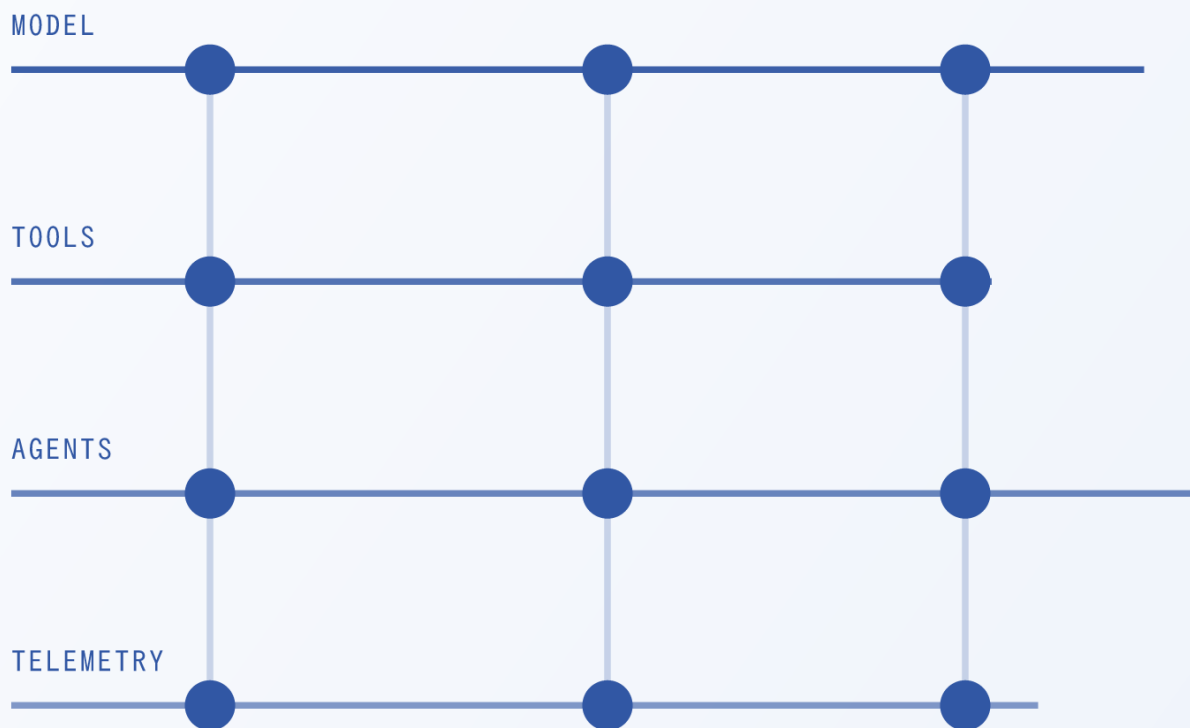


AN OPEN AGENT SYSTEMS BOOK

Designing Portable Agents

Provider-Neutral Agent Systems in TypeScript with Open Standards



Gino Tesei

Experimental edition • 2026-10-07

Open Responses • MCP • A2A • OpenTelemetry

Table of contents

About this book	1
A research agent, from mixed inputs to a verified deck	1
A second path: inspect a document without a model	2
Runnable book-owned examples	3
Implementation and evidence map	3
Claims and evidence	6
Working glossary	6
License	7
I. Part I – The System in Action	8
1. A Portable Research Agent, End to End	9
1.1. The working task	9
1.2. What the demonstration traverses	10
1.3. The portability problem	10
1.4. Five boundaries in one run	11
1.5. Why “provider-neutral” does not mean provider-blind	12
1.6. Failure modes visible in the case study	12
1.7. Security considerations	13
1.8. What the recording proves	13
1.9. How to read the rest of the book	14
1.10. Pattern summary	14
1.11. Exercises	14
1.12. Further reading	14
1.13. Chapter contract	14
2. Pi as a Thin Interactive Client	16
2.1. The client/server split	16
2.2. The reviewed launcher	16
2.3. Why --no-tools matters	17
2.4. Portable model routes	17
2.5. Invocation-local attachments	18
2.6. /inspect-document: a different path	18
2.7. Progress is presentation, not authority	19
2.8. Artifact verification and saving	19
2.9. Sessions, privacy, and retention	19
2.10. Demonstration: model-independent inspection	20
2.11. Deterministic evidence	20
2.12. Failure modes	20
2.13. Pattern summary	21
2.14. Exercises	21
2.15. Chapter contract	21
3. Agent Configuration and Reviewed Capabilities	22
3.1. One configuration, one interpretation	22

3.2.	A reviewed static catalog	23
3.3.	Strict YAML admission	23
3.4.	Limits are server ceilings	24
3.5.	Skills and tools are separate selections	24
3.6.	What cannot appear in <code>agent.yaml</code>	25
3.7.	Selecting a coordinator	25
3.8.	Explicit local customization	25
3.9.	Declaration is not authorization	26
3.10.	Startup is the failure boundary	26
3.11.	Deterministic evidence	26
3.12.	Failure modes	27
3.13.	Pattern summary	27
3.14.	Exercises	28
3.15.	Chapter contract	28
4.	Documents, Images, and the Ingestion Boundary	29
4.1.	One path for every admitted input	29
4.2.	Preserve content order or reject it	29
4.3.	Byte admission comes before parsing	30
4.4.	Supported document profiles	30
4.5.	The Xberg worker boundary	31
4.6.	Canonical-text fallback	32
4.7.	Image admission and recipients	32
4.8.	Invocation-local identity	33
4.9.	<code>/inspect-document</code> reuses the same representation	33
4.10.	Cancellation and worker termination	33
4.11.	Malformed and hostile content	34
4.12.	Demonstration: the BCAUSS paper	34
4.13.	Deterministic evidence	35
4.14.	Pattern summary	35
4.15.	Exercises	35
4.16.	Chapter contract	35
II.	Part II — The Portable Model Runtime	36
5.	The Provider Adapter Boundary	37
5.1.	The problem	37
5.2.	Context and forces	37
5.3.	Separate the application port from the SDK seam	38
5.4.	Bind deployment identity before generation	39
5.5.	The boundary output	41
5.6.	Assign every translation an owner	41
5.7.	Normalization is an information-loss decision	43
5.8.	Provider-specific variation: a failed run is architectural evidence	43
5.9.	Failure modes	44
5.10.	Security considerations	45
5.11.	Observability handoff	45
5.12.	How we test portability	45
5.13.	Portability and evidence ledger	47

5.14. Pattern summary	47
5.15. Exercises	48
5.16. Further reading	48
5.17. Chapter contract	48
6. Open Responses: Requests, Items, Events, and Conformance	50
6.1. From chat turns to trajectories	50
6.2. Requests carry policy as well as input	50
6.3. Items preserve identity	51
6.4. Items are state machines	51
6.5. Semantic events	52
6.6. A reducer, not a callback pile	52
6.7. Terminal states	56
6.8. Snapshot consistency	56
6.9. Unknown and extension events	57
6.10. SDK events and the HTTP protocol are different contracts	57
6.11. Four different status statements	58
6.12. Security considerations	58
6.13. How we test the model	59
6.14. Portability and evidence ledger	59
6.15. Pattern summary	59
6.16. Exercises	60
6.17. Further reading	60
6.18. Chapter contract	60
7. Streaming, State, and Termination	61
7.1. A stream has three layers	61
7.2. Stream all meanings, not only text	61
7.3. Ordered reduction	62
7.4. Backpressure is an end-to-end policy	62
7.5. Cancellation has a direction	64
7.6. Partial output is provisional	65
7.7. Snapshot consistency	65
7.8. Disconnect is a stream-local signal	66
7.9. Errors before and during streaming	66
7.10. Retry is outside the stream reducer	66
7.11. Deterministic streaming tests	66
7.12. Portability and evidence ledger	67
7.13. Pattern summary	68
7.14. Exercises	68
7.15. Further reading	68
7.16. Chapter contract	68
8. The Typed and Authorized Tool Loop	70
8.1. The loop	70
8.2. Define one portable tool contract	70
8.3. Schema is not TypeScript	71
8.4. Validation is still not authorization	71
8.5. Correlation is mandatory	72
8.6. Streamed arguments	73

8.7. Execute under limits	74
8.8. Return structured results	74
8.9. Continue or stop	74
8.10. Tool errors	75
8.11. Parallel tool calls	75
8.12. The shared SDK-backed loop	76
8.13. Portability and evidence ledger	77
8.14. Security checklist	78
8.15. Pattern summary	78
8.16. Exercises	79
8.17. Further reading	79
8.18. Chapter contract	79
 III. Part III — Skills and the Compositional Research Toolchain	 80
9. Skills: Portable Procedures Without Hidden Authority	81
9.1. Procedure is not authority	81
9.2. Static admission before progressive disclosure	82
9.3. Integrity-bound package and declared resources	82
9.4. The Northstar presentation workflow	83
9.5. Separate presentation_builder authorization	83
9.6. Semantic input, server-owned composition	83
9.7. Missing values and numeric normalization	84
9.8. Artifact construction and validation	84
9.9. Verified handoff and safe replacement	85
9.10. What visual review does—and does not—prove	85
9.11. Failure modes	86
9.12. How we test portability	86
9.13. Pattern summary	86
9.14. Exercises	87
9.15. Chapter contract	87
 10. web_search: Broad Discovery Under a Budget	 88
10.1. Search is a tool, not ambient model knowledge	88
10.2. Search-model binding	88
10.3. Two budgets, two meanings	89
10.4. Completion-order admission under concurrency	89
10.5. Candidate normalization	90
10.6. Completion receipts before authority	90
10.7. Provisional search progress	90
10.8. Fitting the final output	91
10.9. Privacy and recipient disclosure	91
10.10. Why candidates are not admitted evidence	91
10.11. Current evidence and its limitations	92
10.12. Failure modes	92
10.13. Pattern summary	92
10.14. Exercises	93
10.15. Chapter contract	93

11. read_source: From URL to Admitted Representation	94
11.1. One narrow public-read operation	94
11.2. URL admission before DNS	94
11.3. DNS and address screening	95
11.4. Pinned transport with hostname TLS validation	95
11.5. Redirects are new destinations	95
11.6. Media type, framing, and encoding	96
11.7. One byte ceiling, applied twice	96
11.8. Shared ingestion, not a second parser stack	97
11.9. Invocation-local source identity	97
11.10. Invocation-local caching and single flight	97
11.11. Progress and privacy	98
11.12. Retrieved content remains data	98
11.13. Retrieval is not truth or entailment	98
11.14. Northstar handoff	99
11.15. Failure modes	99
11.16. Pattern summary	99
11.17. Exercises	100
11.18. Chapter contract	100
 12. code_interpreter: Bounded Local Computation	 101
12.1. One portable computation contract	101
12.2. Fresh worker construction	101
12.3. Invocation-local mounts, not arbitrary files	102
12.4. The Python execution boundary	102
12.5. Scheduling and concurrency	102
12.6. Deadlines and cancellation	103
12.7. Output fitting	103
12.8. Progress without code disclosure	104
12.9. Numeric normalization and missingness	104
12.10. Worked Northstar calculation	104
12.11. Current sandbox limitations	105
12.12. Evidence and testing	105
12.13. Failure modes	106
12.14. Pattern summary	106
12.15. Exercises	106
12.16. Chapter contract	106
 13. Grounding, Citations, and Research Completion	 108
13.1. From tools to an evidence pipeline	108
13.2. Five Northstar evidence classes	108
13.3. Identity, locator, and support are separate	109
13.4. Representation-bound locators	110
13.5. Multi-premise support: the Northstar comparison	110
13.6. Calculations preserve upstream provenance	112
13.7. Generated image observations remain generated	112
13.8. Retrieved guidance: provenance without automatic truth	112
13.9. When is the research run complete?	113
13.10. A completion matrix	113
13.11. Editing invalidates relations	114

13.12. Local-file citations and ephemeral identity	114
13.13. Conflict and absence	115
13.14. Evaluation without entailment overclaiming	115
13.15. Security and privacy	115
13.16. How we test completion	116
13.17. Failure modes	116
13.18. Pattern summary	116
13.19. Exercises	117
13.20. Chapter contract	117
 IV. Part IV — Portability in Production	 118
 14. Capabilities, Fallbacks, and Mixed-Provider Workflows	 119
14.1. The problem	119
14.2. Context and forces	119
14.3. Four different capability statements	120
14.4. A promise needs a scope	121
14.5. An executable admission policy	122
14.6. The implemented mixed-provider workflow	123
14.7. Fallback task profiles	125
14.8. Fallback is a policy transition	125
14.9. Provider-specific variation: feature conjunctions	126
14.10. Measure the semantic gap	126
14.11. Failure modes	127
14.12. Security considerations	128
14.13. Telemetry contract	128
14.14. How we test portability	128
14.15. Portability and evidence ledger	129
14.16. Pattern summary	129
14.17. Exercises	129
14.18. Further reading	130
14.19. Chapter contract	130
 15. Errors, Retries, Recovery, and Durable State	 131
15.1. The recovery problem	131
15.2. Recovery constraints	131
15.3. Two commitments: visibility and effects	132
15.4. Safe errors and protected diagnostics	132
15.5. Provider-specific variation: successful HTTP is not successful execution	133
15.6. Cancellation is a cooperation protocol	133
15.7. One deadline, one retry owner	134
15.8. A tested recovery decision, not a hidden retry loop	135
15.9. Reconciliation examples	136
15.10. Idempotency is an operation contract	137
15.11. Failure modes	137
15.12. Security considerations	138
15.13. Telemetry contract	138
15.14. How we test portability	138
15.15. Portability and evidence ledger	139

15.16.Durable state, memory, and replay	139
15.17.Integrated pattern summary	147
15.18.Exercises	147
15.19.Further reading	148
15.20.Chapter contract	148
16. Observability with OpenTelemetry	149
16.1. Why agent telemetry needs its own contract	149
16.2. Implemented trace topology	149
16.3. Explicit context propagation	151
16.4. Closed attribute policy	152
16.5. Errors, cancellation, and unknown effects	153
16.6. Concurrency	153
16.7. Deterministic exporter and evidence	153
16.8. Production exporter gate	154
16.9. Metrics and logs	154
16.10.Progress is not telemetry	155
16.11.Failure modes	155
16.12.Pattern summary	155
16.13.Exercises	156
16.14.Chapter contract	156
17. Security, Evaluation, and Production Gates	157
17.1. Start with assets and trust boundaries	157
17.2. Threat model	158
17.3. Authentication and tenant ownership	158
17.4. Authorization and approval	158
17.5. Network egress and SSRF	159
17.6. Document and parser isolation	159
17.7. Production code-execution isolation	160
17.8. Prompt injection and untrusted content	160
17.9. Provider and recipient disclosure	161
17.10.Secrets and diagnostics	161
17.11.Rate, cost, deadline, and concurrency controls	162
17.12.Artifact and retained-state security	162
17.13.Evaluation stack	163
17.14.Fault-injection program	164
17.15.Release approval gate	164
17.16.Rollback is a state transition	165
17.17.Current blocked production claims	165
17.18.Pattern summary	166
17.19.Exercises	166
17.20.Chapter contract	166
V. Part V — Open Ecosystem and Validation	167
18. MCP and A2A: Transport and Delegation Boundaries	168
18.1. Four boundaries, not one “agent stack”	168
18.2. MCP’s purpose and lifecycle	169
18.3. MCP admission record	170

18.4. A2A's purpose and lifecycle	170
18.5. Transport versus authority	171
18.6. Identity and confused-deputy risk	171
18.7. Delegation boundaries	172
18.8. Data recipients and disclosure	172
18.9. Cancellation, errors, and reconciliation	173
18.10. Protocol translation loss	173
18.11. Negotiated capability is not semantic portability	174
18.12. Telemetry without payload export	174
18.13. Evidence and conformance limits	174
18.14. Failure modes	175
18.15. Pattern summary	175
18.16. Review questions	175
18.17. Chapter contract	176
19. Production Case Study and Formal Conformance	177
19.1. The integrated system	177
19.2. Exact implementation and evidence baselines	178
19.3. Input and document boundary	178
19.4. Provider and recipient routes	179
19.5. Search, retrieval, and research completion	179
19.6. Computation boundary	180
19.7. Skill and presentation boundary	180
19.8. Recovery and durable state	180
19.9. Telemetry boundary	181
19.10. Four conformance statements	181
19.11. Official Open Responses suite result	182
19.12. Missing required behavior	182
19.13. Decision on missing conformance features	183
19.14. Consolidated evidence matrix	183
19.15. Blocked-work register	184
19.16. What the case study proves	185
19.17. Release checklist for the integrated profile	185
19.18. Pattern summary	185
19.19. Exercises	186
19.20. Chapter contract	186
Index	189

About this book

Designing Portable Agents: Provider-Neutral Agent Systems in TypeScript with Open Standards is an executable guide to separating application intent from model providers, tool transports, agent protocols, and deployment infrastructure.

See the system first

A research agent, from mixed inputs to a verified deck

Before reading the architecture, watch the working system. The approved demonstration follows the companion reference through ordered image/PDF/XLSX admission, bounded computation, generated image observations, public web discovery, source reading, an evidence-separated brief, progressive skill loading, and digest-verified PowerPoint construction.

```
2. Main message: the 72 Wh claim matches across the specification and the workbook, but no one has checked it independently.
3. What the image model saw: the label names the prototype but shows no battery rating.
4. Prototype A summary: its battery (72 Wh), runtime (9 h), mass (1.8 kg) and display (14 in) all match the specification.
5. Runtime bar chart: Prototype A at 9 h and Prototype B at 7.5 h. The 8.25 h mean is labelled as an average of two prototypes, not of repeated tests.
6. FAA comparison: both prototypes fall in the up-to-100 Wh band, which needs no airline approval. This is labelled as my inference, not an FAA ruling.
7. Still unverified: four gaps remain before anyone can rely on these claims.

Notes:
- No new checks: the deck reuses findings from my last answer. Your attachments weren't sent again this turn, so I didn't re-read them.
- FAA date: the FAA page wasn't re-read either, so its "last updated October 2, 2026" date is noted on the slide as coming from the earlier check.
- No label photo: the deck doesn't include the label photo, because the image wasn't available this turn.
- No recommendation slide: the deck builder pointed out that there's no recommendation or next-steps slide. I left one out because this is a findings brief. I can add one, for example "commission independent capacity testing," if you'd like.

File details: 36,120 bytes · SHA-256 a802aa4a08747087e2fa9036a700fd946ef5825f63aae66ff623e02b2146f9c2

Saved northstar-evidence-brief.pptx · sha256
a802aa4a08747087e2fa9036a700fd946ef5825f63aae66ff623e02b2146f9c2.

~/var/portable-agents-reference (main)
↑94k ↑5.1k 4.7%/1.0M (auto) anthropic/claude-opus-5-5 • medium
```

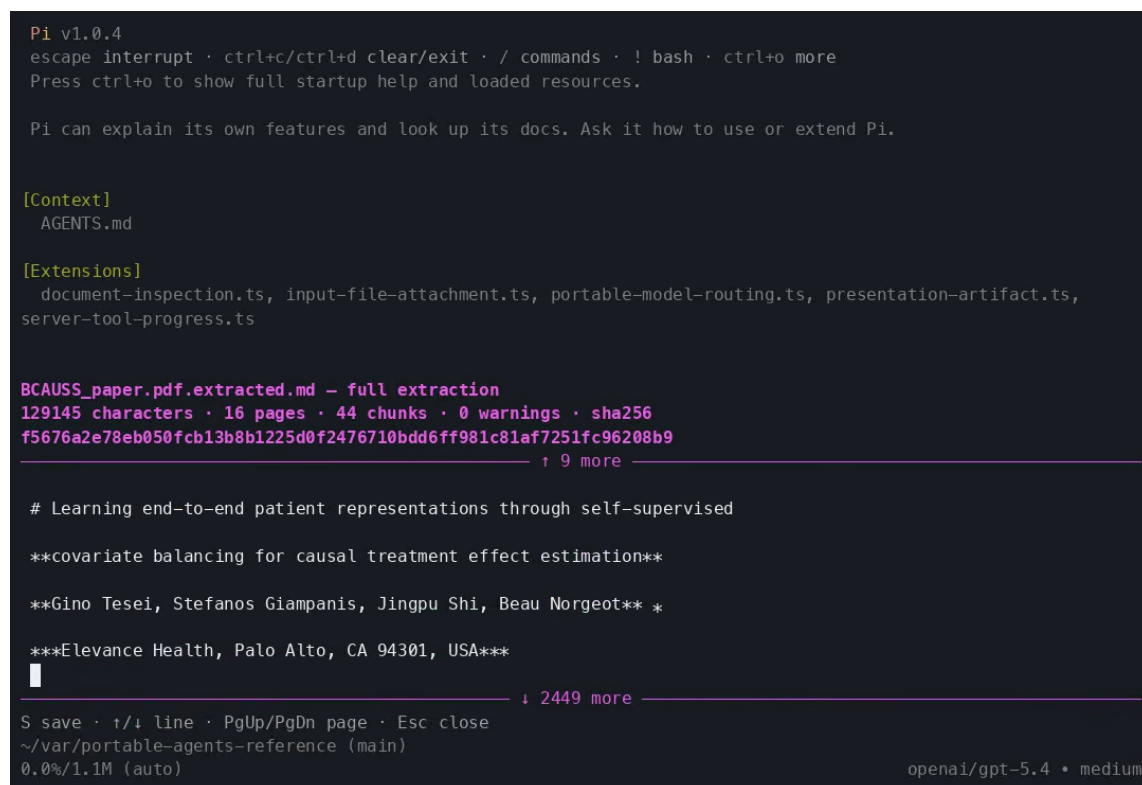
Figure 1.: Final frame of the approved Portable Agents Pi research demonstration. The terminal shows the completed evidence brief and bounded presentation workflow.

