news about client configuration is that it is small. The bad news is that a one-line path mistake can make the whole surface look haunted. Use the pack-local entrypoint rule and keep the config literal. Here are the three host examples I consider canonical.

Claude Code uses the plugin-root placeholder. That matters because the host installs the pack into its own managed location and resolves `${CLAUDE_PLUGIN_ROOT}` at runtime.

claude-code.mcp.json

```
{
  "mcpServers": {
    "asd": {
      "command": "node",
      "args": ["${CLAUDE_PLUGIN_ROOT}/mcp/bin/server.mjs"]
    }
  }
}
```

OpenCode uses a command array. I like this because it makes the local process contract explicit: executable first, arguments second, no shell interpolation games.

opencode.config.json

```
{
  "$schema": "https://opencode.ai/config.json",
  "mcp": {
    "asd": {
      "type": "local",
      "command": ["node", "mcp/bin/server.mjs"]
    }
```

```
    }
  }
```

Codex supports both a registration command and a TOML block. I usually show both because teams often want an imperative install step for onboarding and a declarative block for repository documentation.

codex-mcp-add.sh

```
codex mcp add asd -- node "$PWD/platform/packs/codex/mcp/bin/server.mjs"
```

codex.mcp.toml

```
[mcp_servers.asd]
command = "node"
args = ["mcp/bin/server.mjs"]
```

All three examples obey the same structural rule: `node` launches the pack-local `mcp/bin/server.mjs` entrypoint. That is the point. Host syntax varies. Server identity does not. Once mounted, each host discovers the same 30 tools, the same 228 resources when corpus roots are present, and the same 10 prompts.

What these configs do *not* prove is native host execution quality. They prove mount syntax. That is useful, but it is a smaller claim. Host-specific receipts are the place to prove the full UI path. The MCP config itself only gets you to a correctly mounted server.

A Complete Worked Session

Now for the part that actually persuades me. I do not want an MCP chapter that stops at static configuration. I want one complete session that begins with protocol

initialization, discovers the surface, reads the mounted context, writes real engagement state, and ends at factcheck. The easiest honest way to show that is as a small shell script that speaks JSON-RPC over stdio and parses the responses as it goes.

This example assumes you already emitted the pack to `/tmp/asd-packs`. It initializes the server, lists tools, lists the first resource page, fetches the Mapper prompt, creates a project, captures one quote, reads the method contract, writes a minimal journey artifact with three cited nodes, records a pass review row that cites the same evidence id, and then runs factcheck. Nothing is hidden behind a host UI. That is why this is such a useful diagnostic ritual.

worked-mcp-session.sh

```
export ASD_WORKSPACE=/tmp/asd-ch17-mcp
python3 - <<'PY'
import json
import os
import subprocess

server = subprocess.Popen(
    ["node", "/tmp/asd-packs/mcp/mcp/bin/server.mjs"],
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
    stderr=subprocess.PIPE,
    text=True,
    env={**os.environ, "ASD_WORKSPACE": os.environ["ASD_WORKSPACE"]},
)

def request(payload):
    server.stdin.write(json.dumps(payload) + "\n")
    server.stdin.flush()
    return json.loads(server.stdout.readline())

init = request({
    "jsonrpc": "2.0",
    "id": 1,
    "method": "initialize",
    "params": {
        "protocolVersion": "2025-06-18",
        "capabilities": {},
        "clientInfo": {"name": "chapter-17-demo", "version": "1.0.0"}
```

```
    }
})
print("initialize", init["result"]["serverInfo"])

server.stdin.write(json.dumps({
    "jsonrpc": "2.0",
    "method": "notifications/initialized"
}) + "\n")
server.stdin.flush()

tools = request({"jsonrpc": "2.0", "id": 2, "method": "tools/list"})
print("tools", len(tools["result"]["tools"]))

resources = request({"jsonrpc": "2.0", "id": 3, "method": "resources/list"})
print("resources-page-1", len(resources["result"]["resources"]), resources["result"]

prompt = request({
    "jsonrpc": "2.0",
    "id": 4,
    "method": "prompts/get",
    "params": {"name": "mapper"}
})
print("prompt", prompt["result"]["messages"][0]["role"])

created = request({
    "jsonrpc": "2.0",
    "id": 5,
    "method": "tools/call",
    "params": {
        "name": "sd_project_create",
        "arguments": {
            "name": "Chapter 17 Demo",
            "client": "Fictional composite",
            "sector": "public-transport",
            "service": "disruption recovery",
            "phase": "discover"
        }
    }
})
project_id = created["result"]["structuredContent"]["project"]["projectId"]
print("project", project_id)

captured = request({
    "jsonrpc": "2.0",
    "id": 6,
    "method": "tools/call",
    "params": {
```

```
        "name": "sd_evidence_add",
        "arguments": {
            "projectId": project_id,
            "source": "call-821004",
            "kind": "quote",
            "body": "Told to wait 15 min for next bus; nobody held the tram."
        }
    }
})
evidence_id = captured["result"]["structuredContent"]["evidence"]["id"]
print("evidence", evidence_id)

method_contract = request({
    "jsonrpc": "2.0",
    "id": 7,
    "method": "tools/call",
    "params": {
        "name": "sd_method_run",
        "arguments": {
            "projectId": project_id,
            "methodId": "journey-from-transcripts"
        }
    }
})
print("method", method_contract["result"]["structuredContent"]["method"]["id"])

nodes = [
    {
        "id": "plan",
        "label": "Plan journey",
        "fields": {"emotion": "3", "touchpoint": "website"},
        "evidence": [{"evidenceId": evidence_id}]
    },
    {
        "id": "wait",
        "label": "Wait at stop",
        "fields": {"emotion": "2", "touchpoint": "stop"},
        "evidence": [{"evidenceId": evidence_id}]
    },
    {
        "id": "transfer",
        "label": "Missed connection",
        "fields": {"emotion": "1", "touchpoint": "transfer"},
        "evidence": [{"evidenceId": evidence_id}]
    }
]
written = request({
```

```
    "jsonrpc": "2.0",
    "id": 8,
    "method": "tools/call",
    "params": {
        "name": "sd_artifact_write",
        "arguments": {
            "projectId": project_id,
            "methodId": "journey-from-transcripts",
            "title": "Current-state journey",
            "nodes": nodes
        }
    }
})
artifact_id = written["result"]["structuredContent"]["artifact"]["id"]
print("artifact", artifact_id, written["result"]["structuredContent"]["status"])

reviewed = request({
    "jsonrpc": "2.0",
    "id": 9,
    "method": "tools/call",
    "params": {
        "name": "sd_review",
        "arguments": {
            "projectId": project_id,
            "artifactId": artifact_id,
            "verdict": "pass",
            "rows": [
                {
                    "rowKind": "proposal",
                    "severity": "low",
                    "message": "Keep the disruption notice visible at the transfer d
                    "evidenceId": evidence_id
                }
            ]
        }
    }
})
print("review", reviewed["result"]["structuredContent"]["verdict"], reviewed["result

factcheck = request({
    "jsonrpc": "2.0",
    "id": 10,
    "method": "tools/call",
    "params": {
        "name": "sd_factcheck",
        "arguments": {"projectId": project_id}
    }
```

```
})
print("factcheck", factcheck["result"]["structuredContent"]["pass"])

server.stdin.close()
server.wait(timeout=5)
PY
```

The session matters because it proves discovery and mutation in one run. `tools/list` confirms the server mounted the 30-tool registry. `resources/list` confirms page-one discovery on a 100-row page and therefore a real corpus surface. `prompts/get` confirms the role contract is mounted, not assumed. Then the project workflow proves the write path: create, capture evidence, fetch the method contract, write a draft, pass review with an evidence-cited row, and replay with factcheck.

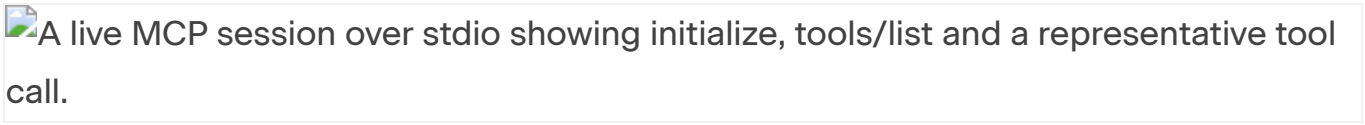


Figure 17.1 **A real stdio MCP session.** The capture proves that discovery and a representative tool call were observable over the MCP surface; it does not, by itself, prove native host UI behavior, long-running orchestration, or phase completion.

If you want to inspect the workflow more mechanically, reduce it to six checkpoints. First, `initialize` must return `asd-mcp` and the advertised capability buckets. Second, `tools/list` must count 30 tools. Third, `resources/list` must return page one with 100 rows when corpus roots are present. Fourth, `prompts/get` must resolve one of the ten roles. Fifth, the create-to-review path must finish with a final artifact. Sixth, `sd_factcheck` must return `pass: true`. If any one of those fails, I do not trust the mount yet.

Notice what I did not do here. I did not call `sd_phase_advance`. That is intentional. Passing factcheck is necessary. It is not sufficient. Phase gates remain human-authorized. The MCP session can prepare the state. It cannot impersonate the decision.

Errors, Boundaries and Receipts

A good MCP server is defined as much by how it refuses work as by how it accepts it. ASD uses the standard JSON-RPC codes for framing errors, then a small server-defined band for runtime boundaries. The split is clean: protocol mistakes stay protocol mistakes; tool failures stay tool failures.

CODE	MEANING	TYPICAL TRIGGER
`-32700`	Parse error	Malformed JSON on the wire
`-32601`	Method not found	Unknown protocol method name
`-32602`	Invalid params	Bad cursor, missing required fields, unknown prompt or tool arguments
`-32603`	Internal error	Unexpected handler failure
`-32000`	Server-defined boundary	Initialize required before discovery or calls
`-32001`	Server-defined boundary	Tool call timed out
`-32002`	Server-defined boundary	Resource not found or equivalent fail-closed lookup refusal

One subtle but important behavior sits beside that table: runtime tool failures do not have to become JSON-RPC failures. A bad review payload or a rejected tool contract can come back as a valid `tools/call` result with `isError: true`. That keeps the transport stable while still making the runtime refusal explicit.

> **CAUTION:** Never confuse a clean MCP transport with a successful ASD operation. A valid JSON-RPC envelope can still contain a rejected tool result. Read the content, the structured payload, and the gate state before you declare success.

The second boundary is the six-bundle receipt story. Current ASD release evidence names six emitted MCP bundles with fresh disposable stdio proof: `buzz`, `claude-code`, `codex`, `kimi`, `mcp`, and `opencode`. The framing is very specific. Each bundle is proved pack-locally for `initialize`, all 30 tools, all 228 resources, all 10 prompts, and one representative project workflow ending in clean factcheck. That is strong proof of bundle integrity. It is not proof that every native host UI executed the bundle end to end.

That distinction saves a lot of nonsense. A disposable receipt can prove that the Codex bundle exposes the same registry as the generic MCP bundle. It cannot prove that a particular Codex desktop build rendered the mount correctly in a specific user account. Host execution needs host-specific evidence. Pack integrity needs pack-local evidence. ASD keeps those claims separate on purpose.

The final boundary is ACP. The MCP server can record orchestration plans, dispatch attempts, handoffs, human gates, and error rows in the shared record vocabulary. ACP itself remains `asd-acp/1`, record-only, and fail-closed. The supported capability JSON is `{"protocolVersion":"asd-acp/1","capabilities":["record-only"]}`. If a client asks for `host-agent-spawn` on the generic MCP surface, the correct result is a structured unsupported error. No simulation. No silent fallback. No fairy tale about a spawned worker that never existed.

I am strict about that line because orchestration theater is dangerous. A handoff record is valuable. A dispatch attempt record is valuable. A typed same-project artifact pointer is valuable. None of them should be mislabeled as live transport or live process execution unless the host proves exactly that path. The MCP chapter is where I want that honesty to become muscle memory.

So the safe mental model is this. MCP gives ASD a portable runtime face. The 30 tools are the executable contract. The 228 resources and 10 prompts make the corpus and role model visible. The client configs are small and literal. The session transcript is inspectable. The error codes are typed. The receipt story is specific. ACP remains record-only. Human phase gates remain human. That is a serious platform surface. It is also exactly the kind of restraint that keeps agentic service design trustworthy when the engagement gets large.

CHAPTER 16

Service Quality Management & CX Maturity Governance

CHAPTER 18

Plugins & Host Packs: Claude Code and Beyond

NEXT CHAPTER

Plugins & Host Packs: Claude Code and Beyond

Install the whole practice into your coding agent

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 18

Plugins & Host Packs: Claude Code and Beyond

Install the whole practice into your coding agent

Reading time

19 min

The emitter and the pack tree

The clean mental model is this: `platform/` is the source of truth, and every host pack is a projection. I start from ten canonical roles, eighty-one skill sources that ship as eighty executable skills after the authored `service-blueprinting` flagship replaces its generated twin, seven authored templates, and the shared runtime registry of thirty `sd_*` tools. The compiler turns that inventory into release data, then the emitter lays it out for each host without changing the underlying practice.

That is why the pack tree matters. It prevents the usual integration drift where every client team hand-copies prompts, renames folders, tweaks configs, and silently forks the method. ASD refuses that pattern. The canonical inventory also carries 228

corpus resources, 10 role prompts, 38 registered artifact-kind contracts, and 39 catalog vocabulary rows. Those counts belong to the platform, not to any single host. A host pack either receives that inventory in a form it can load, or it receives an explicitly bounded subset. Nothing is implied.

The distribution story follows the same rule. The monorepo emits the installable targets, then `scripts/publish-distribution-repos.mjs` republishes them into two standalone mirrors: `imehr/asd-plugin` for coding-agent host packs and native Claude Code marketplace install, and `imehr/asd-mcp` for the standalone MCP server. Each mirror carries a `provenance.json` file that records the source commit it came from. If that provenance record is missing or stale, I stop trusting the copy immediately.



Figure 18.1 **Emitter writing the pack tree.** This capture proves that `emit-packs.mjs` generated the distributable targets from the canonical source tree in one run. It does not prove that any native host loaded or executed those packs.

emit-packs.sh

```
pnpm --dir suite install
node platform/compiler/compile.mjs
node platform/compiler/generate-skills.mjs
node platform/compiler/emit-packs.mjs --output /tmp/asd-packs
```

That four-command sequence is the foundation for everything else in this chapter. `compile.mjs` refreshes the canonical manifest and capability inventory. `generate-skills.mjs` renders the executable method skills. `emit-packs.mjs` projects the same practice into twelve targets: `agy`, `buzz`, `claude-code`, `cli`, `codex`, `dsh`, `grok`, `kimi`, `mcp`, `omp`, `opencode`, and `pi`. The output directory is deliberately disposable because emission is a build product, not a place for hand-maintained edits.

> **My take:** The fastest way to corrupt an agentic practice is to patch the generated pack because it feels quicker than fixing the source. I have done it. It always comes back to bite. If the change belongs to the practice, I make it in `platform/agents`,

`platform/skills`, `platform/templates`, or the emitter itself, then I regenerate every surface.

That regeneration rule is more than tidy engineering. It is how ASD keeps one method contract across many shells. A Claude plugin, a Kimi project mirror, and a bare CLI install should differ in loading shape, not in what counts as acceptable evidence, what the Critic can veto, or who owns the phase gate. The whole point of the emitter is to keep those invariants boring.

The twelve-target matrix

Once I accept that a pack is a distribution boundary rather than a second implementation, the twelve-target matrix becomes easy to read. Some targets bundle the full runtime with roles, skills, and MCP wiring. Some are prompt-only or neutral trees by design. Some are native installs. Some are repo conventions. The job is not to force them into false equivalence. The job is to say, precisely, what each one contains and what evidence backs it.

TARGET	WHAT IT CONTAINS	HOST SHAPE	CURRENT EVIDENCE BOUNDARY
`agy`	`review-session.md` carrying the ASD Critic posture for review work; no bundled runtime.	Review-session contract for AGY or Herdr style use.	Not a native plugin; live AGY prompt execution remains unverified.
`buzz`	Keyless Persona Pack with 10 persona definitions, 80 skill references, pack-relative MCP config, vendored MCP bundle, and compatibility projections.	Native BUZZ Persona Pack plus pack-local MCP.	Pack-local MCP boot is evidenced; authenticated channel membership, exact mention routing, visible replies, and durable ASD

TARGET	WHAT IT CONTAINS	HOST SHAPE	CURRENT EVIDENCE BOUNDARY
			record binding need separate live receipts.
`claude-code`	`.claude-plugin/plugin.json`, plugin-root `.mcp.json`, `agents/`, `skills/`, `commands/asd.md`, vendored `mcp/`, capability manifest.	Native Claude Code plugin and marketplace surface.	Marketplace installation and structural validation are available; native session execution still requires its own host receipt.
`cli`	`asd` launcher plus `cli/bin/cli.mjs` and shared runtime data.	Standalone terminal runtime.	Isolated pack-local CLI boot and representative workflow verified.
`codex`	`AGENTS.md`, `.codex/config.toml` block, vendored `mcp/`.	Repository conventions plus MCP registration, not a native plugin manifest.	MCP bundle has fresh stdio evidence; Codex-native loading and execution remain host-bounded.
`dsh`	Capability pointer and profile-facing distribution for the `@asd/ui` Studio adapter.	DeepSeek Harness Studio integration surface.	Pinned receipt exists for DSH `0.1.0-rc.5` and `@asd/ui` `0.5.0`; support is explicit, including one unsupported capability row.
`grok`	Ten role prompts under `prompts/`; no MCP bundle, no executable runtime.	Prompt-only interoperability target.	Prompt format inspected; native Grok discovery and live workflow unverified.
`kimi`	`.kimi-code` agents, skills, MCP bundle, `mcp.json`, and project	Project-level Kimi mirror with	Pack-local MCP boot is evidenced;

TARGET	WHAT IT CONTAINS	HOST SHAPE	CURRENT EVIDENCE BOUNDARY
	conventions.	workspace trust.	native Kimi loading remains a separate host claim.
`mcp`	`mcp/bin/server.mjs`, `asd-mcp.json`, optional shim, corpus payload.	Standalone stdio MCP server.	Isolated stdio initialize and representative workflow verified.
`omp`	Neutral `agents/<role>/instructions.md` and `skills/<skill>/SKILL.md` trees only.	Source tree to copy into OMP discovery paths.	No bundled MCP or runtime; native OMP loading and execution unverified.
`opencode`	`agent/`, `skills/`, vendored `mcp/`, and `opencode.json` command-array config.	Project adapter for OpenCode.	MCP bundle has fresh stdio evidence; native OpenCode discovery and execution remain unverified.
`pi`	Neutral role and skill trees, mirroring canonical content without runtime packaging.	Copy-only source material for Pi discovery paths.	No MCP or executable runtime in target; Pi host format and execution are unverified.

There is a pattern hiding in that table. Runtime-equipped packs carry the same MCP engine and, when the corpus roots are present, expose 30 tools, 228 resources, and 10 prompts. Prompt-only and neutral targets do not pretend otherwise. That is why I read pack docs as capability boundaries, not marketing pages. A target either bundles MCP wiring, or it does not. A target either has a native install story, or it asks me to copy source material into host discovery paths. Both are valid. Confusing them is how false claims get published.

> **CORE PRINCIPLE:** The host matrix is a truth table, not a feature grid. Read each target by what it concretely installs, which runtime it actually carries, and which proof artifacts exist for that specific layer.

Claude Code: native plugin and marketplace

Claude Code is the reference native plugin surface because it takes the whole practice in one install: ten role subagents, eighty executable method skills, the `/asd` orchestration command, and a vendored MCP server rooted at the plugin itself. I do not need a global `asd` binary. I do not need a sibling checkout. I do not need to guess where the server lives. The plugin carries its own wiring.

claude-plugin-install.sh

```
claude plugin marketplace add imehr/asd-plugin
claude plugin install asd@asd-platform
```



Figure 18.2 **Claude Code marketplace install.** This capture proves that the ASD plugin can be added from `imehr/asd-plugin` and installed as `asd@asd-platform` in Claude Code. It does not prove that a later interactive Claude session executed every role or tool path inside the host.

That two-command path is the canonical user install. It is the one I recommend whenever the marketplace is available, because it keeps distribution honest. The published plugin comes from the emitted mirror repository, and its `provenance.json` tells me exactly which source commit it was synced from.

From a source checkout, I can install the same plugin through the local packs tree. The important detail is that this is still a plugin install, not a manual copy

masquerading as one. Claude Code reads the marketplace manifest from the emitted packs directory and installs the generated plugin by its marketplace id.

claude-plugin-from-source.sh

```
REPO=/path/to/agentic-service-designer
claude plugin marketplace add "$REPO/platform/packs"
claude plugin install asd@asd-platform
```

Inside the installed plugin, the critical file is the plugin-root `.mcp.json`. It points the host at the vendored server with a path that remains valid after installation:

plugin-root.mcp.json

```
{
  "mcpServers": {
    "asd": {
      "command": "node",
      "args": ["${CLAUDE_PLUGIN_ROOT}/mcp/bin/server.mjs"]
    }
  }
}
```

That one configuration block carries a lot of meaning. `${CLAUDE_PLUGIN_ROOT}` resolves inside the installed plugin, so the vendored server is self-contained. No PATH registration. No global npm install. No “clone the monorepo next to your host” footnote. Claude gets a native plugin manifest under `.claude-plugin/plugin.json`, the ten role files under `agents/`, the eighty method skills under `skills/`, the orchestration shortcut at `commands/asd.md`, and the runtime bundle under `mcp/`. Everything arrives together.

There is still a fallback when the marketplace is unavailable: emit the packs, copy the Claude pack into a project, merge the copied `.mcp.json` with the project configuration, and replace `${CLAUDE_PLUGIN_ROOT}` with a project-relative path. I use

that only when I have to. It is a practical source-level fallback, but it is not the same claim as a native plugin install.

> **My take:** When I am evaluating a host for real team use, I look for one boring property first: can it install the whole pack without hand wiring? Claude Code passes that test. That is why it is the reference surface in this chapter, even though the runtime underneath is shared with the CLI and MCP server.

Role mentions and the `/asd` command

Once the plugin is installed, the surface I actually work with is simple. The canonical role mentions are `@orchestrator`, `@researcher`, `@synthesizer`, `@mapper`, `@ideator`, `@prototyper`, `@measurer`, `@critic`, `@presenter`, and `@librarian`. Those ten names are stable across the pack ecosystem. They are not cosmetic aliases. They encode authority, inputs, outputs, veto rights, and human checkpoints drawn from the shared role contract.

The `/asd` command is the orchestrator shortcut layered on top of that role roster. I use it when I want the platform to frame an engagement from a brief, route work in the expected order, keep every write on the tool path, and stop at the points where a human decision is required. I do not use it because I want magic. I use it because it is the shortest route to a disciplined loop.

claude-session.txt

```
/asd
@researcher run contextual-inquiry for lighthouse-transit and add every verbatim quo
@mapper run journey-from-transcripts over the register and write the current-state j
@critic review the journey-map before it can go final.
```

That sequence shows the core interaction model. The command frames. The role mentions make the active discipline explicit. The skills in the background provide the SOP layer. The vendored MCP server persists the work. The runtime gates do the enforcement. The host shell stays conversational, but the storage path is deterministic: evidence first, method run, artifact write, Critic pass or veto, factcheck, then a human phase advance. The role prompt cannot bypass that sequence.

The role distinctions stay sharp for a reason. The Researcher gathers evidence and provenance. The Mapper writes journeys, blueprints, ecosystems, and mental models. The Critic does not “give feedback” in the hand-wavy workshop sense; it writes evidence-cited review rows and can veto a draft back to revision. The Presenter assembles from final artifacts only. The Librarian checks corpus grounding and pack drift. Once I start thinking in those operational contracts, the host UI becomes much less mysterious.

> **CAUTION:** Do not read conversational delegation as transport. ACP is `asd-acp/1` and record-only. The platform can record orchestration plans, dispatch attempts, handoffs, retries, and human gates, but it must not be described as silently spawning external agents just because a role prompt asked for work.

That warning is load-bearing. A role mention is content plus host capability. It is never a blanket license to invent subprocess behavior. If the host can dispatch, the receipt must say so. If it cannot, ASD records the boundary explicitly rather than faking it. I would rather have an honest refusal than a slick demo that lies about what happened.

OpenCode, Codex, Kimi, Grok, AGY, OMP, and Pi

Claude Code is the cleanest native plugin story, but it is not the only useful surface. The other packs exist because real teams already live in different shells. The winning pattern is consistency at the practice layer and specificity at the host layer.

OpenCode gets a proper project adapter. I copy `agent/` into `.opencode/agents/`, `skills/` into `.opencode/skills/`, place the vendored runtime under a project-local `mcp/`, and merge the emitted `opencode.json` block that runs `["node", "mcp/bin/server.mjs"]`. It is a strong fit when the team wants local project agents and an explicit MCP command array. The boundary is honest: emitted config and bundle are verified, but host loading stays unverified until an OpenCode-specific acceptance receipt says otherwise.

Codex receives repository conventions plus MCP, not a native plugin manifest. The important files are `AGENTS.md`, the emitted TOML block for `.codex/config.toml`, and the vendored server copied to a stable path such as `.asd-mcp/bin/server.mjs`. I like this shape because it meets Codex where Codex already works: repo rules in plain text, runtime tools through MCP. I do not overclaim it. The Codex pack does not promise native subagent discovery just because the role prompts exist.

Kimi is a project mirror under `.kimi-code/`. It carries the same ten agents, the same eighty skills, the same vendored MCP bundle, and a project-level `mcp.json` that points at `.kimi-code/mcp/bin/server.mjs`. The extra operational requirement is workspace trust. If the project has not been trusted, the pack is present on disk but not yet in a state where the host can be treated as live.

Grok is intentionally lighter. It ships prompt files only. No bundled MCP. No runtime. If I want Grok to work against the ASD engine, I pair the role prompt with a separate MCP client that points at the standalone `mcp` pack. That is still useful. Prompt-only does not mean worthless. It means exactly what it says.

AGY is narrower still. Its payload is a review-session contract that teaches the AGY or Herdr lane how to behave like an ASD Critic: inspect evidence, read acceptance rules, respect host capability boundaries, and keep human authority explicit. It is excellent for adversarial review. It is the wrong thing to call a native AGY plugin, because it is not one.

OMP and **Pi** are neutral trees. They ship canonical role and skill source material in host-friendly folder layouts, but no runtime bundle and no MCP wiring. I use them when I want to port the practice into a host whose loading rules I control separately.

They are deliberately conservative. The pack gives me the content and refuses to pretend the runtime question is solved.

Beneath all of those host adapters sit the two lowest-level surfaces: the standalone CLI and the standalone MCP server. When I need to prove the loop itself rather than the host sugar around it, I drop to one of those layers first.



Figure 18.3 **CLI creating the first evidence rows.** This capture proves that the emitted terminal pack can create a project and add evidence in an isolated workspace using the shared runtime. It does not prove that any higher-level host pack invoked the same loop inside its own UI.

cli-first-loop.sh

```
PACK=/tmp/asd-packs/cli
mkdir -p "$HOME/.local/bin"
ln -sf "$PACK/asd" "$HOME/.local/bin/asd"
export PATH="$HOME/.local/bin:$PATH"
export ASD_WORKSPACE="$(mktemp -d)/projects"
asd project-create --name "Metrovale Access" --client "Fictional composite" \
  --sector public-transport --service "step-free journeys" --phase discover
PROJECT_ID=metrovale-access-12345678
asd evidence-add --projectId "$PROJECT_ID" --source interview-01 --kind quote \
  --body "I cannot tell whether the lift is working before I travel."
```

I keep that CLI loop in the chapter because it exposes the shared engine with nowhere to hide. The same gate logic appears here that later surfaces through Claude Code, OpenCode, Kimi, Codex, or a standalone MCP client. Evidence kinds are still exactly `quote`, `telemetry`, and `document`. A failing artifact write still persists as draft with machine-readable failure rows. A Critic pass still needs an evidence-cited proposal row. A veto still needs a defect and a fix. The host can change. The contract cannot.

What verified means, and what it does not

This is where most agent-platform documentation goes soft. ASD is stricter. It separates three proof layers that many teams lazily collapse into one sentence.

First layer: emission. Emission proves the compiler generated the expected files and pack tree from the canonical source. That is what Figure 18.1 shows. It tells me the build product exists and that the generated layouts line up with the declared targets. Emission alone does not prove a native host loaded anything.

Second layer: pack-local execution. For six emitted MCP bundles — `buzz`, `claude-code`, `codex`, `kimi`, `mcp`, and `opencode` — the current release matrix has fresh disposable stdio evidence. In each case the evidence covers `initialize`, the full set of 30 tools, 228 paginated resources, 10 prompts, a representative project-to-evidence-to-artifact-to-evidence-cited-review-to-factcheck workflow, and clean stderr with no ambient executable fallback. That is strong proof. It is still only pack-local proof.

Third layer: native host execution. This is the layer that says a particular product version, in a particular integration mode, loaded the pack and exercised the behavior being claimed. Claude marketplace installation is one example. The DSH receipt is another. A BUZZ live channel walkthrough is another, but only for the exact bounded slice it captured. Host receipts are where I learn what really happened in the shell a human will use.

The practical consequence is simple: I never let a lower layer impersonate a higher one. A pack-local MCP initialize transcript does not become proof that BUZZ delivered a mention to the right runtime. A generated plugin manifest does not become proof that Claude executed a research session. A DSH compatibility pointer does not become proof that every provider and every chat mode is supported. I stay literal.

host-verification.sh

```
REPO=/path/to/agentic-service-designer
claude plugin validate "$REPO/platform/packs/claude-code"
```

`claude plugin validate` is a good example of the distinction. It is meaningful. It proves the emitted Claude pack matches the host's structural expectations for a plugin. It is also not a session receipt. Validation tells me the door is shaped correctly. It does not tell me a user walked through it and completed a real loop.

> **My take:** I trust receipts that name the host, the version, the path, and the exact behavior exercised. Everything else is build evidence. Useful, necessary, and insufficient on its own.

That discipline also keeps the language around BUZZ and DSH clean. BUZZ Article 37 shows an observed live slice: a private channel, genuine orchestrator reply and relay handoffs, and a Canvas index. It does not prove MCP persistence, durable Critic verdict rendering, reload or cleanup, or ACP spawn. DSH has a pinned receipt for its verified profile, slot set, test suite, and loopback Studio smoke. It does not claim every provider, chat surface, or external MCP-client combination under the sun. The boundaries are not weaknesses. They are the proof that the platform is being documented honestly.

Choosing a surface per engagement

So which surface should I choose? I start from the engagement constraint, not from the shininess of the host. If I need a native collaborative coding shell with the full pack intact, I choose Claude Code. If I need a machine-facing runtime for another client, I choose MCP. If I want the rawest possible proof path or CI integration, I choose the CLI. If a team already works inside a different host, I install the corresponding adapter and keep the evidence claims narrow.

ENGAGEMENT NEED	BEST SURFACE	WHY I CHOOSE IT	WHAT I STILL VERIFY SEPARATELY
I want the fullest native ASD experience in one install.	Claude Code plugin	It loads roles, skills, `/asd`, and vendored MCP together through a real plugin boundary.	Interactive host session behavior for the exact Claude environment I am using.
I need scripted runs, CI gates, or the cleanest possible runtime proof.	CLI	The shared engine is exposed directly with no host abstraction in the way.	Project workspace isolation and the artifact loop for the scenario at hand.
I need host-agnostic tool access from another app or agent client.	MCP	It is the standalone JSON-RPC runtime with the same 30 tools, 228 resources, and 10 prompts.	The client-specific config path and corpus-root availability.
The team already operates in OpenCode and wants project-local agents.	OpenCode	The adapter mirrors roles and skills into the host's discovery paths and keeps MCP local to the project.	Native OpenCode loading and invocation in the current host version.
The team uses Codex and wants repo conventions plus tool access.	Codex	`AGENTS.md` plus MCP registration fits Codex's operating model cleanly.	Desktop or CLI host loading for the exact Codex build in use.
The team uses Kimi and accepts workspace-trust mechanics.	Kimi	The `.kimi-code/` mirror carries the same practice contract with a project-local server.	Trust prompts, discovery, and host execution in the active Kimi version.
I need a multi-agent network surface with explicit identities and mentions.	BUZZ	The Persona Pack gives me ten canonical personas plus local MCP, ready for owner-bound runtime materialization.	Identity audit, channel membership, exact mention routing, reply attribution, Canvas binding.

ENGAGEMENT NEED	BEST SURFACE	WHY I CHOOSE IT	WHAT I STILL VERIFY SEPARATELY
I need a Studio shell with sidebar, settings, and bounded capability projection.	DSH	The `@asd/ui` adapter plugs into a verified Harness profile and exposes the platform through Studio slots.	Compatibility revision, profile lifecycle, and the specific live receipt scope.
I only need role posture or a neutral source tree, not bundled runtime.	Grok, AGY, OMP, or Pi	These targets separate prompt content or neutral file layouts from runtime claims on purpose.	Any real MCP client, native discovery, or execution path I add around them.

If I boil that decision down to one sentence, it is this: choose the thinnest surface that still matches the work you need to do, then verify the exact layer you plan to rely on. For teaching, debugging, and CI, I often start with the CLI or standalone MCP because the receipts are easiest to read. For live design work inside a coding agent, Claude Code is the reference because the whole pack lands intact. For adjacent hosts, I treat the emitted adapter as source material plus bounded evidence, then I earn stronger claims only by running the host-specific acceptance path.

That habit keeps the platform honest. It also makes it much easier to extend. Once I understand that a pack is just a deterministic projection of one canonical practice, I stop arguing about which host is “the real one.” The real one is the practice contract. The host is the shell I can validate today.

NEXT CHAPTER

BUZZ: Multi-Agent Service Design on the Open Agent Network

Ten personas, exact mentions, one Canvas

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03

CHAPTER 04

Service-Dominant Logic & Value Co-Creation in the Age of AI

The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 19

BUZZ: Multi-Agent Service Design on the Open Agent Network

Ten personas, exact mentions, one Canvas

Reading time

18 min

BUZZ for Service Design Teams

When I use BUZZ for Service Design, I am not looking for a prettier chat client. I am looking for the missing collaboration surface between a human designer and a role team. BUZZ gives me five primitives that matter in practice. First, a relay stores signed events. Second, every human or runtime appears as an identity with a 64-hex public key. Third, I get private stream channels with explicit membership. Fourth, I can mention one exact member by pubkey and see whether the reply came back from that same identity. Fifth, I get one per-channel Canvas document that acts as an index over the work.

The sharp edge is the boundary. BUZZ is where conversation, roster, attribution, and visible handoffs live. It is not the ASD store. A message can be genuine, signed, and useful, while still proving nothing about whether an artifact persisted through `sd_artifact_write`, whether a review passed, or whether a human gate receipt exists. I keep repeating that because the failure mode is seductive. Once a channel looks alive, people start treating the transcript as the system of record. That is exactly what ASD is designed to prevent.

> **CORE PRINCIPLE:** In BUZZ, the message proves conversation, the pubkey proves authorship, the Canvas proves navigability, and the ASD workspace proves the work. If I collapse those into one claim, I lose the audit trail.

The generated BUZZ Persona Pack carries the same canonical inventory as the other emitted ASD targets: 10 role prompts, 80 shipped skills from 81 sources, 30 `sd_*` tools, 228 pack-local MCP resources when the corpus is present, 38 registered artifact kinds, 39 catalog rows, 7 authored templates, and 12 emitted targets across the wider platform. That inventory matters because it tells me the pack is aligned with the rest of ASD. It does *not* tell me that a BUZZ-native host executed every tool, that every persona is materialized as a healthy runtime, or that a given channel has the right members. Those are separate proofs.

BUZZ PRIMITIVE	WHAT IT PROVES	WHAT IT DOES NOT PROVE
Relay event	A signed message, membership change, or Canvas update happened.	That an ASD artifact was persisted or reviewed.
Public-key identity	Which human or runtime authored the event on the wire.	That the identity belongs to the right persona definition without an audit join.
Private stream channel	The engagement has an explicit member list and conversation boundary.	That every member is healthy, unique, or authorized to mutate the ASD store.

BUZZ PRIMITIVE	WHAT IT PROVES	WHAT IT DOES NOT PROVE
Exact mention	A request was addressed to one chosen runtime public key.	That the runtime's reply cited resolvable workspace ids.
Canvas	The channel exposes a shared artifact index.	That the index itself is persistence, or that a phase may advance.

This is why BUZZ fits Service Design so well. Service Design is already a team sport with visible handoffs, specialist roles, and human judgment at the hard edges. BUZZ preserves that shape. The orchestrator can delegate. The critic can block. The librarian can report drift. I can hold the phase gate myself. The channel shows the social topology honestly instead of flattening it into one assistant voice.

Personas, Identities and Public Keys

The Persona Pack is deliberately keyless. That design choice is the whole safety story. The pack definition tells BUZZ what a persona is supposed to be: canonical slug, prompt, runtime metadata, pack-local MCP wiring. It does not mint a public key, start a runtime, or join a channel. The owner does that later. So I keep two nouns separate: the *definition* and the *runtime identity*.

The ten canonical persona slugs are exact and boring on purpose:

ROLE	PERSONA SLUG	WHAT I EXPECT IT TO OWN
Orchestrator	`sd-orchestrator`	Method sequence, handoffs, visible phase-gate preparation
Researcher	`sd-researcher`	Evidence register and provenance-grounded asks
Synthesizer	`sd-synthesizer`	Insight, affinity, and persona work
Mapper	`sd-mapper`	Journey maps, blueprints, and ecosystem structures

ROLE	PERSONA SLUG	WHAT I EXPECT IT TO OWN
Ideator	`sd-ideator`	HMWs, concepts, and opportunity framing
Prototyper	`sd-prototyper`	Prototype plans, walkthroughs, and tests
Measurer	`sd-measurer`	KPI chains and service measurement logic
Critic	`sd-critic`	Evidence-cited review rows and veto posture
Presenter	`sd-presenter`	Report/export assembly from final artifacts only
Librarian	`sd-librarian`	Canon, provenance, and method drift checks

Those slugs are not mention targets. They are pack definitions. A live runtime becomes real only when the owner materializes it in BUZZ, assigns a provider/model/runtime host, and gets an identity-backed public key. The honest join key is `runtime.persona_id`. Each runtime should expose exactly one `runtime.persona_id` that matches exactly one pack definition. Then, and only then, I can say that a runtime public key represents `sd-mapper` or `sd-critic`.

This is where people get sloppy. They see a green card in the UI labeled “Maya” and assume the orchestrator exists, is healthy, is the only orchestrator instance, and is ready to receive a mention in the current channel. None of that follows. A definition may exist with no runtime. A runtime may exist but be inactive. Two runtime instances may report the same persona id. A healthy runtime may still be absent from the channel I care about. A slug is stable metadata. A pubkey is the actual routing surface.

> **My take:** “Slug equals agent” is the BUZZ version of “branch equals deployment.” It feels true right up until the day you send a mention to the wrong runtime and cannot honestly prove who answered. I would rather spend two minutes auditing than publish one fake attribution.

The pack itself stays safely small in scope. It installs through BUZZ's native Persona Pack workflow. It carries relative MCP wiring so the installed pack can point to

`mcp/bin/server.mjs` without hardcoding a developer machine path. What it never does is write BUZZ application storage directly, create identities, or bind members to a channel. That is owner-authorized runtime work outside the emitted pack. Good. The compiler should emit definitions, not side effects.

The Safe Sequence, End to End

I run BUZZ engagements in one safe sequence and I do not improvise it. First I install and validate the Persona Pack through the native BUZZ workflow. Second I owner-materialize the ten runtimes. Third I create the private engagement channel. Fourth I add the audited runtime pubkeys as members. Fifth I verify the membership and resolve display names from the exact pubkeys. Sixth I post the human kickoff and send a scoped mention to one chosen runtime. Seventh I persist any actual ASD records through the pack-local MCP or the CLI, not through chat. Eighth I link those records from Canvas. Ninth I stop at the human phase gate.

The first live BUZZ commands are straightforward. I use `buzz channels list` to inspect existing `sd/` channels, then create a dedicated private stream for the engagement and add the roster by pubkey:

buzz-safe-sequence.sh

```
buzz channels list

buzz channels create \
  --name sd/<engagement> \
  --type stream --visibility private \
  --description "Fictional ASD engagement; synthetic evidence only."

buzz channels add-member --channel <channel-uuid> --pubkey <runtime-pubkey>
buzz channels add-member --channel <channel-uuid> --pubkey <runtime-pubkey>
# repeat until the audited roster is complete
```

```
buzz channels members --channel <channel-uuid>
buzz users get --pubkey <runtime-pubkey>
```

Notice what is absent. There is no “deploy team” shortcut in my acceptance story. There is no broad “forceNewInstance” wave. There is no trust fall from slug to mention target. I want the exact runtime public key in hand because the public key is what the relay will later attach to the response.



Figure 19.1 **Private engagement channel.** The capture proves that a dedicated private stream was created for the engagement and that membership is an explicit act; it does not prove that every listed runtime is healthy, unique, or yet tied to persisted ASD records.

After the roster is in place, I post a kickoff that states the fictional boundary, the project id, the authority boundary, and the stop condition. Then I send the first exact runtime mention. The content can include a friendly display-name reference, but the automation contract rides on `--mention <pubkey>`:

buzz-orchestrator-mention.sh

```
buzz messages send --channel <channel-uuid> \
  --mention <orchestrator-runtime-pubkey> \
  --content "@sd-orchestrator — start Discover coordination for project <project-id>
```

That sentence does three jobs. It names one exact addressee. It constrains the evidence boundary. It gives the runtime a stop line. BUZZ works best when every ask is auditable against a clear deliverable and a clear refusal condition. A vague prompt may still produce prose. It will not produce proof.



Figure 19.2 **Owner kickoff.** The capture proves that the engagement starts from a scoped human post addressed to the channel and then to a chosen runtime; it does not prove that the resulting work persisted in the ASD store until separate ids and receipts exist.

Once replies start landing, I separate collaboration from persistence on purpose. If Maya replies with a workplan and Liam replies with evidence ids, those replies are claims until the ids resolve in the workspace. If I want a project, evidence, artifact, review, handoff, or gate record, I create it through the pack-local MCP or the CLI. BUZZ stays the coordination surface. The ASD store stays the record. That cut is the whole method.

Fail-Closed Identity Audit

Before I trust any mention target, I run the read-only identity audit. The command is simple:

buzz-live-identity-audit.sh

```
node scripts/buzz-live-identity-audit.mjs \
  --input ./audit.json \
  --channel-id <channel-uuid> \
  --output ./identity-audit.json
```

The important part is what happens inside. The audit joins Persona Pack definitions, live runtimes, optional channel membership, and optional mention targets. It checks for missing definitions, missing runtimes, multiple definitions, multiple runtime instances for one persona, missing runtime pubkeys, runtime error states, inactive runtimes, missing channel membership, ambiguous mention targets, and mismatched mention recipients. It reports a safe remediation plan instead of silently choosing for me.

The script is aggressively bounded. It is read-only and plan-only. It never loads credentials. It never fetches a private key. It never contacts BUZZ on its own. It has no mutation surface. It rejects `--apply` and `--confirm` outright. Its only allowed BUZZ CLI inspection step, when membership checking is in scope, is `buzz channels members --channel <channel-uuid>`. That is it.

> **CAUTION:** The audit refuses to "help" by picking one runtime when multiple instances expose the same `persona_id`. That refusal is a feature. The moment the tool auto-chooses among ambiguous runtimes, your later mention-to-reply claim becomes soft and editorial instead of operational.

I like this audit because it refuses to flatter me. If the runtime is missing, it says missing. If the membership is absent, it says absent. If a proposed mention's pubkey does not match the chosen runtime pubkey, it fails. In other words, it behaves like the rest of ASD: fail closed, generate a repair plan, and stop short of mutation.

Case: Metrovale OneRecovery (Observed Slice)

Article 37 is the right place to learn restraint. The Metrovale OneRecovery material is valuable precisely because it is bounded. In the observed private channel `sd/metrovale-one-recovery`, the owner first saw the ten ASD personas and then added general Claude and Codex advisory identities, producing a 13-member roster. The slice includes a genuine orchestrator reply, relay-confirmed orchestrator-to-researcher and orchestrator-to-mapper handoffs, synthetic researcher and mapper content, a Canvas artifact index, and advisory Claude/Codex storyboard responses. The human Discover gate remained blocked.

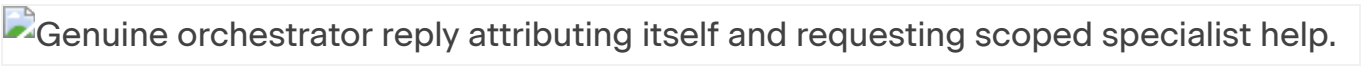


Figure 19.3 **Attributed orchestrator reply.** The capture proves that a runtime-authored reply came back with visible attribution and named the next role handoffs; it does not prove workspace persistence, a passing review, or phase completion on its own.

The boundary list is explicit and I keep it explicit here. Metrovale does **not** prove MCP project-store persistence, durable artifact ids, a visible Critic verdict, ACP spawn or transport, host execution, reload, recovery, cleanup, phase completion, a presenter reply, or a rendered deck. The generated offline Canvas document for Article 37 is

useful, but it is verified offline rather than BUZZ-live. That distinction matters. If I blur it, I teach the wrong lesson.

There is a second lesson in Metrovale. A Canvas row can honestly show `observed`, `draft`, or `planned` without pretending everything is finished. In the captured register, the project charter and authority were observed in BUZZ, the starter evidence and a journey-map outline were observed responses, the advisory storyboard existed only as BUZZ replies with no deck file, and the research plan, service blueprint, measurement plan, review-veto log, executive report, and knowledge register remained planned. That is honest operational publishing.

Lab: Aster Grove Makerspace Live Run

Article 39 is the chapter's live lab because it closes the loop cleanly. The client is fictional. The BUZZ lane is real. The tutorial proves what a full, bounded session looks like when I insist on ids, receipts, and phase discipline. Here is the condensed twelve-step run I want practitioners to copy.

Create an isolated ASD project. In the article, the project id is `aster-grove-makerspace-6a8498ef`. Isolation matters because it prevents one engagement from smearing ids into another.

Register synthetic evidence. Ten rows go in first: quotes, telemetry, and documents. Each one gets a SHA-256 id that later replies must cite.

Write artifacts through the review gate. Stakeholder map, journey, KPI tree, and change plan are persisted through the store; the blueprint deliberately fails once, gets a Critic veto, then is remediated on the same artifact id.

Create the private BUZZ channel and roster. The live channel is `sd/aster-grove-makerspace` with 13 members: owner, ten ASD runtimes, and two coding-agent advisors.

Post the owner kickoff. The kickoff states that the service is fictional, names the project id, states the human authority boundary, and keeps the channel from

becoming an evidence register by accident.

Mention the orchestrator by public key. Maya receives a scoped Discover coordination ask addressed to her runtime pubkey, rather than merely to her display name.

Dispatch specialist work in parallel. Liam gets provenance checking. Kai gets failure-demand mapping. Their replies come back with evidence citations that can be checked against the store.

Record the human gate in the ASD store first. The owner advances Discover to Define with approval receipt `ag-gate-001` only after factcheck is green.

Ask the Critic to verify the persisted review history. Marcus checks the blueprint veto/remediation chain against the store and cites the actual review-veto-log and evidence row.

Render the Canvas index from the workspace. The channel index is generated from store records, not typed in the channel.

Set the Canvas live, get it back, and verify the round trip. The retrieved document matches the generated one modulo a trailing newline, and verify passes all checks.

Continue the loop without fabrication. Maya opens Define, Nadia resolves the synthesis method registry, Rowan sees only the Canvas and not the authoritative store, refuses to fabricate, and the session pauses honestly instead of inventing evidence.

That last step is my favorite because it shows the boundary holding under pressure. Rowan had enough information to bluff. He had the registry. He had the Canvas. He had context. He still declined to invent because he did not have the authoritative workspace. That is what an AI-native Service Design stack should do when access is partial.

asd-human-gate.sh

```
asd phase-advance --projectId aster-grove-makerspace-6a8498ef \
  --toPhase define --idempotencyKey astergrove-gate-discover-01 \
  --approvalReceipt '{"receiptId":"ag-gate-001","approvedBy":"mehran","decision":"ap
```

The channel announcement comes after this store receipt exists. Never before. Agents prepare the decision. I authorize the phase move. That is not a culture preference. It is a protocol boundary.

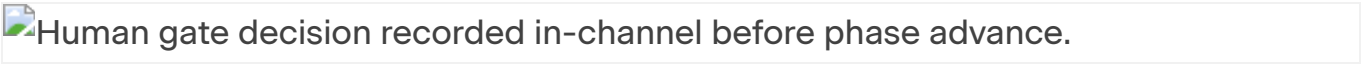


Figure 19.4 **Human gate decision.** The capture proves that the phase decision is announced visibly in-channel with a named human receipt behind it; it does not mean the channel itself authorized the phase move.

The Critic step is just as important. Marcus does not wave through the blueprint because the conversation felt coherent. He checks the persisted review chain: the veto row, the cited evidence, the same-id remediation, and the final review state. Article 39 makes this concrete by using a deliberate defect in the blueprint lane name and then proving the repair on the same artifact id.

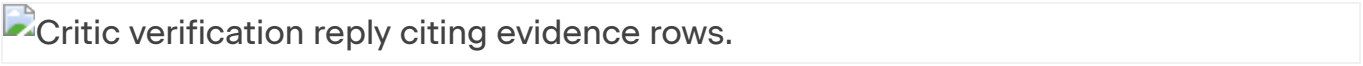


Figure 19.5 **Critic verification.** The capture proves that the Critic cites concrete evidence and review rows when validating the persisted loop; it does not prove that every future artifact in the channel has passed review unless its own ids resolve the same way.

By the time the Aster Grove run reaches its freeze frame, the workspace shows 10 evidence rows, 11 artifacts, 1 typed handoff, and 10 planned kinds. Ten of the artifacts are final with Critic passes. One `insight-set` draft appears without a binding channel reply, so its authorship is left unresolved in the ledger instead of being guessed after the fact. That single choice tells you the chapter's whole ethic. Truth beats completeness theatre.

Canvas: The Shared Index

The BUZZ Canvas is one page per channel. I use it as the front door to the engagement, never as the store itself. The generator that keeps it honest is `scripts/buzz-canvas-index.mjs`. It renders from persisted workspace records, refuses

dangling ids, accepts only registered planned kinds, and can later prove that the document survived a live set/get round trip on the relay.

The lifecycle is mechanical:

buzz-canvas-roundtrip.sh

```
node scripts/buzz-canvas-index.mjs render \
  --project-dir tmp/buzz39/projects/aster-grove-makerspace-6a8498ef \
  --channel sd/aster-grove-makerspace \
  --channel-id 361023a9-31cb-4b40-845f-19d64f7827ee \
  --fictional \
  --planned research-plan --planned interview-guide --planned opportunity-matrix \
  --planned hmw-set --planned concept-set --planned prototype-plan \
  --planned walkthrough-script --planned storyboard --planned test-report \
  --planned recovery-playbook \
  --out articles/39/canvas

buzz canvas set --channel 361023a9-31cb-4b40-845f-19d64f7827ee --content - \
  < articles/39/canvas/aster-grove-makerspace.md

buzz canvas get --channel 361023a9-31cb-4b40-845f-19d64f7827ee \
  > retrieved-canvas.md

node scripts/buzz-canvas-index.mjs verify \
  --canvas-dir articles/39/canvas \
  --project-dir tmp/buzz39/projects/aster-grove-makerspace-6a8498ef \
  --retrieved retrieved-canvas.md --fictional
```

The render step writes a document with twelve contract sections. The verify step checks five things: content hash, presence of all twelve sections, stale-link resolution, relay round trip, and drift against a fresh render. It is a proper integrity loop, not a screenshot ritual.

CANVAS SECTION	WHY IT EXISTS
1. Charter and data boundary	States project identity and reminds readers that the Canvas is navigation, not proof.

CANVAS SECTION	WHY IT EXISTS
2. Phase and human-gate state	Shows the current phase and any approval receipts on record.
3. Participant roster	Lists durable participation from records, not mere channel presence.
4. Evidence register	Indexes evidence sources and ids.
5. Artifact register	Lists kind, id, status, owner, review state, and evidence coverage.
6. Critic findings and remediation	Replays review-veto logs and same-id remediation state.
7. Typed handoff queue	Shows persisted handoff records and unsupported spawn boundaries.
8. Decisions, assumptions, and risks	Makes blocked gates, missing artifacts, and open drafts visible.
9. Prototypes and tests	Indexes prototype-plan, walkthrough-script, storyboard, and test-report artifacts.
10. KPI ownership	Shows who owns measurement artifacts and targets.
11. Final report and recovery playbook	Indexes delivery-side artifacts without inventing them.
12. Index integrity	Carries the round-trip content hash and verification posture.

The planned rows matter more than they first appear. In Aster Grove and Metrovale, planned kinds such as `research-plan`, `interview-guide`, `opportunity-matrix`, `hmw-set`, `concept-set`, `prototype-plan`, `walkthrough-script`, `storyboard`, `test-report`, and `recovery-playbook` can appear as `planned` only because they are registered kinds. If I try to smuggle an unregistered label into the index, render fails closed. If I leave a stale id in place, verify fails closed. That is exactly how an index should behave.


Canvas index set on the channel from the generator output.

Figure 19.6 **Canvas set live.** The capture proves that the generated index was pushed to the channel and treated as the visible front door to the engagement; it does not turn the Canvas into the ASD store or authorize any artifact or phase by itself.

> **My take:** Teams love a shared page because it feels like closure. I use that instinct, then discipline it. The Canvas gives everyone one place to land, but it keeps forcing them back to the underlying ids, statuses, and review states. That is the difference between an index and folklore.

Boundaries: What BUZZ Does Not Prove

Let me end bluntly. BUZZ does not prove workspace persistence unless the cited project, evidence, artifact, review, handoff, or gate ids resolve in the ASD store. It does not prove ACP transport because ACP is record-only in ASD and `host-agent-spawn` is unsupported. It does not prove that a runtime public key maps to the correct persona without the read-only audit. It does not prove that a phase advanced because only a human approval receipt can do that. It does not prove that a rendered deck, prototype, or report exists simply because someone discussed one in-channel.

That is not a limitation to hide. It is the operating model. BUZZ is the collaboration lane on the open agent network. The ASD workspace is the persistent rigor engine. ACP records typed coordination and stops there. Human gates stay human. Once I accept those boundaries, BUZZ becomes incredibly powerful: exact role mentions, visible handoffs, live attribution, a shared Canvas front door, and an honest record of what the team knew at each moment. Once I ignore those boundaries, it becomes another chat transcript pretending to be a project system.

So the practical rule is simple. Mention by pubkey. Verify by audit. Persist through MCP or CLI. Index through Canvas. Advance only with a human receipt. Everything else is commentary.

CHAPTER 18

Plugins & Host Packs: Claude Code and Beyond

CHAPTER 20

DeepSeek Harness Studio: The Integrated Workbench

NEXT CHAPTER

DeepSeek Harness Studio: The Integrated Workbench

ASD inside a browser studio

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 20

DeepSeek Harness Studio: The Integrated Workbench

ASD inside a browser studio

Reading time

27 min

Harness + Adapter: Division of Labour

The first thing I want clear is authorship of responsibility. DSH is the host. It owns the browser shell, the profile manager, the slot system, the settings surface, and the web server that serves the Studio on loopback. ASD does not fork the host and it does not patch the host source tree in place. The shipped pointer is `platform/packs/dsh/`, and the actual integration package is `@asd/ui`. That package composes into the official DSH profile beside `@deepseek-ai/dsh-base` and `@deepseek-ai/dsh-web-app`. If you remember one sentence from this section, remember this one: DSH supplies the room; `@asd/ui` brings in the workbench furniture and wires it back to the canonical ASD engine.

That arrangement keeps the architecture honest. The shared ASD payload stays the shared ASD payload: 10 roles, 80 shipped skills from 81 authored sources, 30 canonical `sd_*` runtime tools, 228 corpus resources, 10 role prompts, 38 registered artifact-kind contracts, 39 catalog rows, and 7 authored templates. The DSH surface does not invent a second project store, a second review system, or a second fact-check engine. The adapter validates the compatibility seam, registers the canonical tools and skills, injects the shared role contract into the system prompt, mounts the Studio browser, and routes the work back into the same workspace-backed service layer used by CLI and MCP. In other words: same governed practice, different shell.