[Read the transcript](#) · [Extended alt description](#) · [Evidence and limitations](#)

Northstar Field Console is fictional. This is dated live-provider and public-network evidence—not deterministic output, provider equivalence, production certification, or full Open Responses conformance.

A second path: inspect a document without a model

The second approved demonstration shows `/inspect-document` opening, navigating, and explicitly saving a bounded local extraction without invoking the configured model. The input is Gino Tesei and coauthors' public 2023 *Journal of Biomedical Informatics* [paper on self-supervised covariate balancing](#), supplied by its author. The PDF is not included in the book or reference repository.



```

Pi v1.0.4
escape interrupt · ctrl+c/ctrl+d clear/exit · / commands · ! bash · ctrl+o more
Press ctrl+o to show full startup help and loaded resources.

Pi can explain its own features and look up its docs. Ask it how to use or extend Pi.

[Context]
  AGENTS.md

[Extensions]
  document-inspection.ts, input-file-attachment.ts, portable-model-routing.ts, presentation-artifact.ts,
  server-tool-progress.ts

BCAUSS_paper.pdf.extracted.md – full extraction
129145 characters · 16 pages · 44 chunks · 0 warnings · sha256
f5676a2e78eb050fcb13b8b1225d0f2476710bdd6ff981c81af7251fc96208b9
↑ 9 more

# Learning end-to-end patient representations through self-supervised

**covariate balancing for causal treatment effect estimation**

**Gino Tesei, Stefanos Giampanis, Jingpu Shi, Beau Norgeot** *

***Elevance Health, Palo Alto, CA 94301, USA***
█
↓ 2449 more

S save · ↑/↓ line · PgUp/PgDn page · Esc close
~/var/portable-agents-reference (main)
0.0%/1.1M (auto)
openai/gpt-5.4 · medium

```

Figure 2.: Pi's read-only document viewer displaying the bounded extraction of a public research paper.

[Read the interaction transcript](#) · [Extended alt description](#) · [Evidence and limitations](#)

The 90-second run reports 16 pages, 44 chunks, zero extraction warnings, and a digest before requiring confirmation to replace the Markdown output. It demonstrates that exact local run—not universal PDF fidelity, scientific verification, or a model capability. Article text and publisher formatting retain their respective rights and are not relicensed by inclusion.

The five-part path is:

1. [A Portable Research Agent, End to End](#)
2. [Pi as a Thin Interactive Client](#)
3. [Agent Configuration and Reviewed Capabilities](#)
4. [Documents, Images, and the Ingestion Boundary](#)
5. [The Provider Adapter Boundary](#)
6. [Open Responses: Requests, Items, Events, and Conformance](#)
7. [Streaming, State, and Termination](#)
8. [The Typed and Authorized Tool Loop](#)

9. **Skills: Portable Procedures Without Hidden Authority**
10. **web_search: Broad Discovery Under a Budget**
11. **read_source: From URL to Admitted Representation**
12. **code_interpreter: Bounded Local Computation**
13. **Grounding, Citations, and Research Completion**
14. **Capabilities, Fallbacks, and Mixed-Provider Workflows**
15. **Errors, Retries, Recovery, and Durable State**
16. **Observability with OpenTelemetry**
17. **Security, Evaluation, and Production Gates**
18. **MCP and A2A: Transport and Delegation Boundaries**
19. **Production Case Study and Formal Conformance**

Chapter 16 intentionally remains evidence-gated until tested OpenTelemetry instrumentation exists. Chapter 19 retains the current **not conformant** Open Responses classification rather than converting compatibility into a release claim.

The same Quarto Markdown sources produce the navigable HTML edition and KOMA-Script/LaTeX PDF. Small book examples are independently tested where executable; the companion reference implementation is maintained in a separate repository and remains private until its release gates pass.

- **Publication status:** nineteen-chapter technical preview; implementation and production-readiness claims remain evidence-scoped
- **Standards checked:** foundational review 2026-10-07-09; later claims carry their own dates and pins
- **Current reviewed snapshot:** portable-agents-reference commit 45a903c84c30ac1bcc9636c91e4166c500 package version 0.4.0
- **Latest immutable release:** v0.4.0, commit 5ecc4f01e61858f16e42e5c2c34edac234855d6d
- **Historical baseline retained where cited:** v0.3.0, commit 1c7a18a8434e5143cbe9d050b8252a6ead1f9b51
- **Canonical book source:** book/

Runnable book-owned examples

Download the independently authored [reader examples](#) and retain its [content manifest](#). With Bun 1.3.9, extract the archive, enter portable-agents-examples, then run `bun run test` and `bun run northstar`. No dependency installation, repository credential, provider key, or live endpoint is needed. The manifest identifies the exact distributed source/test snapshot by file hashes; it is not a signature. PDF readers can obtain the same downloads from the [book website](#), under **Runnable book-owned examples**.

These examples implement the deployment binder, scoped capability admission, recovery decision, SQLite transcript experiment, locator/card boundary, and deterministic two-measurement Northstar composition. They are book designs, not APIs claimed to exist in the reference release. Successful local composition is not live-provider equivalence, general claim support, production authorization, or conformance.

Implementation and evidence map

The map separates four evidence classes that must not be collapsed:

1. **Immutable release evidence** refers to the exact v0.4.0 or historical v0.3.0 tag.
2. **Current snapshot evidence** refers to exact post-release commit [45a903c](#), which includes the approved demonstrations and later skill/configuration work.
3. **Book-owned deterministic evidence** refers only to source and tests distributed with this book.
4. **Planned or blocked work** has no implementation claim, even when the architecture is specified.

Ch.	Boundary	Primary implementation/evidence	Status and limitation
1	End-to-end Northstar workflow	Current snapshot README.md and approved demo metadata	Dated live-provider/public-network sample; not deterministic or equivalent across providers
2	Pi client	Current launcher and five extensions under scripts/ and extensions/pi/	Pi 1.0.4 client path; not Open Responses conformance
3	Strict agent configuration	Current agent.yaml , parser, and configuration tests	Static reviewed catalog; no principal authentication or dynamic plugins
4	Document/image ingestion	Current src/input/ , src/ingestion/ , and inspection evidence	Implemented local profile; invocation-local identity, not durable citations
5	Provider adapters	Immutable v0.4.0 model/provider adapters; historical v0.3.0 live reports where explicitly cited	Shared application boundary; no broad semantic equivalence
6	Open Responses lifecycle	Current src/open-responses/ and pinned protocol/suite records	Compatible HTTP/SSE subset; not conformant
7	Streaming and termination	Immutable bounded consumer plus current termination/diagnostic tests	Local bounds and cancellation; no universal end-to-end network-backpressure proof

Ch.	Boundary	Primary implementation/evidence	Status and limitation
8	Typed authorized tools	Immutable tool runtime plus current server progress implementation	Validation and policy hooks are implemented; application authentication remains external
9	Skills and presentations	Current src/skills/ , presentation tool, worker, and tests	Post-v0.4.0 snapshot; skill grants no executable authority
10	web_search	Current search tool/backends, budgets, and invariant tests	Bounded discovery; candidates are not read-source evidence
11	read_source	Current source reader, screened retrieval, ingestion integration, and tests	Public HTTPS profile; retrieved content remains untrusted
12	code_interpreter	Current fresh Pyodide worker, scheduler, mounts, and tests	Bounded local computation; not an OS sandbox
13	Grounding and completion	Current evidence modules plus book-owned Northstar support computation	Provenance/locator mechanisms; no general entailment verifier
14	Capability and fallback policy	Current portability modules plus book-owned capability-admission example	Scoped policy and limited evidence; not provider parity
15	Recovery and durable state	Current recovery/transcript modules plus book-owned SQLite experiment	Local history and reconciliation rules; not a production effect ledger
16	OpenTelemetry	No OpenTelemetry implementation in the reviewed snapshot	Blocked pending spans, propagation, redaction, concurrency, and sanitized trace evidence
17	Security and evaluation gates	Distributed controls and deterministic adversarial tests	Partial controls only; no production certification

	Ch.	Boundary	Primary implementation/evidence	Status and limitation
	18	MCP and A2A	Architecture and pinned public specifications	Not implemented; no interoperability claim
	19	Integrated case and conformance	Approved demos plus pinned official-suite wrapper/results	Current classification: compatible subset; not conformant

The v0.3.0 tag remains the immutable source for dated 2026-10-08 adapter reports and the teaching baseline used in several substantive runtime chapters. Those records are not rewritten as v0.4.0 runs. Conversely, ingestion, Pi inspection, skills, presentations, and the published hero are tied to the current post-release snapshot rather than backdated into either release tag.

Claims and evidence

The book uses seven evidence labels:

- **specified** — stated by a pinned external specification;
- **designed** — proposed by the book’s architecture;
- **implemented** — present in a named immutable release, fixed source snapshot, or tested book source; a snapshot is not a reviewed release;
- **deterministically tested** — covered by a named network-free fixture;
- **live-tested** — exercised against a named provider, model, configuration, and date;
- **conformant** — passed a pinned official conformance runner; and
- **evaluated** — measured for semantic quality or safety under a published method.

A schema resemblance is not conformance. A common provider interface is not semantic equivalence. Every portability claim should name both its dimension and evidence label. Formal Open Responses conformance is intentionally reserved for the concluding validation rather than imposed as an initial implementation constraint.

Working glossary

- **Response:** one model operation and its authoritative terminal outcome.
- **Item:** an identifiable message, tool call, or other unit in a response trajectory.
- **Event:** an ordered state transition or incremental update.
- **Step:** one model generation phase within a bounded tool loop.
- **Turn:** an application-level conversational exchange; it may contain several model steps.
- **Run:** the request-scoped orchestration that owns policy, budgets, and cancellation.
- **Task:** delegated work with its own lifecycle, especially at an agent-to-agent boundary.
- **Call:** one correlated proposal to invoke a named tool.
- **Result:** the single terminal application outcome associated with a call.

License

Book prose, original diagrams, and cover artwork are licensed under [CC BY-NC-ND 4.0](#). Executable code and build scripts are licensed under [Apache-2.0](#). Third-party material remains under its respective terms.

Part I.

Part I — The System in Action

1. A Portable Research Agent, End to End

Part I · The system in action

Demonstration baseline: reviewed 2026-10-22 · Reference snapshot: 45a903c

Start with the result, not a framework diagram. A user gives one agent an image, a fictional specification, and a workbook; asks it to research, calculate, and separate evidence classes; then asks for a PowerPoint deck. The visible workflow is compact. The contracts underneath it are the subject of this book.

1.1. The working task

Northstar Field Console is a synthetic device created for the demonstration. Its inputs are independently authored fixtures rather than production or confidential material.

Input	Admitted representation	What it can establish
northstar-console.png	Reviewed image bytes and invocation-local image identity	A generated observation about visible labels; not verified product truth
northstar-specification.pdf	Bounded extracted text	What the fictional specification declares: 72 Wh capacity, 1.8 kg mass, a 14-inch display, and an 8–10 hour target
northstar-measurements.xlsx	Profiled workbook extraction	Synthetic measured runtimes of 9 hours for Prototype A and 7.5 hours for Prototype B

The first recorded turn attaches all three inputs and asks:

Build a short evidence brief. Use `image_understanding` for the visible product label, inspect the attached specification and measurements, use `web_search` to find current official US guidance for portable-device battery capacity, use `read_source` on the most relevant official URL, and use `code_interpreter` to calculate the mean measured runtime. Separate generated image observations, attached declarations, measured values, retrieved public evidence, calculations, and anything still unverified. Do not invent citations or treat this fictional product as real.

The follow-up is deliberately short:

make a short pptx deck about those findings

That second turn does not reattach the original files. It asks a reviewed skill to organize findings already present in the conversation and a separately authorized tool to construct a bounded artifact. The distinction matters: prior conversational findings can be reused, while invocation-local file identities are not silently promoted into durable cross-request storage.

1.2. What the demonstration traverses

Pi interaction

- loopback Open Responses request
- ordered input admission
- provider-neutral coordinator
- server-owned tools
 - image observation
 - web discovery
 - source reading
 - bounded computation
- evidence-labelled synthesis
- reviewed presentation procedure
- bounded presentation builder
- digest-verified local save

The client presents the interaction. It does not own provider credentials, tool registration, document extraction, search budgets, source admission, Python execution, or presentation composition. Those decisions remain on the server side of the boundary.

The result deliberately uses different evidence labels:

- **generated observation** for image-model output;
- **attached declaration** for statements found in a supplied document;
- **measured value** for values represented in the workbook;
- **retrieved public evidence** for material admitted from a screened HTTPS source;
- **calculation** for output from a fresh bounded worker; and
- **unverified** when the available operations do not establish a claim.

A polished paragraph or slide must not erase these distinctions.

1.3. The portability problem

An application can call two models through similar TypeScript interfaces and still fail to be portable. Portability has at least five dimensions:

Dimension	Question
Transport	Can requests, streams, cancellation, and limits cross the connection correctly?
Provider	Can the selected deployment be replaced without leaking provider objects into the application core?
Protocol	Are request items, events, lifecycle, errors, and state interpreted consistently?

Dimension	Question
Feature	Does the selected route support the tools, ordered inputs, output shape, and budgets required by this task?
Semantic	Does the replacement still produce acceptable task outcomes under a published evaluation?

Structural compatibility is the narrowest claim. It does not prove account access, feature availability, conformance, safety, or semantic equivalence. Every replacement claim therefore names the environment, task profile, and meaning it is intended to preserve.

A **portable agent system** keeps application intent, trusted policy, tool authority, evidence identity, and terminal meaning stable while provider- or transport-specific code is confined to reviewed boundaries. Replacement remains an evidence-scoped claim, not a provider promise.

1.4. Five boundaries in one run

1.4.1. Client boundary

Pi owns terminal interaction and its local session. The reviewed launcher forces model traffic through portable-agents routes and disables Pi-local executable tools. Chapter 2 owns that client/server division.

1.4.2. Configuration boundary

`agent.yaml` selects only reviewed tool and skill identifiers plus bounded settings. It does not load arbitrary code, packages, paths, endpoints, credentials, or prompts. Chapter 3 owns static selection; invocation-time authorization remains Chapter 8.

1.4.3. Input boundary

Files are admitted in order, checked by format-specific limits, and assigned invocation-local identities. Text extraction is not citation support, and an image-model observation is not provenance. Chapter 4 owns ingestion.

1.4.4. Model boundary

The application sends one typed request to a selected deployment binding. Provider adapters translate that request and normalize selected output meanings. Chapter 5 owns translation; Chapters 6 and 7 own protocol lifecycle and stream-local state.

1.4.5. Tool and evidence boundary

Generated tool arguments are proposals. Server policy authorizes a call, validates input and output, enforces limits, and correlates one terminal result. Chapters 8–13 develop tools, skills, retrieval, computation, and research completion without transferring authority to model text.

1.5. Why “provider-neutral” does not mean provider-blind

Provider identity remains useful for construction, diagnostics, capability declarations, and evidence. The mistake is allowing it to determine application policy everywhere.

A provider adapter may need to:

- construct the correct SDK client;
- choose a provider protocol;
- map ordered content and tool history;
- preserve or reject provider-specific continuation state;
- normalize finish reasons and usage without inventing absent values; and
- sanitize failures before they cross the application boundary.

The application still decides which deployment alias is permitted, which capabilities the task requires, which tools are authorized, and whether a fallback changes the user-visible promise. Chapters 5 and 14 keep those responsibilities separate.

1.6. Failure modes visible in the case study

1.6.1. Fluent output without admitted evidence

A coordinator can write a plausible brief without reading the workbook or official source. Fluency is not completion. Chapter 13 makes required evidence operations and unresolved gaps explicit.

1.6.2. Tool names mistaken for authority

Mentioning `code_interpreter` in a prompt cannot enable it. Selection, current authorization, budgets, validation, and cancellation remain server-owned.

1.6.3. Search mistaken for source reading

Search produces bounded candidates and notes. A cited factual claim requires an admitted representation from `read_source` and a separate support judgment.

1.6.4. Attachments mistaken for durable memory

The three fixture files are invocation-local. A later request must resend them unless the application has a separately designed retained-source contract. Conversation text is not a trusted cross-request asset registry.

1.6.5. A successful deck mistaken for verified research

The presentation builder can produce a structurally valid artifact while the underlying claims remain incomplete or incorrectly labelled. Artifact integrity and evidence quality are different gates.

1.6.6. Similar APIs mistaken for conformance

The current server implements a useful Open Responses-compatible subset. Missing compaction and WebSocket lifecycle/state behavior keep the formal classification **not conformant**. Chapters 6 and 19 preserve that distinction.

1.7. Security considerations

Every external or generated representation remains untrusted: prompt text, attachments, extracted document text, image observations, search results, retrieved pages, Python output, skill resources, and generated presentation content. Labels inside those representations cannot grant authority.

The architecture therefore keeps these controls outside the model:

1. trusted configuration selects reviewed capabilities;
2. request admission enforces byte and structural limits;
3. current policy authorizes each executable operation;
4. credentials and recipients are server-owned;
5. workers have explicit deadlines and output ceilings;
6. source and artifact identities are integrity-bound; and
7. diagnostics exclude sensitive content by default.

This reduces authority confused by prompt injection but is not a complete production security certification. Chapter 17 owns the final production gates, informed by OWASP guidance on prompt injection and excessive agency (OWASP, n.d.; “OWASP Top 10 for Agentic Applications 2026,” n.d.).

1.8. What the recording proves

The approved hero is dated live-provider and public-network evidence for its exact metadata, route, inputs, and visible behavior. It shows:

- ordered attachment notices;
- server-owned progress cards;
- differentiated evidence labels;
- a bounded calculation;
- screened discovery and source reading;
- progressive skill loading;
- presentation construction; and
- byte-count and SHA-256 verification before local save.

It does **not** establish deterministic output, broad semantic quality, provider equivalence, production suitability, complete privacy, or formal Open Responses conformance. The generated deck was not visually inspected during the recorded session. Those exclusions are part of the evidence, not footnotes to omit.

1.9. How to read the rest of the book

Part I separates the visible client, static configuration, and ingestion boundary. Part II develops the portable runtime. Part III composes skills and tools into a research workflow. Part IV handles substitution, recovery, observability, security, and evaluation. Part V places MCP and A2A at their proper boundaries and closes with an integrated case study and formal conformance status.

For every implementation statement, ask: which immutable release or source snapshot, which test or demonstration, which configuration, which date, and which limitation?

1.10. Pattern summary

Begin with one observable task. Keep interaction, configuration, input admission, model translation, tool authority, evidence, and artifact handling as separate contracts. Port only what the application can name and test; disclose every semantic reduction.

1.11. Exercises

1. Classify each Northstar statement as generated observation, attached declaration, measured value, retrieved evidence, calculation, or unverified.
2. Replace the coordinator deployment on paper. List which contracts should remain unchanged and which claims require new evidence.
3. Remove `read_source` from the selected tools. Rewrite the task outcome without pretending that search candidates were read.
4. Identify the exact point at which a generated tool call becomes an authorized execution.
5. Explain why a digest-verified .pptx can still contain unsupported claims.