LAYER	WHAT IT OWNS IN THE DSH SURFACE
DeepSeek Harness `0.1.0-rc.5`	Profile lifecycle, slot system, loopback web host, conversation shell, settings host, sidebar host, and browser runtime.
`@asd/ui` `0.5.0`	ASD host adapter, Studio client, role prompt registration, skill registration, settings plugin item, conversation and sidebar integration, loopback RPC bridge, and compatibility enforcement.
`@asd/core`	Projects, evidence, artifacts, canvases, acceptance rules, Critic review, fact-check, phase-gate records, and ACP envelopes.
`@asd/engine`	The canonical 30 `sd_*` tool definitions and execution handlers.
`@asd/knowledge`	The method corpus, template catalog, artifact-kind contracts, provenance, and the 228-resource library exposed across surfaces.
`@asd/agents`	The ten-role registry and bounded dispatch instructions the host can register.
`@asd/runtime`	The shared CLI and MCP runtime used by non-DSH surfaces. It remains the reference engine; it is not a second Studio.

My take: I distrust host integrations that blur their seams. When I cannot tell whether a feature belongs to the shell, the adapter, or the shared engine, I know I will not be able to debug it six months later. The DSH surface is good precisely because the line is visible.

The package README states this plainly: this pointer is not the old monolithic plugin, not a standalone web app, and not evidence that every emitted host pack runs inside DSH. The documentation page repeats the same discipline. One installed `@asd/ui` package. One composed profile. One loopback Studio. One canonical tool registry behind it. That is a cleaner story than the earlier historical descriptions, and it is the story that survives contact with receipts.

Installing the Profile

The profile lifecycle has to use the host's official commands. No copying files into random folders. No editing the Harness checkout. No pretending a manifest pointer is an installer. The verified shape is the ordinary DSH lifecycle: add, update, remove, then launch the web host with the same profile. The placeholders below are intentional because the package or local bundle path is deployment-specific. What matters is the lifecycle shape, not a fake universal URL.

dsh-profile-lifecycle.sh

```
# Run from the DeepSeek Harness checkout.
dsh plugin --profile <profile> add <asd-ui-package-or-bundle>
dsh plugin --profile <profile> update <asd-ui-package-or-bundle>
dsh plugin --profile <profile> remove <asd-ui-package-or-bundle>

dsh web --profile <profile> --port <loopback-port>
```

For the pinned receipt, the profile name is `asd-live-review2`. The verified compatibility revision is `3f88a0f63f9788959b4e889b78653798bb7d6aae+asd-phase5-e94f006f2283`. I treat that revision string as a contract, not a convenience label. If my local host revision differs, I am outside the receipt. I can still experiment, but I am no longer entitled to reuse the claim that the rc.5 seam was proven.

The release gate is equally concrete. Official add/update/remove/reinstall must succeed. The six required slots must be present. The package must still be portable after install. The Studio must load on loopback. The project count and workspace root must be the expected disposable values in the smoke run. ASD-owned accessibility violations at serious or critical level must stay at zero for that smoke. Shutting down the server must remove the temporary workspace created for the check. That is why the profile story is worth documenting. It is lifecycle evidence backed by a receipt from a real machine.

> **CORE PRINCIPLE:** Treat the host's installer as part of the product surface. If add, update, remove, or reinstall is shaky, the integration is shaky, even if the UI looks polished after a lucky boot.

There is another quiet benefit to doing it this way. Because the adapter bundle carries its own inventory and compatibility expectations, the boot can fail loudly when the Harness seam drifts. The code checks the exact rc.5 package versions it depends on, validates the role, skill, template, and corpus inventories, and refuses to register the web routes when the host is not loopback. That is the correct failure mode. A partial Studio that appears to work is worse than a hard stop.

Studio Slots: Activity, Conversation, Settings

The receipt records six host slots: `sidebar.activity.action`, `sidebar.activity.panel`, `conversation.view`, `conversation.showView`, `settings.plugin.item`, and `layout.openActivity`. Three of those are user-visible surfaces. Three are the navigation seams that make the visible surfaces coherent. Together they let ASD appear as a first-class citizen inside the Harness instead of as a floating afterthought.

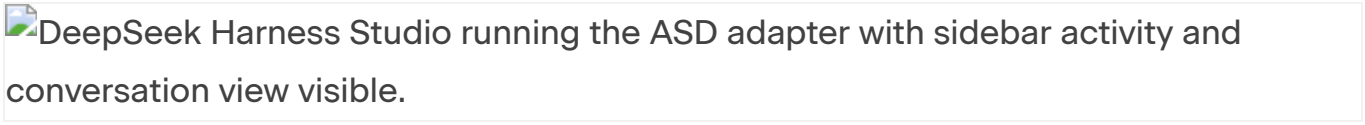

DeepSeek Harness Studio running the ASD adapter with sidebar activity and conversation view visible.

Figure 20.1 **ASD mounted inside the full shell.** This capture proves that the Service Design activity and the Studio conversation view coexist inside the DeepSeek Harness shell; it does not prove that every provider, chat trajectory, or role dispatch path was exercised in that run.

SLOT	FUNCTION	WHAT I EXPERIENCE IN THE UI
`sidebar.activity.action`	Registers ASD as a sidebar activity.	A Service Design button appears in the sidebar activity rail.
`sidebar.activity.panel`	Registers the keyed panel body for that activity.	Clicking the Service Design button opens the docked workbench panel.
`conversation.view`	Adds ASD as a center-column conversation tab.	The Studio appears beside Chat and the host's other conversation views.
`conversation.showView`	Switches a scoped session into the Studio view.	*Open in Studio* deep links can jump the user from a tool card into the right workbench tab.
`settings.plugin.item`	Registers the ASD settings card.	Settings → Plugins shows the ASD entry where workspace root and default phase live.
`layout.openActivity`	Opens the sidebar activity programmatically.	Project or artifact deep links can open the panel to the correct section instead of dumping me at a generic home state.

The important thing is that the Studio is mounted twice over one shared store. I can open it as a center-column tab or as a sidebar activity panel, and both surfaces read the same project, evidence, artifact, review, and report state. Article 36 describes this well: tool cards rendered in the chat for `sd_project_create`, `sd_evidence_add`, `sd_artifact_write`, `sd_artifact_get`, `sd_artifact_list`, and `sd_review` can carry an *Open in Studio* action that lands on the exact project or artifact the tool call touched. That is workbench design, not screen dressing. The browser and the tool runtime are pointing at the same object.

There are currently three user-facing entry points I rely on: the sidebar Service Design activity, the Studio conversation view, and the Settings → Plugins card. Those are the surfaces I touch. The other two names that sound more abstract — `conversation.showView` and `layout.openActivity` — are why the handoff feels integrated instead of clumsy. Without them, every click-out from chat would strand me in the wrong place.

Loopback Routes and the RPC Allowlist

The workbench lives on two loopback routes only: `/asd/app` serves the static Studio assets and `/asd/rpc` exposes a bounded JSON-RPC channel. The adapter refuses to register either route unless the Harness web server host is exactly `127.0.0.1`. That is not a suggestion. It is a boot-time guard. If the host is bound elsewhere, `@asd/ui` stops. I like that decision because it makes the threat model legible. This Studio is a local workbench, not a quietly internet-facing admin surface.

The static side is deliberately dull. The handler serves the Studio files under `/asd/app`, returns `index.html` for the bare prefix, sets the right content types, and rejects traversal attempts such as raw `/asd/app/../../../../etc/passwd` requests with a 404 instead of walking out of the asset root. It also rejects unsafe verbs. That sounds mundane. Good. Static serving should be boring.

The interesting boundary is the RPC channel. The source code labels it a closed allowlist: no eval, no passthrough, and nothing that writes. The methods validate their own parameters and read through `AsdService`. The allowlist covers project summaries, artifact and evidence reads, method and template browsing, review rows, canvases, fact-check projections, orchestration status, report generation, and related read models. Every browser request must be a `POST` with JSON content type. Cross-site browser requests are rejected. If an `Origin` header is present, it must be the loopback Harness origin on `http://127.0.0.1:<port>`. Request bodies are capped at 64 KB. Bad input resolves to a client error instead of a silent fallback.

loopback-routes-probe.sh

```
PORT=3788

curl -i "http://127.0.0.1:${PORT}/asd/app"

curl -i \
  -X POST "http://127.0.0.1:${PORT}/asd/rpc" \
  -H 'content-type: application/json' \
  --data '{"method":"getProjectsSummary"}'
```

The read-only posture of `/asd/rpc` is easy to miss if you only look at screenshots. The Studio can inspect and coordinate. The canonical tool runtime still owns mutations. That is the right split. I do not want an ad hoc browser method sneaking around the acceptance gate or the review contract. If the browser needs to show a review board, it asks for review rows. If I need to write an artifact, review it, or advance a phase, I go through the canonical operation path where the same rules fire as they do in CLI and MCP.

> **CAUTION:** Loopback does not mean “security solved.” Any local browser surface still deserves explicit origin checks, verb limits, parameter validation, and size caps. The DSH adapter earns trust because the route boundary is narrow, not because it is visually hidden.

Capability Projection: 24 Rows, One Unsupported

The capability projection is where the DSH workbench stops being a vibe and starts being a machine-readable contract. The current `capabilities.json` for the DSH surface contains 24 rows. Twenty-three are supported. One is explicitly unsupported: `phase.gate.status`. That unsupported row is a feature, not an embarrassment. It tells me the boundary is enforced by proof rather than optimism.

The supported rows are not a claim that the Studio has 23 shiny buttons and therefore "covers the whole platform." They say something more technical and more valuable: if the Studio advertises one of these operations, it executes the exact canonical tool definition named in the row. The source lists them one by one: `project.list`, `project.create`, `project.open`, `project.status.set`, `evidence.add`, `evidence.import`, `artifact.list`, `artifact.get`, `artifact.write`, `canvas.add`, `canvas.list`, `canvas.remove`, `method.list`, `template.list`, `template.get`, `schema.get`, `review.list`, `review.run`, `factcheck.run`, `orchestration.status`, `orchestration.dispatch`, `orchestration.handoff`, and `phase.advance`.

Two absences matter. `method.run` is a canonical tool concept, but it is not a separate direct Studio capability row. Neither is `evidence.query`. The browser can still expose supported query surfaces for those concepts, but clients must not invent new direct capability IDs. And when a client asks for something unsupported or unknown, the adapter fails closed. The documentation names the error contract clearly: `CAPABILITY_UNSUPPORTED`, no silent downgrade, no "close enough" reroute to some other method. The core capability source puts the same philosophy in code: *return a supported operation or fail loudly; never silently downgrade*.

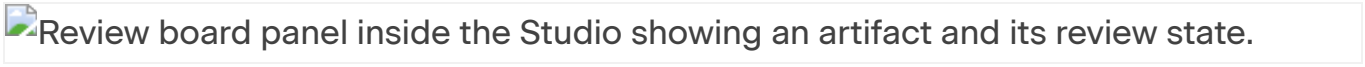

Review board panel inside the Studio showing an artifact and its review state.

Figure 20.2 **The review board inside the workbench.** This capture proves that review state, acceptance re-runs, and linked evidence are surfaced inside the Studio; it does not prove that every host/provider combination completed the full case or that a visible row is automatically final and exportable.

CAPABILITY ROW	CANONICAL TOOL	STATUS	WHY IT MATTERS
`project.list`	`sd_project_list`	supported	Studio dashboards read the same project inventory as every other surface.
`project.create`	`sd_project_create`	supported	New projects are canonical records, not

CAPABILITY ROW	CANONICAL TOOL	STATUS	WHY IT MATTERS
			browser-local drafts.
`project.open`	`sd_project_open`	supported	Opening a project resolves against the shared workspace.
`project.status.set`	`sd_project_status_set`	supported	Kanban lifecycle moves ride the canonical status tool; legal transitions are enforced by the engine, not the board.
`evidence.add`	`sd_evidence_add`	supported	Evidence writes still hit the canonical kind and provenance rules.
`evidence.import`	`sd_evidence_import`	supported	Bulk capture remains an engine operation, not a custom UI loophole.
`artifact.list`	`sd_artifact_list`	supported	Artifact indexes stay consistent across Studio, CLI, and MCP.

CAPABILITY ROW	CANONICAL TOOL	STATUS	WHY IT MATTERS
`artifact.get`	`sd_artifact_get`	supported	The browser reads the same stored artifact object the tools wrote.
`artifact.write`	`sd_artifact_write`	supported	The acceptance gate stays on the write path.
`canvas.add`	`sd_canvas_add`	supported	External canvases are registered in the project store, not in browser memory.
`canvas.list`	`sd_canvas_list`	supported	The Studio can read linked canvases through the shared registry.
`canvas.remove`	`sd_canvas_remove`	supported	Canvas cleanup is explicit and canonical.
`method.list`	`sd_method_list`	supported	The method browser is reading the same corpus-backed manifest list.

CAPABILITY ROW	CANONICAL TOOL	STATUS	WHY IT MATTERS
`template.list`	`sd_template_list`	supported	Template discovery is shared across surfaces.
`template.get`	`sd_template_get`	supported	The authored template body comes from the shared pack.
`schema.get`	`sd_schema_get`	supported	Kind contracts are surfaced through the same schema path.
`review.list`	`sd_review_list`	supported	The board is a projection of the real review state.
`review.run`	`sd_review`	supported	Pass and veto remain canonical review operations.
`factcheck.run`	`sd_factcheck`	supported	The Studio can show deterministic gate output, not a separate browser opinion.

CAPABILITY ROW	CANONICAL TOOL	STATUS	WHY IT MATTERS
`orchestration.status`	`sd_orchestration_status`	supported	Plan traces are shared and inspectable.
`orchestration.dispatch`	`sd_orchestration_dispatch`	supported	Dispatch state is persisted, bounded, and visible.
`orchestration.handoff`	`sd_orchestration_handoff`	supported	Handoffs remain typed and recorded, never mere chat.
`phase.advance`	`sd_phase_advance`	supported	Phase movement is still explicit and receipt-bearing.
`phase.gate.status`	—	unsupported	The engine exposes phase advance, not a separate canonical direct status capability. The adapter says so instead of pretending otherwise.

If you inspect the review board in Figure 20.2, you can see why this contract matters. Review is not decorative chrome around chat. The row carries status, gate, verdict, linked log artifacts, and acceptance failures. A refused write can sit there as a draft

with explicit reasons. A passed review can sit there with evidence-cited rows. The board is useful because the operations beneath it are named and bounded.

Portable Bundles and Isolation

The DSH receipt is strong because it verifies properties most teams would leave implicit. It does not stop at “the plugin installed.” It pins the bundle hash, counts the bundle entries, checks for absolute paths and symlinks, proves byte-identical output across distinct checkout roots, records the official remove-and-reinstall cycle, and confirms that the live smoke used a disposable workspace that vanished cleanly when the server stopped. That is the level of evidence I want for a workbench that is supposed to travel between machines and profiles.

dsh-receipt-summary.json


```
{
  "package": {
    "name": "@asd/ui",
    "version": "0.5.0",
    "bundleSha256": "8c7aa34f02c1df50030cf2faa6c0dd23098c38afbf1248d8607ef07fc815748
    "entryCount": 502,
    "absolutePaths": false,
    "sourceTreeImports": false,
    "symlinks": false
  },
  "verification": {
    "uiTests": 120,
    "workspaceTests": 701,
    "bundleSafety": "pass"
  },
  "profile": {
    "name": "asd-live-review2",
    "slotCount": 6,
    "officialRemoveReinstall": "pass"
  },
  "liveSmoke": {
    "projectCount": 0,
```

```
      "pluginOwnedSeriousOrCriticalAxeViolations": 0,
      "serverStopped": true,
      "temporaryWorkspaceRemoved": true
    }
  }
```

The portable-bundle claim has three parts. First, the archive is self-consistent: 502 entries, no absolute source paths, no source-tree imports, no symlinks. Second, the build is stable: the receipt records byte-identical outputs from distinct absolute roots and a stable SHA-256 for the package bundle. Third, the runtime footprint is disposable: the smoke runs with a loopback server, an empty project count, and a temporary workspace that is removed on shutdown. That is what I mean by an integrated workbench. It is not merely mounted inside the shell. It is installable, movable, inspectable, and removable without dragging half a source tree behind it.

Accessibility belongs in the same proof set. The receipt records zero ASD-owned serious or critical accessibility violations in the live smoke. That phrase is carefully scoped. It does not mean the whole Harness had zero violations. The receipt even records bounded host-owned exceptions separately. I appreciate that level of honesty. When a platform tells me what belongs to it and what does not, I can improve the right thing instead of laundering host defects through the plugin's success story.

> **My take:** Portable bundle safety sounds like release-engineering housekeeping until the day you need to prove that a "plugin" was not quietly importing half your checkout from `/Users/<checkout>`. Then it becomes design governance.

The Center-Tab Studio

Until the Studio v2 merge, the center tab was a respectable read surface: project summaries, artifact views, the review board. Useful, but passive. Article 38, *The Center-Tab Studio: Live Service Design Journeys Inside the DeepSeek Harness Shell*,

is now live, and it documents what the merge changed: the center tab became an engagement surface in its own right. Three workspaces share it — a kanban board for the engagement lifecycle, a knowledge-base Library browser, and a canvas preview workspace. Each reads the same workspace-backed store the rest of this chapter describes. None of them forks it.

The kanban board renders every engagement as a project card carrying its design phase, draft and veto badges projected from the review state, and the latest status note. Moving a card is a first-class operation, not a visual trick. Drag-and-drop and the keyboard Move menu both call the `project.status.set` capability row, which executes `sd_project_status_set` under its legal transition rules: `briefed → active → gated → blocked → archived`, `archived` reachable from any state, `blocked ↔ active`, and same-status calls treated as no-ops unless a note accompanies them. The board updates optimistically so the interaction feels immediate, and it rolls back visibly when the engine refuses a transition. The property I care about is that the refusal path exists at all. A board that let me drag a project into any column would be a board I could not trust.

The Library browser turns the 228-resource knowledge base into something navigable from inside the shell: methods, templates, and article READMEs, each carrying its provenance. The detail that earns its place is cited use-in-project — a Library entry shows which engagements have actually applied it, so reuse is visible evidence instead of an anecdote about a method being popular. The canvas preview workspace does the equivalent job for canvases. Compare mode puts two states of a canvas side by side. The evidence spotlight rail keeps the cited records beside the canvas instead of a click away. SVG and PNG export take a view into a deck or a review pack without a screenshot of a screenshot.

Article 38 walks this surface end to end on two paired engagements: Aurelia Trade Finance, a trade-finance modernisation, and Meridian Oncology Partners, an oncology-network coordination problem. Both are fictional composites, per the series convention, and the run is bounded in the same way the Wattle Grove walkthrough below is bounded. The Studio v2 surface is demonstrated in that bounded run, and the per-kind typed schemas for richer artifact views remain Phase 2 work. What the article proves is the loop — board, Library, canvas, review — holding together over

one shared store. It does not prove that every artifact kind already has a bespoke typed renderer.

> **My take:** A kanban board over a governed store is the first project-tracking UI I have actually enjoyed working in, because the columns are enforced by the engine. The drag is optimistic. The truth is not.

The Live Build, Running

Everything above this line describes the integration as a contract. As I write this, the contract is running on my desk: the `asd-studio-v2` profile serving on `127.0.0.1:4685`. The host's own boot manifest is machine-checkable, so I checked it. The host booted at revision `ad3ba0951ddb` with 39 plugin entries, and among them sits `@asd/ui` at revision `367432b4e475`, its client served from `/plugins/@asd/ui/client.js` at roughly 707 kilobytes. Reading the served bundle itself confirms the surfaces this chapter describes by name: the Agentic SD Studio center tab registered through the `conversation.view` seam, the Kanban lifecycle including the *briefed* column, the Library with its cited *Use in project* action, the canvas workspace, and the jobs-to-be-done kind. That is source-level proof of what the host is running. It is not, by itself, proof of what a model turn did — that proof stays with Article 38's receipts. Two different questions, two different evidence kinds, and I keep them apart the way I keep the DSH receipt apart from the bounded prior workflow evidence.

Four pieces of the v2 build deserve their own paragraphs here because they changed how I actually work, and none of them is a new place to put a button.

The composer knows the engagement. The input dock chip reads the project — name, phase, gate state, draft count — instead of just its label, and its Project-actions menu drafts phase-aware instructions: a "suggest next step" framed by the orchestrator with an explicit wait-for-confirmation, or a gate report. The posture is the

same one the create form established: the Studio drafts, the human reads, edits, and sends. Nothing fires on a click that I have not read first.

The guided interview. Selecting a template or a method in the Studio used to mean hoping the model guessed the context. Now it starts an interview. The drafted instruction tells the model to review existing project evidence first, ask me focused questions one at a time, and only then write the artifact, run the critic, and report the verdict — with a hard guard requiring at least three answered questions before any artifact is created. All seven authored templates, the original six plus jobs-to-be-done, open their interview with a question. In the acceptance run, one journey-map interview ran five question rounds before it produced anything. The interview is where the model's ignorance becomes visible early, while it is still cheap.

The right panel, taken seriously. The host reserves a single details seat beside the conversation, and the contract says one occupant renders there. The v2 build takes the seat deliberately: an artifact write opens its sanitized SVG render with badges and an Open-in-Studio deep link; a review opens its verdict card; a factcheck opens a pass/fail table; anything else falls back to the host's default faithfully. The receipt line I keep retelling: the first live boot of this build failed loudly because the host already held the seat at priority 0, and the fix was to register at −10, because the lowest rank renders. Activation, not registration, is where takeover seams bite — this chapter has now shown that twice.

Six new kinds. The artifact-kind registry grew from 32 to 38: jobs-to-be-done, day-in-the-life, touchpoint-inventory, service-scenario, expectation-map, actor-network-map. Each carries a closed schema, a deterministic renderer, and a method manifest, and each entered through the same gated write path as the original 32 — the live turn wrote two of them through it. The kind count is the least interesting number in that sentence. The interesting part is that expansion did not require a new door.

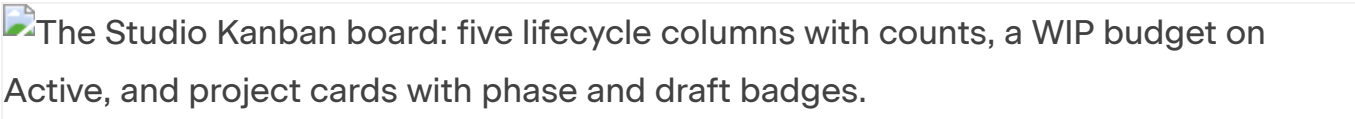

The Studio Kanban board: five lifecycle columns with counts, a WIP budget on Active, and project cards with phase and draft badges.

Figure 20.3 **The engagement Kanban.** Five lifecycle columns, a WIP budget on Active, drag handles and Move-to menus that execute `sd_project_status_set` — never a direct write. Captured 2026-08-23 against the studio v2 build; see Article 38 for the receipt.

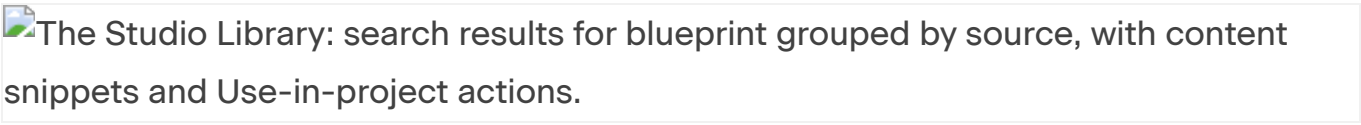


Figure 20.4 **The Library as a human surface.** The 228-resource corpus the model reads, searchable, with every hit carrying a provenance-cited *Use in project* action that drafts a librarian instruction into the composer. Same capture run as Figure 20.3.

> **My take:** A build you can interrogate is worth three demos you can watch. The boot manifest answers *what is running* in one curl. The receipts answer *what it did*. A workbench chapter that cannot answer both, separately, is marketing.

Case: Wattle Grove Community Care

Wattle Grove Community Care is the right case for this chapter because it shows both the promise and the discipline of the workbench. The client is fictional and composite. The evidence is synthetic and declared as such. The current DSH integration receipt and the bounded prior workflow evidence are deliberately separated. That separation is the whole lesson. The workbench lets me run a serious service-design engagement in one host surface, but it also forces me to label exactly which parts were proven by the current rc.5 integration and which parts were proven earlier through bounded workflow receipts.



Figure 20.5 **The ASD settings entry.** This capture proves that `@asd/ui` registers as a first-class Plugins settings item with ASD-owned configuration; it does not prove that another DSH revision or another profile inherits the same compatibility receipt.

I like to begin the walkthrough with the settings surface because it tells me the integration is real before I even touch the case. The ASD plugin item is where the workbench exposes its own namespace and lets me set workspace root and default phase. From there, the Wattle Grove engagement follows the shared ASD loop. The designer creates the project, captures eight evidence records, pulls the relevant

methods, writes typed artifacts, reviews them, fact-checks them, and stops at a human gate.

The service problem is concrete. Wattle Grove is an in-home aged-care provider with roughly 1,900 clients and 240 support workers. Clients keep seeing unfamiliar workers. Families are not told about changes. Schedulers trade continuity away whenever coverage gets tight, because an uncovered visit is a contractual breach while continuity is a quality the organisation cares about but does not measure. The evidence pack for the case is small on purpose: two client interviews, two family interviews, two support-worker interviews, one scheduler interview, and one operational telemetry figure. The workbench does not need a fake data mountain. It needs cited records that can anchor each artifact node.

From there the method sequence is the canonical one. Stakeholder mapping comes first because the quotes come from four different groups whose interests conflict. The resulting map has six stakeholders. The care recipient and family carer have the highest lived stake, but the scheduler and the regulator carry the highest operational influence. That asymmetry matters because it tells me continuity failures are not random UX glitches. They are downstream of backstage incentives.

The journey map then renders the service as six evidence-backed stages. The quotes are the story. *“I get dressed and wait by the door from eight.” “I have never met most of these clients and there are no notes.” “Eleven different girls came. I stopped telling them where things are kept.” “Nobody comes until eleven and no one rings.” “I only find out Mum had a new carer when she tells me she was upset.” “I took the day off. They arrived at 2pm. I lost a day’s pay.”* Read those together and the case becomes legible. The platform is not inferring a sentimental narrative. It is forcing each stage to resolve back to an opened record.

The blueprint deepens the diagnosis. Article 36 reports twenty nodes across five lanes, with eleven fail or wait points. Worker arrives with no client context. Run-sheet generation excludes care-plan data. Client meets a stranger again. No dynamic resequencing or overrun alert. Family not notified. Compliance measures completion, ignores continuity. Complaint signal never reaches rostering policy. That is a workbench doing actual service design. I can move between the conversation, the

evidence register, the artifact view, and the review board without losing the thread because the store underneath is one store.

Then comes the moment I care about most. Somebody asks for a better story than the evidence supports. In the bounded prior workflow, the designer explicitly asks for a current-state journey stage saying the client receives a proactive evening SMS with the worker's name and photo and feels reassured, even though there is no evidence for that state. The platform refuses the write. This is the kind of refusal I want from an integrated workbench: precise, persistent, and public to the human reviewer.

probe-refusal.json


```
{
  "accepted": false,
  "status": "draft",
  "artifact": {
    "id": "ef0c010c06af4559a1c0ad588c4a0fba",
    "kind": "journey-map",
    "status": "draft",
    "check": {
      "pass": false,
      "failures": [
        { "rule": "min-nodes", "message": "expected at least 3 nodes, found 1" },
        { "rule": "min-evidence-per-node", "nodeId": "stage-proactive-sms", "message
        { "rule": "field-non-empty", "nodeId": "stage-proactive-sms", "message": "no
      ]
    }
  }
}
```


The phrasing matters. The write is refused because `accepted` is false. The artifact is also persisted because the status is still `draft`. The failed attempt remains visible on the review board with the machine-readable reasons attached. Nothing has been silently "fixed." Nothing has been silently discarded. Nothing has been silently promoted. The human holding the gate can see the exact attempted distortion and the exact reason it failed. That is why the review board in Figure 20.2 matters so much. It is where the workbench surfaces the moral discipline of the engine.

> **My take:** A gate that only says no to the model is theatre. The Wattle Grove refusal matters because it said no to the person holding the keyboard.

Remediation happens in place. The same artifact can be rewritten with valid cited nodes, then reviewed with an evidence-cited pass that promotes it to final. A veto does the opposite: it demotes the artifact to draft and writes a linked `review-veto-log`. And the phase gate still belongs to a human. The workbench prepares the decision, shows the evidence, and records the outcome. It does not seize authority.

That is why Wattle Grove is a good workbench case. It is not a glossy demo about a plugin opening a pane. It is a bounded proof that the Studio can carry a real engagement shape — project, evidence, methods, artifacts, reviews, reports — while preserving the harder property: refusal of unevidenced improvement theatre.

Receipt Boundaries

By this point you should have a feel for the workbench. Now I want to state the limits as sharply as the capabilities. The receipt proves one profile composition for DSH `0.1.0-rc.5` with `@asd/ui` `0.5.0`, profile `asd-live-review2`, the pinned compatibility revision, six host slots, official lifecycle success, portable bundle safety, 120 UI tests, 701 suite tests, loopback Studio smoke, workspace isolation, and zero ASD-owned serious or critical accessibility violations in that smoke. It also links bounded prior workflow evidence showing the evidence → artifact → review/remediation → pass → human-gate shape.

It does **not** prove every provider and chat dispatch combination. It does **not** prove every external MCP client inside the host. It does **not** prove every future rc build, every DSH profile, or every possible role execution path. It does **not** turn ACP into a transport; ACP remains record-only. It does **not** make BUZZ messages or Canvas

documents appear in the ASD workspace automatically. And it does **not** let me round up from one bounded case to a universal “the host is fully live” claim.

THE RECEIPT PROVES	THE RECEIPT DOES NOT PROVE
Official add/update/remove/reinstall works for the pinned profile.	Every DSH profile or future revision will behave identically.
The Studio mounts through six required host slots and serves on loopback.	Every provider or chat path in the host has been exercised end to end.
The direct capability projection is 24 rows with one explicit unsupported row.	Unsupported capabilities are secretly available through some hidden fallback.
The `@asd/ui` bundle is portable: 502 entries, stable SHA-256, no absolute paths, no symlinks.	Any local experiment outside the pinned compatibility revision inherits the same portability claim automatically.
The live smoke passed with zero ASD-owned serious or critical accessibility violations and disposable workspace cleanup.	The entire host shell had zero issues, or that accessibility is “finished.”
Wattle Grove shows the governed workflow shape and the write-path refusal behavior.	Current host chat/provider dispatch completed the Wattle Grove engagement end to end.

That last row is the one I care about most. Boundary honesty is not a legal disclaimer stapled on after the fact. It is part of the product design. When a workbench tells me what it cannot yet prove, I can trust the things it says it has proved. DSH and `@asd/ui` earn that trust here. The shell is real. The adapter is real. The workbench is real. The limits are real too. That combination is what makes this surface useful enough to rely on.

CHAPTER 19

BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 21

SOPs-as-Code: The AI Execution Playbook

NEXT CHAPTER

SOPs-as-Code: The AI Execution Playbook

Methods become executable, versioned, reviewed

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 21

SOPs-as-Code: The AI Execution Playbook

Methods become executable, versioned, reviewed

Reading time

13 min

From Wiki Pages to Manifests

Most design teams already have SOPs. They just do not call them that. They live in onboarding docs, copied prompts, old Miro boards, and the phrases senior people repeat in review: “show me the quote,” “split current from future state,” “do not merge the journey and the blueprint,” “who owns this change.” That is useful craft, but it is a terrible runtime. A new designer cannot inspect it cleanly. A machine cannot execute it safely. CI cannot fail it before a weak artifact reaches a client. The result is uneven quality disguised as creative freedom.

ASD treats the method itself as a design object. The machine source of truth lives in manifest files under the knowledge package. The readable projection lands in the

portable `SKILL.md` surface. The runtime then enforces the boring parts humans routinely forget under deadline pressure: evidence minimums, non-empty fields, forbidden placeholders, review gates, typed handoffs, and a deterministic fact-check before any human phase advance. That is the important shift. The library is not a folder of admired templates. It is an executable operating discipline.

Once I encode a method this way, three things change immediately. First, the method becomes inspectable. A junior designer can see the real contract instead of reverse-engineering it from examples. Second, the method becomes testable. I can run fixtures, compare runs, and tighten failure classes. Third, the method becomes governable. When something drifts, I can point to the manifest, the review rows, the factcheck output, and the commit that changed the rule. The practice stops relying on memory and starts relying on versioned judgment.

> **My take:** A prompt is not an SOP. A prompt tells a model how to talk. An SOP tells a practice what must be true before the talking counts.

That distinction is why article 29 mattered so much in the series. It showed the framework article itself becoming the client: one lead designer, one part-time design-ops owner, three reviewers, and a growing library of methods that had to survive live runs across sectors. The lesson was blunt. Prose-only methods create prose-only quality. Once the method has a machine tail, the practice can improve without depending on who happened to brief the agent that day.

Anatomy of a Method Manifest

The shipped runtime source of truth today is JSON. I use YAML here because it is the clearest teaching format for the anatomy. The engine reads machine manifests, not nicely formatted examples. The key idea is simple: every field exists because something downstream needs it. Inputs tell the runtime what must already be in the

project. Steps preserve sequence and role ownership. The output schema constrains shape. Acceptance criteria turn quality into checks. The review gate makes approval explicit instead of cultural. If a field cannot change runtime behavior, it probably does not belong in the contract.

service-blueprinting.manifest.yaml

```
id: service-blueprinting
phase: synthesis
owner: mapper
inputs:
  - kind: evidence-set
    min: 4
    note: frontline observations and process documentation
steps:
  - role: mapper
    instruction: Map customer actions in order and cite evidence for each node.
  - role: mapper
    instruction: Add physical evidence, frontstage, backstage, and support-process l
  - role: critic
    gate: true
    instruction: Reject placeholder lanes or uncited nodes.
output_schema:
  kind: service-blueprint
  required_fields:
    - lane
    - label
    - evidence_refs
acceptance_criteria:
  - type: min-nodes
    value: 4
  - type: min-evidence-per-node
    value: 1
  - type: field-non-empty
    field: lane
  - type: forbid-field-values
    field: lane
    values:
      - unknown
      - tbd
      - n/a
review_gate: critic
```

That example is deliberately small, but it already captures the difference between a method description and an executable contract. “Blueprint well” is vague. “Reject placeholder lanes or uncited nodes” is enforceable. The service-blueprinting flagship skill adds depth about lane craft, current-versus-future separation, and failure points. The manifest keeps the machine truths: minimum evidence, required fields, and the fact that a Critic veto keeps the artifact in draft. Both layers matter. The human-readable flagship adds judgment. The manifest defines what the runtime can actually police.

MANIFEST PART	WHY IT EXISTS	WHAT BREAKS WITHOUT IT
`inputs`	Declares required evidence or upstream artifacts before a run starts.	Agents silently invent missing context.
`steps`	Preserves role sequence and explicit gate moments.	Outputs become style-driven instead of method-driven.
`output_schema`	Constrains the shape of what can be written.	Artifacts sprawl into persuasive prose with no stable structure.
`acceptance_criteria`	Turns “quality” into pass/fail checks on write.	Review degrades into taste and memory.
`review_gate`	Names who must inspect the draft before it can be final.	Agent output slides straight into delivery culture unchecked.

The deeper point is that manifests do not eliminate judgment. They relocate it. The designer still decides where the human gate sits, which fields are non-negotiable, what counts as evidence, and which failures must fail closed. That is exactly the right layer for human craft. I do not want humans wasting judgment on whether a lane is blank. I want them spending judgment on whether the line of visibility is drawn honestly, whether the evidence is structurally representative, and whether the service change should exist at all.

> **CAUTION:** The brittle version of SOPs-as-code is as dangerous as the loose version. If you over-specify every sentence, the method becomes unusable across contexts. Fix the

> invariants. Leave interpretation only where evidence-backed judgment is legitimate.

The 80-Skill Library by Phase

The library currently ships **80 skills from 81 sources**. That distinction matters. There are 80 generated projections plus one authored flagship source, but only 80 shipped skills because `service-blueprinting` is present twice in source form and only once in the emitted pack. The compiler applies a flagship-wins rule: it walks the authored flagship tree before the generated tree and takes first-write-wins for the same skill id. That is how the platform keeps one executable contract while still allowing depth where a method genuinely needs more human judgment. The library also grew with the Studio v2 artifact kinds, jobs-to-be-done among them, so the phase counts below reflect the current shipped set rather than the original release.

The phase distribution is not cosmetic inventory. It tells you how the practice is weighted. Research and synthesis remain the thickest layers because evidence quality still determines almost everything downstream. Delivery and measurement are narrower in count, but they are where governance gets real: standards, rollout plans, audits, service recovery, KPI trees, and the gates that keep a change honest after the prototype applause dies down.

PHASE	SHIPPED SKILLS	TYPICAL EXAMPLES	WHAT THE PHASE CONTRIBUTES
research	19	`contextual-inquiry`, `transcript-mining`, `shadowing-frontline`	Turns raw field evidence into reliable inputs.
synthesis	15	`affinity-diagramming`, `behavioural-personas`, `service-blueprinting`	Converts evidence into structures the team can reason over.
ideation	13	`hmw-formulation`, `orchestration-canvas`, `story-`	Generates options without losing the trace

PHASE	SHIPPED SKILLS	TYPICAL EXAMPLES	WHAT THE PHASE CONTRIBUTES
		`mapping`	back to evidence.
prototyping	12	`wizard-of-oz`, `service-roleplay`, `cognitive-walkthrough`	Tests hypotheses before rollout claims harden.
delivery	10	`frontline-enablement`, `service-standard-authoring`, `wave-gated-rollout`	Turns concepts into governed operating change.
measurement	11	`good-services-audit`, `kpi-tree-design`, `service-recovery-design`	Measures whether the service actually got better.

The twelve-skill canon the series leans on makes the architecture even clearer. Three research methods, six synthesis methods, one ideation method, one delivery method, one measurement method. The canon is smaller than the whole library. It is the recurring spine pulled by `sd_method_run` across articles and cases. In other words: the 80-skill count proves breadth; the canon proves continuity.

skills-library.snapshot.json


```
{
  "skillSources": 81,
  "skillsShipped": 80,
  "phaseCounts": {
    "research": 19,
    "synthesis": 15,
    "ideation": 13,
    "prototyping": 12,
    "delivery": 10,
    "measurement": 11
  },
  "flagshipReplacement": {
    "id": "service-blueprinting",
    "rule": "flagship-wins",
    "generatedTwinShipped": false
```

```
    }
  }
```

This is also where the flagship authored skill earns its keep. The generated twin for `service-blueprinting` is perfectly serviceable as a machine projection. The authored flagship adds what the manifest cannot carry elegantly: why the line of visibility is a design decision, why current and future state must be separate artifacts, why backstage cannot be a dumping ground, and why recovery rows matter. ASD keeps both source forms in agreement, then ships the deeper one. I like that pattern because it respects both sides of the practice: determinism where machines need determinism, prose depth where humans need craft.

Evolving SOPs Through Review

The wrong mental model is “we wrote the SOP, so the problem is solved.” No. An SOP library is a live service. It takes change requests, accumulates defects, gets challenged by new domains, and eventually retires bad assumptions. Article 29 made this explicit with five gates: source stability, executable shape, fixture survival, publish readiness, and operating review. That is the right governance pattern because method quality does not fail in one place. It fails at the source, the schema, the live run, the release ritual, and the telemetry trend.

When I change a method, I am changing institutional behavior. So the review has to be more serious than “this wording feels clearer.” The creator authors the change. The reviewer tests method integrity. The approver owns production risk. If the run depends on live consent, political trade-off, clinical or legal judgment, or other irreducibly human accountability, the manifest says so. The method can still exist, but as a human-run checkpoint rather than an autonomous agent routine. That boundary is not a limitation. It is design maturity.

sop-library.pull-request.yaml

```
change:
  skill_id: service-blueprinting
  version: 1.3.0
  creator: service_design_practice
  reviewer: critic
  approver: journey_owner
  required_checks:
    - generated-skill-sync
    - fixture-regression
    - provenance-review
    - factcheck-green
  release_rule: reject-if-any-required-check-fails
```

The authored-versus-generated split sharpens this review discipline. A generated skill changes when its manifest changes or the generator changes. A flagship skill can also change because the prose guidance got better, even while the machine acceptance stays the same. That means the team has to review two failure classes separately: runtime contract drift and judgment guidance drift. Good teams name that difference. Weak teams muddle it, then wonder why the artifact still passes the machine while the practitioner experience gets worse.

There is another advantage to this explicit review loop: institutional memory stops decaying into anecdote. Article 29 mined 31 draft SOPs from the corpus and accepted 26 into reusable form. That acceptance number matters because it proves selectivity. Five of those drafts stayed local because they were too context-bound. I would rather keep a method local than pretend it is generic when it is really a disguised memory of one project. Reusability is earned, not declared.

My take: The most valuable SOP fields are usually the boring ones. `approver`, `source_ref`, `review_cadence`, and `retirement_rule` do more to protect quality than another paragraph of lofty system-prompt prose.

CI Enforcement and Institutional Memory

Review alone is not enough because humans get tired, optimistic, and busy. The library needs a write path that enforces what the review claims to care about. ASD does this in layers. The method run returns the steps and acceptance contract. The artifact write checks structure and acceptance. The Critic review decides pass or veto. Then `sd_factcheck` replays integrity, acceptance, and review before the human phase gate. If that final check fails, CI fails. Cleanly. Loudly. With exit code 1.

ci-factcheck.sh

```
asd method-run --projectId framework --methodId service-blueprinting
asd artifact-write --projectId framework --kind service-blueprint --file artifacts/c
asd review --projectId framework --artifactId blueprint-001 --verdict pass --rows re
sd_factcheck framework
if [ $? -ne 0 ]; then
  echo "factcheck failed"
  exit 1
fi
```

The important subtlety is what happens on failure. ASD does not quietly discard the bad artifact. Failed acceptance persists as draft with machine-readable failure rows. That is a much better learning surface. The team can inspect the defect, repair the exact rule that failed, and rerun without pretending the mistake never happened. It is the same philosophy I want in service design more broadly: fail closed, keep the evidence, and make remediation explicit.

.github/workflows/sop-library-ci.yaml

```
name: sop-library-ci
on: [pull_request]
jobs:
  validate:
    runs-on: ubuntu-latest
    steps:
```

```
      - uses: actions/checkout@v4
      - run: node platform/compiler/generate-skills.mjs
      - run: node platform/compiler/emit-packs.mjs --output /tmp/asd-packs
      - run: sd_factcheck framework
```

That CI loop is not glamorous. Good. It should not be. Boring enforcement is exactly what lets designers spend energy on more interesting problems. Once the machine handles mermaid validity, field completeness, evidence minimums, registry integrity, and review-state consistency, the human reviewer can focus on the real questions: did we encode the right boundary, is the method still faithful to the books and proof cases, and does this automation leave dignity, safety, or accountability exposed?

Institutional memory then becomes cumulative instead of sentimental. Every merged method carries provenance. Every failure leaves a record. Every release has a diff. Every emitted pack can tell you what shipped. The library is still authored by people, but it no longer depends on people remembering what changed three quarters ago. That is the real dividend of SOPs-as-code. The practice starts compounding.

Case: The Framework Itself

Article 29 made the framework itself the case study, which is exactly why it works so well as the spine for this chapter. The team was small: one lead designer, one design-ops owner, three reviewers, and an AI stack used as execution surface rather than authority. The initial build took eight weeks. Then came six pilot iterations. The headline gains were not magical. They were procedural. First-pass SOP success moved from 50% in iteration 1 to 100% in iterations 5 and 6. Mean human review time fell from 44 minutes to 15. Junior-designer ramp to a safe first draft dropped from roughly four weeks to eight working days. None of that happened because the prompts got more lyrical. It happened because the method became inspectable, testable, and governable.

The blueprint worked example in article 29 is especially useful because it exposes the whole stack in miniature. Source methods and proof points were mined from the series. Mined patterns were accepted or rejected. A YAML contract was drafted. Fixture generators were built. A flagship blueprint SOP was tested against fixed cases and live runs. CI enforced validation. CODEOWNERS and approvers held releases. Telemetry then determined whether a method should be retained, revised, or retired. That is more than documentation. It is a service operating model for practice quality itself.

What I find most convincing is the humility built into the design. The article does not claim every method should be autonomous. It explicitly names the work that should stay human: interviewing, consent handling, politically sensitive reframing, final go/no-go choices, and other decisions where consequence outruns formal acceptance criteria. That honesty is one reason the framework scales. Teams trust a system more when it admits its boundaries.

> **CORE PRINCIPLE:** Encode the method deeply enough that the machine can fail safely, then stop. Everything beyond that boundary is design judgment, not a schema bug.

SOPs-as-code builds a practice where good judgment survives handoff, where new people inherit more than vibes, where a method can improve through review, and where AI acceleration stays accountable to evidence. That is the execution playbook. And once you have it, the rest of the platform starts to make more sense: the skill library, the Critic, the factcheck gate, the human phase receipt, even the closing argument of this book. They are all answers to the same question. How do we make rigor compound? It also gives the team a durable answer when a client, reviewer, or future maintainer asks what changed, why it changed, and which rule now enforces it.

CHAPTER 20

DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 22

Frontline Enablement & Employee Experience

NEXT CHAPTER

Frontline Enablement & Employee Experience

The internal service that makes external ones possible

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01

CHAPTER 02

The AI-First Service Design Operating System

The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 22

Frontline Enablement & Employee Experience

The internal service that makes external ones possible

Reading time

12 min

Internal Service Quality Drives External Value

Service designers sometimes talk about employee experience as if it were a morale initiative running beside the “real” customer work. Calder is a useful corrective because the business logic was brutally direct. Advisers in the pilot branches were spending 41 administrative minutes per customer interaction, switching tools constantly, making 6.8 internal chase actions per day, and still being expected to sound calm, competent, and local. Customer dissatisfaction was not arriving from nowhere. It was the downstream expression of internal service failure.

The right frame here is the internal service profit chain. Internal service quality shapes employee capability and confidence. That affects external value, which then affects

customer loyalty and growth. Calder's leaders finally stopped treating that as soft language because the baseline numbers were too expensive to ignore. Adviser eNPS sat at -18. First-contact quality was 62%. Repeat-customer NPS in the pilot branches was only +4. Staff turnover was trending badly. The frontline was acting as a shock absorber for organizational seams.

METRIC	BASELINE	PILOT OUTCOME	WHAT MOVED IT
Admin minutes per customer interaction	41	24	One start point, fewer duplicate notes, visible ownership
Tool switches per complex interaction	13	5	Federated desktop shell and rationalized task paths
Adviser eNPS	-18	+11	Reduced friction plus coaching infrastructure
First-contact quality	62%	79%	Approved-source knowledge with citations
Status inquiries per 100 active SME cases	34	16	Named-owner SLA tracker and receipt-confirmed progress
New-hire time to competence	16 weeks	9 weeks	Scenario labs, coaching cadence, clearer escalation paths

That table matters because it stops the internal-service conversation drifting into sentiment alone. Calder did care about morale. But morale improved because the service around the adviser improved. The bank reduced friction, restored visible ownership, and made knowledge retrieval safe enough to use under pressure. In other words, internal quality became an operating condition, not an HR slogan.

My take: If the frontline has to compensate for your organizational joins with memory, charm, and heroics, you do not have a people problem. You have a service-design problem hiding inside the org chart.

This is also why internal-service work belongs in the same evidence discipline as customer-facing work. The moment I say “employee enablement” without showing the task path, the backstage dependency, the metric, and the owner, I am back in poster territory. ASD helps because it forces the internal redesign to produce typed artifacts, explicit standards, and review gates. The bank cannot simply say “we trained people.” It has to show what changed in the service.

Frontline Blueprinting

The central move in Calder was to blueprint the employee experience with the same seriousness we usually reserve for the customer journey. The branch adviser became the primary actor for this internal service. That sounds obvious, but it changes the artifact deeply. The blueprint has to show what the adviser sees, what the bank asks them to do, what backstage teams owe them, what support processes make the work possible, and where invisible delays force dishonest promises. Once you draw that honestly, “training” stops being the main lever. The real levers are ownership, orchestration, knowledge access, and backlog priorities.

Calder's team resisted the seductive but false answer of “one system.” The evidence showed that a full platform replacement would take years and would probably fail politically. So the design team built what the article called the one useful lie of the unified adviser desktop: one place to start, one case context, one note spine, one approved knowledge lane, while the systems of record remain separate underneath. I like that phrase because it names the design move honestly. The frontline experience can simplify before the architecture is fully unified, but only if the simplification is disciplined rather than decorative.

internal-blueprint.json

```
{
  "artifactKind": "service-blueprint",
  "state": "future",
```

```
  "rows": [
    {
      "step": "S2 Prepare",
      "physicalEvidence": "Case timeline",
      "employeeAction": "Prepare for appointment once",
      "frontstage": "Context panel with products, notes, and status",
      "backstage": "Case and product APIs",
      "support": "Knowledge-content sync"
    },
    {
      "step": "S4 Escalate",
      "physicalEvidence": "SLA tracker",
      "employeeAction": "Promise honest next step",
      "frontstage": "Named-owner escalation and receipt state",
      "backstage": "Credit and compliance response queue",
      "support": "Internal SLA review"
    }
  ]
}
```

Notice what is happening in that excerpt. The customer is still present, but the blueprint is now exposing what the employee must rely on to keep the promise intact. The physical evidence is not a customer brochure. It is the case timeline, the answer card, the manager digest. The frontstage is more than a screen. It is the visible behavior the adviser can safely perform because the backstage is finally legible enough to support it.

INTERNAL SERVICE STEP	FRONTLINE PROMISE	BACKSTAGE DEPENDENCY	DESIGN IMPLICATION
Start the day	I know what needs attention first.	Aggregated open-case view across systems	Federated desktop home
Prepare for an appointment	I can enter the interaction once with enough context.	Case, product, and note history joined cleanly	Single case context panel
Answer a policy question	I can answer safely or escalate honestly.	Approved sources with citations and freshness	Knowledge assistant with “not approved” fallback

INTERNAL SERVICE STEP	FRONTLINE PROMISE	BACKSTAGE DEPENDENCY	DESIGN IMPLICATION
Hand a case to support	I can promise a next step without bluffing.	Named owner and visible SLA	Tracked escalation and receipt state
Coach and improve	I learn from patterns instead of firefighting anecdotes.	Manager digest and branch-level metrics	Weekly coaching cadence

Frontline blueprinting is therefore less about diagramming and more about accountability. If a backstage team cannot be named, the frontline will end up lying politely. If a knowledge source cannot be approved, the adviser will improvise. If a manager ritual does not exist, the pilot will decay into optional tooling. The blueprint lets me expose those truths early enough to redesign the service instead of blaming the staff.

> **CORE PRINCIPLE:** Blueprint the internal promise, not the customer flow alone. A frontline experience improves when the invisible service behind the employee becomes visible, owned, and testable.

Enablement Kits as Typed Artifacts

Enablement work becomes messy when everything is dumped into one folder called “training materials.” ASD gives me a cleaner way to think about it. The runtime already knows certain artifact kinds: service-blueprint, service-standard-set, walkthrough-script, storyboard, prototype-plan, KPI tree, review-veto-log, and so on. An enablement kit is not yet a single registered runtime kind, and I do not pretend otherwise. In practice, I assemble it as a governed bundle of typed artifacts, each with its own acceptance logic and owner. That honesty matters. The bundle is real. The runtime kind is still pending.

For Calder, the core bundle centered on the `frontline-enablement` delivery skill, which takes a service blueprint or standard set and emits a `service-standard-set`. That output is crucial because it defines the behavior the frontline is expected to perform, the measure that proves it exists, and the owner who is accountable for keeping it current. Around that core sit the training and adoption artifacts: walkthrough scripts for the new flow, roleplay packs for difficult scenarios, a manager coaching digest, scenario labs for new hires, and approved-answer cards for policy-heavy tasks.

enablement-kit.bundle.yaml

```
bundle_id: calder-branch-enable-v1
registered_runtime_kind: false
parts:
  - artifact_kind: service-standard-set
    purpose: define frontline behaviors and measures
  - artifact_kind: walkthrough-script
    purpose: guide adviser rehearsal of the new flow
  - artifact_kind: storyboard
    purpose: rehearse difficult branch scenarios safely
  - artifact_kind: prototype-plan
    purpose: test the desktop and escalation path before rollout
  - artifact_kind: kpi-tree
    purpose: measure adoption and service effect
owner: branch_network_lead
review_gate: critic
```

The advantage of this bundle model is practical. A bank can change the coaching script without rewriting the service standard. A branch pilot can update a scenario lab without pretending the desktop shell changed. The Critic can veto an unowned standard while leaving a prototype plan intact. That is the kind of modularity I want in operational design: one internal service, many artifacts, clear boundaries.

A useful enablement kit also makes adoption falsifiable. Calder did not merely hand branches a polished desktop mock and hope behavior changed. The service-standard-set named the measure for each frontline behavior and banned owner values like `tbd` or `unassigned`. That matters because a standard without an owner is not a standard. It is a wish. The frontline-enablement skill encodes that directly: the

Critic rejects any standard with no owning role, and the artifact stays draft until somebody accountable is named.

This is where typed artifacts help the employee-experience conversation grow up. A coaching prompt, a roleplay pack, a standard set, and a dashboard may all travel together, but they should not all be judged by the same acceptance rule. The service-standard-set needs measures and owners. The walkthrough-script needs the task path. The storyboard needs scenario fidelity. The KPI tree needs causal clarity. Once teams separate those concerns, enablement stops being a blob and starts becoming a maintainable internal product.

The most interesting kit component in Calder was the approved-source knowledge assistant. The design team refused the more impressive-looking but unsafe version of "AI that can infer the answer." Instead, the assistant was bound to approved policy packs, procedures, FAQs, credit playbooks, KYC notes, and coaching scenarios. If the source pack could not support an answer, the artifact had to say so explicitly. In regulated branch work, graceful incompleteness beats fluent fabrication every time.

knowledge-assistant.prompt.txt

```
SYSTEM ROLE: CALDER APPROVED KNOWLEDGE ASSISTANT
Allowed sources only: policy pack, approved procedures, current branch FAQs,
credit playbooks, KYC process notes, coaching scenarios marked approved.
When asked a question:
1. Retrieve only approved sources in the current data zone.
2. Cite the source title and freshness date.
3. If the answer is absent or ambiguous, say: not approved in source pack.
4. Route the adviser to the named escalation path.
```

That text block is not runtime theater. It is the exact difference between enablement and risk creation. A helpful answer card without citations becomes branch folklore in machine clothing. An approved answer card with freshness metadata becomes part of the internal service quality layer. That is the bar.

Feedback Loops into the Register

Enablement fails when the organization treats rollout as the end of the design. Calder worked because the pilot branches kept feeding evidence back into the project register. Advisers were not passive recipients of the new service. Their friction, uncertainty, edge cases, and coaching needs were captured as evidence rows and fed into backlog decisions. In ASD language, the loop stayed alive: capture evidence, run the method, write the artifact, review it, factcheck it, then let the human decide the next gate.

This is where the works-council guardrails mattered. The bank did not collect individual ranking telemetry or covert performance traces. It captured branch-level patterns, roleplay findings, manager notes, and structured feedback about blockers. That design choice protected trust and made the evidence better. People are much more honest when they know the point is to improve the internal service rather than sort them into winners and losers.

frontline-feedback.csv

```
week,branch,signal_type,issue,owner_class,route
2026-07-20,BR014,quote,"Credit-owner status still invisible after handoff",backstage
2026-07-20,BR031,telemetry,"Knowledge-assistant fallback rate rose on bereavement ca
2026-07-27,BR052,quote,"Manager huddle helped, but scenario lab lacks SME edge cases
2026-07-27,BR018,telemetry,"Duplicate note entry returned in mortgage complaints pat
```

That kind of table does two useful things. First, it preserves the evidence kind honestly: quote versus telemetry. Second, it routes the issue to the right remediation path instead of flattening every problem into “training needed.” Calder's best insight here was that many apparent coaching problems were actually design problems. A branch manager should not be asked to coach around a broken ownership model or a weak content pack. The feedback loop made that visible.

The coaching cadence then became part of the service architecture. Weekly branch huddles surfaced immediate blockers. Fortnightly side-by-side observations

deepened skill. Monthly cross-branch calibration prevented local myths from hardening into policy. A journey council escalation path ensured structural blockers reached owners with enough authority to fix them. That is what I mean by feedback into the register: a route from lived use to governed change.