1.12. Further reading

Read the Open Responses specification as a protocol target rather than a synonym for one provider API (“Open Responses Reference 2026-04-24,” n.d.). Use the NIST Generative AI Profile for evaluation and source-review obligations (National Institute of Standards and Technology 2024), and OWASP’s prompt-injection and excessive-agency guidance for authority boundaries (OWASP, n.d.; “OWASP Top 10 for Agentic Applications 2026,” n.d.).

1.13. Chapter contract

- **Scope:** the complete Northstar workflow and the portability problem it exposes.
- **Canonical claims:** portability is layered; interfaces alone do not establish behavioral equivalence.
- **Migration inputs:** former Chapter 1 portability definitions and the approved Northstar demonstration.
- **Evidence gates:** dated demonstration metadata, immutable reference snapshots, and independently reproducible local examples.
- **Demonstration slots:** the approved Pi research recording on the landing page.

- **Source-claim ledger:** standards and implementation claims remain dated, versioned, and scoped.
- **Exclusions:** detailed client, ingestion, runtime, tool, skill, recovery, and conformance mechanics belong to later chapters.

2. Pi as a Thin Interactive Client

Part I · The system in action

Tested host: Pi 1.0.4 · Current reviewed reference snapshot: 45a903c · Demo review: 2026-10-22

Pi makes the system usable without becoming its authority boundary. The terminal owns interaction and local sessions; the Bun server owns provider credentials, portable routes, input admission, executable tools, and evidence-producing operations.

2.1. The client/server split

The reviewed path is intentionally asymmetric.

Pi owns	Portable server owns
terminal editing and rendering	provider client construction and credentials
/model interaction	admitted portable model routes
local session files and compaction	request parsing and ordered input admission
explicitly loaded UI extensions	server tool selection, policy, budgets, and execution
attachment references for the current invocation	document/image ingestion and invocation-local asset identities
progress-card presentation	the actual search, retrieval, computation, and builder operations
artifact preview/save interaction	short-lived artifact creation and integrity metadata

This is a thin-client claim, not a claim that Pi contains no logic. Pi still parses interaction, maintains its session, builds requests, renders streams, and runs trusted client extensions. “Thin” means that application authority is not derived from the terminal or generated prompt text.

2.2. The reviewed launcher

The current launcher is `scripts/run-pi.sh`. It performs a fixed sequence:

1. resolve the repository root;
2. require the pinned Pi executable installed under `extensions/pi`;
3. create an isolated, ignored Pi agent directory;
4. copy the reviewed model registry into that directory;
5. load exactly five repository extensions;
6. select the portable-agents provider and reviewed OpenAI route; and
7. pass `--no-tools` before any additional user arguments.

The five extensions are:

Extension	Client responsibility	Authority it does not receive
portable-model-routing.ts	keep reviewed model choices on portable-agents routes	arbitrary provider or endpoint selection
input-file-attachment.ts	encode reviewed local attachments for the current invocation	durable file identity or arbitrary file traversal
document-inspection.ts	present a bounded local extraction	model/tool authority or citation status
presentation-artifact.ts	verify, preview, and save a returned artifact	permission to build arbitrary server files
server-tool-progress.ts	render server progress as expandable cards	executable tools or model-context instructions

Every extension is local code with the user’s privileges and must be reviewed as such. Loading it is not equivalent to downloading an untrusted plugin at runtime: paths are fixed by the launcher and are not selected by model output or agent .yaml.

2.3. Why --no-tools matters

Pi includes its own executable tool surface. That is useful in coding sessions but would create a second authority plane in this research workflow. The reviewed launcher disables those tools so generated calls must use the server-owned tool loop.

--no-tools does not sandbox Pi, restrict operating-system permissions, or make extensions inert. It has a narrower, testable meaning: built-in Pi tools are not available to the model in this launched session. The repository extensions still perform their reviewed UI and attachment functions, while executable research capabilities remain behind the server.

A model refusal is not proof of this boundary. Evidence comes from the launch arguments, registered tool surface, server tests, and observed correlated calls.

2.4. Portable model routes

Pi’s registry uses a loopback base URL ending in /v1 and a non-secret placeholder API key. Provider credentials remain in the server process. A route such as:

```
portable-agents/openai/gpt-5.4
```

means “send the request to the portable server and let its trusted routing policy select the reviewed provider adapter.” Choosing Pi’s built-in openai/. . . route directly would bypass server-owned attachments and tools even if the visible model name looked similar.

The routing extension remaps a reviewed built-in Anthropic selection to the corresponding portable-agents/anthropic/. . . route. An unmapped selection restores a reviewed portable route; if rerouting cannot be completed, the session stops instead of silently falling through to a direct provider. Registry context windows and output limits are client routing metadata, not live capability or account-access evidence.

The reviewed Pi 1.0.4 path uses HTTP POST `/v1/responses` with JSON or SSE. It does not use the Open Responses WebSocket transport or `/v1/responses/compact`. Pi performs its own local session compaction. These facts explain client behavior; they do not relax the formal conformance gaps reported in Chapter 19.

2.5. Invocation-local attachments

Typing an attachment reference causes the extension to inspect a supported local regular file, enforce its client-side count and byte limits, and add a standard ordered file/image content part to the current request. The server performs its own authoritative admission.

The reviewed extension supplements Pi's normal fd-based completion because ignored fixture files do not otherwise appear. It exposes only supported, non-empty regular files from the current working directory and does not recursively enumerate ignored paths.

Attachment identity ends with the invocation. A later turn must resend the file if it needs the bytes again. The conversation may retain extracted statements, but those statements are not equivalent to a trusted retained file, page map, or citation identity. This is why the presentation follow-up can reuse research findings but cannot silently remount the original workbook.

2.6. `/inspect-document`: a different path

`/inspect-document` path is a local command, not a prompt asking the model to read a file. The command starts a fixed Bun child, runs the same bounded document-ingestion component, and opens the complete admitted extraction in a read-only viewer.

The viewer:

- begins at the first line;
- supports bounded line and page navigation;
- reports filename, media type, size, digest, extraction mode, page/chunk counts, and warnings;
- saves only after an explicit command;
- confirms before replacing an existing extraction; and
- closes without sending document text to the model.

`/inspect-document --save` path performs the bounded extraction and save path without opening the interactive viewer. The canonical output name ends in `.extracted.md`; it does not overwrite the source document.

The command does not establish scientific correctness, universal PDF fidelity, page-perfect citations, or safe publication of the extracted content. It exposes what the reviewed ingestion path produced.

2.7. Progress is presentation, not authority

Long research operations need visible activity without leaking raw arguments or results. The server emits bounded progress summaries with operation identity and lifecycle. The Pi extension renders those summaries as expandable custom cards stored outside model context.

The extension registers no executable tool. A progress card cannot start, approve, retry, or cancel an operation. It also must not expose search queries, source bodies, document text, credentials, or raw provider errors. Resume tests keep cards associated with the correct session and verify that rendering does not duplicate execution.

This user-visible protocol is distinct from OpenTelemetry. Chapter 8 owns progress semantics; Chapter 16 remains blocked until operator telemetry is implemented and tested.

2.8. Artifact verification and saving

The presentation builder returns an artifact descriptor rather than placing arbitrary bytes at a model-supplied path. The descriptor binds an artifact ID, media type, byte count, SHA-256 digest, expiry, and loopback retrieval route.

The Pi artifact extension:

1. retrieves the short-lived artifact from the loopback server;
2. verifies the exact byte count and SHA-256 digest;
3. allows preview without claiming visual quality;
4. writes the selected local destination with restrictive mode 0600; and
5. requires confirmation before replacement.

The model cannot supply server paths, local save paths, package names, macros, OLE objects, external relationships, or executable presentation code. Integrity verification proves that the saved bytes match the server descriptor; it does not prove the claims in the deck or compatibility with every presentation suite.

2.9. Sessions, privacy, and retention

PI_CODING_AGENT_DIR separates this workflow from the user's default Pi configuration and sessions. It is organizational isolation, not an operating-system sandbox. Session records can contain prompts, generated answers, tool summaries, source notes, and local paths. They should remain private even when ignored by Git.

--no-session avoids durable Pi session storage for that run, but also prevents later resumption. Deleting the example directory removes copied registry and local sessions; it does not revoke provider credentials held elsewhere or delete server/provider records. A project directory, terminal scrollbar, downloaded artifact, and operating-system logs may have separate retention.

2.10. Demonstration: model-independent inspection

The approved landing-page recording supplies this chapter's clearest negative capability evidence. It shows a public 16-page paper opened, navigated, and explicitly saved with:

```
44 chunks
129,145 extracted characters
0 extraction warnings
0 model calls
0 provider calls
0 public-network calls
```

The displayed extraction digest is part of the recording metadata. The PDF itself is not distributed by the book. Gino Tesei approved visible use because he is an author and the paper is public; article text and publisher formatting are not relicensed.

This recording proves the exact local route and reviewed artifact. It does not prove that every PDF extracts correctly, that the paper's claims are true, or that an on-screen but unused model contributed to the result.

2.11. Deterministic evidence

The current reference provides three credential-free acceptance commands for the client path:

```
bun run test:pi
bun run test:pi-progress
bun run test:pi-inspection
```

They use isolated client directories and synthetic loopback servers. Together they exercise text and stream behavior, tool/usage display, continuation and cancellation, progress ordering and resume isolation, command registration, metadata-only inspection notification, zero provider calls, save behavior, and cleanup. They do not replace manual TUI review for resizing, colors, keyboard ergonomics, or every host terminal.

2.12. Failure modes

- **Direct-provider selection:** the visible model works but bypasses the server boundary.
- **Implicit attachment retention:** a later turn appears to reuse bytes that no longer have an admitted identity.
- **UI card as evidence:** rendered progress is mistaken for proof that an operation completed correctly.
- **Client configuration as capability proof:** a registry entry is treated as account availability.
- **Local session as trusted memory:** editable display history is treated as an authenticated effect ledger.
- **Artifact digest as semantic review:** matching bytes are mistaken for verified claims or visual quality.

2.13. Pattern summary

Keep the interactive client replaceable. Give it enough responsibility to present, attach, inspect, and save, but keep credentials, model routing, input admission, executable capabilities, evidence operations, and policy on the server side of a visible protocol boundary.

2.14. Exercises

1. Launch Pi without `--no-tools`. Identify the new authority plane and the evidence needed before accepting it.
2. Compare an `@file` turn with `/inspect-document file`. List every model, provider, and network recipient in each path.
3. Design a session-retention notice that distinguishes Pi files, provider retention, downloaded artifacts, and server invocation state.
4. Explain why a model name shown in `/model` is not evidence that the request used the portable route.
5. Define the checks required before replacing an existing `.pptx` file.

2.15. Chapter contract

- **Scope:** client/server responsibility, portable model selection, fixed extension loading, invocation-local attachments, progress cards, local inspection, artifact verification, and Pi session privacy.
- **Canonical claims:** Pi is an optional interactive client; server authority survives only when routing stays under `portable-agents` and local executable tools remain disabled.
- **Migration inputs:** former client/server material, the Pi 1.0.4 implementation notes, and the approved document-inspection evidence.
- **Evidence gates:** deterministic Pi acceptance, exact launcher/extension snapshots, and manually reviewed recordings.
- **Demonstration slots:** the approved model-independent inspection recording; the hero supplies the complementary attachment/progress path.
- **Source-claim ledger:** client behavior is tied to Pi 1.0.4 and reference snapshot 45a903c; formal protocol status remains separate.
- **Exclusions:** provider translation belongs to Chapter 5, protocol lifecycle to Chapter 6, executable tool authority to Chapter 8, and durable state to Chapter 15.

3. Agent Configuration and Reviewed Capabilities

Part I · The system in action

Configuration implementation: reference snapshot 45a903c · Parser ceiling: 65,536 bytes

Configuration should make the agent smaller and more predictable. The checked-in `agent.yaml` selects reviewed identifiers and bounded settings; it cannot load code, introduce a destination, provide credentials, or turn generated text into authority.

3.1. One configuration, one interpretation

`agent.yaml` is the sole implicit live-agent configuration. The server reads one bounded local file before it starts serving requests. It does not merge several files, interpolate environment variables, fetch includes, or accept request-time changes.

The root shape is intentionally small:

```
version: 1

server:
  host: 127.0.0.1
  port: 8787

coordinator:
  model: openai/gpt-5.4
  maxOutputTokens: 128000

inputFiles:
  maxFiles: 4
  maxDecodedBytesPerFile: 5242880
  maxDecodedBytesPerRequest: 10485760
  extractionTimeoutMs: 60000

skills:
  - name: business-tech-presentation

tools:
  - name: image_understanding
    provider: openai
    model: gpt-5.4
    maxCalls: 10
  - name: web_search
    model: gpt-5.4
    maxCalls: 300
  - name: read_source
    access: public-https
    maxCalls: 100
  - name: code_interpreter
    maxCalls: 10
```

```
- name: presentation_builder  
  maxCalls: 2
```

The complete checked-in file supplies every reviewed limit. The excerpt shows the principle: names select statically compiled components; fields narrow their operation.

3.2. A reviewed static catalog

A tool name is accepted only when the parser has an explicit branch for that exact identifier:

```
image_understanding  
web_search  
read_source  
code_interpreter  
presentation_builder
```

The reviewed skill catalog currently contains:

```
business-tech-presentation
```

This is not dynamic discovery. A configuration entry cannot point to a directory, npm package, URL, script, module, prompt file, class name, or executable. Adding a new identifier requires a source change, review, tests, and a new immutable revision.

The same rule applies to settings. Each tool branch accepts only fields meaningful to that implementation. For example, `read_source` can select `public-https`, call count, timeout, and response ceiling. It cannot accept arbitrary headers, a proxy, credentials, a private-network exception, or a custom fetch module.

3.3. Strict YAML admission

The loader limits the file to 65,536 bytes and parses it with unique-key enforcement. Startup rejects:

- duplicate mapping keys;
- YAML aliases;
- unsupported tags or parser warnings;
- unknown root, server, coordinator, input, skill, or tool fields;
- unknown skill and tool names;
- duplicate skill or tool selections;
- a missing explicit `tools` array;
- too many selected entries;
- values of the wrong YAML type; and
- out-of-range limits.

Failing on unknown fields is important. Ignoring `maxCall` when the reviewed field is `maxCalls` would silently remove the operator's intended bound. Treating the typo as startup failure keeps the effective policy observable.

The loader converts YAML to a plain value only after rejecting aliases. There is no anchor-based reuse, merge-key expansion, or tag-directed construction. This is a defense against ambiguous configuration, not a claim that YAML parsing alone secures the application.

3.4. Limits are server ceilings

inputFiles and per-tool settings are trusted deployment policy. A request or model-generated argument cannot raise them.

Limit family	Examples	Enforcement owner
Listener	loopback host, port	startup and server construction
Coordinator	selected provider/model route, maximum output tokens	route admission and model adapter
Input	file count, encoded/decoded bytes, extracted characters, retained bytes	ordered-input and asset registry
Extraction	timeout and process-wide concurrency	ingestion worker controller
Image	bytes, dimensions, pixels, frame count	image admission and decoder
Tool	calls, sources, timeout, code/output size	request-scoped tool runtime
Presentation	calls, slides, text, artifact bytes, TTL	builder and artifact store

Bounds compose; they do not replace one another. Passing the HTTP body limit does not imply that every contained file passes decoded-byte or extraction limits. A configured timeout does not authorize a tool call, and a call count does not prove the selected provider can satisfy the request.

3.5. Skills and tools are separate selections

A skill is reviewed non-executable procedure. A tool is a separately authorized executable capability. Their configuration arrays remain distinct:

```
skill → reviewed instructions and declared resources
tool  → compiled executor with schema, policy, limits, and effects
```

The current presentation procedure requires the bounded presentation builder. Startup therefore rejects either half selected alone. This dependency check prevents two misleading states:

- guidance that promises an unavailable builder; and
- a builder selected without the reviewed procedure expected by this application profile.

The dependency is static and application-specific. Loading the skill later does not register the tool, spend its budget, choose a provider, or widen any limit. Chapter 9 owns progressive disclosure and the full presentation case study.

3.6. What cannot appear in agent.yaml

The configuration intentionally excludes:

- API keys, bearer tokens, cookies, or credential-file references;
- arbitrary endpoint or redirect destinations;
- request headers;
- host filesystem paths;
- modules, plugins, packages, commands, or executable code;
- arbitrary prompts or model-generated policies;
- remote assets or presentation templates;
- dynamically discovered tools; and
- tenant, principal, or approval assertions.

Provider credentials and reviewed endpoint overrides, when needed, enter the trusted server process through the deployment environment. Keeping them out of YAML prevents an otherwise shareable capability selector from becoming a secret bundle. Environment ownership is not intrinsically safe; production deployment still needs a credential and configuration control plane.

3.7. Selecting a coordinator

The coordinator value has a bounded provider/model form with a reviewed provider prefix. This selects a route supported by server code; it does not accept a URL. The server rejects an unconfigured prefix rather than guessing or silently falling back.

A selected route is still only a declaration. Account access, regional availability, context limits, input types, and task quality require separate evidence. Chapter 14 owns feature admission and fallback. Chapter 5 owns provider construction and translation.

3.8. Explicit local customization

An operator can copy the reviewed file and point the server to one alternative path:

```
cp agent.yaml agent.local.yaml
PORTABLE_AGENTS_CONFIG=agent.local.yaml bun run serve:live
```

There is no merge with the checked-in file. The selected file must satisfy the complete schema. An omitted tools key fails rather than restoring defaults.

The former implicit server.yaml name is rejected with a migration error. It can be loaded only through an explicit PORTABLE_AGENTS_CONFIG path. This prevents stale legacy configuration from being silently interpreted as the current policy.

Legacy WEB_SEARCH_* environment overrides are also rejected. Search enablement and limits belong to the YAML tool selection; provider credentials and approved endpoint settings remain in the server environment. Mixing both mechanisms would make the effective configuration difficult to review.

3.9. Declaration is not authorization

Four decisions must remain distinct:

1. **Selection:** startup includes a reviewed component in this deployment.
2. **Capability evidence:** the chosen route has sufficient dated evidence for the task profile.
3. **Invocation authorization:** the current principal and request may perform this operation now.
4. **Execution admission:** arguments, budgets, cancellation state, and dependencies pass immediately before the effect.

`agent.yaml` performs the first decision and supplies ceilings used by the fourth. It does not identify a principal or grant per-request authority. A selected tool can still be denied, exhausted, cancelled, or rejected for invalid input. Conversely, prompt text cannot select a tool absent from the startup catalog.

A configured capability is **available for policy consideration**, not automatically authorized. Generated arguments remain a proposal until the server validates current trusted context and admits execution.

3.10. Startup is the failure boundary

Malformed or contradictory configuration fails before the listener accepts work. Startup errors identify the field category without echoing secrets or the full file. The process does not continue with a partial tool list, guessed defaults, or a disabled validation branch.

Useful startup diagnostics include the effective loopback address, coordinator routes, and selected reviewed tools/skills. They should not include API keys, endpoint credentials, prompt bodies, or complete configuration text.

This design favors availability loss over policy ambiguity. A production supervisor may restart a corrected deployment; it should not reinterpret invalid policy while requests are active.

3.11. Deterministic evidence

The reference tests cover:

- defaults and the complete checked-in file;
- unknown and duplicate tools;
- unknown and duplicate skills;
- extra fields at every relevant level;
- malformed numbers and limit ranges;
- aliases, tags, and duplicate YAML keys;
- missing explicit tool selection;
- presentation skill/builder dependency mismatch;
- loopback-only binding;
- legacy filename and environment rejection; and
- startup wiring from parsed settings to the selected compiled tools.

These tests establish parser and construction behavior for the named revision. They do not establish production principal authentication, secret management, deployment approval, or live provider capability.

3.12. Failure modes

3.12.1. Configuration as plugin loader

A path or package field turns declarative selection into code execution. Reviewed IDs avoid that hidden authority expansion.

3.12.2. Permissive unknown fields

Ignoring a typo can remove a limit while startup appears healthy. Strict field sets make drift fail visibly.

3.12.3. Tool selection as blanket permission

A selected tool still needs request-scoped authorization and execution admission. Startup cannot know the future principal, arguments, effect history, or cancellation state.

3.12.4. Skill text as capability grant

Instructions can shape model behavior but cannot register an executor or widen policy. Treating text as authority recreates prompt injection at the configuration boundary.

3.12.5. Capability declaration as evidence

A model route listed in configuration may be unavailable or semantically unsuitable. Replacement evidence belongs to Chapter 14, not the YAML parser.