> **My take:** A frontline dashboard without a remediation path is surveillance with nicer typography. Do not collect feedback that cannot change an owner, backlog, or service standard.

There is a second reason I like this loop. It gives the frontline a legitimate route to challenge the design. In many organizations, employees are asked for feedback but not given any visible proof that the feedback can alter standards, tools, or ownership. Calder's branch backlog lanes, journey-council escalation path, and monthly calibration review changed that. A branch manager could point to a content gap, a dashboard anomaly, or an SLA breach and know exactly which artifact or owner had to respond. That is internal service quality behaving like a service, not a suggestion box.

Case: Calder Bank

Calder's case is a strong spine for employee experience because it sits in the awkward middle that real organizations inhabit. The bank could not replace the core. It could not weaken KYC, AML, credit, or complaint rules. It could not rank people individually without blowing up trust. It still had to improve the service. So the redesign stayed stubbornly operational: one useful start point, one safer knowledge lane, one visible escalation path, one coaching rhythm, one evidence loop.

That internal-service lens also clarified how article 23 connects to earlier Calder work. Article 06 had already exposed onboarding delay, document loops, and status blindness. Article 16 had already framed LaunchPad, Guided Start, and Accountant

Connect as promising customer-facing concepts. This chapter shows what must exist behind those concepts if they are going to survive live branch reality. Guided Start only works when the adviser can see named ownership and visible heartbeat. LaunchPad only works when document receipt is confirmed and re-entry is reduced. The concept is the promise. Frontline enablement is what makes the promise performable.

The enablement kit for the pilot branches therefore included three layers. First, structural artifacts: the internal blueprint, the service standards, the role clarity, the SLA model. Second, rehearsal artifacts: desktop walkthroughs, roleplay scripts, scenario labs, answer cards. Third, operating artifacts: coaching digests, adoption metrics, branch backlog lanes, and review rituals. When those three layers aligned, the numbers moved for the right reasons. Advisers spent more time in useful customer conversation. New hires became competent faster. Status chases dropped. First-contact quality improved. And the bank began speaking about internal service quality instead of “resilience.” That language shift is not cosmetic. It means the organization has started locating the real source of failure.

> **CAUTION:** Do not let enablement become a euphemism for coping better with a bad system. If the frontline needs constant coaching to survive the same unowned seams, the redesign is incomplete.

What I take from Calder is simple. Employee experience belongs inside service design because the frontline is part of the service, not a delivery afterthought. Once you accept that, the artifacts get sharper. A blueprint shows the invisible work behind honest behavior. A service-standard-set defines what good looks like in the moment of use. A coaching loop keeps capability alive after the prototype team leaves. And a governed feedback register stops “staff comments” from dissolving into folklore. That is the internal service that makes the external one possible.

CHAPTER 21

SOPs-as-Code: The AI Execution Playbook

CHAPTER 23

The 15 Principles of Good Public Services

NEXT CHAPTER

The 15 Principles of Good Public Services

Lou Downe's bar, enforced by machine checks

Continue Reading →

ASD ' 26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 23

The 15 Principles of Good Public Services

Lou Downe's bar, enforced by machine checks

Reading time

12 min

Why Good Services Matters More With Agents

Agents amplify both quality and harm. If the underlying service is coherent, evidence-linked, and owned, they can make it easier to keep standards current, surface failures quickly, and route work across channels consistently. If the underlying service is weak, they accelerate the wrong thing: faster dead ends, more confident misinformation, smoother exclusions, and prettier versions of organizational confusion. Public service cannot afford that trade.

The Fernwood planning case made this concrete. Residents were not mainly asking for a prettier website. They wanted to know whether they needed permission, what documents were required, how long the route would take, what changed if the council

asked for amendments, and why a decision had been made. The old service was legal enough to function, but not clear enough to trust. That distinction matters. A lawful service can still be a bad service.

Good Services gives the practice a bar that is human, operational, and politically grounded at the same time. Easy to find. Explains purpose. Sets expectations. Completes the outcome. Familiar. No prior knowledge required. Agnostic of organizational structure. Minimum possible steps. Consistent. No dead ends. Usable by everyone equally. Responds to change quickly. Explains why decisions are made. Easy to get human assistance. Requires the right knowledge from the user. I like this list because none of it is decorative. Every principle forces a service team to reveal whether it has designed the whole service or only one channel.

> **My take:** The fastest way to spot fake service maturity is to ask whether the team can point to the artifact, evidence, owner, and gate for each principle. If they cannot, the principle is still a slogan.

ASD's contribution is not to replace the principles with a machine rubric. It is to make the principles operational enough that the machine can help enforce them. Principle 13, for example, does not become “write better copy.” It becomes a demand for a decision-explainer artifact, comprehension testing, and an owner who can remediate weak explanations. Principle 14 does not become “show a phone number somewhere.” It becomes a cross-channel requirement that human assistance is visible and joined to the same service state. That translation is where the work gets serious.

The Fifteen Principles as Checks

Fernwood treated each principle as a measurable property rather than an aspiration. That is the pattern I recommend everywhere in public service. Start with the principle.

Translate it into one observable test in plain service language. Require evidence before a pass score. Link the principle row to the artifact that proves the response exists. Name the owner who can fix a failure. Then make the gate fail when evidence or ownership disappears. Principle, checklist, evidence, artifact, gate. That chain is what prevents public-service standards from turning into poster rhetoric.

The principles also work well as a portfolio language because they cut across channels and departments. A team that scores only the web form will miss the call-centre script, the PDF letter, the assisted-digital route, the committee decision note, and the backlog rule that silently reintroduces the same burden next month. By forcing an artifact link and an owner per principle, the audit keeps the conversation whole-service. That is especially important in government, where a resident experiences one service while the organization experiences five directorates and three systems.

good-services.scorecard-row.json

```
{
  "principle_id": "P13",
  "principle": "Explains why decisions are made",
  "observable_test": "A refused applicant can explain the decision, evidence, and ne
  "evidence_ref": "quote:resident-test-08",
  "artifact_ref": "decision-explainer-template",
  "owner": "planning_service_owner",
  "gate_status": "pass"
}
```

#	PRINCIPLE	MACHINE-CHECKABLE AUDIT QUESTION	TYPICAL PROVING ARTIFACT
1	Easy to find	Can users reach the service from natural-language search or the home entry point without insider vocabulary?	Navigation audit, findability test, search log
2	Explains purpose	Can users state who the service is for and what it does after a short exposure?	Start-page comprehension test

#	PRINCIPLE	MACHINE-CHECKABLE AUDIT QUESTION	TYPICAL PROVING ARTIFACT
3	Sets expectations	Are time, cost, documents, stages, and likely next steps visible before commitment?	Checklist, route explainer, timeline artifact
4	Completes the outcome	Does the service cover the whole user goal rather than ending at internal submission?	Journey map, service blueprint, end-state flow
5	Works in a familiar way	Does the service reuse recognizable patterns instead of forcing relearning?	Pattern inventory, interaction review
6	Requires no prior knowledge	Can a first-time user proceed without internal jargon, codes, or policy literacy?	Plain-language audit, first-time-user test
7	Agnostic of organizational structures	Can the user complete the route without understanding departmental seams?	Service blueprint, seam audit
8	Requires minimum possible steps	Have duplicate requests, avoidable loops, and unnecessary approvals been removed?	Step-count audit, backlog diff
9	Consistent throughout	Do channels, documents, and messages agree on state, language, and next steps?	Cross-channel parity check
10	Has no dead ends	Does every failure, timeout, and refusal route lead somewhere useful?	Unhappy-path specification
11	Usable by everyone, equally	Do accessibility and assisted-digital routes offer comparable completion power?	Accessibility audit, assisted-digital walkthrough
12	Responds to change quickly	Can the service handle amendments, updates, and reopened cases without restart?	Change-handling flow, SLA test

#	PRINCIPLE	MACHINE-CHECKABLE AUDIT QUESTION	TYPICAL PROVING ARTIFACT
13	Explains why decisions are made	Can users explain the decision, supporting evidence, and next step afterward?	Decision explainer, comprehension test
14	Easy to get human assistance	Is human help visible, reachable, and joined to the same service record?	Support script, shared timeline, escalation path
15	Requires the right knowledge from the user	Does the service ask only for facts users can reasonably know?	Data-burden audit, form review

That table is intentionally blunt. It lets a team challenge weak claims immediately. “We improved findability” means nothing unless you can point to the search path and the task-success evidence. “We made it more inclusive” means nothing unless the assisted-digital route can complete the same outcome with the same underlying case state. “We explain decisions” means nothing unless users can actually explain them back.

One practical benefit of this translation is that it gives multidisciplinary teams a shared argument surface. Content designers, policy leads, accessibility specialists, service owners, and engineers can disagree over one observable test instead of arguing from separate abstractions. That is a healthier conflict. It turns “this feels clearer” into “can a resident actually complete the route, explain the decision, and recover from failure?” Good public-service standards become stronger when the disagreement is forced down to evidence.

planning-service-standard-gate.yaml

```
name: planning-service-standard-gate
checks:
  - principles-register-complete
  - accessibility-ci-pass
  - unhappy-paths-covered
  - decision-explainer-present-where-needed
  - missing-owner-for-principle-regression
```

```
  fail_on:
    - unsupported-pass-score
    - inaccessible-core-task
    - cross-channel-parity-gap
```

Once the principles are expressed this way, the machine can do useful work without pretending to own public judgment. It can check whether the row exists, whether evidence is present, whether owners are named, whether unhappy paths are covered, whether accessibility CI passed, whether an unsupported pass score slipped in. What it cannot do alone is decide whether the service is politically just, proportionate, or trustworthy in a context where law, democratic accountability, and lived vulnerability collide. That remains human work, as it should.

Running `good-services-audit`

The platform ships `good-services-audit` as a measurement-phase method, and it also carries a more explicit sibling called `good-services-audit-15`. For most teams the important point is the same: the audit is not a mood board. It consumes a live or shipped service blueprint or journey map plus post-launch feedback or support logs, then emits an `insight-set` that the Critic must review before it can become final. The runtime contract is intentionally tight. At least three nodes. At least one evidence reference per node. The `theme` field must be non-empty. Anything weaker stays draft.

Teams often assume the audit belongs at the end of delivery, as if Good Services were a compliance stamp. Fernwood shows why that is too late. The most useful moment to run the audit is whenever a service definition, prototype, or standards pack could still be reshaped cheaply. In practice I use it in three modes. Early, as a baseline that exposes structural gaps. Midstream, as a design gate that blocks decorative progress. Late, as a governance instrument that proves whether the live service is actually honoring the bar it claims.

That three-mode use is one reason the sibling methods coexist sensibly. `good-services-audit` is the general measurement instrument in the shipped library. `good-services-audit-15` is the more explicit fifteen-principle variant when the scorecard itself needs to be the direct working artifact. I would rather have both than force one method to pretend every public-service context is identical.

good-services-audit.run.sh

```
asd method-run --projectId fernwood --methodId good-services-audit
asd artifact-write --projectId fernwood --kind insight-set --file audits/good-servic
asd review --projectId fernwood --artifactId audit-015 --verdict pass --rows reviews
sd_factcheck fernwood
```

I like this method because it keeps public-service quality from floating away from the operational record. The audit can start with a principles scorecard, but it has to end with evidence-cited findings that someone can act on. If Principle 10 fails, where is the dead end? Which screen, letter, or phone transfer created it? If Principle 15 fails, what internal knowledge is the resident being asked to supply? If Principle 12 is partial, which change route is still breaking? Those questions drive the artifact toward service repair instead of ceremonial scoring.

good-services-audit.finding.yaml

```
theme: decision-explanation
principle_id: P13
severity: high
evidence_refs:
  - quote:resident-test-08
  - document:decision-letter-v2
finding: Residents can repeat the decision outcome but not the reason chain or appea
recommended_artifact: decision-explainer-template
owner: planning_service_owner
```

The output being an `insight-set` is a smart choice. It means the audit does not pretend to be the final fix. It surfaces where the service violates the bar, cites evidence, and hands structured insight back to the practice. Mapper, Presenter, or

Orchestrator can then turn those findings into blueprint changes, standards updates, backlog items, or a governance intervention. The audit is a measuring instrument, not a decorative certificate.

> **CAUTION:** Never let a public-service audit pass on interface attractiveness alone. A principle can only pass when the evidence proves the service behaves that way in use.

The human checkpoint is equally important. Before the audit runs, the designer confirms scope, inputs, consent, and intended decision. Before handoff, the designer approves the evidence-backed findings and their open gaps. That is what keeps the method from drifting into automated scorekeeping detached from public consequences. Measurement is still design, not clerical work.

Case: Fernwood Council Phase 2

Fernwood's second-phase planning work is one of the clearest demonstrations in the book because it treats the fifteen principles as a live operating standard. The council did not simply list the principles in a deck. It translated each one into a row in a standards register with a test, evidence requirement, artifact link, owner, and merge gate. Then it redesigned the citizen account and planning route so those rows could actually pass.

Principle 3, “sets expectations,” became a requirement that the start page show route, drawings, fees, consultation, and end-to-end stages before the resident committed. Principle 13, “explains why decisions are made,” became a four-part explanation template tested with refused and conditioned applicants. Principle 14, “easy to get human assistance,” became a structural rule that phone and in-person advisers worked in the same draft and timeline as the digital route. That is the move I want every public-service team to copy. Translate principle language into service mechanics.

PRINCIPLE FOCUS	BEFORE	AFTER	WHAT CHANGED
P3 Sets expectations	Residents entered with vague expectations about time, fees, and drawings.	Start page exposed route, documents, fees, and stages before start.	Checklist plus route explainer artifact
P13 Explains decisions	Legally sufficient letters, weak comprehension	Decision explainer paired reasons, evidence, and next step in plain language	New decision artifact plus comprehension test
P14 Human assistance	Phone help sat outside the digital record	Human support wrote into the same draft and timeline	Cross-channel parity rule
P12 Responds to change	Amendments disappeared into email	Planning changes improved, but cross-service propagation remained partial	Honest partial score instead of decorative pass

That last row is especially instructive. Fernwood did not claim perfection. Principle 12 remained partial because full cross-service propagation into council tax and waste was still on the roadmap. I trust that kind of scorecard more than a wall of passes. Public services earn legitimacy when they are explicit about what has improved, what is still partial, and what evidence would be required to claim more.

The before-and-after audit made the improvement visible without flattening it into one vanity number. Before redesign, the planning service was strong on statutory completion and weak on findability, expectation-setting, decision explanation, assisted digital, and right-knowledge burden. After redesign, the new planning route could credibly pass most principles because the artifacts, tests, and owners existed. Residents could find the service more easily. Decision explanations became understandable. Progress checks per case fell. Complaint pressure eased. None of that required mystical AI. It required disciplined service design with machine-checkable gates.

> **My take:** In public service, honesty about a partial score is a quality signal. Decorative pass rates destroy trust faster than a visible unresolved gap.

Fernwood phase 2 is also a reminder that the principle bar applies to governance as well as interface copy. The planning committee chair, digital lead, and service owner all needed the same scorecard language so design, policy, and delivery could argue over one thing. That is why the standards register mattered so much. It gave the organization one public-service quality grammar instead of several competing interpretations of "good enough." Once that grammar existed, GitHub issues, PR gates, accessibility checks, and decision-explainer reviews all pointed at the same standard rather than inventing their own.

The result was a service that became easier to govern because it became easier to inspect. Residents could find the route more easily. Officers could see which principle a change was meant to improve. The service owner could challenge a pass score that lacked evidence. And the design team could show why a partial remained partial instead of hiding behind optimism. That is exactly what machine-checkable quality should do in public service: make accountability more discussable.

Fernwood phase 2 also shows how the public-service standard interacts with ASD's wider loop. The evidence register grounded the claims. The standards register made the principles operational. The good-services audit surfaced remaining gaps. The Critic stopped unsupported pass scores. `sd_factcheck` forced integrity, acceptance, and review to agree before a human gate. In other words, the principles were not bolted on after delivery. They were woven into the method spine.

That is why this chapter matters beyond local government. Any organization delivering high-stakes, low-choice, multi-channel services can borrow the pattern. Take the fifteen principles. Write the observable tests. Require evidence before pass. Link the row to an artifact. Name the owner. Gate the merge. Then audit again after the prototype glow fades. Good services are good when the service can survive that audit repeatedly, across channels, owners, and political pressure. That is how a principle becomes a public operating standard instead of a slogan on the wall. It is

also how design earns a durable seat in governance. Public trust gets a stronger operating basis.

CHAPTER 22

Frontline Enablement & Employee Experience

CHAPTER 24

Mastercase: Meridian Bay Ferries

NEXT CHAPTER

Mastercase: Meridian Bay Ferries

One fictional engagement across every surface

Continue Reading →

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 24

Mastercase: Meridian Bay Ferries

One fictional engagement across every surface

Reading time
19 min

The Client and the Mess

Meridian Bay Ferries runs the kind of service that executives still describe as straightforward because the route map looks neat on a slide. Five terminals. Nine vessels. Forty-three weekday sailings. Commuters in the morning, school groups at ten, tourists all afternoon, luggage spikes whenever a cruise ship anchors offshore, and tide windows that refuse to negotiate with anyone's quarterly targets. On paper, the operator's punctuality promise is simple: board quickly, depart on time, land to a waiting shuttle. In practice, the promise breaks three times before the customer leaves the dock.

The first break sits in ticketing. The booking flow and the confirmation email say one thing about boarding cut-off times; gate staff and harbour operations live by another. The second break sits on the dock. Gate readers stamp passengers against scheduled departure, while ramp crews load against physical readiness and whatever the harbourmaster is doing with berth sequence that hour. The third break shows up onboard and after arrival. A ferry can leave "on time" in the reporting system, miss the safe tide slot, arrive late enough to strand island passengers after the last shuttle, and still look green in the dashboard until someone reconciles the journey by hand. That is failure demand dressed as punctuality.

The stakeholder map was crowded from day one. Crew unions needed assurance that any automation would respect the safety walk, roster rules, and fatigue boundaries. The council harbourmaster controlled berth changes and tidal windows. The regional tourist board owned a stream of public complaints and reputational pressure. A third-party maintenance contractor held defect and reset notifications that often surfaced too late for passenger communications. Inside Meridian Bay itself, operations, ticketing, dock supervisors, and the call-centre had each built local workarounds. Nobody owned the promise end to end.

> **My take:** Punctuality is never a single timestamp. In transport services it is a chain of promises crossing channels, people, physical constraints, and exceptions. When the operator measures only departure, it congratulates itself for journeys customers still experience as broken.

I framed the brief narrowly enough to finish and honestly enough to matter. I was not redesigning fleet strategy, procurement, or harbour infrastructure. I was redesigning the punctuality promise across ticketing, dock operations, and onboard recovery communications. Success meant fewer false on-time signals, fewer stranded passengers after disruptions, and a clearer human owner whenever the system shifted from promise to recovery.

That is why this chapter works well as a mastercase. It demands every surface without letting any one surface pretend to be the whole practice.

Surface Plan: Which Tool Where

I kept the engagement in one workspace, then chose surfaces by situation. Four of them were human-facing: CLI, Claude Code plugin, BUZZ, and DSH Studio. The standalone MCP server acted as the automation seam. I did not split the project by host. Every surface opened the same `meridian-bay` project and resolved the same evidence ids.

SURFACE	WHERE I USED IT	WHY IT BELONGED THERE	PRIMARY OUTPUTS
CLI	Project scaffold, dockside evidence capture, gate receipts	I could capture quotes and documents on the go, under noise, without opening a host UI. It is the fastest path from field note to content-addressed record.	`quote`, `telemetry`, `document` evidence; human `phase-advance` receipts
Claude Code plugin	Discover and Define synthesis with role mentions	The role roster is legible there. I could invoke `@researcher`, `@synthesizer`, `@mapper`, `@critic`, and `@librarian` against the same store while reading method contracts inline.	`research-plan`, `stakeholder-map`, `insight-set`, `journey-map`, `opportunity-matrix`
MCP server	Scripted corpus research and CI fact-check	JSON-RPC is the clean seam for repeatable lookups and machine-run checks. I used it to query the knowledge corpus and to replay `sd_factcheck` in CI without pretending a chat host had executed anything.	Corpus lookups, fact-check rows, reproducible report/export steps
BUZZ channel	Dock-ops rehearsal coordination with a Canvas index	The rehearsal needed visible multi-party coordination: owner, runtime roster, scoped mentions, and a navigable Canvas. Conversation stayed in BUZZ. Persistence stayed in ASD.	Channel thread, Canvas index, owner-verified round trip, rehearsal coordination
DSH Studio	Board readout session	The board needed a review board, evidence browser, artifact renders,	`executive-summary`, `artifact-dossier`,

SURFACE	WHERE I USED IT	WHY IT BELONGED THERE	PRIMARY OUTPUTS
		and reports in one place. Studio reads the same store as the canonical tools, so the readout did not require a second export ritual.	`traceability-matrix`, review board state

> **CORE PRINCIPLE:** Choose surfaces by working context, never by status. The field tool captures. The host tool synthesizes. The collaboration tool coordinates. The studio reads out. The evidence chain stays singular.

That distinction mattered because Meridian Bay had already suffered from tool sprawl. Ticketing had its own truth. Dock control had its own truth. Maintenance had its own truth. I refused to create a fourth or fifth local spreadsheet truth just because a host made it convenient. The whole point of ASD is that surface variety does not require evidence fragmentation.

Discover: Evidence at Ferry Scale

I started where transport redesign usually gets dodged: the dock in bad weather, early enough to watch the promise fail before anyone has time to explain it away. The CLI handled the first hour because it was the only surface I trusted on a windy pier with gloves on. I created the project, opened it, and started dropping raw records straight into the register. Every evidence id was content-addressed over `(source, kind, body)`. That meant the same passenger sentence from two different interviews stayed two different records, while re-importing the same timetable paragraph stayed idempotent.

dock-capture.sh

```
export ASD_WORKSPACE="$HOME/.asd/projects"
PID=meridian-bay
asd project-create --name "Meridian Bay Ferries" --client "Meridian Bay Ferries" --s
asd project-open --projectId "$PID"
asd evidence-add --projectId "$PID" --source passenger-int-04 --kind quote --body "T
asd evidence-add --projectId "$PID" --source gate-readers-2026-02 --kind telemetry -
asd evidence-add --projectId "$PID" --source timetable-v47 --kind document --body "W
```

By the end of Discover, the core register held twelve records I trusted enough to synthesize from. Five were verbatim quotes, three were telemetry extracts, and four were documents. I am showing the ids because the hash discipline matters here; the case falls apart if you cannot see how one quote becomes one stage, one defect row, or one blueprint step later.

EVIDENCE ID	KIND	SOURCE
`86861ea2c6d2d8e149426ccf4d3378541369fa43fa3bfe708a68ba98e6f690fe`	`quote`	passenge 04

EVIDENCE ID	KIND	SOURCE
`8b1b1d6fcc22c7a018fdffa3d82301af0b8aa7819a10bf51755fa599e9017e4d`	`quote`	deckhanc 02
`45a1897278eea02652d9c069157c6c736dcd558408384439fc9a63a09a9fcae3`	`quote`	callcentre 11
`aa30fd7acec2031246ab648f5d3c29f37f4ea6cee27c1f4387102bff0587a2b3`	`quote`	union-workshop
`c226ce2a23ee8ea6f7a73c906955f524c5f4cd98fbb74a4392abb4612c3c9000`	`quote`	harbourm int-01
`5041cc2df5c5f973663a6f9e78a9490aba792524453035f11ab4441c15d2b72b`	`telemetry`	gate-reac 2026-02
`acc0c8ef64c5526de8506e6001a0d63a2de6337aacef6859bed0168de831563e`	`telemetry`	manifest-reconcile 2026-02
`2f388bea2d5b291e26e213c06b2ae1ec24cec1fd6eb245b65041b59010fea207`	`telemetry`	dock-turnaroun study-20

EVIDENCE ID	KIND	SOURCE
`ecd0f76d9fa4fd133d372462a04e0c187d292b78efad728de3d656fd73620fd8`	`document`	timetable
`3a39001efbc20cbc7fbf53257d10c1feb422969d9ffe218993bd1484ffba6b33`	`document`	spring-tic table-ma 2026
`8d3999a2d07095031344424d6255626ff8864735fd62b34805d44a0d30ffae81`	`document`	maintena sla-2026-
`3aead43aab03b0f5a2d38fceec97e4dd3c4fa07aa7eba723a3af60798710ddc7`	`document`	tourist-bc complain

Once the raw evidence was in the store, I moved into the Claude Code plugin for Discover synthesis. That switch was intentional. I wanted the role roster visible and the method canon close at hand. `@researcher` ran the contextual inquiry contract against the dock and passenger records. `@librarian` checked that the cut-off mismatch really belonged in evidence as a document conflict and not as my interpretation. The result was a `research-plan` artifact with id `1fb65eb4a19f9ac2cbb7d8e1231c3c91` and a `stakeholder-map` artifact with id `2b9b9a84e3111e95b04dcf8be21f3bd0`. Both stayed draft until the Critic passed them. Both passed cleanly because the evidence was strong and the claims stayed humble.

The standalone MCP server also entered here, quietly. I used it for scripted corpus lookups against the public transport plugin case in Article 31 and the BUZZ boundary article in Article 37, plus a CI-side fact-check stub. That work never became evidence by itself. It gave me method discipline and bounded language. The ferry case still had to earn its own register.

Define: Synthesis Under Review

Define happened inside the Claude Code plugin because this was the phase where role handoffs helped more than a plain shell. I wanted a visible thread showing who did what: `@researcher` consolidating ferry-specific patterns, `@synthesizer` clustering tensions, `@mapper` drafting a current-state journey, and `@critic` checking every artifact before it could go final. This is where the platform's ten-role shape earns its keep. The work stays legible.

claude-code-session.txt

```
@researcher Pull only Meridian Bay evidence ids and separate dock, ticketing, onboar
@synthesizer Cluster them into tensions. No causes without evidence.
@mapper Draft the current-state commuter and visitor journey. Mark gaps explicitly.
@critic Review every node and reject any stage whose cited evidence does not resolve
@librarian Check method provenance and keep article-derived language out of the arti
```

The current-state journey landed as artifact `36a1d93b6b6fffadaaa97ebeafd5591d`. The insight set landed as `12c7b3ee619b0513c088ab997facd793`. The opportunity matrix landed as `506c6f55332d6102f45638e45f812114`. The Critic's passes were useful because they forced a distinction I see teams blur constantly: lateness, uncertainty, and recovery are related, yet they are not the same defect. Meridian Bay's biggest wound was not absolute delay. It was false certainty. The system repeatedly promised "on time" after the journey had already entered a recovery state.

That synthesis produced three framing moves. First, the punctuality promise had to be rewritten as a journey promise, not a departure promise. Second, berth control and maintenance comms had to be treated as service-shaping backstage actors, not external noise. Third, compensation could not be designed as a finance feature alone; it was a recovery mechanism entangled with manifest accuracy, gate logic, and harbour authority.

The human Define gate mattered more than the artifacts. I presented the current journey and the stakeholder tensions to Meridian Bay's operations director, customer director, and a union safety delegate. They approved the problem framing under receipt `mbf-gate-define-002`: the engagement would redesign the punctuality and recovery promise without pretending to solve fleet capital constraints. That decision sounds modest. It saved the project. Ambition without a tractable frame is how service programs die politely.

Develop: Concepts, a Veto, a Remedy

Develop is where people get seduced by speed. Meridian Bay gave me an especially tempting trap: automate refunds whenever gate scans and arrival telemetry showed a broken promise. It looked efficient. Finance liked it. Marketing loved the story. Passengers would probably have liked the headline. The concept-set artifact, `87d6f25ebca1c64025f434df72fbcad5`, originally included an *auto-compensate on promise break* concept alongside a tide-aware boarding promise and a dock controller console.