3.13. Pattern summary

Use one bounded, strict, declarative selector over a static reviewed catalog. Reject ambiguity at startup. Keep credentials, destinations, code loading, principal identity, capability evidence, and invocation authorization in their own trusted boundaries.

3.14. Exercises

1. Add a hypothetical `email_sender` tool. List the source changes required without adding a module or path field to YAML.
2. Explain why `tools: []` is safer than an omitted `tools` key that means “use defaults.”
3. Construct a duplicate-key YAML example whose permissive interpretation would widen a limit.
4. Identify which of selection, capability evidence, invocation authorization, and execution admission should reject an exhausted call budget.
5. Design a migration from one configuration version to another without silently accepting old fields.

3.15. Chapter contract

- **Scope:** `agent.yaml`, static reviewed skill/tool selection, strict syntax and fields, bounded settings, dependency checks, startup failure, and the distinction between selection and authority.
- **Canonical claims:** configuration selects reviewed compiled components and ceilings; it cannot load code or authorize a future invocation.
- **Migration inputs:** former capability/configuration material and the current strict live-server parser.
- **Evidence gates:** deterministic parser, wiring, dependency, and startup-failure tests for an immutable snapshot.
- **Demonstration slots:** the hero shows the effective selected tool/skill profile; configuration enforcement is established by tests rather than model behavior.
- **Source-claim ledger:** current skill identity and dependencies are tied to snapshot `45a903c`; older `v0.4.0` parser evidence remains historical where applicable.
- **Exclusions:** provider translation belongs to Chapter 5, skill procedure to Chapter 9, capability substitution to Chapter 14, and principal authorization to Chapters 8 and 17.

4. Documents, Images, and the Ingestion Boundary

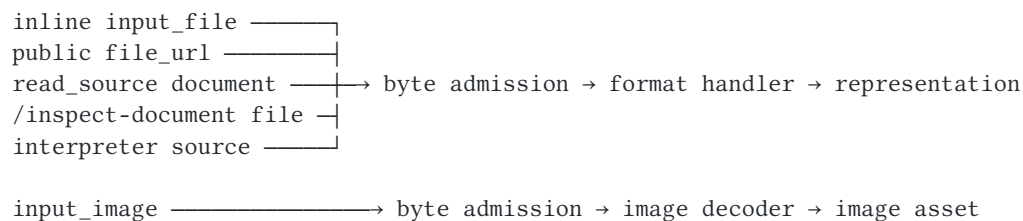
Part I · The system in action

Ordered-input target: Open Responses 2026-04-24 · Xberg profile: v2 PDF/XLSX · Reference snapshot: 45a903c

A file is not text, evidence, or model input merely because the client attached it. The ingestion boundary must identify bytes, preserve order, detect format, enforce structural limits, produce one reviewed representation, and keep that representation invocation-local.

4.1. One path for every admitted input

The reference reuses one DocumentIngestionService across several entry points:



Reusing the service prevents a local inspection command, URL reader, and model attachment path from quietly applying different PDF limits. Entry points still have separate authority: `/inspect-document` may read a local path selected by the user, while an HTTP request cannot turn a file-name into a host path.

The implementation target follows the Open Responses ordered content shape for `input_text`, `input_file`, and `input_image` (“Open Responses Reference 2026-04-24,” n.d.). This is an implemented local profile with additional restrictions, not full protocol conformance.

4.2. Preserve content order or reject it

A mixed user message can interleave instructions, files, images, and more text. Flattening all text first and appending extracted documents later changes meaning. The parser therefore retains message and content indices as origin metadata.

A simplified internal order is:

```
input_text
input_file -> admitted document marker/part
input_text
input_image -> admitted image marker/part
```

Unknown items, unknown content parts, extra fields in supported message/content objects, malformed discriminators, and unsupported media fail explicitly. They are not silently discarded. System and developer arrays remain text-only in the reviewed profile; known function-call history has its own strict mapping.

The local file rule requires exactly one non-empty `file_data` or `file_url`. The pinned schema permits both or neither, so this exclusive-or rule is a documented local security restriction rather than a statement about what all conforming implementations must do.

4.3. Byte admission comes before parsing

Admission treats supplied metadata as a claim. It detects media from bytes, compares reviewed declared types where present, normalizes a bounded display filename, computes a digest, and refuses unsupported or ambiguous input before invoking a parser.

The checked-in profile includes ceilings for:

Resource	Checked-in ceiling
files per request	4
encoded characters	7,000,000
decoded bytes per file	5 MiB
decoded bytes per request	10 MiB
extracted characters per file	5 MiB-equivalent character ceiling
extracted characters per request	10 MiB-equivalent character ceiling
retained invocation bytes	50 MiB
extraction deadline	60 seconds
concurrent extractions	1 process-wide
image bytes	5 MiB
image dimension	8,192 pixels
image pixels	40,000,000
image frames	1
filename length	240 characters

The HTTP application body limit is 8,000,000 bytes, below the larger schema-only base64 maximum. Near-schema-limit experiments showed material request-memory amplification, so the implementation publishes a narrower profile rather than pretending the schema maximum is operationally safe.

Limits are layered. A request can pass the listener ceiling and still fail decoded bytes, archive expansion, image pixels, extracted characters, or retained-invocation capacity.

4.4. Supported document profiles

The current profile admits:

- HTML;
- PDF;
- DOCX;

- XLSX;
- UTF-8 text and Markdown;
- JSON;
- JSONL/NDJSON;
- CSV; and
- TSV.

JSON and line-delimited JSON are structurally parsed. Text-like formats retain complete admitted UTF-8 text. HTML, PDF, DOCX, and XLSX use pinned Xberg extraction in a killable worker. PDF and XLSX use explicit checked-in profiles because their security and fidelity risks differ from simple passthrough.

Every document result records:

- invocation-local source ID;
- content digest;
- sanitized filename;
- detected media type;
- complete canonical text;
- extraction mode and method;
- warning count;
- page and chunk counts;
- ingestion/profile revision; and
- original admitted bytes retained only for reviewed invocation-local uses.

The representation says `completeInputExamined: true` only after the complete admitted input passes its handler. A warning, timeout, malformed structure, unsupported encryption, partial extraction, or excess output fails instead of returning text labelled complete.

4.5. The Xberg worker boundary

Xberg 1.3.8 runs in a child worker rather than the main request process. The server passes one reviewed profile; callers cannot supply extraction options, model paths, plugins, or layout strategies.

Common profile controls include:

- no extraction cache;
- Markdown output;
- one concurrent extraction;
- 60-second parser timeout;
- 20 MiB archive/content ceiling;
- 20:1 compression ratio;
- 100 archive entries;
- 64 nesting levels;
- bounded entities and parser iterations; and
- 5 MiB embedded-file ceiling.

The PDF profile extracts pages, tables, metadata, annotations, footnotes, and document structure while removing repeating headers/footers and excluding form fields, watermarks, and embedded

images. Markdown chunks use a 4,000-character target with 200-character overlap and repeated table headers.

PDF OCR uses local Tesseract fallback when native text is inadequate. Layout-model inference remains disabled because no model artifacts have been independently admitted. Enabling it would add packages, model bytes, resource use, and potentially new recipients; it is not a harmless configuration toggle.

The XLSX profile extracts structured tables/document shape with 6,000-character chunks and no overlap. It does not run OCR. A spreadsheet table is still untrusted data and may contain formulas, misleading labels, or instruction-like text.

4.6. Canonical-text fallback

Not every selected model route has reviewed native file support. For a user-message document, the server can replace the file part in place with its complete bounded canonical text and an explicit untrusted-document marker. This preserves order and allows a text-capable route to reason over the admitted representation.

Canonical-text fallback does **not** mean:

- the provider received the original PDF or workbook;
- native provider file semantics were tested;
- layout was perfectly preserved;
- page-level citation identity exists; or
- extracted statements are verified facts.

PDF and XLSX are sent to provider adapters as bounded text/plain extraction in the reviewed Pi path. Original bytes remain invocation-local for reviewed mounts and inspection. Function-output files and images follow stricter capability declarations because their provider mappings differ.

4.7. Image admission and recipients

Reviewed image formats are PNG, JPEG, single-frame GIF, and WebP. Admission sniffs bytes, compares media claims, bounds dimensions/pixels/frames, and requires complete decoding through Sharp. SVG, animated images, and multipage images are outside the current profile.

An image asset records immutable bytes, digest, dimensions, frame count, detail level, origin, ingestion revision, and invocation-local source ID. The source ID—not a path, URL, or byte string—is the only selector accepted by the `image_understanding` tool.

For the mixed-provider bridge, an Anthropic coordinator sees a text marker containing the admitted source ID rather than image bytes. A separately authorized tool call can send exactly one admitted image and one focused question to a fixed OpenAI recipient selected by server configuration. The result identifies that recipient and labels its observations generated: `true`.

This is recipient minimization, not image truth. Generated observations are not provenance, authenticity, identity proof, coordinate-preserving OCR, or evidence that a public source was consulted. Image text remains untrusted data, never instructions.

4.8. Invocation-local identity

Source IDs combine request identity, content digest, and ingestion revision. Assets are immutable after admission and disappear on completion, failure, cancellation, or stream disposal. There is no trusted cross-request asset identity.

This design prevents a later request from guessing a source ID and retrieving another request's bytes. It also means formal local-file citations remain limited: a transcript can retain extracted words, but it cannot claim a durable file/page/span identity that the application did not preserve.

A tool argument may select an admitted source ID. It cannot introduce a new path, URL, filename, provider, or byte payload. `code_interpreter` receives selected sources at fixed `/inputs` locations; it cannot request an arbitrary host mount.

4.9. `/inspect-document` reuses the same representation

The Pi command accepts one explicit local path, rejects control characters, symbolic links, directories, empty/oversized files, and unsupported formats, then invokes a fixed Bun child with `execFile` and no shell. The child imports the same ingestion service and checked-in Xberg configuration.

The Pi extension owns a private temporary directory. The child writes the complete extraction and an exact metadata record with owner-only permissions. The parent recomputes the digest before viewing or saving, and cleanup is idempotent across success, failure, cancellation, and session shutdown.

Current implemented behavior opens the complete already-bounded extraction at the first line in a read-only viewer. Navigation and explicit save do not add the content to model input, conversation messages, progress, ordinary notifications, or logs. Metadata-only notifications can report bounded counts and the digest.

This command proves representation inspectability. It does not add a second parser, a model-callable local-file tool, or a citation system.

4.10. Cancellation and worker termination

Cancellation has to cross every layer:

```
request/Pi cancellation
  → ingestion service abort
  → child worker termination
  → queue release
  → invocation registry cleanup
  → temporary directory cleanup
```

A queued extraction must stop waiting when cancelled. A running parser must be killable rather than trusted to poll cooperatively. A timeout or cancellation cannot publish partial text as complete. Cleanup may run more than once without widening its ownership scope.

A hard process kill cannot execute ordinary cleanup. The current implementation does not scan and delete unrelated temporary directories at startup; any future scavenger needs an ownership marker, age bound, and separate tests.

4.11. Malformed and hostile content

The boundary fails closed for:

- malformed base64 and data URLs;
- unsupported or mismatched media;
- unsafe public URLs and redirects;
- missing or invalid request framing;
- malformed PDF/DOCX/XLSX archives;
- encrypted or incomplete documents;
- archive bombs and excessive nesting;
- oversized entities or parser work;
- decompression-ratio violations;
- invalid UTF-8 where UTF-8 is required;
- image decode failures and excess pixels/frames;
- parser timeout or cancellation; and
- extraction warnings when the caller requires a complete warning-free inspection.

The text that survives remains untrusted. A document can contain prompt injection, fabricated citations, misleading tables, or data exfiltration instructions. Admission permits bounded use; it does not grant authority or establish truth.

4.12. Demonstration: the BCAUSS paper

The approved model-independent recording uses the public paper *Learning end-to-end patient representations through self-supervised covariate balancing for causal treatment effect estimation*, supplied by an author. The book distributes the reviewed recording, poster, transcript, alt description, and metadata—not the PDF or extracted article text.

The recorded run reports:

```
16 pages
44 chunks
129,145 extracted characters
0 warnings
SHA-256 f5676a2e78eb050fcb13b8b1225d0f2476710bdd6ff981c81af7251fc96208b9
0 model/provider/public-network calls
```

This evidence is dated and format-specific. It demonstrates one local PDF path, navigation, explicit save, and metadata. It does not establish universal PDF fidelity, scientific validity, exact publisher layout preservation, page-span citations, or safe extraction of every malformed file.

4.13. Deterministic evidence

The current tests cover ordered mixed content, strict unknown-part rejection, file-source XOR, base64 admission, request and decoded limits, media detection, JSON/text handlers, generated PDF/DOCX/XLSX fixtures, archive limits, image decoding, provider translation, canonical-text fallback, asset cleanup, interpreter mounts, cancellation, worker timeout, inspection metadata, permissions, digest equality, and zero-provider inspection behavior.

Synthetic transport tests can show that ordered parts reach adapter mappings. They do not establish live model quality or universal provider-native file behavior. The official Open Responses suite currently has no comprehensive file matrix; file support alone cannot establish conformance.

4.14. Pattern summary

Admit bytes once, preserve content order, apply a reviewed format profile in a killable worker, produce one immutable invocation-local representation, and expose only source IDs to tools. Keep extraction, provider delivery, citation identity, and claim support as separate assertions.

4.15. Exercises

1. Trace a PDF from Pi attachment to provider input. Mark every point where bytes become text or metadata.
2. Explain why a 5 MiB file limit does not protect against every archive or image bomb.
3. Design a retained-source identity that could support later local citations without exposing one tenant's document to another.
4. Compare native provider file delivery with canonical-text fallback. Name the lost semantics.
5. Specify cancellation tests for a request waiting behind another extraction.

4.16. Chapter contract

- **Scope:** ordered file/image admission, byte and structural limits, PDF/DOCX/XLSX/text profiles, canonical-text fallback, image decoding, extraction identity, local inspection, cancellation, and recipient disclosure.
- **Canonical claims:** an admitted representation is bounded and identifiable but remains untrusted; extraction is not citation support or truth verification.
- **Migration inputs:** ingestion material formerly mixed into grounding plus the current ordered-input, Xberg, image, and inspection implementations.
- **Evidence gates:** deterministic malformed/oversized/cancellation fixtures, exact profile revisions, provider-mapping tests, and approved local inspection evidence.
- **Demonstration slots:** the BCAUSS /inspect-document recording; the Northstar hero supplies mixed image/PDF/XLSX admission.
- **Source-claim ledger:** profile claims are tied to Open Responses 2026-04-24, Xberg 1.3.8 profile v2, and reference snapshot 45a903c.
- **Exclusions:** public URL acquisition belongs to Chapter 11, tool authorization to Chapter 8, claim support to Chapter 13, and durable cross-request identity remains unimplemented.

Part II.

Part II — The Portable Model Runtime

5. The Provider Adapter Boundary

Part II · The portable model runtime

Primary-source review: 2026-10-08-09 · Immutable implementation baseline: v0.4.0 · Historical live reports: v0.3.0

The Northstar research flow now works. Its schemas are checked, calls are correlated, and streams have terminal meaning. Then an operator changes the coordinator deployment. The code still compiles, but token policy, protocol details, and representable features can change. The difficult question is no longer how to write an adapter. It is which decisions the adapter is allowed to make.

A **provider adapter boundary** separates a stable application model contract from the construction, configuration, and translation decisions needed to execute it on a particular provider deployment. Replacement is controlled when those decisions can change without silently changing the application's authority or required meanings.

5.1. The problem

In Part I, the Northstar runtime comparison had one purpose: compare two synthetic measurements without treating provisional output as a final answer. We now want the same purpose to survive a different model deployment. The tool implementation, authorization decision, and result correlation should not be rewritten merely because a different model proposes the calls.

There are two tempting solutions. One is to duplicate the whole agent loop for each provider. The other is to move every provider setting into one universal request object. The first duplicates the invariants established in Chapters 6–8; the second makes the core dependent on a growing catalogue of vendor features. Neither tells us what substitution actually preserves.

The architectural task is to divide *decisions*, not just files. Which layer chooses the endpoint? Which one supplies a token default? Who interprets tool arguments? Who decides that an unsupported feature may be dropped? Who owns the retry budget after a tool has executed? A change is portable only when the relevant owner can change it under an explicit contract.

This chapter follows the immutable v0.4.0 adapter implementation and uses the older v0.3.0 reports only for the live runs they actually record. It extends [Chapter 1's model boundary](#) into deployment construction, examines the shared SDK seam, and shows where normalization loses information. Chapter 14 owns capability admission and fallback; Chapter 15 owns recovery.

5.2. Context and forces

One place for invariants. Within one authority domain, shared tool policy, cancellation propagation, event normalization, and iteration limits need one owner and implementation. Duplicated loops make a provider change also a policy change. Client-local tools remain a separate execution domain; a server cannot acquire their authority by advertising their schemas.

A visible place for differences. Required request fields, model construction, credential headers, endpoint policy, and provider-specific features cannot always be made identical. Their differences need an owner, not a comment claiming that the SDK handles everything.

A finite application contract. A useful port describes what the application depends on. It need not represent every feature a provider offers. But narrowing the contract creates a responsibility to reject unsupported requirements or disclose what is lost.

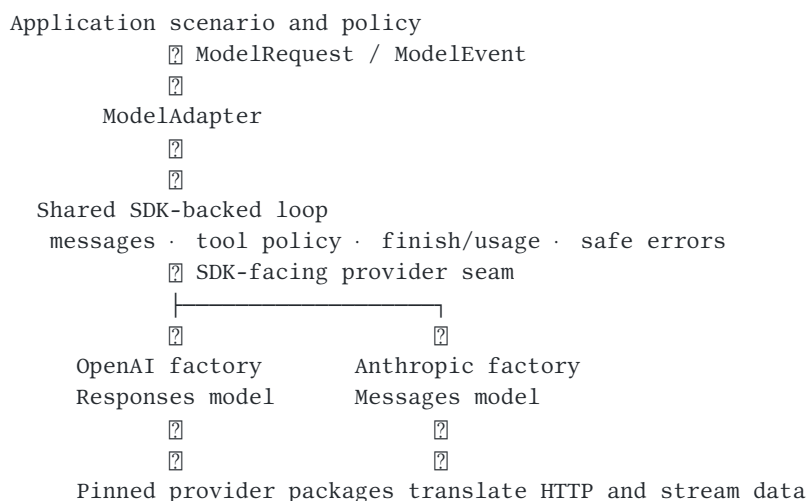
A bounded replacement claim. A TypeScript assignment proves structural compatibility, not that a particular account can call a model, that a gateway implements the selected protocol, or that the Northstar comparison remains correct.

Reading rule. For each setting and response field, ask three questions: who chooses it, who translates it, and what evidence shows that its meaning survived?

5.3. Separate the application port from the SDK seam

The reference has two interfaces with different audiences. The application-facing `ModelAdapter` in `src/model.ts` exposes a diagnostic provider label, model-scoped capability declarations, and a stream of application events. Its request includes typed history and optional tools, trusted tool context, output-token limit, step limit, and abort signal. Callers need not construct an AI SDK message or retain a provider SDK client.

The internal `AiSdkProviderBoundary` additionally exposes `languageModel(modelId)` and `maxOutputTokens(request)`. The former returns an AI SDK `LanguageModel`. This is intentional implementation coupling: the seam is not a claim that the reference could replace its orchestration library without changing internal code. What remains stable is the application-facing request/event boundary.



This diagram describes the reference baseline, not every agent architecture. A direct protocol implementation would replace the SDK-facing layer and take responsibility for framing, validation, and provider-specific lifecycle behavior. Merely replacing a factory does not transfer those responsibilities to application code.

The shared implementation in `src/agent.ts` converts typed history to SDK messages, registers validated tools, invokes `streamText`, and projects the resulting stream. `AiSdkModelAdapter`

delegates its `stream` method to that same function. Neither provider subclass contains its own tool-continuation loop.

Share mechanics whose meaning must remain invariant. Isolate decisions whose value depends on the provider or deployment. Keep an SDK type at the SDK seam rather than pretending that all implementation dependencies have vanished.

An interface can therefore be provider-neutral without its implementation being dependency-free. The test of a boundary is whether the application can substitute an implementation at that boundary, not whether every layer beneath it is also interchangeable.