The Critic vetoed that concept, correctly. The problem was not sentiment. The problem was missing failure-path evidence. The concept assumed the scanner

history could distinguish passenger-caused misses from operator-authorized gate reopens, berth holds, freight interruptions, paper concession reconciliations, and late manifest merges. My own evidence set did not support that claim. A fast team would have shipped it anyway and called the defects “edge cases.” The ASD loop does not let me get away with that.

review-rows.json


```
[
  {
    "rowKind": "defect",
    "severity": "high",
    "rule": "failure-path-evidence",
    "message": "The automated-refund concept assumes gate scans distinguish missed b
    "fix": "Add dock-operator evidence for gate reopen scenarios and replace automat
    "evidenceId": "45a1897278eea02652d9c069157c6c736dcd558408384439fc9a63a09a9fcae3"
  }
]
```


The veto persisted as `review-veto-log` artifact `edaba7bc9072c412b2fe5e2b9ae9bd0e`, row `r1`, linked back to the concept set by `reviewOf`. That row did exactly what I want a review row to do. It named the violated rule. It explained the risk in operational language. It proposed a concrete fix. It demoted the concept back to draft without deleting the work or hiding the argument. Meridian Bay’s team could see the refusal, read it, and answer it.

> **CAUTION:** The most dangerous automation ideas are the ones that sound compassionate. Automatic refunds, automatic rebookings, automatic reassurance messages. If the evidence cannot distinguish service failure from service exception, you automate unfairness at scale.

Remediation required more evidence, not louder rhetoric. I went back to the dock and added two new quote records: `08ce9338a6966bcb54ee3954313d2d733c0084500e25a465cee66e9e4ee05ea6` from the dock supervisor who sometimes reopens a gate on harbourmaster instruction, and `36c217132b4d2ac18b333a71068d08c871064918f2ef5c15e422978e07c04633` from the gate agent who still reconciles family bookings and paper concession tickets by hand after departure. Those two records changed the concept materially. I rewrote it from auto-compensate to *compensation proposal with human override*.

That rewrite also clarified the six-lane future blueprint. The sixth lane was not there to make a moral call. It was there to support bounded operational decisions: propose a revised ETA once berth change and tide evidence aligned, flag likely compensation candidates after manifest reconciliation, draft a shuttle hold recommendation, and route accessibility exceptions to a human owner without pretending the model could decide boarding eligibility. The automation moved from action to decision support wherever the service crossed fairness, safety, or public authority boundaries.

The remediated concept passed on second review. Better, the union stopped resisting the entire blueprint conversation. Once the safety walk stayed human, the critics inside the system became more willing to let the agent lane exist at all.

Deliver: Blueprint, Rehearsal Channel, Board Readout

Deliver is where cross-surface continuity stops being elegant theory and becomes project hygiene. The future-state `service-blueprint` artifact, `8a620aca21b3db982525afcfef649ea3`, held the familiar five canonical lanes—Physical Evidence, Customer Actions, Frontstage Interactions, Backstage Operations, Support Processes—plus the sixth Agent Decision lane. I used `service-blueprinting` to shape the operational flow, then `evidence-gated-blueprint` to force the stronger gate before readout.

The sixth lane carried only decisions that could be justified with resolvable evidence and handed to named humans when risk rose. It never owned phase authority. It never overruled the harbourmaster. It never bypassed the dock supervisor on accessibility or safety. Its job was disciplined assistance.

AGENT DECISION LANE STEP	WHAT THE AGENT DOES	GUARDRAIL	HUMAN OWNER
Recalculate passenger promise after berth change	Propose revised ETA and boarding cut-off when berth, tide, and turnaround evidence align	Suppressed if harbourmaster update is missing or contradictory	Dock controller
Flag compensation review	Suggest compensation review after manifest reconciliation and arrival miss are both present	No auto-payout; blocked on unresolved manual ticket states	Call-centre lead
Draft shuttle hold request	Prepare a recommended hold window for island shuttles on severe late arrivals	Only advisory; tourist board and shuttle operator see evidence summary	Arrival coordinator
Accessibility exception routing	Surface the likely impact of a berth or gate change on booked assistance cases	No autonomous boarding or denial decision	Dock supervisor

The rehearsal itself moved into BUZZ. I created a private channel for the dock-ops exercise, materialized the runtime roster, verified membership by public key, then posted three scoped mentions: one to the orchestrator for sequence control, one to the researcher for unresolved evidence questions, and one to the critic for failure-mode scrutiny. That BUZZ slice is written here in observed style because that is how a real rehearsal thread feels to the team, but I am keeping the claim bounded. The channel showed coordination and Canvas navigation. The ASD workspace remained the only source of record.

buzz-rehearsal.sh

```
CH=8d35f071-1af4-4dc2-a4bc-8f7c308c3f39
ORCH=9c2f8e65c7d94eab81d7bcf4dbf6fdb6c5ec6d5e4eeeb6d6488cf7dbe81ab122
RESEARCHER=244b0e3cd4c1d143d7ce0c9fb9fa1dd3cf7af9a46b95f7df8e5d2333d6c5771e
CRITIC=7f4a2fdab7d8c8b9df55e4fd3bc96be9d7f6b6be7dbab7ca45be4f5e10a9d0cc
buzz messages send --channel "$CH" --mention "$ORCH" --content "Run the dock-ops reh
buzz messages send --channel "$CH" --mention "$RESEARCHER" --content "Check whether
buzz messages send --channel "$CH" --mention "$CRITIC" --content "Challenge the six-
node scripts/buzz-canvas-index.mjs render \
  --project-dir ~/.asd/projects/meridian-bay \
```

```
  --channel sd/meridian-bay-rehearsal --fictional \
  --planned prototype-plan --planned test-report --planned recovery-playbook \
  --out ./canvas
buzz canvas set --channel "$CH" --content - < ./canvas/meridian-bay.md
buzz canvas get --channel "$CH" > retrieved-canvas.md
node scripts/buzz-canvas-index.mjs verify --canvas-dir ./canvas --project-dir ~/.asd
```

The Canvas round trip mattered because it kept the BUZZ thread navigable without turning the channel into a second database. The generated index pointed back to the actual evidence register, the blueprint, the veto log, the prototype plan, and the pending rehearsal outputs. Planned rows used only registered kinds: `prototype-plan`, `test-report`, and `recovery-playbook`. No fake `phase-gate-decision` row. No improvised deck artifact. No hand-written summary pretending to be the store.

The board readout happened in DeepSeek Harness Studio. That was the right surface because executives needed to read the project, not to watch me operate it. Studio's dashboard opened `meridian-bay`; the Evidence tab showed the full register with kind filters; the Review tab showed the blueprints, the clean passes, and the resolved veto log; the Reports tab gave me the `executive-summary`, `artifact-dossier`, and `traceability-matrix` in one place. Studio reads the same store the CLI and plugin write. I did not have to stage a separate board pack in some forgotten folder.

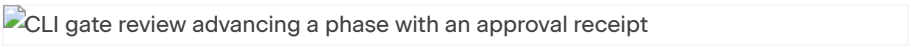


Figure 24.1 **CLI phase advance with approval receipt.** The capture proves that a human-approved phase transition is recorded with a receipt payload in the shared ASD workspace. It does not prove that the fictional Meridian Bay engagement itself ran live on this exact host or that another surface authorized the same gate.

One awkward but healthy detail surfaced during the board readout: DSH's direct capability projection does not expose a bespoke `phase.gate.status` row. Good. That forced the board to read the actual review board state, the fact-check, and the explicit approval receipts rather than relying on a decorative green badge. Governance stayed visible as evidence and human decision, exactly where it belongs.

The Traceability Thread

The final move was the traceability report. I exported it because Meridian Bay's board had already lived through one consulting program where the answer to every challenge was "trust the synthesis." That phrase should be retired. I wanted a reader to travel from a passenger quote to a journey stage, from a journey stage to a concept change, from that change to a blueprint step, and from the blueprint step to the final board pack without leaving the project.

gate-and-reports.sh

```
PID=meridian-bay
asd export-report --projectId "$PID" --kind executive-summary
asd export-report --projectId "$PID" --kind artifact-dossier
asd export-report --projectId "$PID" --kind traceability-matrix
asd factcheck --projectId "$PID"
asd phase-advance --projectId "$PID" --toPhase deliver \
```

```
    --idempotencyKey meridian-bay-gate-develop-03 \
    --approvalReceipt '{"receiptId":"mbf-gate-003","approvedBy":"elise.nguyen","decisi
```

Those exports were useful because they made the case inspectable by people outside the design team. The `executive-summary` told the board what to decide. The `artifact-dossier` showed what existed and in what state. The `traceability-matrix` showed why the recommendation existed at all.

DECISION OR CLAIM	EVIDENCE ANCHORS	ARTIFA
Boarding cut-off promise must be harmonized across channels	`86861ea2c6d2d8e149426ccf4d3378541369fa43fa3bfe708a68ba98e6f690fe`, `ecd0f76d9fa4fd133d372462a04e0c187d292b78efad728de3d656fd73620fd8`	`journ` `36a1c` → `ser` `8a620`
Turnaround target cannot stay at eleven minutes on Dock 3	`2f388bea2d5b291e26e213c06b2ae1ec24cec1fd6eb245b65041b59010fea207`, `aa30fd7acec2031246ab648f5d3c29f37f4ea6cee27c1f4387102bff0587a2b3`	`insig` `12c7b` → `opp` `506c6` → `ser` `8a620`
Compensation stays human-approved	`45a1897278eea02652d9c069157c6c736dcd558408384439fc9a63a09a9fcae3`, `08ce9338a6966bcb54ee3954313d2d733c0084500e25a465cee66e9e4ee05ea6`, `36c217132b4d2ac18b333a71068d08c871064918f2ef5c15e422978e07c04633`, `acc0c8ef64c5526de8506e6001a0d63a2de6337aacef6859bed0168de831563e`	`conce` `87d61` + `rev` `edaba` → `ser` `8a620`
Harbourmaster berth changes must feed passenger promises quickly	`c226ce2a23ee8ea6f7a73c906955f524c5f4cd98fbb74a4392abb4612c3c9000`, `3a39001efbc20cbc7fbf53257d10c1feb422969d9ffe218993bd1484ffba6b33`	`stake` `2b9b9` → `ser` `8a620` → `pro`
Tourist-board reputation risk is a service recovery issue, not merely a PR issue	`3aead43aab03b0f5a2d38fceec97e4dd3c4fa07aa7eba723a3af60798710ddc7`, `45a1897278eea02652d9c069157c6c736dcd558408384439fc9a63a09a9fcae3`	`insig` `12c7b` → `rec` Canva

Notice what never changed across surfaces. The ids stayed the ids. BUZZ referenced them. Studio rendered them. The plugin wrote against them. The CLI advanced phase around them. The MCP lane replayed checks over them. That is the compounding value of a single evidence chain: the case becomes easier to challenge, easier to trust, and easier to resume.

The approval receipts told the same story. `mbf-gate-001` moved the project from Discover to Define after the evidence scope and problem frame were accepted. `mbf-gate-002` moved it into Develop after the current-state journey and tensions were signed off. `mbf-gate-003` moved it into Deliver only after the veto had been remediated, the blueprint had passed, and the rehearsal plan existed. No phase moved because an agent felt confident. Every move carried a human name, rationale, timestamp, and idempotency key.

What Compounds

Meridian Bay Ferries is an illustrative composite. I need to say that plainly because the discipline of this book depends on honest boundaries. The BUZZ slice here is bounded. It shows the observed collaboration style, the Canvas index pattern, and the channel logic. It does not ask you to imagine that a message became persistence by magic. The DSH readout is equally bounded. Studio gives a board-grade reading surface over the same store. It does not replace the review gate, and it does not hide the human decision behind a UI flourish.

What does compound, though, is very real. The same project can survive field capture, agent synthesis, adversarial review, rehearsal coordination, executive scrutiny, and later reuse because the evidence thread never breaks. The first edition of service-design digitization gave teams prettier artifacts. This model gives them continuity. A dock supervisor can challenge a concept and see the exact quote that changed it. A union delegate can inspect the veto that stopped unsafe automation. A board member can ask why compensation stayed manual and get an answer that traverses evidence, review row, blueprint step, and receipt. That is a different level of organizational memory.

> **My take:** The strongest thing about an agentic service-design practice is not speed. It is that the practice becomes replayable under pressure. When the weather turns, the ferry is late, the channel is noisy, and the board wants an answer by five, you can still show what you know, what you do not know, who decided, and why.

That is why I end the case here, at the point where most teams would start calling the work “implementation.” The mastercase has already shown the compounding move: one workspace, several surfaces, one evidence chain, one veto that improved the design, one set of human gates that stayed human. Everything after this—pilot tuning, KPI drift, new recovery patterns—can continue in the same project without reopening the trust problem. That is the whole bet of ASD. Methods run. Evidence stays put. People remain accountable.

CHAPTER 23

The 15 Principles of Good Public Services

CHAPTER 25

The Future of Service Design: The Agentic Practice

NEXT CHAPTER

The Future of Service Design: The Agentic Practice

What compounds when methods run

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

CHAPTER 25

The Future of Service Design: The Agentic Practice

What compounds when methods run

Reading time

12 min

The Practice After Automation

The first thing that compounds in an agentic practice is not speed. It is continuity. A conventional project often loses its memory after each handoff: research sits in one tool, synthesis in another, delivery in tickets, and learning in whatever the team manages to remember six weeks later. ASD tightens that loop. Evidence is content-addressed. Methods are executable. Artifacts are typed. Review verdicts are stored. Factcheck can rerun the whole project. Human phase receipts say who decided what. That stack does save time, yes, but the deeper gain is that the service-design practice stops forgetting itself.

That continuity changes the economics of rigor. Once the cost of re-running a method, refreshing a blueprint, rebuilding a standards pack, or challenging a decision falls, teams can afford to stay honest for longer. The blueprint can be one day stale instead of forty-two. The review can happen before release instead of after harm. A junior designer can inherit method safety rather than only “tips.” A practice lead can see whether quality is actually improving or merely sounding more polished. The whole discipline gets less theatrical.

WHAT COMPOUNDS	OLD MODE	AGENTIC MODE
Evidence continuity	Quotes, metrics, and decisions fragment across tools	Evidence register stays queryable across methods and surfaces
Method quality	Good practice depends on who briefs the work	Executable SOPs preserve the non-negotiables
Review discipline	Late, opinion-heavy, and inconsistent	Critic rows, factcheck, and human gates stay durable
Institutional memory	Folklore, copied prompts, remembered caveats	Versioned packs, provenance, and reusable artifacts

The series kept returning to this from different angles. Article 29 showed methods becoming executable. Article 22 showed service quality turning into a standing operating rhythm. Article 23 showed principles becoming merge gates. Article 24, the mastercase, shows cross-surface continuity as one workspace rather than several disconnected performances. By the time I get to this closing chapter, the pattern is hard to ignore: the practice gets stronger when its judgments are made more visible, more reviewable, and more portable.

> **My take:** The real promise of AI in service design is not “fewer hours.” It is “more of the right memory survives contact with delivery.”

Compounding also changes what “reuse” means. In the older mode, reuse often meant dragging yesterday's slide or framework into tomorrow's workshop. In the

agentic mode, reuse means something stricter: the method keeps its provenance, the artifact kind keeps its contract, the review row keeps its evidence, and the emitted pack keeps its lineage. That is a much higher-quality reuse pattern. It creates memory with accountability attached instead of memory by copy-paste.

practice-loop.json

```
{
  "loop": [
    "evidence",
    "method-run",
    "typed-artifact",
    "critic-review",
    "factcheck",
    "human-phase-gate"
  ],
  "properties": {
    "deterministic_rendering": true,
    "draft_on_failure": true,
    "approval_receipt_required": true
  }
}
```

None of this means every service-design team needs to become a software platform team. It does mean the practice has to become infrastructural in its thinking. The output is no longer just the workshop summary or the final deck. It is the durable operating layer that keeps a service legible after the project team leaves. That is a different professional ambition. I think it is the right one.

Open Problems and Phase 2 Boundaries

There are also straightforward inventory facts worth holding onto because they keep the conversation concrete. The current practice contract spans 10 canonical roles, 80 shipped skills from 81 sources, 30 runtime `sd_*` tools, 228 corpus resources, 10

prompts, 38 registered artifact-kind contracts, 39 vocabulary rows, 7 authored templates, and 12 emitted targets. I do not cite those numbers as a vanity metric. I cite them because scale changes the design problem. Once the library is that broad, hand-waved consistency is no longer enough.

ASD is already useful, but it is also unfinished in visible ways. I do not see that as a weakness. I see it as part of the discipline's honesty contract. A mature platform should say which capabilities are live, which are bounded, and which are still architectural intent. Several of the most interesting problems sit exactly in that gap.

OPEN PROBLEM	WHAT EXISTS NOW	WHY IT STILL MATTERS
Per-kind typed schemas	38 registered artifact-kind contracts, but per-kind runtime typed JSON schemas remain a Phase 2 boundary.	The vocabulary is richer than the currently enforced live forms.
ACP transport	`asd-acp/1` is record-only and fail-closed.	Typed records exist, but generic transport and spawn are not proven.
BUZZ persistence proofs	Article 37 proves a bounded observed slice with real mentions and Canvas indexing.	It does not yet prove durable workspace persistence, Critic verdicts, or reload/recovery end to end.
Host execution evidence	Selected packs and six fresh stdio bundles are evidenced; other hosts remain structural until a host receipt exists.	Emission is not the same thing as verified execution.

The first boundary is subtle but important. The platform knows 38 artifact-kind contracts and 39 catalog vocabulary rows, yet it is explicit that not every catalog row is already a live runtime form. That is good discipline. It stops teams from mistaking a documented idea for an enforced schema. Phase 2, in this context, means deeper per-kind typed runtime schemas that carry more of the artifact's structure directly into the write path. That would shrink ambiguity further, especially for report-like and orchestration-heavy outputs.

The ACP boundary is equally important. `asd-acp/1` already gives typed record envelopes for identity, capability negotiation, session, handoff, orchestration, and related events. But the current contract is record-only. A record can say a host-agent spawn was requested or unsupported. It does not itself create transport, relay, or execution. That might sound restrictive. I think it is actually a sign of maturity. Too many systems pretend a logged intention is the same as a verified capability. ASD refuses that shortcut.

acp-capability.json

```
{
  "protocolVersion": "asd-acp/1",
  "capabilities": [
    "record-only"
  ]
}
```

BUZZ exposes the same design ethic. The native Persona Pack is real. Ten keyless persona definitions are real. The safe operational sequence is real. The observed article slice is real. What is not yet fully proven is durable workspace persistence across the whole loop, visible Critic verdict persistence, ACP spawn, and recovery/reload/cleanup claims. That gap is not an embarrassment. It is exactly the kind of evidence boundary a trustworthy platform should publish before the market forces it to overclaim.

buzz-proof.sh

```
node scripts/buzz-live-identity-audit.mjs   --input ./audit.json   --channel-id 123e
buzz channels members --channel 123e4567-e89b-12d3-a456-426614174000
```

Then there is host execution evidence more broadly. The emitted targets are twelve: `agy`, `buzz`, `claude-code`, `cli`, `codex`, `dsh`, `grok`, `kimi`, `mcp`, `omp`, `opencode`, and `pi`. That generated breadth is useful. It is not a license to claim each host has been proven live in the same way. Selected surfaces have stronger evidence. Others are still

structural projections awaiting runtime receipts. I want more of those receipts, not more confidence theater.

Per-kind typed schemas are the most immediate boundary because they determine how far the runtime can move from general discipline into truly artifact-specific guarantees. Today the platform can enforce shared contracts such as node minimums, evidence minimums, review gates, and non-empty fields. That already eliminates a lot of weak work. But richer typed schemas per artifact kind would let the platform express deeper rules natively: lane dictionaries for blueprints, explanation structures for decisions, or stronger form semantics for reports and orchestration outputs. That is Phase 2 work I want precisely because it would make the human review more meaningful, not less.

Richer ACP transport is the second open problem. The current `asd-acp/1` layer is intentionally conservative. It records typed handoffs and capability events, but it does not pretend to be a universal session bus. A richer future ACP would need stronger proofs around spawn, relay, host capability discovery, and durable cross-host execution semantics. In other words, it would need to earn transport status instead of inheriting it rhetorically from its event vocabulary.

Live BUZZ persistence proofs are similarly specific. The bounded observed slice is already valuable because it proves the social choreography of identity, mentions, and Canvas linkage more honestly than many systems prove their whole runtime. Still, the unresolved questions remain real: what persists into the ASD workspace under reload, what survives recovery, how visible Critic verdicts remain, and how cleanup behaves over longer-lived channels. Those are not edge cases. They are the difference between a compelling live demo and a durable operating surface.

> **CAUTION:** Emission proves packaging. It does not prove execution. A credible agentic practice must be more conservative with runtime claims than with build claims.

The Designer's Changing Craft

So what becomes of the designer when more of the method spine is executable? I think the craft gets sharper, not thinner. The designer becomes less valuable as a human template expander and more valuable as a framer of boundaries, tests, owners, risks, and consequences. Someone still has to decide which uncertainty matters. Someone still has to smell when a metric is becoming coercive. Someone still has to decide where a human gate must harden because dignity, law, or politics outruns the schema.

That means the future service designer needs a wider competence stack. Classic service craft is still the floor: research quality, synthesis, journey logic, blueprint literacy, facilitation, prototyping, measurement. On top of that sit prompt anatomy, evidence literacy, repository discipline, tooling judgment, and governance design. I do not mean that every designer must become an engineer. I mean the designer has to understand enough of the substrate to shape how service decisions survive contact with software.

> **CORE PRINCIPLE:** In an agentic practice, the designer's highest leverage move is often defining the boundary of automation, not performing the automation.

The human line also gets more specific. Framing stays human. Ethics stays human. Vulnerability and dignity trade-offs stay human. Final escalation language in high-stakes domains stays human. Phase advance stays human. Taste stays human too, though I use that word carefully. By taste I mean the practiced judgment that knows when a service is technically consistent but socially wrong, when a clean-looking flow is hiding a burden shift, or when an audit pass still feels untrustworthy because the service has optimized the wrong thing.

The same broadening happens in team leadership. A future design lead needs to know when a method should become part of the shared library, when it should remain engagement-local, when a gate should fail closed, and when a team is mistaking

throughput for value. Those are product decisions about the practice itself. They require strategic patience, operational literacy, and enough humility to publish a boundary instead of covering it with language.

This is why I resist the lazy story that AI makes service design either obsolete or effortless. It does neither. It removes some mechanical drag. It raises the value of explicit judgment. It punishes vague ownership. It rewards designers who can hold evidence, operations, policy, and human experience in one frame at once. That is a demanding future. It is also a better one than the old theater of static artifacts and implied wisdom.

designer-judgment-boundary.yaml

```
must_remain_human:
  - service_framing
  - vulnerability_and_dignity_tradeoffs
  - final_escalation_language
  - approval_receipt_for_phase_gate
  - consequential_go_no_go_decisions
agent_supports:
  - evidence_synthesis
  - artifact_drafting
  - deterministic_checks
  - standards_regression
  - telemetry_refresh
```

In other words, the future designer is not the last manual craftsperson standing between machines and a deck. The future designer is the author of service constraints, the keeper of accountability, the builder of reviewable methods, and the person who notices when the whole system is getting good at the wrong thing. That is a serious profession. I want more of it, not less.

Build in the Open

If this book has one practical closing instruction, it is this: build the practice in the open enough that your claims can be challenged. That does not mean exposing private client data. It means publishing the method boundaries, the emitted packs, the receipts, the provenance, and the explicit unknowns. ASD already points in that direction. The distribution repos separate concerns cleanly: `imehr/asd-plugin` carries the coding-agent host packs and native Claude Code plugin surface; `imehr/asd-mcp` carries the standalone MCP server. Both are emitted from the canonical suite, and both can ship a `provenance.json` file that records the source commit. The public website at `sammozaffari.github.io/the-agentic-service-designer-site` makes the practice legible outside the repo.

provenance.json

```
{
  "distribution": "imehr/asd-plugin",
  "sourceCommit": "e94f006f2283",
  "emittedTargets": [
    "claude-code",
    "codex",
    "buzz"
  ]
}
```

I care about this because service design has a long history of hiding method quality behind charisma, bespoke decks, and selective storytelling. Build-in-the-open is the opposite move. Show the pack. Show the contract. Show the receipt. Show the unresolved boundary. Show the exact place where you stopped claiming more than the evidence allowed. That is how a field becomes trustworthy under automation.

The repos and the website are distribution surfaces, but the deeper commitment is cultural. When a method changes, the diff should be reviewable. When a gate fails, the reason should be inspectable. When a host is only structurally emitted, the docs should say so. When BUZZ proves one bounded slice and no more, the caption should say so. When ACP is record-only, the team should resist the urge to narrate it as transport. The future practice is more agentic and more precise about truth.

That open posture is why I care so much about the distribution split. `imehr/asd-plugin` and `imehr/asd-mcp` are not marketing side projects. They are accountability surfaces. The plugin repo proves what host packs actually receive. The MCP repo proves the standalone server contract. The website makes the practice legible to people who will never read the monorepo. Together they create a public trail from source practice to emitted distribution to human-readable explanation. That is how an agentic design system earns trust over time.

publish-distribution-repos.sh

```
node scripts/publish-distribution-repos.mjs
node platform/compiler/emit-packs.mjs --output /tmp/asd-packs
```

> **My take:** Build in the open because it disciplines the builder. A public provenance trail is a better teacher than private confidence.

There is also a cultural reason to prefer this explicitness. Once a practice starts publishing its receipts and its missing proofs side by side, internal conversations improve. Engineers stop treating design as handoff theater. Operators stop treating standards as abstract ideals. Designers stop pretending confidence can substitute for traceability. The work becomes calmer because the claims get smaller and truer.

That is where I want to leave this second edition. The agentic practice is not a fantasy of autonomous designers replacing human work. It is a way of making service design more durable, more explicit, more reviewable, and more accountable under real operating conditions. The tools are finally good enough to force the question. The answer, I think, is to encode what should be encoded, verify what can be verified, publish the evidence boundaries, and keep the consequential human judgments visible. If we do that, the field does not shrink in the age of agents. It grows up. And a grown-up discipline is exactly what services shaped by software, policy, operations, and AI now require. That is the future this platform is trying to practice in public.

CHAPTER 24

Mastercase: Meridian Bay Ferries

APPENDIX A

The ASD Method & Artifact Catalog

NEXT APPENDIX

The ASD Method & Artifact Catalog

Reference map of the 80-skill library, the 39-row artifact vocabulary, and the 38-kind runtime registry

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

APPENDIX A

The ASD Method & Artifact Catalog

Reference map of the 80-skill library, the 39-row artifact vocabulary, and the 38-kind runtime registry

Reading time

15 min

> **CORE PRINCIPLE:** In ASD, a method is executable only when a manifest, an output kind, acceptance rules, and a review gate all exist together. A memorable label in a workshop deck does not count as a runtime capability.

Method Library by Phase

The method library follows the six-phase practice loop rather than a grab bag of prompts. That matters. A researcher should not have to guess whether a method is meant to generate evidence, cluster it, turn it into concepts, or carry it into delivery. The counts are fixed: research 19, synthesis 15, ideation 13, prototyping 12, delivery 10,

measurement 11. Together they total the 80 shipped skills exposed by the CLI, MCP server, host packs, and Studio adapter.

The more subtle count is the source count. The repository holds 81 SKILL.md sources because the compiler keeps both the generated and the authored forms for one method, then resolves them with a flagship-wins rule. The method that gets that treatment is `service-blueprinting`. I am calling it out because it signals what deserves hand-authored depth in this platform: the work that most often collapses into diagram cosmetics unless the instructions are unusually explicit about lanes, waits, failures, handoffs, and evidence discipline.

PHASE	SKILLS	TYPICAL PURPOSE
research	19	Capture grounded evidence, plan studies, and structure interviews and observation.
synthesis	15	Turn raw evidence into patterns, maps, personas, and evidence-backed current-state structure.
ideation	13	Generate concepts, frame options, and connect insight to service propositions.
prototyping	12	Plan trials, write walkthroughs, test concepts, and gate prototype learning.
delivery	10	Translate proven concepts into standards, rollout logic, responsibility maps, and future-state blueprints.
measurement	11	Audit quality, define metrics, and connect service performance to business and frontline outcomes.

Research (19). The research phase is narrower than many teams expect. It is not a vague “discovery” bucket. It is the part of the system that builds trustworthy evidence, plans the next evidence capture, and denies unsupported claims before they leak forward. Most methods here write `insight-set` or `research-plan` outputs because the runtime wants the evidence register to stay legible.

SKILL ID	NAME	OUTPUT KIND	MINIMUM DECLARED INPUT	SOURCE REFS
`actor-network-mapping`	Actor Network Mapping	`actor-network-map`	`stakeholder-map` or `ecosystem-map` artifact ≥1	2
`contextual-inquiry-observation`	Contextual Inquiry & In-Context Observation	`insight-set`	evidence-set ≥1	2
`contextual-inquiry`	Contextual Inquiry	`insight-set`	evidence-set ≥3	2
`critical-incident-technique`	Critical Incident Technique	`insight-set`	evidence-set ≥1	2
`cultural-probes`	Cultural Probes	`insight-set`	evidence-set ≥1	2
`day-in-the-life-mapping`	Day-in-the-Life Mapping	`day-in-the-life`	evidence-set ≥2	2
`diary-study`	Diary Study	`insight-set`	evidence-set ≥1	2
`discussion-guide-design`	Discussion / Interview Guide Design	`interview-guide`	`research-plan` artifact ≥1	2
`expectation-mapping`	Expectation Mapping	`expectation-map`	evidence-set ≥2	2
`failure-demand-analysis`	Failure Demand Analysis	`insight-set`	evidence-set ≥1	2
`mystery-shopping`	Mystery Shopping	`insight-set`	evidence-set ≥1	2
`poems-field-observation`	POEMS Field Observation	`insight-set`	evidence-set ≥1	2

SKILL ID	NAME	OUTPUT KIND	MINIMUM DECLARED INPUT	SOURCE REFS
`research-plan-design`	Research Plan Design	`research-plan`	text brief	2
`secondary-competitive-audit`	Secondary & Competitive Research Audit	`insight-set`	evidence-set ≥1	2
`service-safari`	Service Safari	`insight-set`	evidence-set ≥1	2
`shadowing-frontline`	Shadowing Frontline Staff	`insight-set`	evidence-set ≥1	2
`stakeholder-mapping`	Stakeholder Mapping	`stakeholder-map`	evidence-set ≥2	2
`touchpoint-inventory-audit`	Touchpoint Inventory Audit	`touchpoint-inventory`	evidence-set ≥2	2
`transcript-mining`	Transcript Mining	`insight-set`	evidence-set ≥5	2

Synthesis (15). Synthesis is where the platform becomes opinionated. It does not treat clustering, personas, journeys, and blueprints as presentation choices. Each one is a separate contract with different minimums, different owner roles, and different failure modes. This is also where the flagship authored skill sits, because blueprinting is the moment sloppy teams start inventing process steps they cannot cite.

SKILL ID	NAME	OUTPUT KIND	MINIMUM DECLARED INPUT	SOURCE REFS
`aeiou-coding`	AEIOU Coding	`affinity-cluster-set`	evidence-set ≥3	2
`affinity-diagramming`	Affinity Diagramming	`insight-set`	evidence-set ≥5	2
`behavioural-personas`	Behavioural Persona	`persona-set`	evidence-set ≥5 plus `insight-`	3

SKILL ID	NAME	OUTPUT KIND	MINIMUM DECLARED INPUT	SOURCE REFS
	Construction		`set` artifact ≥1	
`ecosystem-value-mapping`	Ecosystem Value Mapping	`ecosystem-map`	evidence-set ≥3	4
`empathy-mapping`	Empathy Mapping	`empathy-map`	evidence-set ≥2	2
`evidence-register-build`	Evidence Register Build	`insight-set`	evidence-set ≥3	2
`experience-mapping`	Experience Mapping	`journey-map`	evidence-set ≥3	2
`insight-statement-formulation`	Insight Statement Formulation	`insight-set`	evidence-set ≥2	2
`jobs-to-be-done`	Jobs-to-be-Done Statement Construction	`insight-set`	evidence-set ≥2	2
`journey-from-transcripts`	Journey Mapping from Transcripts	`journey-map`	evidence-set ≥3	2
`jtbd-map-authoring`	Jobs-to-be-Done Map Authoring	`jobs-to-be-done`	`insight-set` artifact ≥1	2
`mental-model-alignment`	Mental Model Alignment	`journey-map`	evidence-set ≥3	3
`mental-model-diagramming`	Mental Model Diagramming	`mental-model-map`	evidence-set ≥3	2
`opportunity-matrix-2x2`	Opportunity Matrix (Impact x Feasibility)	`opportunity-matrix`	`insight-set` artifact ≥1	1
`service-blueprinting`	Service Blueprinting	`service-blueprint`	evidence-set ≥4	3

My take: The authored `service-blueprinting` skill is a tell. When a platform hand-authors one method and leaves the rest generated, it is quietly admitting where practitioners still need judgment coaching. In this case the answer is obvious: blueprinting fails when teams confuse a diagram with evidence-backed operating truth.

Ideation (13). The ideation phase is narrower than an innovation theatre workshop. ASD only keeps methods that can preserve traceability back to evidence or forward to a typed artifact. That is why wild idea generation sits next to scoring, canvas work, and concept briefs inside one bounded set.

SKILL ID	NAME	OUTPUT KIND	MINIMUM DECLARED INPUT	SOURCE REFS
`bodystorming`	Bodystorming	`concept-set`	text brief	2
`brainwriting-635`	Brainwriting 6-3-5	`concept-set`	`hmw-set` artifact ≥1	2
`business-model-canvas-build`	Business Model Canvas Build	`business-model-canvas`	`value-proposition-canvas` artifact ≥1	2
`co-design-workshop`	Co-Design Workshop	`concept-set`	evidence-set ≥1	2
`concept-brief-authoring`	Concept Brief Authoring	`concept-set`	`concept-set` artifact ≥1	2
`hmw-formulation`	How Might We Question Formulation	`hmw-set`	`insight-set` artifact ≥1	2
`morphological-concept-matrix`	Morphological Concept Matrix	`concept-set`	text brief	2
`nuf-triple-lens-filter`	NUF (New-Useful-Feasible) Triple-Lens Filter	`concept-set`	`concept-set` artifact ≥1	1

SKILL ID	NAME	OUTPUT KIND	MINIMUM DECLARED INPUT	SOURCE REFS
`orchestration-canvas`	Orchestration Canvas	`ecosystem-map`	`ecosystem-map` artifact ≥1 plus evidence-set ≥1	3
`service-scenario-authoring`	Service Scenario Authoring	`service-scenario`	`concept-set` or `persona-set` artifact ≥1	2
`speed-dating-concepts`	Speed Dating Concepts	`concept-set`	`concept-set` artifact ≥2	2
`story-mapping`	Story Mapping	`story-map`	`journey-map` or `insight-set` artifact ≥1	2
`value-proposition-canvas-build`	Value Proposition Canvas Build	`value-proposition-canvas`	evidence-set ≥2	2

Prototyping (12). ASD treats prototyping as service rehearsal and evidence production, not UI polish. Several methods here produce `test-report` or `decision-matrix` outputs rather than a new concept artifact. That is deliberate. The goal is to learn, narrow, and either move forward or kill a weak idea without lying about the evidence.

SKILL ID	NAME	OUTPUT KIND	MINIMUM DECLARED INPUT	SOURCE REFS
`cognitive-walkthrough`	Cognitive Walkthrough	`test-report`	`prototype-plan` artifact ≥1	2
`concierge-pilot`	Concierge Pilot	`prototype-plan`	`service-blueprint` artifact ≥1	2
`desirability-testing`	Desirability Testing	`test-report`	`concept-set` and `prototype-plan` artifacts ≥1 plus evidence-set ≥3	2

SKILL ID	NAME	OUTPUT KIND	MINIMUM DECLARED INPUT	SOURCE REFS
`desktop-walkthrough`	Desktop Walkthrough	`walkthrough-script`	`service-blueprint` and `concept-set` artifacts ≥1	2
`heuristic-evaluation`	Heuristic Evaluation	`test-report`	`prototype-plan` artifact ≥1	2
`pilot-test-report`	Pilot Test Report	`test-report`	evidence-set ≥3	2
`prototype-ladder-gating`	Prototype Ladder Gating	`decision-matrix`	`test-report` artifact ≥1	2
`rite-iteration`	RITE Iteration	`decision-matrix`	evidence-set ≥1	2
`service-roleplay`	Service Roleplay	`storyboard`	`journey-map` and `concept-set` artifacts ≥1	2
`storyboard-authoring`	Storyboard Authoring	`storyboard`	evidence-set ≥1	2
`usability-testing-moderated`	Moderated Usability Testing	`test-report`	`prototype-plan` artifact ≥1 plus evidence-set ≥3	2
`wizard-of-oz`	Wizard of Oz Prototyping	`prototype-plan`	`concept-set` artifact ≥1	2

Delivery (10). Delivery is where the method system stops flattering speculative work. Everything here assumes a design artifact already exists and now has to become a rollout plan, standard, queue decision, or future-state operating logic. The methods are fewer because they are downstream of earlier proof.

SKILL ID	NAME	OUTPUT KIND	MINIMUM DECLARED INPUT	SOURCE REFS
`blueprint-to-backlog`	Blueprint to Backlog	`roadmap`	`service-blueprint` artifact ≥1	2
`change-pack-authoring`	Change Pack Authoring	`change-plan`	`service-blueprint` and `roadmap` artifacts ≥1	2
`evidence-gated-blueprint`	Evidence-Gated Service Blueprint	`service-blueprint`	evidence-set ≥5 plus `journey-map` artifact ≥1	4
`frontline-enablement`	Frontline Enablement	`service-standard-set`	`service-blueprint` and `service-standard-set` artifacts ≥1	2
`future-state-blueprinting`	Future-State Blueprinting	`service-blueprint`	`service-blueprint` artifact ≥1	2
`implementation-roadmap`	Implementation Roadmap	`roadmap`	`roadmap` and `change-plan` artifacts ≥1	2
`operations-queue-analysis`	Operations Queue Analysis	`decision-matrix`	evidence-set ≥3	2
`raci-definition`	RACI Definition	`raci-matrix`	`service-blueprint` and `roadmap` artifacts ≥1	2
`service-standard-authoring`	Service Standard Authoring	`service-standard-set`	`service-blueprint` and `journey-map` artifacts ≥1	2

SKILL ID	NAME	OUTPUT KIND	MINIMUM DECLARED INPUT	SOURCE REFS
`wave-gated-rollout`	Wave-Gated Rollout	`roadmap`	`roadmap` artifact ≥1	2

Measurement (11). Measurement closes the loop. Some methods here audit the lived experience. Others connect service change to KPI structure, maturity, recovery, or profit-chain logic. None of them are decorative. If I cannot point to a measurable service consequence, ASD assumes I am still describing intent, not performance.

SKILL ID	NAME	OUTPUT KIND	MINIMUM DECLARED INPUT	SOURCE REFS
`customer-typology-mapping`	Customer Typology Mapping	`persona-set`	evidence-set ≥3	2
`cx-maturity-assessment`	CX Maturity Assessment	`maturity-assessment`	evidence-set ≥2	2
`experience-audit`	Experience Audit	`experience-audit`	`service-blueprint` and `journey-map` artifacts ≥1 plus evidence-set ≥2	2
`good-services-audit-15`	Good Services 15-Principle Audit	`insight-set`	`service-blueprint` and `journey-map` artifacts ≥1 plus evidence-set ≥2	2
`good-services-audit`	Good Services Audit	`insight-set`	`service-blueprint` and `journey-map` artifacts ≥1 plus evidence-set ≥2	2
`kpi-tree-design`	KPI Tree Design	`kpi-tree`	`service-blueprint` and `journey-map` artifacts ≥1	2

SKILL ID	NAME	OUTPUT KIND	MINIMUM DECLARED INPUT	SOURCE REFS
`service-profit-chain-mapping`	Service Profit Chain Mapping	`service-profit-chain`	evidence-set ≥2	2
`service-quality-diagnostic`	Service Quality Diagnostic	`experience-audit`	evidence-set ≥3	2
`service-recovery-design`	Service Recovery Design	`recovery-playbook`	evidence-set ≥3	2
`three-tier-measurement`	Three-Tier Measurement	`kpi-tree`	`service-blueprint` and `kpi-tree` artifacts ≥1	2
`wwhtbt-evidence-gate`	What-Would-Have-To-Be-True Evidence Gate	`decision-matrix`	`service-blueprint` and `roadmap` artifacts ≥1	2

Artifact Vocabulary: Canonical, Alias, Pending

The vocabulary catalog has 39 rows, but they do not all mean the same thing. This is the trap I see readers fall into first. A familiar label in the catalog is not automatically a live runtime kind. The catalog deliberately mixes three statuses: 18 canonical labels that name shipped runtime kinds, 13 aliases that keep practitioner language alive without multiplying schemas, and 8 pending concepts that remain vocabulary only until registry, template, renderer, and skill gates exist.

That split is healthy. Teams speak in aliases all the time. They say “insight register” instead of `insight-set`, “standards audit” instead of `service-standard-set`, and “storyboard walkthrough” instead of `storyboard`. ASD lets the language survive in the catalog while keeping the runtime set small enough to reason about. Pending rows

stay visible so I can talk about future platform scope without pretending it already runs.

CANONICAL LABEL	RUNTIME KIND	WHY IT MATTERS
`research-plan`	`research-plan`	Turns research intent into a typed project instrument.
`interview-guide`	`interview-guide`	Keeps interviews structured and reviewable.
`stakeholder-map`	`stakeholder-map`	Captures the actor field early.
`ecosystem-map`	`ecosystem-map`	Shows value flows and dependencies across actors.
`persona-set`	`persona-set`	Stores behaviour-backed personas, not demographic fiction.
`empathy-map`	`empathy-map`	Preserves the classic says/does/thinks/feels lens.
`journey-map`	`journey-map`	Represents stages, channels, emotion, pain, and evidence.
`jobs-to-be-done`	`jobs-to-be-done`	Records what customers hire the service to do as evidence, not feature wishlists.
`service-blueprint`	`service-blueprint`	Holds the system view of customer, frontstage, backstage, and support work.
`story-map`	`story-map`	Frames narrative structure for concept development.
`value-proposition-canvas`	`value-proposition-canvas`	Connects pains, gains, and service fit.
`business-model-canvas`	`business-model-canvas`	Links service logic to economic logic.
`prototype-plan`	`prototype-plan`	Defines what a trial is supposed to test.

CANONICAL LABEL	RUNTIME KIND	WHY IT MATTERS
`test-report`	`test-report`	Records what actually happened during evaluation.
`decision-matrix`	`decision-matrix`	Gives scored comparison structure.
`experience-audit`	`experience-audit`	Audits a live service against experience criteria.
`opportunity-matrix`	`opportunity-matrix`	Ranks opportunities for sequence and leverage.
`review-veto-log`	`review-veto-log`	Captures the Critic's administrative verdict rows.

ALIAS LABEL	RESOLVES TO RUNTIME KIND	WHY THE ALIAS EXISTS
`insight-register`	`insight-set`	Practitioners often think in registers before they think in set schemas.
`affinity-diagram`	`affinity-cluster-set`	Keeps the common wall-clustering label.
`mental-model`	`mental-model-map`	Shorter language for the same executable contract.
`experience-map`	`journey-map`	Lets teams speak broadly without creating a second map schema.
`orchestration-canvas`	`ecosystem-map`	Preserves the practice lens while reusing the actor-flow runtime kind.
`hmw-register`	`hmw-set`	Common workshop phrasing for HMW work.
`concept-brief`	`concept-set`	Many teams frame concept artifacts as briefs.
`storyboard-walkthrough`	`storyboard`	Signals use in service rehearsal.
`kpi-set`	`kpi-tree`	Shorter language for a structured measurement tree.

ALIAS LABEL	RESOLVES TO RUNTIME KIND	WHY THE ALIAS EXISTS
`standards-audit`	`service-standard-set`	Readable practice label for standards work.
`maturity-model`	`maturity-assessment`	Keeps the familiar maturity-language surface.
`recovery-plan`	`recovery-playbook`	Lets operational teams ask for a plan and receive the richer playbook type.
`frontline-enablement`	`service-standard-set`	Maps enablement work onto the standards runtime contract.

PENDING LABEL	CURRENT STATUS	WHY IT STAYS VISIBLE
`design-brief`	pending	Useful practice term, but no shipped executable kind yet.
`reframe-register`	pending	Captures framing work the platform documents but does not yet execute directly.
`co-design-plan`	pending	A real workshop concept that still lacks full runtime support.
`evaluation-framework`	pending	Named in practice vocabulary without a live artifact contract.
`executive-report`	pending	Handled today as a rendered report kind rather than a typed artifact kind.
`presentation-deck`	pending	Visible as a practice deliverable, not yet a runtime artifact contract.
`orchestration-plan`	pending	Operational records exist, but the practice-facing taxonomy row remains pending.
`phase-gate-decision`	pending	Human gate records exist operationally, but the user-facing vocabulary row still waits for a full canonical promotion.

Registered Runtime Kinds

The runtime registry contains 38 artifact-kind contracts. That is the number the engine can persist, export, and look up through schema metadata today. I state it this way because the registry is not the same thing as the 39-row vocabulary. The clean mental model is this: 18 catalog rows are canonical and executable, 13 catalog rows are aliases that collapse onto existing executable kinds, 8 rows are still pending, and 10 runtime kinds exist only in the registry because they are operationally useful without being part of the public template promise.

Those ten registry-only kinds are `actor-network-map`, `change-plan`, `day-in-the-life`, `expectation-map`, `raci-matrix`, `roadmap`, `service-profit-chain`, `service-scenario`, `touchpoint-inventory`, and `walkthrough-script`. They are real runtime contracts. They are simply outside the 39-row practice-catalog claim. That is why the arithmetic feels strange until you separate vocabulary from execution.

ALIAS-BACKED RUNTIME KIND	RUNTIME TITLE	NODE LABEL	SURFACED PRACTICE LABEL(S)
`affinity-cluster-set`	Affinity Cluster Set	cluster	`affinity-diagram`
`concept-set`	Concept Set	concept	`concept-brief`
`hmw-set`	How Might We Question Set	question	`hmw-register`
`insight-set`	Insight Set	insight	`insight-register`
`kpi-tree`	KPI Tree	kpi	`kpi-set`
`maturity-assessment`	CX Maturity Assessment	dimension	`maturity-model`
`mental-model-map`	Mental Model Map	tower	`mental-model`

ALIAS-BACKED RUNTIME KIND	RUNTIME TITLE	NODE LABEL	SURFACED PRACTICE LABEL(S)
`recovery-playbook`	Service Recovery Playbook	scenario	`recovery-plan`
`service-standard-set`	Service Standard Set	standard	`standards-audit`, `frontline-enablement`
`storyboard`	Storyboard	frame	`storyboard-walkthrough`

REGISTRY-ONLY KIND	RUNTIME TITLE	NODE LABEL	BOUNDARY
`actor-network-map`	Actor Network Map	actor	Real runtime kind, outside the 39-row catalog promise.
`change-plan`	Change Plan	action	Real runtime kind, outside the 39-row catalog promise.
`day-in-the-life`	Day-in-the-Life Map	moment	Supports research evidence structure without widening the practice vocabulary.
`expectation-map`	Expectation Map	expectation	Operationally useful across research and measurement handoffs, but not an authored template claim.
`raci-matrix`	RACI Matrix	assignment	Operationally useful in delivery, but not an authored template claim.
`roadmap`	Implementation Roadmap	milestone	Shipped in the registry and renderer set without a flagship template claim.
`service-profit-chain`	Service Profit Chain Map	link	Supports measurement work without widening the practice vocabulary.
`service-scenario`	Service Scenario	beat	Supports ideation and rehearsal while staying registry-only.
`touchpoint-inventory`	Touchpoint Inventory	touchpoint	Shipped in the registry and renderer set without a flagship template claim.

REGISTRY-ONLY KIND	RUNTIME TITLE	NODE LABEL	BOUNDARY
`walkthrough-script`	Walkthrough Script	step	Supports prototyping and rehearsal while staying registry-only.

> **CAUTION:** Do not inflate the runtime registry to 39 by counting aliases or pending rows as executable kinds. That mistake cascades into bad host-pack claims, bad schema assumptions, and bad migration work.

Authored Templates

The authored template tier is deliberately small: seven fillable forms, each backed by a Markdown source in `platform/templates/` and a runtime-aware metadata record in `suite/packages/knowledge/data/templates/`. These are the forms I can hand to a human team before any agent touches the project. They are also the templates most likely to survive a workshop and still compile into useful runtime artifacts later.

TEMPLATE ID	RUNTIME KIND	PHASE	SOURCE PATH
`stakeholder-map`	`stakeholder-map`	research	`platform/templates/05-stakeholder-map/template.md`
`ecosystem-map`	`ecosystem-map`	research	`platform/templates/06-ecosystem-map/template.md`
`insight-register`	`insight-set`	synthesis	`platform/templates/07-insight-register/template.md`
`persona-set`	`persona-set`	synthesis	`platform/templates/09-persona-set/template.md`
`journey-map`	`journey-map`	synthesis	`platform/templates/11-journey-map/template.md`

TEMPLATE ID	RUNTIME KIND	PHASE	SOURCE PATH
`service-blueprint`	`service-blueprint`	synthesis	`platform/templates/12-service-blueprint/template.md`
`jobs-to-be-done`	`jobs-to-be-done`	synthesis	`platform/templates/39-jobs-to-be-done/template.md`

That seven-template set is intentional. ASD is not promising a fillable form for every vocabulary row. It is promising depth where teams repeatedly need scaffolding: stakeholders, ecosystems, insights, personas, journeys, blueprints, and jobs. Everything else must earn authored-template status through the same contract discipline as a new runtime kind.

The Admin Record: review-veto-log

`review-veto-log` is the cleanest example of the platform's governance posture. It is a registered runtime kind, it is canonical in the vocabulary, and it is intentionally administrative. Users do not fill it out as a workshop artifact. The Critic writes it when a review happens. That distinction matters because the log is the memory of why a pass was granted or a veto was issued, and the rest of the pipeline can replay it without guessing.

The record is always final from birth. That is the point. A veto log that can drift in and out of draft state would be governance theatre. Instead, `sd_review` appends rows across verdicts, keeps row numbering stable, links the log back to the reviewed artifact through `reviewOf`, and enforces the basic discipline: a veto needs at least one defect row with a fix, a pass cannot smuggle in defect rows, and a pass must cite evidence in at least one proposal row.

FIELD OR RULE	MEANING
`kind = review-veto-log`	The artifact is an administrative control record, not a user-authored deliverable.
`status = final`	The log itself is final when written so the decision trail cannot hide behind draft status.
`reviewOf`	Links the log to the reviewed artifact id.
`rowKind`	`defect` or `proposal`; vetoes require at least one defect.
`fix`	Mandatory on defect rows so a veto is actionable.
`evidenceId`	When supplied, it must resolve in the project evidence store.
Human gate boundary	The Critic can pass or veto artifacts, but phase advance remains human-only. A good review log prepares a decision; it never authorizes one.

review-veto-log.example.json


```
{
  "kind": "review-veto-log",
  "status": "final",
  "reviewOf": "7f9fefdc44f5469195106e739112f9ae",
  "nodes": [
    {
      "id": "r1",
      "label": "All three nodes cite zero evidence rows; the acceptance gate cannot
      "fields": {
        "rowKind": "defect",
        "severity": "high",
        "rule": "min-evidence-per-node",
        "message": "All three nodes cite zero evidence rows; the acceptance gate can
        "fix": "Re-derive each node from the register with verbatim quotes attached"
      }
    }
  ]
}
```

If I had to reduce this appendix to one sentence, it would be this: ASD is strict about names because names decide what can be executed, what can be reviewed, and what remains honest future scope. The method count, the vocabulary count, the runtime-kind count, and the template count are all true at the same time once you stop forcing them to mean the same thing.

CHAPTER 25

The Future of Service Design: The Agentic Practice

APPENDIX B

CLI & MCP Tool Specification

NEXT APPENDIX

CLI & MCP Tool Specification

All 30 sd_* tools, the client wiring rules, the protocol facts, and the resource surface that sits behind them

Continue Reading →

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

APPENDIX B

CLI & MCP Tool Specification

All 30 sd_* tools, the client wiring rules, the protocol facts, and the resource surface that sits behind them

Reading time

13 min

> **CORE PRINCIPLE:** The CLI parser is generated from the same JSON schemas that the MCP server advertises over `tools/list`. That is why the terminal surface and the stdio surface stay aligned instead of drifting into two separate products.

CLI Command Families

The canonical terminal form is short and boring. Good. I want a runtime surface that can be scripted without ceremony: `asd <tool-name> [--flag value]`, with the short name derived from the engine tool by stripping `sd_` and replacing underscores with hyphens. The pack-local install path is fixed, Node 20 or newer is required, and the launcher never falls back to a global executable or a sibling checkout. That last point matters in release verification because the bundle is supposed to prove it can stand on its own.

install-cli-pack.sh

```
node platform/compiler/emit-packs.mjs --output /tmp/asd-packs
PACK=/tmp/asd-packs/cli
mkdir -p "$HOME/.local/bin"
ln -sf "$PACK/asd" "$HOME/.local/bin/asd"
export PATH="$HOME/.local/bin:$PATH"
asd --version
asd help
```

Flags map one-to-one onto tool arguments. Strings stay strings. Numbers, booleans, arrays, and objects are validated before the tool runs. Arrays and objects travel as JSON literals on the command line. Unknown flags, missing required flags, positional arguments, and invalid enum values exit 1 with a real error instead of a fuzzy best effort.

FAMILY	CLI COMMAND	ENGINE ALIAS	FLAGS	WHA
Project	`asd project-list`	`sd_project_list`	—	Lis in t
Project	`asd project-create`	`sd_project_create`	`--name --client --sector --service [--phase]`	Cre `pr` plu `ar` `re` `sc`
Project	`asd project-open`	`sd_project_open`	`--projectId`	Op an me art
Project	`asd project-status-set`	`sd_project_status_set`	`--projectId --status [--statusNote]`	Se en life the col ind the `st` en `ac` `ga` `bl` `ar`
Evidence	`asd evidence-add`	`sd_evidence_add`	`--projectId --source --kind --body`	Ad `te`

FAMILY	CLI COMMAND	ENGINE ALIAS	FLAGS	WHA
				do rec
Evidence	`asd evidence-import`	`sd_evidence_import`	`--projectId --path --source [--encoding] [--maxBytes] [--onError]`	Imp CS evi ins AS
Evidence	`asd evidence-query`	`sd_evidence_query`	`--projectId [--search] [--kind]`	Se pro reg
Artifacts	`asd artifact-list`	`sd_artifact_list`	`--projectId [--kind]`	Lis typ
Artifacts	`asd artifact-get`	`sd_artifact_get`	`--projectId --artifactId`	Fet in f no ref sta
Artifacts	`asd artifact-write`	`sd_artifact_write`	`--projectId --methodId --title --nodes [--edges] [--references] [--artifactId]`	Wr pro thr ac
Methods	`asd method-list`	`sd_method_list`	`[--phase] [--producesKind]`	Lis fro
Methods	`asd method-run`	`sd_method_run`	`--methodId --projectId`	Re me ste ac
Review	`asd review`	`sd_review`	`--projectId --artifactId --verdict --rows`	Re pa evi rev
Review	`asd review-list`	`sd_review_list`	`--projectId [--status]`	Sh bo dra art

FAMILY	CLI COMMAND	ENGINE ALIAS	FLAGS	WHA
Export	`asd export-report`	`sd_export_report`	`--projectId --kind [--generatedAt]`	Re de HT
Export	`asd export-artifact`	`sd_export_artifact`	`--projectId --artifactId [--width]`	Re art de SV
Knowledge	`asd knowledge`	`sd_knowledge`	`[--query] [--phase] [--source] [--resourceQuery] [--resource] [--type]`	Qu kin op res
Factcheck	`asd factcheck`	`sd_factcheck`	`--projectId`	Re ac rev 1 o fa
Templates/schema	`asd template-list`	`sd_template_list`	—	Lis au ter
Templates/schema	`asd template-get`	`sd_template_get`	`--id`	Re au ter bo
Templates/schema	`asd schema-get`	`sd_schema_get`	`--kind`	Re reg art
Templates/schema	`asd practice-get`	`sd_practice_get`	—	Re ma rea co
Canvas	`asd canvas-add`	`sd_canvas_add`	`--projectId --label --url [--tool]`	Lin de
Canvas	`asd canvas-list`	`sd_canvas_list`	`--projectId`	Lis ca
Canvas	`asd canvas-remove`	`sd_canvas_remove`	`--projectId --canvasId`	Re ca

FAMILY	CLI COMMAND	ENGINE ALIAS	FLAGS	WHA
				wit the
Orchestration	`asd orchestration-plan`	`sd_orchestration_plan`	`--projectId --idempotencyKey --title --steps`	Per ind pla
Orchestration	`asd orchestration-status`	`sd_orchestration_status`	`--projectId [--planId]`	Re dis har
Orchestration	`asd orchestration-dispatch`	`sd_orchestration_dispatch`	`--projectId --planId --stepId --idempotencyKey --capability [--outcome] [--error] [--retryable] [--retryOf]`	Per dis
Orchestration	`asd orchestration-handoff`	`sd_orchestration_handoff`	`--projectId --planId --idempotencyKey --from --to --condition [--artifactId] [--artifactKind]`	Per rol an art
Phase	`asd phase-advance`	`sd_phase_advance`	`--projectId --toPhase --idempotencyKey --approvalReceipt`	Re ap tra

Outside the 30 engine tools, the CLI also exposes `asd help` and `asd --version`. Everything else is the registry. That separation keeps the book simple: if a command begins with a practice verb such as `project-create`, `method-run`, or `phase-advance`, it is really the schema-backed tool under the hood.