5.4. Bind deployment identity before generation

A model name is not a complete execution target. At least four identities matter:

Identity	What it selects	What it does not prove
Application selection	A trusted deployment binding or routing decision	Provider availability
Provider model ID	The model passed to the provider factory	Endpoint, account, or region
Protocol family	Responses, Messages, or another API	Complete feature fidelity
Credential scope	The principal and service authority used for the call	Authorization of the user or its tools

The deployment also carries endpoint configuration, dependency versions, and operational limits. Two clients using the same model ID can still have different credentials, regions, gateways, or policy. A routing prefix is therefore not a proof of deployment equivalence.

In `v0.4.0`, application code chooses an adapter object and supplies the provider model ID in the request. The tagged `OpenAI factory` constructs a provider with the supplied API key and optional endpoint, organization, project, and fetch implementation, then explicitly selects `.responses`. The `Anthropic factory` supplies its API key, optional endpoint and fetch implementation, then selects `.messages`.

These are real TypeScript implementation examples: inspect the constructors together, then inspect the common `stream` delegation. Their significance is not that the classes have similar names. They bind different protocol construction behind one application port. The pinned packages are `ai 7.0.131`, `@ai-sdk/openai 4.0.87`, and `@ai-sdk/anthropic 4.0.75`; current SDK documentation may describe additional settings that these wrappers do not expose.

The OpenAI provider documentation’s “Responses Models” and “Chat Models” sections distinguish those APIs even within the same package (Vercel, n.d.-b). Changing only a base URL cannot make a Chat-Completions-only gateway implement Responses. Select a protocol implementation first; then verify that the chosen endpoint accepts it.

5.4.1. A small, tested deployment-binding example

The following is an **independently authored, executable book design**, not the reference server's routing API. It permits the application to select an exact deployment alias from trusted configuration. An adapter has already been constructed; the selection cannot supply a new endpoint or credential. The generic adapter value can be a reference `ModelAdapter`, but the binder does not require SDK types.

```
export interface Deployment<Adapter> {
  readonly alias: string;
  readonly modelId: string;
  readonly adapter: Adapter;
}

export type ModelTarget<Adapter> = Readonly<Pick<Deployment<Adapter>, "modelId" |
  ↳ "adapter">>;

export class DeploymentSelectionError extends Error {
  constructor(readonly code: "invalid_deployment" | "unknown_deployment") {
    super(`Deployment selection failed (${code}).`);
    this.name = "DeploymentSelectionError";
  }
}

export const bindDeployments = <Adapter>(
  deployments: readonly Deployment<Adapter>[],
): ((alias: string) => ModelTarget<Adapter>) => {
  const targets = new Map<string, ModelTarget<Adapter>>();
  if (deployments.length === 0) throw new
    ↳ DeploymentSelectionError("invalid_deployment");
  for (const { alias, modelId, adapter } of deployments) {
    if (
      !/^[a-z][a-z0-9_-]{0,63}$/.test(alias) ||
      modelId.length === 0 ||
      modelId !== modelId.trim() ||
      targets.has(alias)
    ) {
      throw new DeploymentSelectionError("invalid_deployment");
    }
    targets.set(alias, Object.freeze({ modelId, adapter }));
  }
  return (alias) => {
    const target = targets.get(alias);
    if (target === undefined) throw new DeploymentSelectionError("unknown_deployment");
    return target;
  };
};
```

The complete source is [code/deployment-binding.ts](#). Tests cover exact adapter identity, opaque model IDs containing punctuation, duplicate and invalid declarations, unknown selections, and mutation of the original configuration after binding. Unknown aliases produce a fixed error rather than model-name inference or silent fallback.

This is a deliberately small boundary experiment. It is not account authentication, per-principal deployment authorization, endpoint validation, or capability discovery. Trusted configuration remains a precondition. If different readers may use different deployments, the application must

authorize the selection before resolving it. Freezing a binding does not freeze or isolate the adapter object itself.

Constructing the lookup once also keeps registry scanning out of the event loop: construction is linear in deployment count; selection uses a map lookup. That is a complexity statement about this helper, not a production latency benchmark. A deployment registry does not justify adding a service, cache, or background refresh loop before there is a concrete requirement.

The current [live-server snapshot](#) instead routes configured prefixes such as `openai/<model-id>` and rejects unknown prefixes. It is not an alias allowlist of individual models. Both designs make selection explicit, but their admitted target sets differ. Do not describe this book helper as a control already enforced by that server.

5.5. The boundary output

This chapter's output is deliberately narrow: an exact deployment binding plus a translation result whose retained and lost meanings are documented. It does not decide whether a fallback is acceptable, whether a failed attempt may repeat, whether history may be replayed, or whether a source supports a claim.

Fact leaving this boundary	Owner that consumes it
exact deployment/adapter identity	Chapter 14 capability admission
normalized request and event meanings	Chapters 6–8 runtime contracts
explicit unsupported or lost meaning	caller task policy and Chapter 14
sanitized provider failure category	Chapter 15 recovery policy
provider/SDK response identity, when present	Chapter 15 history without treating it as a run ID

Selecting a deployment cannot authorize a tool, and a translation cannot certify evidence. The [worked trace in Chapter 13](#) uses a scripted adapter precisely so local policy composition is not confused with a live-provider claim.

5.6. Assign every translation an owner

Walk a request through the reference before deciding where an extension belongs.

Decision or transformation	Reference owner	Architectural consequence
Select adapter/deployment	Caller or composition layer	Do not infer authority from prompt text
Construct model, bind credentials and endpoint	Provider factory and SDK	Configuration is trusted deployment input
Convert messages, calls, and results	Shared <code>modelMessage</code> function	One history conversion policy for both factories
Validate schemas and apply tool hooks	Shared loop and tool runtime	Provider replacement need not replace execution authority

Decision or transformation	Reference owner	Architectural consequence
Serialize provider request and parse wire events	Pinned provider package	A mock model does not prove wire correctness
Normalize finish, usage, and failures	Shared loop	Review information loss once, across providers
Decide whether to retry, fall back, or continue	Application/run policy	A failure category is not a recovery command

The shared conversion places function calls in assistant messages and function results in tool messages for the SDK. The provider packages then produce their respective wire representations. The application owns the correlation fact; the SDK owns its provider encoding. Rewriting a tool result into prose at the application boundary would discard the fact before either provider receives it.

System instructions illustrate the same separation. The neutral request’s system field is passed separately to `streamText`. The Messages API reference, “Body parameters / system and messages,” uses a top-level system parameter rather than a system input role. The Responses reference, “instructions and input,” exposes its own instruction/role-based forms (Anthropic, n.d.-b; OpenAI, n.d.-c). A common application field does not imply identical wire fields or identical instruction-following behavior.

5.6.1. Required settings versus application budgets

The same API references’ `max_tokens` and `max_output_tokens` field descriptions distinguish an explicit Messages request field from an optional Responses bound including visible output and reasoning tokens (Anthropic, n.d.-b; OpenAI, n.d.-c). The reference preserves that request-shape difference:

- The base adapter returns the request’s `maxOutputTokens` unchanged. If absent, the OpenAI path omits it from the SDK call.
- The Anthropic subclass supplies `request.maxOutputTokens` when present, otherwise its configured default; the implementation default is **1,024**.
- The shared loop includes the SDK setting only when a value exists.

A provider’s required field is not the application’s entire budget policy. The reference default is a construction decision, not a proven safe value for every model or task. A production run still needs an explicit ceiling and a policy for exhausting it. Conversely, imposing one small constant everywhere can preserve request validity while changing whether a reasoning model has enough budget to produce visible text (OpenAI, n.d.-b).

The declared application contract currently has no reasoning-effort, provider-option envelope, or provider-private continuation field. Those omissions are scope boundaries. They are not evidence that every model behaves acceptably with its defaults. Adding a setting requires deciding whether it expresses application intent, deployment policy, or a provider-specific extension. Arbitrary JSON passed through to a provider is not a substitute for that decision: it enlarges the authority and compatibility surface without defining its semantics.

5.7. Normalization is an information-loss decision

Let a provider response be richer than the application projection. Mapping both into a common event union can be useful even when the mapping is lossy. The important distinction is between information that the application has explicitly excluded and information it needs but the adapter silently discards.

The reference maps text, complete tool calls, tool outcomes, finish reasons, and optional input/output token counts. It does not project every provider item, reasoning artifact, source annotation, warning, raw finish detail, or token breakdown. Its response identifier is updated from SDK step metadata; consumers must not assume that this optional provider/SDK response ID is the same thing as a durable application run ID.

Provider APIs have different termination vocabularies. Anthropic streaming includes message and content-block lifecycle events; Open Responses uses its own response/item/content vocabulary (Anthropic, n.d.-c; “Open Responses Specification 2026-04-24,” n.d.). The SDK translates provider-specific outcomes before the shared loop maps its normalized finish reason. A normalized stop or length is therefore the result of two translations, not a raw provider assertion.

The current common loop’s unknown case prevents an unfamiliar SDK finish reason from being mislabeled as ordinary completion. It does not preserve the raw reason for diagnosis. Likewise, omitting an unknown token count avoids inventing a measured zero, but retaining two counts is not complete accounting of reasoning, caching, or provider billing.

Warnings deserve the same discipline. The `streamText` return reference, “warnings,” exposes unsupported-setting warnings, but the tagged event contract does not carry them (“AI SDK Core,” n.d.-a). A future extension must decide which warnings affect admission, which are safe for public output, and which belong in protected diagnostics. Do not claim that an SDK accepting a setting means the model honored it.

A useful design rule follows: **loss is acceptable only relative to the consumer’s obligations**. For the synthetic Northstar comparison, a text/tool subset may be sufficient. For a product whose answer must contain verified document citations, dropping annotations is a functional defect. For reasoning continuation, dropping opaque compatible state may alter the next turn even if the visible transcript looks unchanged. Chapters 15 and 13 treat those state and evidence obligations directly.

5.8. Provider-specific variation: a failed run is architectural evidence

The retained [failed provider report](#) records two different failures in one fixed text-generation scenario at tested commit 672d00c. OpenAI gpt-5 returned empty output under a 32-token request limit; the custom Anthropic endpoint returned HTTP 404. The report’s normalized usage records do not establish the full internal reason for the empty output. The official reasoning guide describes how output limits can be reached before visible output, but that mechanism must not be promoted into a diagnosis that the report itself did not capture (OpenAI, n.d.-b).

The [passing report](#), at tested commit 1bbf572, uses a 512-token limit and provider-default endpoint configurations. It records three successful exact-match attempts each for gpt-5 and claude-sonnet-4-6.

This is not a controlled experiment isolating one parameter. It is evidence that deployment configuration belongs inside the compatibility claim. A short expected answer does not imply that

all models require only a short generation budget. An endpoint override is not proof that the requested protocol exists there. Both failures could occur with unchanged orchestration code and structurally compatible adapters.

Retaining the failed run is more useful than erasing it behind a successful release label. It identifies assumptions that future substitution tests must exercise. It does not establish live Northstar-tool execution, broad semantic equivalence, or universal availability of either model. The tag containing a report also is not necessarily the commit that produced the report; inspect the recorded commit rather than treating the release directory as the experiment identity.

5.9. Failure modes

5.9.1. A provider adapter that becomes a second agent

If a subclass starts authorizing tools, selecting application memory, or deciding a task-specific stopping condition, it has crossed the ownership boundary. A new provider can then acquire a different authority model under the same application interface. Keep the provider subclass small for a reason: not aesthetic minimalism, but a smaller surface where replacement can change policy.

5.9.2. Endpoint changes presented as model changes

A gateway can change protocol support, authentication, metadata, streaming behavior, or retention. Recording only the model name hides these changes. Test the selected factory/endpoint pair, not a marketing label such as “compatible.”

5.9.3. Silent defaults mistaken for neutral semantics

A missing limit can become a provider default, a factory default, or a deployment guard. Record the effective setting and its owner. The later live bridge clamps output budgets separately; that is not behavior of every v0.3.0 adapter call.

5.9.4. Recovery hidden below observable effects

The reference sets SDK `maxRetries` to zero and does not implement a durable retry workflow. A provider error’s `retryable` flag is a hint, not permission to repeat a tool-bearing operation. HTTP itself distinguishes retry safety from method syntax and warns against automatically retrying non-idempotent requests without additional knowledge (Fielding et al. 2022). A gateway or outer run can still introduce retries; the architecture needs one explicit owner, developed in Chapter 15.

5.9.5. Complete projection mistaken for complete fidelity

The application event union can be exhaustive while its mapping covers only a subset of the SDK stream. Type exhaustiveness protects the chosen contract, not all provider meanings. Keep a loss register: unsupported or dropped field, affected consumer obligation, disposition, test, and version.

5.10. Security considerations

The selected endpoint is a recipient of both credentials and request content. Moving it is a security change even when the adapter type is unchanged. The reference supports endpoint and fetch injection; it does not enforce an endpoint allowlist or bind a credential cryptographically to its intended recipient.

Treat model IDs and deployment aliases as input, but endpoint URLs, organization/project settings, and credentials as trusted configuration. Do not accept a request's arbitrary URL and attach the server's provider key to it. A custom fetch implementation, proxy, redirect, or environment override can affect the actual destination; audit the complete transport path rather than just the constructor argument. Even an allowlisted initial URL is not proof of where every redirected or delegated request goes.

The pinned SDK factories can obtain endpoint defaults from environment variables when an explicit endpoint is absent. Reproducible tests should therefore isolate ambient configuration and record a safe endpoint class. Do not publish keys, internal endpoint URLs, or raw headers as evidence. A model-availability placeholder in a loopback client is not a provider credential.

Client-to-server identity remains independent of server-to-provider identity. The post-release development server is loopback-only and unauthenticated; neither the adapter port nor possession of a provider key authorizes a reader's tool call. Tool context must still originate from trusted application policy, as Chapter 8 establishes.

5.11. Observability handoff

The boundary must expose enough typed information for later instrumentation—deployment alias, provider/model ID under policy, protocol family, adapter revision, effective limit ownership, terminal category, and explicit loss decisions—without logging prompts, credentials, tool payloads, opaque continuation data, or raw provider failures. Chapter 16 owns span names, propagation, sampling, attributes, cardinality, and redaction tests. No OpenTelemetry exporter is claimed here.

5.12. How we test portability

Start with the [current release TypeScript adapter tests](#). They are executable examples of replacement at the SDK seam, not provider-wire emulators.

Tested example	Actual assertion	Limit of the proof
OpenAI model injection	Requested ID reaches the injected factory; output tokens remain unset; normalized text/usage collected	Real HTTP headers and endpoint not exercised
Anthropic model injection	Requested ID reaches the factory; configured default of 768 reaches the mock call	Fixture value is not the implementation default or an optimal budget

Tested example	Actual assertion	Limit of the proof
Common tool continuation	Executor receives 2 and 3; correlated output contains result 5; a second model step produces text; usage totals 12 input and 4 output	Arithmetic fixture, not live Northstar selection or semantic equivalence
Invalid input/output and context	Invalid input does not execute; invalid output produces failure; policy hooks without context do not reach the model	Complete deployed authorization not established
Unknown usage and safe errors	Missing counts remain absent; sensitive test error text is not exposed	Full provider usage and protected diagnostics not established

The provider constructors accept `languageModelFactory`, so these tagged fixtures prove forwarding and shared-loop behavior while bypassing the normal construction path. Duplicate-call prevention has its own runtime fixture; the arithmetic continuation fixture does not itself count executor invocations.

The independently authored [factory and counterexample probe](#) now exercises the **normal tagged constructors** through an injected fetch. No model-factory override is supplied. The probe inspects each POST route, synthetic authentication headers, system/history translation, and effective output setting. It supplies a minimal successful SSE stream, an HTTP 401, and an in-stream failure for each provider. All six paths make exactly one synthetic request and assert either normalized text/finish or a sanitized failure. The [retained report](#) records the baseline and assertion count; it contains no actual credential or endpoint.

```
OpenAI constructor    → /v1/responses → output limit omitted
Anthropic constructor → /v1/messages → max_tokens = 1024
Injected SSE          → SDK decoding → text "portable", finish "stop"
```

Explicit fixture keys and endpoints isolate ambient configuration. Injected transport proves the selected request path and the chosen synthetic responses—not account availability, regional routing, redirects, arbitrary provider events, or live task quality. This is new book-owned verification against unchanged `v0.3.0`, not a test retroactively added to the tag.

Reader access. The [reader examples archive](#) provides the complete approved book-owned designs and tests without repository credentials or dependencies. Reference-dependent probes are included, but require an authorized reference checkout; reference source links are not anonymously accessible in this preview. They are not prerequisites for the book-only test command. With such a checkout and Bun 1.3.9, select the immutable tag:

```
git worktree add --detach ../reference-v0.3.0 v0.3.0
cd ../reference-v0.3.0
bun install --frozen-lockfile
bun run check
# From the extracted reader archive, with the worktree path:
bun probes/reference-boundaries.ts /path/to/reference-v0.3.0
```

The test phase is network-denied and credential-free; installation still requires the pinned dependency artifacts to be available locally or from their registry. Provider live tests are separate commands, can incur charges, and are not required to reproduce these deterministic assertions.

A replacement evidence ladder has four distinct gates: **source/contract review**, **deterministic seam tests**, **dated live deployment checks**, and **task-level evaluation**. Keep its language precise: similar API shape is resemblance; passing the application's tested profile is compatibility; official-suite results are conformance evidence; only satisfying the declared formal profile warrants a conformance claim. Increasing evidence at one gate does not promote the others.

5.13. Portability and evidence ledger

Dimension	Chapter claim	Evidence	Not established
Transport	SDK packages own wire translation behind the application port	Tagged factories/common loop; book-owned normal-factory probe with synthetic fetch/SSE	Full wire fidelity, arbitrary gateway replacement, or live availability
Provider	Two constructions use one common loop	v0.4.0 source/tests plus bounded historical v0.3.0 live text reports	Equivalent tool behavior on every deployment
Protocol	Explicit protocol selection differs from endpoint selection	Source-reviewed .responses and .messages; official API references	Open Responses conformance
Feature	Required settings and retained meanings have identifiable owners	Token-policy fixtures, tool outcomes, unknown-usage tests	Complete metadata, citations, reasoning, or option coverage
Semantic	Deployment configuration can change success without changing the core	Retained failures and passes under named configurations	Broad task-quality or safety equivalence

The deployment binder and factory probe are **implemented** and **deterministically tested** book examples. The former makes a selection; the latter tests real tagged construction with synthetic transport. Neither implements deployment authorization or proves live-provider behavior.

5.14. Pattern summary

Problem: Provider replacement can silently change configuration, authority, or meaning while satisfying the same TypeScript interface.

Pattern: Bind trusted deployment configuration at construction, expose a finite application contract, share invariant mechanics, assign translation ownership, and test both retained meanings and known losses.

Result: A provider change becomes a reviewable substitution with explicit proof obligations rather than an undocumented rewrite of the agent.

Cost: The boundary needs version records, deployment-specific checks, loss accounting, and separate operational and semantic evaluation.

5.15. Exercises

1. Add two deployments with the same provider model ID but different adapter instances. Explain which identity must appear in an incident report and which details must remain private.
2. Extend the factory probe with a malformed synthetic stream frame. Assert a bounded failure, then explain why that test still says nothing about account availability.
3. A product now requires citations. Identify the first boundary that currently loses the needed data and decide whether to extend the port or reject the requirement.
4. Specify an output-budget policy that distinguishes visible output from total generation and names who may change each limit. Do not assume a single numeric value works identically across models.
5. Write a loss-register entry for SDK warnings. Define what should cause admission rejection, what may be disclosed to a caller, and what belongs only in protected diagnostics.

5.16. Further reading

Read the provider API references for request semantics (OpenAI, n.d.-c; Anthropic, n.d.-b), then compare the SDK factory and stream contracts (Vercel, n.d.-b, n.d.-a; “AI SDK Core,” n.d.-a) with the pinned reference implementation. The official reasoning guide explains why a visible answer is not the whole generation budget (OpenAI, n.d.-b). RFC 9110 provides the baseline distinction between a repeatable request and an optimistic retry (Fielding et al. 2022). The [dated claim register](#) names the consequential sections and separates mutable documentation from newly executed synthetic paths. These sources constrain the boundary; they do not certify the implementation.

5.17. Chapter contract

- **Scope:** shared model contracts, deployment binding, translation, and replacement evidence.
- **Canonical claims:** provider neutrality is an application-owned boundary, not an equivalence claim.
- **Migration inputs:** former provider-boundary material and Northstar adapter trace.
- **Evidence gates:** strict adapter tests and dated live compatibility evidence.
- **Demonstration slots:** deterministic adapter fixture; no additional public recording required.
- **Source-claim ledger:** current adapter behavior is tied to v0.4.0; dated live reports remain tied to v0.3.0 and their recorded tested commits.

- **Exclusions:** request/item/event mechanics belong to Chapter 6, streaming to Chapter 7, tools to Chapter 8, fallback to Chapter 14, recovery to Chapter 15, telemetry to Chapter 16, and conformance classification to Chapter 19.

6. Open Responses: Requests, Items, Events, and Conformance

Part II · The portable model runtime

Design pin: Open Responses 2026-04-24 · Official suite pin: 92c12d96 · Current classification: not conformant

A completed string is too small a model for an agent response. One response can contain messages, tool calls, structured data, and failures, each of which changes state while it streams. Items preserve meaning; events preserve change.

An **item** is an identifiable unit in a response trajectory. An **event** reports a state transition or incremental update. A **snapshot** is the materialized state after applying an ordered event prefix.

6.1. From chat turns to trajectories

The familiar chat abstraction is an ordered list of role/content messages. It works for simple text, but an agent run needs more structure. A model can emit a message, request a function, receive its result, continue with another message, and terminate incompletely. Flattening that trajectory into text destroys correlation and lifecycle information.

A response-oriented model separates:

- **input items**, supplied by the caller;
- **output items**, produced during the response;
- **semantic events**, emitted while output changes; and
- a **terminal response object**, which summarizes the authoritative outcome.

Open Responses specifies typed items and semantic events, including output-item, content-part, and text-delta lifecycles (“Open Responses Specification 2026-04-24,” n.d.). This chapter uses those concepts as a design target, not as a claim that the current reference implementation is conformant.

6.2. Requests carry policy as well as input

A useful request is more than a prompt:

Architecture sketch — not the exact reference API:

```
interface ModelRequest {  
  model: string;  
  input: InputItem[];  
  tools?: ToolDefinition[];  
  maxOutputTokens?: number;  
  signal?: AbortSignal;  
}
```

The exact production shape will grow, but each field already has policy consequences. The model identifier selects capability and cost. Tool definitions expose authority. The token limit changes incomplete-response behavior. The signal defines cancellation ownership.

Keep a transcript as typed input items rather than concatenating prose. A message, function call, and function result are different facts even if a provider eventually serializes all three into one request body.

6.3. Items preserve identity

A reducer needs enough identity to correlate updates within the response and replay scope. Distinguish item type, item identifier, call identifier where applicable, and ordered indexes without assuming that every identifier is globally stable. Stream events often carry an output index, item identifier, and content index.

Protocol-derived architecture types — not the exact reference API:

```
interface MessageItem {
  id: string;
  type: "message";
  role: "assistant";
  status: "in_progress" | "completed" | "incomplete";
  content: OutputTextPart[];
}

interface FunctionCallItem {
  id: string;
  type: "function_call";
  callId: string;
  name: string;
  arguments: string;
  status: "in_progress" | "completed" | "incomplete";
}
```

These identifiers have different jobs:

- `id` identifies the output item in the response;
- `callId` correlates a function call with its result;
- `outputIndex` places an item in ordered response output; and
- `contentIndex` places a content part inside an item.

Do not substitute array position for identity. Retries, replay, parallel calls, and incremental rendering all need stable references.

6.4. Items are state machines

In the pinned Open Responses specification, generated items move through states such as `in_progress`, `completed`, and `incomplete` (“Open Responses Specification 2026-04-24,” n.d.). A completed item is closed to further mutation. An incomplete item is also terminal, but it tells the application that output ended before normal completion.

A portable reducer should enforce monotonic state:

```

not present —item added—❏ in_progress —done————❏ completed
                    ❏
                    |—interrupted————❏ incomplete

```

Invalid transitions include:

- applying a text delta before its item and content part exist;
- adding content after an item is completed;
- completing the same item twice;
- changing an item identifier at the same output index; and
- accepting a nonterminal update after the response terminates.

Rejecting invalid transitions is not pedantry. It prevents a malformed or duplicated stream from silently corrupting persisted state.

6.5. Semantic events

Transport chunks are not application events. One network chunk can contain several server-sent events, and one JSON event can arrive across several chunks. Decode transport framing first; then process semantic events.

A text item can follow a lifecycle like this:

```

response.output_item.added
  response.content_part.added
    response.output_text.delta (zero or more)
    response.output_text.done
  response.content_part.done
response.output_item.done
response.completed | response.incomplete | response.failed

```

The specification describes state-transition events and delta events and requires monotonically increasing `sequence_number` values in the stream (“Open Responses Specification 2026-04-24,” n.d.). Sequence numbers let a consumer establish order and detect gaps. They do not make arbitrary reordering safe: a missing “item added” event cannot be repaired by blindly applying a later delta.

6.6. A reducer, not a callback pile

Model the consumer as a deterministic reducer. The following application-level implementation is executable, not a transcription of the Open Responses schema. Its complete source is [code/foundations.ts](#), and deterministic tests cover normal completion, replay, sequence gaps, premature deltas, text mismatch, and EOF without a terminal event.

```

export type ResponseEvent =
  | { type: "response-start"; sequence: number; responseId: string }
  | { type: "text-start"; sequence: number; itemId: string; outputIndex: number }
  | { type: "text-delta"; sequence: number; itemId: string; delta: string }
  | { type: "text-done"; sequence: number; itemId: string; text: string }
  | {
      type: "tool-call";
      sequence: number;
      itemId: string;
    }

```

```

        outputIndex: number;
        callId: string;
        name: string;
        input: unknown;
    }
    | {
        type: "response-end";
        sequence: number;
        status: "completed" | "incomplete" | "failed" | "cancelled";
        usage?: Usage;
    };

export type OutputItem =
    | {
        type: "message";
        id: string;
        outputIndex: number;
        status: "in_progress" | "completed";
        text: string;
    }
    | {
        type: "tool-call";
        id: string;
        outputIndex: number;
        status: "completed";
        callId: string;
        name: string;
        input: unknown;
    };

export type ResponseState = {
    readonly responseId?: string;
    readonly lastSequence: number;
    readonly status: "in_progress" | "completed" | "incomplete" | "failed" | "cancelled";
    readonly output: readonly OutputItem[];
    readonly usage?: Usage;
};

export const initialResponseState = (): ResponseState => ({
    lastSequence: 0,
    status: "in_progress",
    output: [],
});

const invariant = (condition: boolean, message: string): void => {
    if (!condition) throw new Error(message);
};

export const applyResponseEvent = (state: ResponseState, event: ResponseEvent):
    ↳ ResponseState => {
    invariant(state.status === "in_progress", "event after terminal response");
    invariant(event.sequence === state.lastSequence + 1, "non-contiguous sequence
        ↳ number");

    switch (event.type) {
        case "response-start":
            invariant(state.responseId === undefined && state.output.length === 0,
                ↳ "duplicate start");

```

```

    return { ...state, responseId: event.responseId, lastSequence: event.sequence };
case "text-start":
  invariant(state.responseId !== undefined, "response has not started");
  invariant(
    !state.output.some(
      (item) => item.id === event.itemId || item.outputIndex === event.outputIndex,
    ),
    "duplicate item identity",
  );
  return {
    ...state,
    lastSequence: event.sequence,
    output: [
      ...state.output,
      {
        type: "message",
        id: event.itemId,
        outputIndex: event.outputIndex,
        status: "in_progress",
        text: "",
      },
    ],
  };
case "text-delta": {
  const item = state.output.find((candidate) => candidate.id === event.itemId);
  invariant(item?.type === "message" && item.status === "in_progress", "invalid
    ↪ delta");
  const message = item as Extract<OutputItem, { type: "message" }>;
  return {
    ...state,
    lastSequence: event.sequence,
    output: state.output.map((candidate) =>
      candidate.id === message.id
        ? { ...message, text: message.text + event.delta }
        : candidate,
    ),
  };
}
case "text-done": {
  const item = state.output.find((candidate) => candidate.id === event.itemId);
  invariant(item?.type === "message" && item.status === "in_progress", "invalid
    ↪ completion");
  const message = item as Extract<OutputItem, { type: "message" }>;
  invariant(message.text === event.text, "final text differs from accepted
    ↪ deltas");
  return {
    ...state,
    lastSequence: event.sequence,
    output: state.output.map((candidate) =>
      candidate.id === message.id ? { ...message, status: "completed" } : candidate,
    ),
  };
}
case "tool-call":
  invariant(state.responseId !== undefined, "response has not started");
  invariant(
    !state.output.some(
      (item) =>

```

```

        item.id === event.itemId ||
        item.outputIndex === event.outputIndex ||
        (item.type === "tool-call" && item.callId === event.callId),
    ),
    "duplicate tool-call identity",
);
return {
  ...state,
  lastSequence: event.sequence,
  output: [
    ...state.output,
    {
      type: "tool-call",
      id: event.itemId,
      outputIndex: event.outputIndex,
      status: "completed",
      callId: event.callId,
      name: event.name,
      input: event.input,
    },
  ],
};
case "response-end":
  invariant(state.responseId !== undefined, "response has not started");
  invariant(
    event.status !== "completed" || !state.output.some((item) => item.status ===
    ↪ "in_progress"),
    "completed response contains an open item",
  );
  return {
    ...state,
    lastSequence: event.sequence,
    status: event.status,
    ...(event.usage === undefined ? {} : { usage: event.usage }),
  };
}
};

export const reduceResponseEvents = (events: Iterable<ResponseEvent>): ResponseState
↪ => {
  let state = initialResponseState();
  for (const event of events) state = applyResponseEvent(state, event);
  invariant(state.status !== "in_progress", "stream ended without a terminal event");
  return state;
};

```

Pure reduction has practical benefits:

- the same fixture always produces the same snapshot;
- replay can reconstruct state without rerunning tools;
- property tests can generate malformed orders;
- UI projection can be tested separately; and
- persistence can checkpoint an event prefix.

An implementation may mutate an internal builder for performance, but it should preserve reducer semantics at its public boundary.

6.7. Terminal states

A stream ending is not the same as a response completing. EOF can result from success, cancellation, network loss, proxy timeout, or a crashed server. Require an explicit terminal outcome.

A useful portable terminal model distinguishes:

Terminal state	Meaning	May partial output be shown?	May it be treated as final?
completed	Normal terminal response	Yes	Yes
incomplete	Model stopped before full completion	Yes, labeled	No, unless policy accepts it
failed	Provider or protocol failure	Possibly, labeled	No
cancelled	Caller or system stopped work	Possibly, labeled	No

cancelled is an application-level normalization in this chapter, not a claim that every provider or Open Responses uses the same terminal representation. Preserve the raw reason in protected diagnostics, not in a public error body.

6.8. Snapshot consistency

A terminal snapshot should agree with the event history. At completion:

- every completed text part equals the concatenation of its accepted deltas or its authoritative done value;
- every item occupies one output index;
- each function result refers to a known call identifier;
- no item remains `in_progress` after a completed response;
- terminal status and finish reason agree with policy; and
- unknown usage remains explicitly unknown rather than being converted to zero. Both the tested chapter reducer and the v0.4.0 common loop preserve absent input/output token counts; measured zero remains distinct from absence.

The terminal response object may contain a more authoritative final representation than accumulated deltas. Compare the two. A mismatch is a protocol or adapter defect, not an invitation to silently select whichever value is convenient.

i Events are evidence; snapshots are state

Persisting only the snapshot is compact. Persisting an approved event log supports replay and diagnosis. If events contain sensitive content, retention and redaction rules apply to both forms.

6.9. Unknown and extension events

Evolving protocols add item and event types. A portable consumer needs an explicit compatibility policy:

- reject unknown events in strict conformance mode;
- retain and forward namespaced extensions when the application supports them;
- ignore only events documented as nonessential; and
- emit telemetry when an unknown type is observed.

Never coerce an unknown item into text. That can turn a control or safety signal into user-visible content while losing its semantics.

6.10. SDK events and the HTTP protocol are different contracts

The immutable v0.4.0 model loop consumes normalized AI SDK stream parts rather than raw Open Responses events. It maps text, tool calls, explicit tool-result outcomes, finish reasons, optional usage, response identifiers, and sanitized errors. Provider packages own provider-wire translation.

The release's [application reducer](#), the shared model stream, and the current Open Responses HTTP bridge are three separate APIs. The reducer checks item identity, contiguous local sequence, closed-item updates, final-text agreement, and terminal presence. The HTTP bridge translates the application's supported profile into Open Responses-shaped JSON/SSE resources. None is automatically a field-for-field implementation of every specification item and event.

Concept	Specification target	Local application/runtime	Current HTTP profile
Request input	Typed ordered items/content	Typed messages, history, tools, and ordered admitted content	Strict supported subset; unknown supported-object fields and unknown parts reject
Item identity	Output item IDs, call IDs, indexes	Correlated calls/results and checked reducer identity	Stable response/items for implemented text/function paths
Ordering	Monotonic stream sequence	SDK order plus local reducer sequence	Contiguous emitted sequence for the implemented SSE profile
Terminal outcome	Completed, incomplete, or failed response resources/events	Normalized finish/error/cancellation semantics	Schema-shaped terminal resources for supported paths
Usage	Optional measured details	Unknown totals remain absent	null when required totals are unavailable; no invented top-level count

Correspondence establishes an implementation approach, not conformance.

6.11. Four different status statements

Use four labels precisely:

1. **API resemblance:** a route or JSON object looks like `/v1/responses`.
2. **Implemented compatible subset:** named clients and fixtures work within a documented local profile.
3. **Official-suite result:** a pinned runner reports exact passes, failures, and skips against an exact target.
4. **Formal conformance:** the implementation satisfies the declared required profile; this status is reserved until the required suite and lifecycle behavior pass.

The current server has a useful HTTP/JSON/SSE subset and a Pi 1.0.4-compatible path. The opt-in wrapper runs the official CLI from immutable revision `92c12d96d7b61d6d15e2214daa5e9c6000ab6e1c` and preserves its failing exit status. The latest reviewed complete run reports **8 passed, 9 failed, 0 skipped** overall; seven passes and all nine failures contacted the HTTP target.

The remaining failures are three compaction cases and six WebSocket lifecycle/state cases. `/v1/responses/compact` is absent, and the required WebSocket chains, continuation, reconnect recovery, missing-state behavior, and failed-chain eviction are not implemented. Therefore the classification is **compatible subset; not conformant**.

Security limits and strict local input rules are documented restrictions. They must not be weakened merely to increase a suite score. Conversely, passing only schema examples would not establish compaction quality, tenant isolation, durable state, concurrency bounds, production WebSocket hardening, or semantic equivalence. Chapter 19 owns the final dated report; this chapter owns the protocol vocabulary used to interpret it.

6.12. Security considerations

Items can contain untrusted text, URLs, citations, tool arguments, and opaque provider metadata. Treat every field as data:

- render text with output encoding appropriate to its destination;
- validate tool arguments before execution;
- constrain extension sizes and nesting;
- do not deserialize provider metadata into executable classes;
- bound retained events and snapshots; and
- avoid exposing raw upstream failures to callers.

Stable item identity also supports auditability: a policy decision can refer to the exact call, result, or message it evaluated.

6.13. How we test the model

A deterministic item/event test suite should cover:

1. one message with several text deltas;
2. two output items with distinct indexes;
3. a function call and correlated result;
4. a gap or duplicate sequence number;
5. a delta before item creation;
6. completion followed by another event;
7. incomplete and failed terminal states; and
8. an unknown extension under strict and permissive policies.

Protocol conformance tests should remain separate from reducer unit tests. The first proves local invariants; the second proves behavior against a named external specification revision.

6.14. Portability and evidence ledger

Dimension	Chapter claim	Evidence	Not established
Transport	Semantic reduction is independent of byte framing	Book and v0.4.0 reducers accept application events	Complete adversarial HTTP/SSE parsing coverage
Provider	One application event model preserves selected meanings	v0.4.0 normalized calls, results, finish, and usage	Complete provider item fidelity
Protocol	The target derives from Open Responses 2026-04-24	Pinned specification and correspondence table	Conformance
Feature	Text and function-call identity survive reduction	Book and tagged reducer tests for identity, sequence, and terminal consistency	Citations, reasoning, images, and extensions
Semantic	Replaying events preserves state, not answer quality	Replay equality test	Cross-model answer equivalence

6.15. Pattern summary

Problem: Flattening a response to text loses identity, lifecycle, tool correlation, and terminal meaning.

Pattern: Represent response output as typed stateful items, apply ordered semantic events through a checked reducer, and reconcile the terminal snapshot.

Result: Streaming, replay, persistence, tools, and diagnostics can share one coherent model.

Cost: The implementation must enforce lifecycle invariants and evolve with new item and event types.

6.16. Exercises

1. Specify whether duplicate sequence numbers are ignored, rejected, or deduplicated, and justify the replay consequences.
2. Add a citation content part without flattening it into text.
3. Define how a cancelled response differs from an incomplete model response in your application.
4. Write an invariant that detects a function result with no matching call.
5. Decide what your strict and permissive modes do with a namespaced extension event.

6.17. Further reading

Read the pinned Open Responses specification's sections on semantic events, item state machines, streaming, and extensions ("Open Responses Specification 2026-04-24," n.d.), then compare the generated field definitions in its reference ("Open Responses Reference 2026-04-24," n.d.). Keep architectural use separate from a formal conformance claim.

6.18. Chapter contract

- **Scope:** Open Responses request, item, event, lifecycle, and conformance vocabulary.
- **Canonical claims:** reducers require correlated items and exactly one terminal interpretation.
- **Migration inputs:** former responses-as-items-and-events chapter.
- **Evidence gates:** schema fixtures, reducer tests, and official conformance output.
- **Demonstration slots:** deterministic event trace.
- **Source-claim ledger:** design uses the 2026-04-24 protocol artifact; official results use suite revision 92c12d96; adapter reduction uses v0.4.0; each remains a separate claim.
- **Exclusions:** provider translation belongs to Chapter 5, transport backpressure to Chapter 7, tool authority to Chapter 8, durable continuation to Chapter 15, and final release classification to Chapter 19.

7. Streaming, State, and Termination

Part II · The portable model runtime

Streaming sources checked: 2026-10-07

Streaming reduces time to first output, but it turns one response into a state machine whose transitions cross transport, provider, adapter, application, and consumer boundaries. The consumer must preserve order, correlate fragments, propagate cancellation, bound memory, and distinguish a useful partial result from a completed answer.

A **correct stream consumer** produces the same logical terminal state as the corresponding non-streaming response, or reports why it cannot. Rendering early is optional; preserving lifecycle and outcome is not.

7.1. A stream has three layers

Do not conflate these layers:

1. **Transport bytes** arrive in network chunks.
2. **Protocol frames** delimit messages, such as server-sent event data records.
3. **Semantic events** describe text deltas, item changes, tool calls, usage, errors, and terminal state.

Byte boundaries are arbitrary. UTF-8 code points, lines, and JSON values can cross them. A transport parser must buffer incomplete framing before JSON decoding. The semantic reducer from Chapter 6 should never inspect raw chunks.

The Open Responses HTTP stream uses server-sent events carrying JSON event objects and a terminal [DONE] marker (“Open Responses Specification 2026-04-24,” n.d.). That marker closes framing; the semantic terminal event determines whether the response completed, failed, or remained incomplete.

7.2. Stream all meanings, not only text

A text-only stream is tempting. This is an **API-usage fragment**, not a complete consumer:

```
for await (const delta of result.textStream) {  
  render(delta);  
}
```

It is insufficient for an agent loop because it can hide tool calls, tool errors, usage, finish reasons, and abort events. The AI SDK distinguishes a text stream from its complete event stream; errors and nontext parts require the richer stream or callbacks (“AI SDK Core,” n.d.-a).

At the application boundary, use a discriminated union. This is an **architecture sketch**, not the exact v0.4.0 event type:

```

type ModelEvent =
  | { type: "response-start"; responseId?: string }
  | { type: "text-delta"; delta: string }
  | { type: "tool-call"; callId: string; name: string; input: unknown }
  | { type: "tool-result"; callId: string; name: string; output: unknown }
  | { type: "usage"; inputTokens?: number; outputTokens?: number }
  | { type: "finish"; reason: FinishReason }
  | { type: "error"; error: PublicModelError };

```

This proposed union is deliberately smaller than Open Responses. The exact v0.4.0 model event union exposes text deltas, tool calls, correlated tool-result outcomes, and finish events carrying usage and an optional response identifier; failures are thrown as sanitized errors. It does not expose separate response-start, usage, or error events. The sketch separates those meanings for discussion, not as a claim about the release API or protocol conformance.