Two CLI behaviors are worth memorizing because they surprise people coming from softer tools. First, booleans can be bare flags or explicit `true` / `false`, but arrays and objects must be valid JSON literals. Second, exit code 1 is overloaded on purpose: it means either a real usage or runtime error, or a completed gate returning a failing verdict such as `sd_factcheck` with `\"pass\": false`. That makes CI wiring simple. A

green-looking report with a failing top-level gate cannot quietly slip through as success.

MCP Tools by Family

The MCP surface exposes the same 30 tools over newline-delimited JSON-RPC 2.0 on stdio. The naming stays explicit: `sd_project_create`, `sd_method_run`, `sd_review`, and so on. Hosts commonly prefix those names again when they mount the bundle, which is why a Claude Code transcript may show `mcp__asd__sd_method_run` even though the underlying tool remains `sd_method_run`.

What matters operationally is the family semantics. Project tools work on the workspace tree. Evidence tools allow only three evidence kinds: `quote`, `telemetry`, and `document`. Artifact writes always persist honestly: acceptance success still lands as draft when Critic review is outstanding, acceptance failure persists a draft with machine-readable failures, and a dangling evidence reference is rejected before any write. Review and phase tools are the governance spine, and they keep human authority separate from automated checks.

FAMILY	MCP TOOLS	KEY ARGUMENTS	BOUNDARY W REMEMBERIN
Project	`sd_project_list`, `sd_project_create`, `sd_project_open`	`name`, `client`, `sector`, `service`, optional `phase`; or `projectId` for open	Projects ar on disk. Th no hidden database.
Project	`sd_project_status_set`	`projectId`; `status` enum `briefed`, `active`, `gated`, `blocked`, `archived`; optional `statusNote`	Lifecycle st is the Kanb column, independer design pha Legal trans run `briefe` `active → g` `→ blocked` `archived`; `archived` reachable f any state a `blocked ↔` `active` m both ways.

FAMILY	MCP TOOLS	KEY ARGUMENTS	BOUNDARY W REMEMBERIN
			same-statu is a no-op u a note accompani An unknow `projectId` fails, an ille transition is rejected na the legal ta and a non- status exits
Evidence	`sd_evidence_add`, `sd_evidence_import`, `sd_evidence_query`	`kind` is closed to `quote`, `telemetry`, `document`; import adds `path`, `encoding`, `maxBytes`, `onError`	Import neve fetches UR reads only workspace files and fa closed on i roots and malformed inputs.
Artifacts	`sd_artifact_list`, `sd_artifact_get`, `sd_artifact_write`	Write requires `projectId`, `methodId`, `title`, `nodes`; optional `edges`, `references`, `artifactId`	A failed acceptance check is no crash. It is a persisted d with failure
Methods	`sd_method_list`, `sd_method_run`	List can filter by `phase` or `producesKind`; run requires `methodId` and `projectId`	`sd_method` returns the contract. It not do the for you.
Review	`sd_review`, `sd_review_list`	`verdict` is `pass` or `veto`; `rows` carry `rowKind`, `severity`, `message`, optional `rule`, `fix`, `evidenceId`	A veto nee least one d with a fix. A needs at le one eviden cited row a cannot con defects.

FAMILY	MCP TOOLS	KEY ARGUMENTS	BOUNDARY W REMEMBERIN
Export	`sd_export_report`, `sd_export_artifact`	Reports require `kind`; artifact export optionally takes `width`	Both outpu determinist derived file under `renders/`
Knowledge	`sd_knowledge`	`query`, `phase`, `source`, `resourceQuery`, `resource`, `type`	This tool ca search the registry and bundled co in one call.
Factcheck	`sd_factcheck`	`projectId`	It replays integrity, acceptance review. Exit failure is a feature, not exception.
Templates/schema	`sd_template_list`, `sd_template_get`, `sd_schema_get`, `sd_practice_get`	`id` for template lookup, `kind` for schema lookup, no arguments for practice contract	The seven authored templates d imply one template p catalog row
Canvas	`sd_canvas_add`, `sd_canvas_list`, `sd_canvas_remove`	`label`, `url`, optional `tool` where the enum is `figma`, `penpot`, `paper`, `pencil`, `other`	Canvas link project rec They do no replace typ artifact writ
Orchestration	`sd_orchestration_plan`, `sd_orchestration_status`, `sd_orchestration_dispatch`, `sd_orchestration_handoff`	Plan requires `idempotencyKey`, `title`, `steps[{id, role, instruction, maxRetries?}]`; dispatch adds `stepId`, `capability`, optional `outcome`, `error`, `retryable`, `retryOf`; handoff uses `from`, `to`, `condition`, optional `artifactId`, `artifactKind`	`record-or` the generic supported dispatch capability. `host-ager` `spawn` rem an explicit unsupporte path unless host adapt really implements
Phase	`sd_phase_advance`	`toPhase`, `idempotencyKey`, `approvalReceipt{receiptId,`	The only le receipt dec

FAMILY	MCP TOOLS	KEY ARGUMENTS	BOUNDARY W REMEMBERIN
		`approvedBy, decision, rationale, approvedAt}`	for advance `approve`, the human receipt nev gets inferre from a pass artifact.

My take: The orchestration family is deliberately humble. It records plans, dispatch attempts, retries, and handoffs. It does not pretend that writing a plan means the host actually launched an agent. That boundary keeps operational evidence clean.

Resources and Prompts

The MCP surface serves the corpus as resources and the role library as prompts. When the corpus roots are present, `resources/list` returns 228 resources total. The page size is 100, so a client that wants the full inventory must keep paginating by cursor until the list is exhausted. The URI space is intentionally small:

URI PATTERN	WHAT IT SERVES
`asd://books/<book>`	One book overview resource from the 20-book corpus.
`asd://books/<book>/<chapter>`	One chapter resource from the book corpus.
`asd://articles/<NN>`	One article resource from the method and case-study series.

`resources/read` returns exact UTF-8 payload, MIME type, version, size, and logical provenance, never an absolute filesystem path. If the runtime is copied without its corpus roots, the list can legitimately come back empty while the 30 runtime tools still work. That is a visible boundary, not a silent fallback.

PROMPT NAME	ROLE
`orchestrator`	Coordinates plans, handoffs, and human gates.

PROMPT NAME	ROLE
`researcher`	Captures evidence and structures discovery work.
`synthesizer`	Clusters, interprets, and sharpens evidence into insight.
`mapper`	Builds journeys, ecosystems, mental models, and blueprints.
`ideator`	Generates and filters service concepts.
`prototyper`	Designs trials, walkthroughs, and prototype learning loops.
`measurer`	Defines audit, measurement, maturity, and KPI logic.
`critic`	Runs the evidence-cited review gate.
`presenter`	Packages the work for human consumption without changing its facts.
`librarian`	Maintains the catalog and curates knowledge growth.

`prompts/get` returns the shared role contract followed by the role-specific instruction set as one user message. If `ASD_AGENTS_ROOT` is unavailable, the prompt list is empty by design. Again: explicit boundary, no fake fallback.

Client Configurations

The pack-local rule is simple: the emitted bundle's launcher path is relative to the pack itself. When a client runs directly from the installed pack, the relative entrypoint is correct. When a client copies the snippet elsewhere, replace the relative path with an absolute path. The server launcher does not search `PATH` for a spare binary.

asd-mcp.json

```
{
  "mcpServers": {
    "asd": {
      "command": "node",
      "args": ["mcp/bin/server.mjs"]
    }
  }
}
```

Claude Code uses the plugin-root variant:

.mcp.json

```
{
  "mcpServers": {
    "asd": {
      "command": "node",
      "args": ["${CLAUDE_PLUGIN_ROOT}/mcp/bin/server.mjs"]
    }
  }
}
```

OpenCode uses the command-array form:

opencode.json

```
{
  "$schema": "https://opencode.ai/config.json",
  "mcp": {
    "asd": {
      "type": "local",
      "command": ["node", "mcp/bin/server.mjs"]
    }
  }
}
```

Codex supports either the registration command or a TOML block:

codex-mcp-add.sh

```
codex mcp add asd -- node "$PWD/platform/packs/codex/mcp/bin/server.mjs"
```

codex.toml

```
[mcp_servers.asd]
command = "node"
args = ["mcp/bin/server.mjs"]
```

Those four snippets are enough to capture the pack-local rule and the three host-specific variants readers keep asking about. The operational nuance is that each host

chooses its own working directory, which is why the relative bundle path is a rule of the installed pack, not a universal truth of the client machine.

Protocol and Error Codes

The stdio server speaks newline-delimited JSON-RPC 2.0 with MCP revision `2025-06-18`. The lifecycle starts with one `initialize` call before `tools/list`, `tools/call`, `resources/list`, `resources/read`, `prompts/list`, or `prompts/get`. The response advertises tools, resources, and prompts, and identifies the server as `asd-mcp` version `0.1.0`. Tool failures return MCP content blocks with `isError: true`; protocol framing remains valid.

initialize.json

```
{"jsonrpc":"2.0","id":1,"method":"initialize","params":{"protocolVersion":"2025-06-1
```

Default tool timeout is 30 seconds. The frame limit is 1 MiB. Cancellation uses `$/cancelRequest`. Notifications remain silent. None of that is exotic, but it is precisely the kind of detail that matters when a client blames the server for behavior that actually belongs to the transport contract.

CODE	MEANING
`-32700`	Parse error. The incoming JSON frame is invalid.
`-32601`	Method not found.
`-32602`	Invalid params, unknown tool or prompt, bad cursor, or non-object arguments.
`-32603`	Unexpected internal handler failure.
`-32000`	Initialize required before the client can use the surface.
`-32001`	Tool call timed out.
`-32002`	Resource not found.

One more boundary belongs here because it is easy to overclaim: ACP on the MCP surface is `asd-acp/1` with capability `record-only`. The runtime can record

orchestration plans, dispatch attempts, handoffs, and human gates. It does not prove that the host launched a new process. `host-agent-spawn` therefore remains an explicit unsupported path on the generic surface unless a host adapter really wires it up.

Environment and Workspace

`ASD_WORKSPACE` sets the root directory for projects. If I do nothing, the runtime uses `~/.asd/projects`. That default is fine for live use. It is a bad idea for tests, demos, and disposable verification, which is why the docs push isolated temporary roots. The project tree itself is plain: `project.json`, `evidence/index.json`, artifact JSON files, exported renders, and `sops/`. Files are the record of truth.

workspace-isolation.sh

```
export ASD_WORKSPACE="$(mktemp -d)/projects"
mkdir -p "$ASD_WORKSPACE"
asd project-create --name "Metrovale Access" --client "Fictional composite" \
  --sector public-transport --service "step-free journeys" --phase discover
```

The MCP server adds three corpus-root environment variables beside the workspace root: `ASD_BOOKS_ROOT`, `ASD_ARTICLES_ROOT`, and `ASD_AGENTS_ROOT`. They control the served books, articles, and role prompts independently. That means a client can still use the 30 runtime tools even when the book corpus or prompt roots are unavailable.

VARIABLE	DEFAULT OR TYPICAL VALUE	WHAT IT CONTROLS
`ASD_WORKSPACE`	`~/.asd/projects`	Project file root for CLI and MCP runtime operations.
`ASD_BOOKS_ROOT`	pack or checkout path	Book corpus resources under `asd://books/<book>`.
`ASD_ARTICLES_ROOT`	pack or checkout path	Article corpus resources under `asd://articles/<NN>`.
`ASD_AGENTS_ROOT`	pack or checkout path	Role prompt files exposed through `prompts/list` and `prompts/get`.

> **CAUTION:** Release evidence today covers six disposable stdio bundles only: `buzz`, `claude-code`, `codex`, `kimi`, `mcp`, and `opencode`. Each receipt proves initialize, 30 tools, 228 paginated resources, 10 prompts, and a representative project-to-review workflow from an isolated environment. It does not prove that every listed host has executed the pack in its native UI.

The disposable verification pattern is stronger than it looks. Each verified bundle is run from an isolated `HOME`, `PATH`, current working directory, and workspace, then forced through a representative sequence: initialize, inspect the 30 tools, page through the 228 resources, load the 10 prompts, create a project, capture evidence, write an artifact, record an evidence-cited review, and replay factcheck. That is the right level of proof for a transport bundle. It shows pack-local interoperability without pretending that every host UI or every orchestration surface has already been exercised live.

That last sentence is the right place to stop. The CLI and MCP surfaces are fully specified, but they still obey the book's broader honesty rule: prove what ran, say what the receipt covers, and do not promote an emitted pack into host-execution evidence unless the host actually exercised it.

APPENDIX A

The ASD Method & Artifact Catalog

APPENDIX C

The 20-Book Canonical Cross-Reference

NEXT APPENDIX

The 20-Book Canonical Cross-Reference

The knowledge corpus behind the methods, served to hosts as book resources under asd://books/<book>

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14

Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

APPENDIX C

The 20-Book Canonical Cross-Reference

The knowledge corpus behind the methods, served to hosts as book resources under asd://books/<book>

Reading time

9 min

The Corpus

The corpus is intentionally mixed. Some books are hands-on practice manuals. Some are theory, some are business translation, some are method encyclopedias, and some are service-quality classics that keep the platform from collapsing into design theatre. That blend explains the method library better than any single origin story would. Research planning comes from one corner. Journey craft comes from another. Blueprint rigor, CX maturity, service recovery, orchestration, and profit-chain logic each pull from their own shelves.

The important runtime point is simple: these books are not decorative citations. Their chapter syntheses are part of the served resource space. A client can browse them through MCP the same way it browses article notes, and the method manifests cite them directly in structured provenance. That is why a tool like `sd_knowledge` can filter by book id, book chapter, or article number without guessing what the source means.

A practical way to read the corpus is to see four layers at once. First come the field manuals: books that tell me how to plan research, run workshops, map journeys, and blueprint services. Then come the theory books that explain value, systems, and emergence, which stops the runtime from becoming a box of workshop rituals with no model of why services behave the way they do. Then come the management and quality texts that keep operations, economics, and frontline reality in the room. Finally come the method catalogs that widen the tactical repertoire when a project needs something more specific than the usual playbook.

CORPUS FACT	CURRENT VALUE
Books in the served corpus	20
Chapter files across those books	159
Book resource URI forms	`asd://books/<book>` and `asd://books/<book>/<chapter>`
Total MCP resources when article roots are present too	228
Why the corpus exists	To give method manifests, prompts, and knowledge queries a bounded source base instead of hand-wavy references.

I like this split because it keeps the platform from sounding smarter than it is. The runtime can answer “which sources ground this method?” because the source base is finite. It can expose chapter resources through MCP because the corpus has already been normalized into book and chapter files. It can refuse unsupported provenance claims because every manifest is supposed to point back to concrete book chapters or article notes, not an inferred school of thought.

> **CORE PRINCIPLE:** The book corpus is reference material, not license to inflate claims. A chapter served under `asd://books/<book>` proves the knowledge is available to the runtime. It does not prove a host executed any specific method against a live project.

Books to Methods Map

BOOK TITLE	AUTHORS	CORPUS SPAN	REPRESENTATIVE ASD METHODS	PRIMARY ARTIFACT KINDS
This Is Service Design Doing	Marc Stickdorn, Adam Lawrence, Markus Hormess, Jakob Schneider	10 chapters	`research-plan-design`, `stakeholder-mapping`, `storyboard-authoring`, `service-blueprinting`	`research-plan`, `stakeholder-map`, `storyboard`, `service-blueprint`
This Is Service Design Thinking	Marc Stickdorn & Jakob Schneider	6 chapters	`stakeholder-mapping`, `service-safari`, `story-mapping`	`stakeholder-map`, `insight-set`, `story-map`
Service Design: From Insight to Implementation	Andy Polaine, Lavrans Løvlie, Ben Reason	8 chapters	`journey-from-transcripts`, `service-blueprinting`, `prototype-ladder-gating`, `implementation-roadmap`	`journey-map`, `service-blueprint`, `decision-matrix`, `roadmap`

BOOK TITLE	AUTHORS	CORPUS SPAN	REPRESENTATIVE ASD METHODS	PRIMARY ARTIFACT KINDS
Service Design for Business	Ben Reason, Lavrans Løvlie, Melvin Brand Flu	8 chapters	`stakeholder-mapping`, `service-standard-authoring`, `frontline-enablement`, `raci-definition`	`stakeholder-map`, `service-standard-set`, `raci-matrix`
Good Services	Lou Downe	7 chapters	`good-services-audit`, `good-services-audit-15`, `experience-audit`	`insight-set`, `experience-audit`
The Service Innovation Handbook	Lucy Kimbell	8 chapters	`business-model-canvas-build`, `value-proposition-canvas-build`, `implementation-roadmap`	`business-model-canvas`, `value-proposition-canvas`, `roadmap`
Designing for Service: Key Issues	Daniela Sangiorgi & Alison Prendiville, editors	8 chapters	`co-design-workshop`, `desirability-testing`, `experience-audit`, `service-recovery-design`	`concept-set`, `test-report`, `experience-audit`, `recovery-playbook`
Orchestrating Experiences	Chris Risdon & Patrick Quatromoni	8 chapters	`orchestration-canvas`, `ecosystem-value-mapping`, `journey-from-transcripts`	`ecosystem-map`, `journey-map`
Mapping Experiences	James Kalbach	8 chapters	`experience-mapping`, `mental-model-alignment`, `mental-model-diagramming`, `service-blueprinting`	`journey-map`, `mental-model-map`, `service-blueprint`

BOOK TITLE	AUTHORS	CORPUS SPAN	REPRESENTATIVE ASD METHODS	PRIMARY ARTIFACT KINDS
The User's Journey	Donna Lichaw	6 chapters	`story-mapping`, `storyboard-authoring`, `desktop-walkthrough`	`story-map`, `storyboard`, `walkthrough-script`
Design Unbound	Ann M. Pendleton-Jullian & John Seely Brown	10 chapters	`opportunity-matrix-2x2`, `nuf-triple-lens-filter`, `blueprint-to-backlog`	`opportunity-matrix`, `concept-set`, `roadmap`
Service-Dominant Logic	Stephen L. Vargo & Robert F. Lusch	7 chapters	`ecosystem-value-mapping`, `value-proposition-canvas-build`, `business-model-canvas-build`	`ecosystem-map`, `value-proposition-canvas`, `business-model-canvas`
The Experience Economy	B. Joseph Pine II & James H. Gilmore	7 chapters	`experience-audit`, `desirability-testing`, `service-roleplay`	`experience-audit`, `test-report`, `storyboard`
Outside In	Harley Manning & Kerry Bodine	8 chapters	`behavioural-personas`, `experience-audit`, `cx-maturity-assessment`, `customer-typology-mapping`	`persona-set`, `experience-audit`, `maturity-assessment`
Change by Design	Tim Brown	11 chapters	`bodystorming`, `hmw-formulation`, `co-design-workshop`, `wizard-of-oz`	`concept-set`, `hmw-set`, `prototype-plan`
The Design of Business	Roger Martin	6 chapters	`concept-brief-authoring`, `decision-matrix`, `wwhtbt-evidence-gate`	`concept-set`, `decision-matrix`

BOOK TITLE	AUTHORS	CORPUS SPAN	REPRESENTATIVE ASD METHODS	PRIMARY ARTIFACT KINDS
101 Design Methods	Vijay Kumar	9 chapters	`hmw-formulation`, `morphological-concept-matrix`, `opportunity-matrix-2x2`, `business-model-canvas-build`	`hmw-set`, `concept-set`, `opportunity-matrix`, `business-model-canvas`
Universal Methods of Design	Bella Martin & Bruce Hanington	7 chapters	`discussion-guide-design`, `contextual-inquiry`, `affinity-diagramming`, `usability-testing-moderated`	`interview-guide`, `insight-set`, `test-report`
Service Management and Marketing	Christian Grönroos	9 chapters	`service-blueprinting`, `service-quality-diagnostic`, `service-recovery-design`, `frontline-enablement`	`service-blueprint`, `experience-audit`, `recovery-playbook`, `service-standard-set`
The Service Profit Chain	James L. Heskett, W. Earl Sasser, Leonard A. Schlesinger	8 chapters	`kpi-tree-design`, `service-profit-chain-mapping`, `three-tier-measurement`, `frontline-enablement`	`kpi-tree`, `service-profit-chain`, `service-standard-set`

The table is not claiming one-to-one ownership. A single method often draws from several books, and a single book often informs several phases. That is the right shape for a living practice. What the table does give me is a dependable spine: if I need journey craft, I know which shelves are strongest; if I need theory of value co-creation, recovery, maturity, or frontline economics, I know where the runtime's assumptions came from.

I also read the table as a map of tensions inside the practice. *This Is Service Design Doing*, *Service Design: From Insight to Implementation*, and *Mapping Experiences* push the craft toward visible service structure: plans, journeys, blueprints, and prototypes. *Service-Dominant Logic* and *Design Unbound* pull the other way. They remind me that value is co-created, systems are adaptive, and neat diagrams can lie if they freeze living service behavior into a static story. ASD works because it keeps both impulses: operational specificity and theoretical humility.

The business-facing books do a different job. *Outside In*, *The Service Profit Chain*, and *The Design of Business* keep the library accountable to growth, capability, and decision quality. Without them, the measurement phase would drift into generic dashboards or satisfaction slogans. With them, methods such as `kpi-tree-design`, `service-profit-chain-mapping`, and `cx-maturity-assessment` can anchor service change to operating consequences the rest of an organization will actually recognize.

Then there are the quality and frontline texts. *Good Services* and *Service Management and Marketing* are load-bearing because they force the runtime to respect legibility, recovery, and support operations. Those sources are why the platform cares about standards, service recovery, and evidence-backed audits rather than treating delivery as a vague "handoff to implementation." When a blueprint reaches the delivery or measurement phase, these are the books that stop it from becoming a polished fiction.

The method encyclopedias matter too. *101 Design Methods* and *Universal Methods of Design* widen the repertoire without breaking the contract model. They do not flood the runtime with every method ever named. Instead, they justify why a specific subset made the cut and why each retained method still has to pass through outputs, acceptance rules, review gates, and provenance. In other words: broad inspiration upstream, strict runtime discipline downstream.

> **My take:** The method library feels practical because the corpus refuses to stay in one discipline. The best service-design systems always borrow shamelessly from operations,

management, ethnography, and design craft, then force those borrowings back through a common evidence standard.

Reading Paths

If I am orienting a new teammate, I rarely tell them to read the whole corpus in order. I give them a path based on the job in front of them. The runtime encourages the same habit. The method manifests cite the exact sources they need, and the MCP resource surface makes those sources fetchable without a scavenger hunt.

There is another way to use these reading paths. I use them to diagnose blind spots in a team. If a workshop-heavy group keeps producing clever concepts with weak measurement logic, I send them toward the business and quality shelves. If a delivery-heavy team has standards and roadmaps but weak evidence, I push them back to the research and synthesis books. If a technically strong team keeps drawing system diagrams without a service view, I hand them the mapping and orchestration cluster.

For end-to-end service design practice: start with *This Is Service Design Doing*, then pair it with *Service Design: From Insight to Implementation* and *Mapping Experiences*.

For research craft: move through *Universal Methods of Design*, *This Is Service Design Doing*, and the research-heavy chapters of *Outside In*.

For multi-channel and systems work: read *Orchestrating Experiences*, *Mapping Experiences*, and *Service-Dominant Logic* together.

For prototype and concept generation: combine *Change by Design*, *101 Design Methods*, and *The User's Journey*.

For delivery, quality, and operations: use *Service Management and Marketing*, *Good Services*, and *Service Design for Business*.

For business case and measurement: pull from *Outside In*, *The Service Profit Chain*, and *The Design of Business*.

For theory and discipline-level framing: read *Design Unbound*, *Service-Dominant Logic*, and *The Experience Economy*.

What I do not recommend is reading only the books that flatter your home discipline. Designers who skip operations and service quality end up with elegant artifacts that fail in contact with frontline work. Operators who skip journeys, personas, and narrative structure end up with efficient systems nobody can navigate. Strategists who skip field research and prototype testing end up with confident abstractions. The corpus is strongest when it is used as a corrective, not a mirror.

That is why Appendix C belongs in this book at all. It turns provenance from a polite bibliography into an inspectable design decision. When a host asks for a prompt, when a method manifest cites a chapter, or when the knowledge tool returns a matching resource, the platform is leaning on this bounded library. That is the quiet contract under the louder runtime features.

That is enough. Appendix C is supposed to be a compass, not a seminar. The core fact to carry forward is that the platform's method and artifact claims sit on a bounded, inspectable 20-book corpus, and that corpus is available to tools under the same URI space the rest of the runtime already uses.

APPENDIX B

CLI & MCP Tool Specification

ASD ' 26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09
Living Journey Maps: From Transcripts to Sentiment Curves

CHAPTER 10
The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15

CHAPTER 16

Transformative Services, Behavior Change & Recovery Engineering

Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.

← All Chapters

FRONT MATTER

Copyright

Reading time

1 min

CHAPTER 0

The Agentic Service Designer

NEXT CHAPTER

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations

Continue Reading →

ASD '26

The Agentic Service Designer

How AI Agent Teams, Service Blueprints-as-Code, and Autonomous Tooling Are Transforming Service Design from Insight to Operations. By Sam Mozaffari.

FRONT MATTER
Front Matter

FRONT MATTER
Copyright & License

CHAPTER 01
The AI-First Service Design Operating System

CHAPTER 02
The ASD Platform Architecture

CHAPTER 03
Service-Dominant Logic & Value Co-Creation in the Age of AI

CHAPTER 04
The Role Contract, Evidence Registers & Deterministic Fact-Checking

CHAPTER 05
Research Ops at Scale: Mining Transcripts & Field Inquiries

CHAPTER 06
Mixed Methods & Evidence-Backed Segmentation

CHAPTER 07
Mental Models, Alignment & Opportunity Mapping

CHAPTER 08
Ecosystem Value Mapping & Stakeholder Topologies

CHAPTER 09

CHAPTER 10

Living Journey Maps: From Transcripts to Sentiment Curves

The Master Blueprint: Five-Lane Architecture & Line of Visibility

CHAPTER 11
Agentic Service Blueprinting: The Sixth Lane & Autonomous Workflows

CHAPTER 12
Cross-Channel Orchestration & Story Mapping

CHAPTER 13
Structured Ideation & How Might We at Machine Speed

CHAPTER 14
Prototyping Services: Desktop Walkthroughs to Wizard-of-Oz

CHAPTER 15
Transformative Services, Behavior Change & Recovery Engineering

CHAPTER 16
Service Quality Management & CX Maturity Governance

CHAPTER 17
The ASD MCP Server

CHAPTER 18
Plugins & Host Packs: Claude Code and Beyond

CHAPTER 19
BUZZ: Multi-Agent Service Design on the Open Agent Network

CHAPTER 20
DeepSeek Harness Studio: The Integrated Workbench

CHAPTER 21
SOPs-as-Code: The AI Execution Playbook

CHAPTER 22
Frontline Enablement & Employee Experience

CHAPTER 23
The 15 Principles of Good Public Services

CHAPTER 24
Mastercase: Meridian Bay Ferries

CHAPTER 25
The Future of Service Design: The Agentic Practice

APPENDIX A
The ASD Method & Artifact Catalog

APPENDIX B
CLI & MCP Tool Specification

APPENDIX C
The 20-Book Canonical Cross-Reference

©2026 Sam Mozaffari. Free for personal/noncommercial use (CC BY-NC-ND 4.0); commercial license required for business use.