7.3. Ordered reduction

Process semantic events serially unless the protocol explicitly permits independent partitions. A simple **control-flow sketch** provides an important invariant: event n finishes reduction before event $n + 1$ mutates state.

```

for await (const event of adapter.stream(request)) {
  state = applyModelEvent(state, event);
  await publishProjection(state, event);
}

```

`publishProjection` may update a UI or send a downstream event, but it should not mutate canonical state. If publishing is slow, decide whether to block upstream consumption, buffer within limits, or decouple through a bounded queue.

Parallelizing arbitrary event handlers creates races such as a `tool-result` being rendered before its `tool-call`, or a terminal snapshot being stored before the final text delta.

7.4. Backpressure is an end-to-end policy

The Web Streams model includes backpressure: a producer can observe downstream demand rather than pushing without limit (“Streams Standard,” n.d.). But application pipelines often break that chain by reading as fast as possible into an unbounded array or message queue.

Choose explicit limits for:

- unread transport bytes;
- decoded but unapplied events;
- accumulated text and tool arguments;
- pending UI or WebSocket messages;
- concurrent tool calls; and
- persisted event volume per run.

When a limit is reached, the options are to pause, shed nonessential projections, cancel, or fail. “Keep buffering” is not a safe default.

The chapter's executable bounded consumer blocks its producer at a configured event capacity, reduces events serially, awaits a deliberately slow projection, and keeps cancellation observable while either side waits. The complete source is [code/foundations.ts](#); tests assert ordered delivery, a peak queue size no greater than the configured bound, and cancellation while waiting for capacity.

```
export const consumeBounded = async <Event, State>(options: {
  source: AsyncIterable<Event>;
  maximumBufferedEvents: number;
  initialState: State;
  reduce: (state: State, event: Event) => State;
  publish: (state: State, event: Event) => void | Promise<void>;
  signal?: AbortSignal;
}): Promise<{ state: State; peakBuffered: number }> => {
  invariant(options.maximumBufferedEvents > 0, "buffer limit must be positive");
  const queue: Event[] = [];
  let peakBuffered = 0;
  let done = false;
  let failure: unknown;
  let wakeConsumer: (() => void) | undefined;
  let wakeProducer: (() => void) | undefined;

  const wait = (role: "consumer" | "producer"): Promise<void> => {
    if (options.signal?.aborted) return Promise.reject(options.signal.reason);
    return new Promise((resolve, reject) => {
      const abort = (): void => reject(options.signal?.reason);
      const wake = (): void => {
        options.signal?.removeEventListener("abort", abort);
        resolve();
      };
      options.signal?.addEventListener("abort", abort, { once: true });
      if (role === "consumer") wakeConsumer = wake;
      else wakeProducer = wake;
    });
  };

  const producer = (async (): Promise<void> => {
    try {
      for await (const event of options.source) {
        while (queue.length >= options.maximumBufferedEvents) await wait("producer");
        queue.push(event);
        peakBuffered = Math.max(peakBuffered, queue.length);
        wakeConsumer?.();
        wakeConsumer = undefined;
      }
    } catch (error) {
      failure = error;
    } finally {
      done = true;
      wakeConsumer?.();
    }
  })();

  let state = options.initialState;
  while (!done || queue.length > 0) {
    if (options.signal?.aborted) throw options.signal.reason;
    if (queue.length === 0) {
```

```

    await wait("consumer");
    continue;
  }
  const event = queue.shift() as Event;
  wakeProducer?.();
  wakeProducer = undefined;
  state = options.reduce(state, event);
  await options.publish(state, event);
}
await producer;
if (failure !== undefined) throw failure;
return { state, peakBuffered };
};

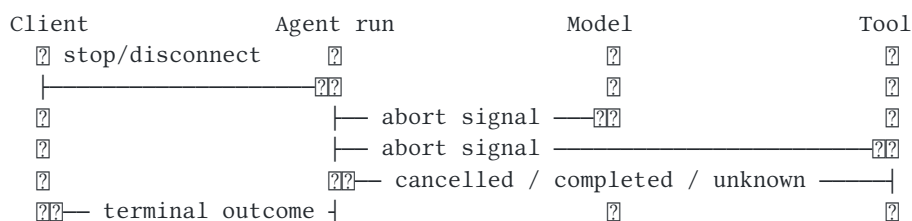
```

⚠ Slow consumers are a correctness condition

A client that reads slowly can turn a healthy model request into memory pressure. Test a deliberately slow sink and prove that buffering stays bounded or cancellation propagates.

7.5. Cancellation has a direction

Cancellation begins with a principal: a disconnected client, a user pressing stop, a deadline, a budget guard, or an operator. Its direction is explicit:



Propagate one signal through every cancellable operation. This **SDK call fragment** omits surrounding construction:

```

const result = streamText({
  model,
  messages,
  abortSignal: request.signal,
  tools,
});

```

The AI SDK accepts an `AbortSignal` for the overall operation and provides abort lifecycle handling (“AI SDK Core,” n.d.-a). Passing the signal only to the model is incomplete. Tool execution, sleeps, retrieval calls, subprocesses, and queue waits must also observe it.

Cancellation does not automatically roll back side effects. If a tool already created a record, stopping the model cannot uncreate it. Tools need idempotency keys, transactions, compensating actions, or a clear “outcome unknown” state.

7.5.1. Cancellation is not failure

Normalize cancellation separately from provider failure. This supports accurate telemetry and retry policy:

- a user cancellation should usually not page an operator;
- a deadline may indicate capacity or policy configuration;
- a provider timeout may be retryable;
- a tool outcome after disconnect may require reconciliation.

Cleanup can need an independent, short timeout. Reusing an already-aborted signal for mandatory cleanup guarantees that cleanup never starts.

7.6. Partial output is provisional

Streaming output may be useful before completion, but it is not yet an authoritative final answer. The following **proposed projection type** labels partial state explicitly:

```
type Projection = {
  text: string;
  provisional: boolean;
  outcome: "running" | "completed" | "incomplete" | "failed" | "cancelled";
};
```

A UI can display provisional text while preventing downstream automation from treating it as final. On failure or cancellation, policy determines whether to retain, hide, or annotate it. The choice should differ by use case: partial creative text may be useful; a partial JSON object or transaction instruction may be unsafe.

Never parse a structured result from an arbitrary text prefix and execute it. Wait for the structured item or terminal validation.

7.7. Snapshot consistency

The reducer should be able to create a snapshot after any accepted prefix, but only a terminal event can mark it final. At termination, verify:

1. all referenced items and calls exist;
2. no text or argument buffer remains syntactically unfinished;
3. terminal status is unique;
4. finish reason is normalized without discarding protected raw diagnostics;
5. input/output usage counts preserve the distinction between unknown and measured zero, as the chapter reducer and v0.4.0 common loop do; and
6. any authoritative final object agrees with accumulated output.

For a completed response, a mismatch between accumulated deltas and final text should fail a consistency check. For an interrupted transport with no semantic terminal event, report an indeterminate or failed outcome rather than manufacturing completed.

7.8. Disconnect is a stream-local signal

Client disconnect, model cancellation, and durable resumption are separate features. This chapter owns the first two only: the handler must define whether reader cancellation/request abort propagates to active model and tool work, release stream-local buffers, and report the most accurate terminal knowledge available to code that can still receive it.

A cancelled connection cannot receive a terminal event on that same connection. It also cannot prove that a remote effect did not occur. Durable run identity, retained event replay, reconnect authorization, unknown-effect reconciliation, and restart policy belong to Chapter 15. Until those contracts exist, advertise a new attempt rather than resumability.

7.9. Errors before and during streaming

Errors can occur:

- before headers, when a normal HTTP error is still possible;
- after streaming begins, when the protocol needs an error event;
- inside tool execution, where a correlated tool error may be model-visible; or
- after a side effect, where the operational outcome is uncertain.

Public errors should carry a stable category, safe message, retry hint, and correlation identifier. Raw provider bodies, headers, prompts, and tool arguments belong only in protected diagnostics under a redaction policy.

The v0.4.0 reference loop disables accidental SDK error logging and maps provider failures into a sanitized `AdapterError`. This prevents raw provider details from reaching callers or default logs. That release does not provide a protected diagnostic store, so correlation and authorized diagnostic capture remain unimplemented operational requirements.

7.10. Retry is outside the stream reducer

A missing terminal event does not authorize another attempt. The stream layer reports what became visible, which calls started, the last accepted sequence, and why terminal meaning is absent. Chapter 15 decides whether that history permits replacement, requires reconciliation, or must stop. Keeping retry policy outside the reducer prevents a transport interruption from silently duplicating tool effects or user-visible text.

7.11. Deterministic streaming tests

A network-free suite should vary chunking independently from semantic events. Feed the same event sequence as:

- one transport chunk;
- one byte at a time;
- boundaries inside UTF-8 and JSON tokens;
- several events in one chunk; and
- a final truncated frame.

Then vary semantic behavior:

- normal text completion;
- tool call and result;
- duplicate or missing event;
- abort between deltas;
- slow downstream consumer;
- error after partial text; and
- EOF without a terminal event.

The immutable `v0.4.0` reference uses AI SDK mock language models for common-loop tests and separate fixtures for its bounded consumer and deterministic HTTP/SSE endpoint. Both the reference and chapter bounded-consumer tests establish local slow-sink queue bounds and queue-wait cancellation, not network-byte limits. The tag retains a basic Pi 1.0.4 text-streaming acceptance record at tested commit `0d54a3e`; extended tool, session, usage-display, and cancellation acceptance belongs to post-release work. Neither client acceptance nor complete-response SSE checks replace adversarial transport-chunking tests.

7.11.1. What the later live bridge actually proves

At the [current reviewed snapshot](#), cancelling the response body's reader aborts the signal supplied to a scripted provider. The handler also forwards an incoming request abort and removes its listener when work ends. This is deterministic handler-level cancellation evidence—not proof that every network disconnect, provider, or side effect stops correctly. A cancelled reader cannot receive a terminal event on that cancelled stream; cleanup and reconciliation must not depend on delivering one.

The live bridge's producer eagerly enqueues SSE records from `start()` without waiting for downstream demand. It does not use the bounded consumer. Its output-token ceiling limits requested generation, not unread transport bytes, event count, or queue size. Collecting text chunks and joining once avoids repeated concatenation, but neither that optimization nor immediate forwarding establishes bounded memory or end-to-end backpressure. Slow-reader HTTP tests and explicit queue admission remain necessary.

The lesson is to keep proofs at their tested boundary: local queue bounds, response-reader cancellation, client compatibility, and network backpressure are four different claims.

7.12. Portability and evidence ledger

Dimension	Chapter claim	Evidence	Not established
Transport	Bytes, framing, and semantic events are separate layers	Tagged deterministic HTTP/SSE checks and basic Pi text acceptance	Full adversarial byte-framing coverage
Provider	Common semantic consumption need not branch on provider event names	<code>v0.4.0</code> injected models and retained dated text-generation checks	All provider events or model configurations

Dimension	Chapter claim	Evidence	Not established
Protocol	Explicit terminal meaning matters beyond transport EOF	Open Responses target and chapter reducer	Conformance or resumable protocol
Feature	Bounded buffering and cancellation are implementable application policies	Book and v0.4.0 local slow-sink and queue-wait abort tests	End-to-end network backpressure
Semantic	Partial output is provisional	State model only	Equivalent final answers

7.13. Pattern summary

Problem: Streaming improves responsiveness but can lose order, lifecycle, cancellation, and terminal truth.

Pattern: Separate transport parsing from semantic reduction, consume the rich event stream serially, propagate cancellation, bound buffering, and reconcile a terminal snapshot.

Result: Early output remains responsive without becoming indistinguishable from completed state.

Cost: The implementation needs explicit state, limits, cleanup policy, and adversarial stream tests.

7.14. Exercises

1. Define a maximum buffer policy for a slow WebSocket client.
2. Decide whether your server cancels model work when an HTTP client disconnects.
3. Write a test where EOF occurs after text but before a terminal event.
4. Identify a tool side effect that requires reconciliation after cancellation.
5. Explain which events your UI may drop under pressure and which it must preserve.

7.15. Further reading

Compare the Open Responses streaming and semantic-event rules (“Open Responses Specification 2026-04-24,” n.d.) with the Web Streams backpressure model (“Streams Standard,” n.d.). For the reference implementation, consult the pinned AI SDK `streamText` contract for complete-stream, error, usage, finish, and cancellation behavior (“AI SDK Core,” n.d.-a).

7.16. Chapter contract

- **Scope:** ordered stream reduction, bounded consumption, cancellation, and termination.
- **Canonical claims:** a stream is a state transition sequence, not a text iterator.
- **Migration inputs:** former streaming chapter plus termination behavior.
- **Evidence gates:** ordering, limits, malformed events, cancellation, and terminal-state tests.

- **Demonstration slots:** deterministic scripted stream.
- **Source-claim ledger:** local reducer/consumer evidence is tied to v0.4.0 and book tests; handler cancellation evidence is tied to snapshot 45a903c; neither is end-to-end network proof.
- **Exclusions:** request/item vocabulary belongs to Chapter 6, tool authorization to Chapter 8, retries and durable cross-request state to Chapter 15, and OpenTelemetry to Chapter 16.

8. The Typed and Authorized Tool Loop

Part II · The portable model runtime

Tool and JSON Schema sources checked: 2026-10-07

A tool call crosses from probabilistic generation into software authority. Treat the model’s proposed name and arguments as untrusted input, validate them against a pinned schema, authorize the action, execute under limits, and correlate exactly one result with the call.

A **typed tool loop** is a bounded state machine that advertises tool contracts, receives proposed calls, validates and authorizes them, executes approved implementations, records correlated results, and gives those results back to the model for the next step.

8.1. The loop

A complete tool step has more stages than “call a function”:

```
advertise schemas
↓
model proposes call(name, callId, input)
↓
resolve name → validate input → authorize → apply limits
↓
execute implementation → validate/encode output
↓
record result(callId, name, output)
↓
continue model generation or stop
```

Every arrow is a failure and security boundary. The model can choose an unknown name, produce malformed input, repeat a call, or request an action outside the user’s authority. The implementation can time out, return invalid data, or complete a side effect after cancellation.

8.2. Define one portable tool contract

A neutral contract can use JSON Schema directly. The following is an **architecture sketch**, not the full v0.4.0 type. That release also carries trusted execution context, authorization and approval hooks, deadlines, and output-byte limits:

```
interface ToolDefinition {
    name: string;
    description: string;
    inputSchema: Record<string, unknown>;
    outputSchema?: Record<string, unknown>;
    execute?: (
        input: unknown,
        context: { abortSignal?: AbortSignal },
    )
```

```
) => unknown | Promise<unknown>;
}
```

Use JSON Schema Draft 2020-12 as the project dialect and declare it when schemas leave the process (“JSON Schema Draft 2020-12,” n.d.). Do not assume a provider supports every keyword. The provider-facing adapter may need a documented compatible subset, while the canonical application schema remains unchanged.

A small schema should be precise:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "prototype": { "enum": ["A", "B"] },
    "metric": { "const": "runtime" },
    "unit": { "const": "hours" }
  },
  "required": ["prototype", "metric", "unit"],
  "additionalProperties": false
}
```

Descriptions help a model select and fill a tool, but validation keywords enforce the machine contract. Test the validator configuration: in JSON Schema, format is annotation by default unless format assertion behavior is enabled (“JSON Schema Draft 2020-12,” n.d.).

8.3. Schema is not TypeScript

A TypeScript type disappears at runtime. An assertion such as `input as MeasurementInput` proves nothing about provider data. Runtime validation must occur before execution, and the validated value—not the original object—should reach the implementation.

Conversely, a runtime schema does not automatically give every TypeScript call site a useful static type. Schema-derived types or typed schema libraries can reduce drift, but generated types must be reproducible and checked in CI.

The AI SDK accepts Zod or JSON Schema input definitions, but its low-level `jsonSchema` helper validates values only when supplied with a validation function (“AI SDK Core,” n.d.-b). Release v0.4.0 supplies that callback using pinned Ajv 8.17.1 and a documented Draft 2020-12 cross-draft keyword profile. Its `invalid-generated-input` fixture asserts zero executions and a correlated failed outcome. The tested chapter runtime makes the validation step explicit without depending on the SDK. Neither a TypeScript cast nor an SDK wrapper alone proves dialect support.

8.4. Validation is still not authorization

A valid request can be forbidden. The following **policy-order sketch** evaluates authorization with trusted context after structural validation:

```
const input = validate(tool.inputSchema, proposedInput);

await authorize({
  principal: run.principal,
  action: `tool:${tool.name}`,
});
```

```

    resource: resourceFrom(input),
    tenant: run.tenant,
  });

const output = await tool.execute(input, { abortSignal: run.signal });

```

Do not let the model supply the principal, tenant, approval result, or policy scope as ordinary tool arguments. Those values come from authenticated application context.

For high-impact tools, insert human approval after validation and policy evaluation but before execution. Bind approval to a digest of the exact tool name, normalized arguments, principal, and expiry; otherwise the approved call can be swapped.

The measurement tool is read-only, but it still has a threat model: the model can invent a prototype field, a caller can cross a tenant boundary, a connector can return adversarial text, duplicate call IDs can trigger repeated cost, and cancellation can leave the measurement's status uncertain. Its trusted request context is never accepted from generated arguments.

8.5. Correlation is mandatory

A tool name is not enough to match calls and results. The same tool can be called twice or in parallel. These **architecture types** carry an opaque callId from proposal through result:

```

type ToolCall = {
  callId: string;
  name: string;
  input: unknown;
};

type ToolResult = {
  callId: string;
  name: string;
  output: unknown;
};

```

The tested chapter runtime below receives trusted request context, validates before authorization, executes once per call ID, validates output, and returns one explicit terminal outcome. It is an application policy example, not an Open Responses or MCP implementation. Its complete source is [code/foundations.ts](#).

```

export type ToolContext = Readonly<{
  requestId: string;
  principal: string;
  tenant?: string;
  callId: string;
  signal?: AbortSignal;
}>;

export type ToolContract<Input, Output> = {
  name: string;
  validateInput: (value: unknown) => Input;
  authorize: (input: Input, context: ToolContext) => void | Promise<void>;
  approve?: (input: Input, context: ToolContext) => void | Promise<void>;
  execute: (input: Input, context: ToolContext) => Output | Promise<Output>;
  validateOutput: (value: unknown) => Output;
};

```

```

};

export type ToolOutcome<Output> =
  | { status: "completed"; output: Output }
  | { status: "failed" | "cancelled" | "outcome_unknown"; code: string };

export class ToolRuntime {
  readonly #run: Omit<ToolContext, "callId">;
  readonly #calls = new Set<string>();

  constructor(run: Omit<ToolContext, "callId">) {
    this.#run = Object.freeze({ ...run });
  }

  async invoke<Input, Output>(
    tool: ToolContract<Input, Output>,
    callId: string,
    proposedInput: unknown,
  ): Promise<ToolOutcome<Output>> {
    if (this.#calls.has(callId)) return { status: "failed", code: "duplicate_call_id" };
    this.#calls.add(callId);
    if (this.#run.signal?.aborted) return { status: "cancelled", code: "cancelled" };

    const context = Object.freeze({ ...this.#run, callId });
    try {
      const input = tool.validateInput(proposedInput);
      await tool.authorize(input, context);
      await tool.approve?.(input, context);
      const output = await tool.execute(input, context);
      return { status: "completed", output: tool.validateOutput(output) };
    } catch {
      return this.#run.signal?.aborted
        ? { status: "outcome_unknown", code: "cancelled_after_start" }
        : { status: "failed", code: "rejected_or_failed" };
    }
  }
}

```

Check that:

- the result references a known pending call;
- the name agrees with the original call;
- one call does not receive two terminal results;
- a result is recorded before the next model step consumes it; and
- replay does not execute an already completed call again.

Provider item identifiers and tool-call identifiers may be different. Preserve both when needed rather than inventing equality.

8.6. Streamed arguments

Some provider protocols stream function arguments as string fragments; SDKs may expose only a validated final input. If your boundary receives fragments, accumulate them by call identifier

under byte and time limits. Do not execute a prefix that happens to parse as JSON. Wait for the protocol's argument-done or tool-call event, parse exactly once, then validate.

Malformed arguments are a model/tool-call error, not an invitation to guess. A repair mechanism, if enabled, is another bounded model step with its own telemetry and disclosure policy.

The application contract should normally emit a complete `tool-call` event. Raw argument deltas belong only in a lower-level protocol lesson or diagnostic stream.

8.7. Execute under limits

Each tool needs policy beyond its schema:

Limit	Purpose
Timeout/deadline	Bound downstream latency
Cancellation signal	Stop cooperative work when the run ends
Concurrency	Protect local and remote resources
Input/output bytes	Prevent memory and context exhaustion
Rate or cost budget	Bound financial and service impact
Network/filesystem sandbox	Limit reachable authority
Idempotency key	Make retries and replay safer

The tool description shown to the model should explain function, not disclose credentials, internal topology, or policy bypasses.

8.8. Return structured results

A tool result is data for the model and possibly the user. Prefer a bounded object with explicit fields over an unstructured log dump. Validate an output schema when downstream code depends on the shape.

Tool output remains untrusted even when it comes from your own connector. A remote page, database row, or MCP server can contain instructions aimed at the model. Label provenance and keep data distinct from system instructions.

Structured model output is related but separate. A response schema constrains the model's final generated object; a tool output schema constrains software returned by a tool. Neither guarantees factual correctness or authorization.

8.9. Continue or stop

After tool results are added to the transcript, the model can produce text or request another tool. The loop needs a hard stop condition. Release v0.4.0 uses the AI SDK's step-count stop condition: four steps by default when executable tools are present, otherwise one, unless `maxSteps` is supplied ("AI SDK Core," n.d.-a).

A production policy can combine:

- maximum model steps;

- maximum total tool calls;
- per-tool call limits;
- elapsed deadline;
- token and monetary budgets;
- repeated-call detection; and
- an application-specific terminal tool or result.

Count completed steps precisely and define whether the initial model call is step one. Ambiguous counters create off-by-one safety defects.

A loop limit is a safety control

A model can repeatedly request a harmless tool and still exhaust tokens, money, or downstream capacity. Enforce limits outside the model and report which guard ended the run.

8.10. Tool errors

Distinguish at least:

- unknown tool;
- invalid arguments;
- unauthorized action;
- approval denied or expired;
- timeout or cancellation;
- execution failure;
- invalid tool output; and
- budget exhausted.

Decide which errors are safe to return to the model. A concise “resource unavailable” result may help it recover; a database stack trace or authorization internals may disclose sensitive information. Public application errors and protected operator diagnostics should be separate objects linked by a correlation identifier.

Do not automatically retry non-idempotent tools. For an ambiguous timeout, the correct result may be “outcome unknown; reconcile” rather than “failed.”

8.11. Parallel tool calls

Parallel execution can reduce latency when calls are independent. It also complicates budgets, ordering, cancellation, and side effects. Before enabling it, define:

- whether calls commute;
- a maximum concurrency;
- deterministic result ordering;
- whether one failure cancels siblings;
- how partial success is represented; and
- whether the next model step waits for every result.

Sequential execution is a reasonable portable default. Parallelism is a capability and policy choice, not a free optimization.

8.12. The shared SDK-backed loop

The immutable `v0.4.0` reference constructs dynamic tools from neutral definitions and supplies them to one `streamText` call. The shared layer registers explicit input/output validators, delegates executable tools to a request-scoped `ToolPolicyRuntime`, propagates the SDK abort signal, and continues across bounded model steps. It emits normalized tool calls, terminal tool-result outcomes, and finish events with optional token counts, and sanitizes surfaced errors. Supplying authorization or approval hooks requires trusted `toolContext` before model execution; the reference does not authenticate that context itself.

The tagged [adapter tests](#) exercise this successful sequence:

1. the model emits a tool call;
2. the SDK parses the arguments and the configured validator accepts them;
3. the runtime applies configured policy hooks and executes the implementation;
4. a completed tool-result outcome is correlated with the call;
5. the shared loop starts the next model step; and
6. final text and aggregate usage reach the same application contract.

The successful fixture does not configure every policy hook. Separate adapter tests reject invalid generated input before execution, report invalid output as a correlated failure, and reject policy-bearing definitions without trusted context before calling the model. Tagged [runtime tests](#) cover authorization/approval ordering, duplicate-call rejection, pre-execution cancellation, execution timeout, invalid output, and registration checks. These are bounded deterministic proofs, not an authentication service, cryptographically bound approvals, parallel-call safety, or side-effect reconciliation. The retained live text-generation checks do not establish live tool-policy behavior.

8.12.1. Tool ownership across the live client/server boundary

The current reviewed live profile disables Pi-local executable tools and registers five server-owned research tools selected by `agent.yaml`: `image_understanding`, `web_search`, `read_source`, `code_interpreter`, and `presentation_builder`. A sixth server-owned operation, `load_skill`, admits reviewed non-executable procedure; it does not become an executable capability.

The server owns each selected tool's schema, policy, progress summaries, request-scoped quota, deadline, execution, and terminal result. Pi renders progress and can retrieve a presentation artifact, but it does not execute these research calls. This differs from a generic Responses client that advertises its own function schemas and later submits outputs; authority belongs to the component that actually executes the operation.

Model-generated fields cannot choose provider credentials, endpoint, arbitrary path, package, executable, or presentation code. Tools that need request assets accept invocation-local source IDs. Current configuration selection still is not principal authentication: the loopback development server is unauthenticated, and production identity/policy remain Chapter 17 gates.

The coordinator output-token ceiling is separate from tool call counts, source budgets, worker deadlines, mounted bytes, result-size limits, and artifact TTL. One configured number never implies complete resource control.

8.12.2. User-visible progress

A long tool can emit bounded progress without exposing its raw arguments or result. The current protocol associates progress with the response and call, uses fixed safe labels/summaries, and lets Pi render expandable cards outside model context. Progress must be monotonic and terminally consistent with the call result.

Progress is not authority, evidence of semantic success, or operator telemetry. Rendering a card cannot execute, retry, approve, or cancel a tool. Sensitive queries, document bodies, image bytes, source IDs, Python code/output, credentials, and raw provider errors stay out of ordinary progress. Chapter 16 separately owns OpenTelemetry once implemented.

8.12.3. Implemented policy and remaining boundary

Control	Reviewed evidence	Remaining boundary
Input validation	v0.4.0 schema callback plus current strict tool-specific schemas	Full provider/dialect parity
Trusted run context	Required for policy hooks; request/call identity forwarded	Production authentication and identity verification
Authorization/approval	Hooks run before execution; denial/order fixtures	Production policy service and cryptographically bound approval
Limits	Steps, per-tool calls, deadlines, cancellation, input/output and artifact ceilings	Complete cumulative cost and production isolation
Output validation	Schema/serialization checks and correlated invalid-output failures	Semantic truth and arbitrary connector safety
Tool result	Calls receive one correlated terminal outcome	Ambiguous remote-effect reconciliation in Chapter 15
Progress	Bounded call-correlated lifecycle cards outside model context	Operator traces and metrics in Chapter 16
Replay	Duplicate IDs rejected within one runtime instance	Durable replay admission and cross-request idempotency in Chapter 15

8.13. Portability and evidence ledger

Dimension	Chapter claim	Evidence	Not established
Transport	Tool policy is independent of local or remote invocation	Tested local port	MCP transport behavior

Dimension	Chapter claim	Evidence	Not established
Provider	Neutral contracts cross the common model loop	v0.4.0 continuation and invalid-input/output fixtures	Live tool-policy equivalence across providers
Protocol	Open Responses function calls guide the design	Pinned source and current compatible HTTP subset	Formal conformance
Feature	Validation, authorization, correlation, and replay are separate controls	Book runtime and tagged adapter/runtime tests	Durable replay, full approval, sandbox, and cost enforcement
Semantic	Structured output does not guarantee truth	No positive claim	Equivalent tool selection or answer quality

8.14. Security checklist

Before enabling a tool:

- ☐ schema dialect and validator behavior are pinned;
- ☐ unknown properties and size limits are deliberate;
- ☐ authenticated context is separate from model arguments;
- ☐ authorization occurs immediately before execution;
- ☐ approval is bound to exact call content where required;
- ☐ cancellation, timeout, and concurrency limits propagate;
- ☐ retries are restricted by idempotency policy;
- ☐ outputs are bounded, validated, provenance-labeled, and treated as untrusted;
- ☐ credentials never enter model-visible schemas or results; and
- ☐ calls, decisions, outcomes, and costs are auditable without logging sensitive content by default.

8.15. Pattern summary

Problem: Model-generated tool calls can be malformed, unauthorized, duplicated, unbounded, or incorrectly correlated.

Pattern: Validate a pinned schema, authorize with trusted context, execute under limits, validate and correlate the result, then continue through a bounded shared loop.

Result: Provider-specific tool syntax becomes a controlled application operation with explicit authority and evidence.

Cost: Each tool needs schemas, policy, limits, error design, idempotency analysis, and deterministic tests.

8.16. Exercises

1. Add an output schema to the measurement tool and define the behavior for invalid connector data.
2. Design an approval token that cannot be replayed with different arguments.
3. Decide how an ambiguous timeout is represented for a payment tool.
4. Add repeated-call detection without blocking two legitimate calls with different inputs.
5. Specify deterministic ordering for three parallel, read-only tool results.

8.17. Further reading

Use JSON Schema Draft 2020-12 for the canonical contract (“JSON Schema Draft 2020-12,” n.d.), the pinned AI SDK tool documentation for the current implementation boundary (“AI SDK Core,” n.d.-b), and the Open Responses function-call item model as the later protocol target (“Open Responses Reference 2026-04-24,” n.d.). Chapter 18 applies the same application policy to tools discovered through MCP (“Model Context Protocol Specification 2025-11-25,” n.d.).

8.18. Chapter contract

- **Scope:** tool schemas, authorization, execution, correlation, and bounded results.
- **Canonical claims:** generated arguments propose an action; only server policy authorizes execution.
- **Migration inputs:** former typed-tool-loop chapter and Northstar measurement tool.
- **Evidence gates:** malformed input, denied authority, duplicate calls, cancellation, and output validation.
- **Demonstration slots:** deterministic local tool trace.
- **Source-claim ledger:** base runtime claims use v0.4.0; server-owned research tools/progress use snapshot 45a903c; API resemblance and compatible-subset behavior are not conformance.
- **Exclusions:** provider translation belongs to Chapter 5, protocol item/event vocabulary to Chapter 6, stream handling to Chapter 7, skill procedure to Chapter 9, retries/reconciliation to Chapter 15, telemetry to Chapter 16, and remote MCP transport to Chapter 18.

Part III.

**Part III — Skills and the
Compositional Research Toolchain**

9. Skills: Portable Procedures Without Hidden Authority

Part III · Skills and the compositional research toolchain

Implementation baseline: business-tech-presentation 1.1.0 · reviewed reference snapshot 45a903c

A skill can explain how to do work without gaining permission to do it. The Northstar presentation workflow makes that separation concrete: reviewed guidance shapes the deck, while a separately authorized tool—and only that tool—may construct a PowerPoint artifact.

9.1. Procedure is not authority

A **skill** is a reviewed, non-executable procedure. It can describe an evidence method, narrative sequence, quality checklist, or accessibility practice. A **tool** is executable capability with a schema, policy gate, limits, progress, and result contract. Confusing the two turns instructional text into an implicit plugin system.

The implemented boundary is deliberately asymmetric:

Concern	Skill	Tool
Primary role	Explain a reviewed procedure	Perform a bounded operation
Representation	Integrity-checked text and declared resources	Compiled server-owned implementation
Selection	Static <code>skills[]</code> entry in <code>agent.yaml</code>	Separate static <code>tools[]</code> entry
Effect	Adds context	May consume compute, network, or create an artifact
Authority	None	Only the authority granted by current policy
Model control	May request a declared resource	May propose typed arguments only

Loading text does not register a function. It cannot choose a provider, read credentials, open a path, install a package, enable network access, execute a bundled script, increase a quota, or alter `agent.yaml`. A procedure can say “inspect evidence before writing,” but it cannot make an inspection tool exist.

Treat skills as integrity-bound documentation and tools as separately authorized effects. Progressive disclosure changes what the model can read, not what the system permits it to do.

9.2. Static admission before progressive disclosure

The live configuration names reviewed IDs, not directories:

```
skills:
  - name: business-tech-presentation

tools:
  - name: presentation_builder
```

The static skill catalog maps `business-tech-presentation` to one reviewed package identity and profile. There is no search path, dynamic import, package resolver, URL, arbitrary prompt, or caller-supplied directory. Unknown and duplicate names fail startup. The current dependency rule also fails startup unless `business-tech-presentation` and `presentation_builder` are selected together. That pairing expresses a reviewed operational profile; it does not collapse their authority.

The server initially gives the coordinator only a short routing description. When presentation work becomes relevant, the coordinator can call `load_skill`. The successful result admits the bound `SKILL.md`, or one declared textual reference or asset, into context. This is **progressive disclosure**: detailed procedure is paid for in context only when needed.

The Pi client renders a successful operation as a `business-tech-presentation · skill · completed` card. Without that event, a transcript does not establish that the skill was loaded. The card is operational evidence of the load, not proof that every instruction was followed.

9.3. Integrity-bound package and declared resources

The skill package has a manifest that records its identity, profile version, and every payload file's role, byte count, and SHA-256 digest. Startup rejects:

- missing or undeclared payload files;
- changed byte counts or digests;
- duplicate, hidden, absolute, or unsafe paths;
- symbolic links;
- unsupported front matter or profile versions;
- oversized content; and
- identity or version mismatch.

Declared resources include the core procedure plus bounded narrative, visual, evidence, accessibility, outline, and theme material. Declaration has two purposes. It makes the review set finite, and it prevents a later filesystem addition from silently becoming model context.

Script files illustrate an important distinction. A manifest can integrity-check a script as package material without granting it execution authority. `load_skill` does not return scripts, and the live skill loader does not run them. Integrity answers “are these the reviewed bytes?” It does not answer “may these bytes execute?”

SHA-256 binding also has limits. It detects a change relative to the reviewed manifest under the collision-resistance assumption; it does not establish that the procedure is correct, safe for every task, or authored by a trusted party. Review, package admission, and execution authorization remain separate gates.

9.4. The Northstar presentation workflow

The research turn distinguishes five evidence classes: attached declarations, workbook measurements, generated image observations, deterministic calculations, and retrieved public guidance. The follow-up—“make a short pptx deck about those findings”—must preserve those distinctions rather than flatten them into a confident marketing narrative.

The skill guides the coordinator through a bounded sequence:

1. state the governing message and intended decision;
2. build an evidence ledger before slide construction;
3. draft answer-first titles that form a coherent sequence;
4. choose a visual form suited to each claim;
5. preserve units, missingness, uncertainty, and source labels;
6. use semantic construction rather than drawing instructions;
7. inspect warnings and the rendered artifact; and
8. describe completion honestly.

This workflow is portable because its important concepts are not PptxGenJS calls or PowerPoint coordinates. They are procedure-level obligations. The executable mapping from those concepts to a .pptx remains server-owned.

9.5. Separate presentation_builder authorization

presentation_builder is selected and authorized as a tool under Chapter 8’s policy loop. The skill cannot summon it when the tool is absent. Conversely, direct tool selection does not turn the skill package into executable code.

The model-facing contract accepts a semantic deck: filename, deck title, purpose, bounded slides, evidence labels, optional speaker notes, and supported semantic content. The initial profile includes title, statement, bullets, comparison, metrics, bounded charts, bounded flows, and admitted-image slides. The model cannot submit:

- JavaScript, TypeScript, Python, HTML, shell, or macros;
- host paths, package names, credentials, endpoints, or remote assets;
- template files, private masters, component IDs, or recipe IDs;
- slide coordinates, arbitrary styles, or unrestricted geometry;
- OLE objects, embedded packages, or external relationships.

Image slides refer only to admitted invocation-local source IDs and require alt text plus an evidence label. Source notes accept bounded labels and optional credential-free HTTPS URLs. The schema is correction-friendly, but runtime parsing is authoritative: provider schema acceptance cannot bypass the implementation’s narrower checks.

9.6. Semantic input, server-owned composition

A semantic slide says what the content means. The server decides how that meaning is composed:

```

semantic slide
  ↓ validate and normalize
reviewed composition recipe
  ↓ select 2-5 reviewed components
bounded slide regions
  ↓ render and inspect
one PowerPoint slide

```

The independently authored gallery contains 312 mappings. Region-aware components receive an existing slide, a fixed region, validated content, and a reviewed variant. They do not create slides themselves. Startup validation rejects unknown component identities, incompatible regions, duplicate occupancy, geometric overlap, and unreviewed template references.

This design keeps model variability above the rendering boundary. A model can choose a comparison or process meaning, but it cannot ask the worker to place an arbitrary object outside the slide, invoke a hidden template, or reach an unreviewed component by name. Server-owned recipes can select reviewed timeline, cycle, network, handoff, six-stage, and linear-flow forms while retaining the same semantic request contract.

The trade-off is deliberate limitation. A bounded vocabulary cannot reproduce every bespoke visual expression. Portability here means controlled composition with known failure modes, not unrestricted design automation.

9.7. Missing values and numeric normalization

Presentation data frequently arrives with missing observations. The implementation distinguishes two uses of `null`:

- conventional `null` in an optional field normalizes to omission; and
- explicit `null` inside a chart series means a missing observation.

A missing chart observation renders as a gap or N/A, never as zero. Zero is a measurement; `null` is absence. Conflating them changes the claim shown by the deck.

Finite numeric strings such as "0.2" or "1e-1" can normalize to numbers before strict validation. Non-finite, malformed, or out-of-range values still reject. Bar charts use a visible zero baseline and support signed values. Line charts show scale, markers, and values. Automatically derived callouts describe neutral first-to-last change or observed range rather than guessing whether higher or lower is desirable.

These transformations improve correction behavior without weakening the boundary. Normalization is explicit and bounded; it is not permission to reinterpret arbitrary prose as data.

9.8. Artifact construction and validation

A killable Bun worker runs the pinned presentation library and emits a candidate OOXML package. Cancellation or deadline expiry terminates the worker because the underlying ZIP generation has no trusted cooperative cancellation contract.

Before admission, the server checks the typed deck limits and inspects the produced package. Validation rejects, among other conditions:

- unexpected slide count;
- macros or active/embedded packages;
- external relationships;
- negative geometry;
- visible text below 9 pt; and
- output above the configured byte ceiling.

Quality analysis separately warns about generic or repeated titles, claim-bearing slides without evidence labels, missing decision/risk framing, repeated layouts, generic chart units, and dense bullets. A warning is not structural acceptance, and structural acceptance is not visual approval.

Completed bytes enter a process-local opaque artifact store. The result contains a random artifact ID, sanitized filename, PowerPoint MIME type, exact byte count, SHA-256 digest, slide count, profile, and expiry. There is no artifact-listing endpoint. IDs expire, and a restart discards the store.

9.9. Verified handoff and safe replacement

The Pi extension observes the completed server-tool event, immediately fetches the short-lived artifact from the loopback /v1 endpoint, refuses redirects, and verifies both exact byte count and SHA-256 before saving. A matching digest without the expected length would not satisfy the handoff; both fields bind the retrieved payload to the completion metadata.

The local file is created with mode 0600. The sanitized default filename does not authorize replacement. If the target already exists, Pi requires confirmation rather than silently overwriting it. Explicit /presentation-artifact <id> and --save <id> operations retain the same verification boundary; preview also requires confirmation and does not expose deck bytes to the model or progress channel.

The opaque ID is short-lived local capability data, not durable source identity. It cannot be used as an attachment on a later invocation, and it should not be published. A saved file becomes ordinary user-owned local state and is outside the server lease.

9.10. What visual review does—and does not—prove

OOXML inspection can prove structural properties of the generated package. It cannot prove that a title wraps well, a chart communicates clearly, color choices work in every display environment, or a presentation application renders every object identically.

Representative acceptance used the exact installed Microsoft PowerPoint for Mac version, PDF export, and rendered-page inspection. The Northstar artifact also received a separate exact-artifact review. Those observations apply to the reviewed bytes, application versions, fonts, and host conditions. They do not establish compatibility with LibreOffice, Keynote, Google Slides, Windows PowerPoint, arbitrary fonts, future versions, or every semantic deck the model may propose.

The published Northstar recording shows skill load, tool completion, and digest-verified save. Its metadata explicitly says the generated deck was not visually inspected during that recorded session. The separately reviewed deck can support a later artifact-specific statement; the recording itself cannot be retroactively promoted into visual-quality evidence.

9.11. Failure modes

Skill-as-plugin. A package directory is scanned and bundled scripts execute. Keep the catalog static and make loading read-only.

Manifest-as-permission. Integrity-checked code is assumed safe to run. Separate byte identity from executable authority.

Semantic escape hatch. A “style” or “template” field accepts arbitrary model content. Keep model input semantic and map it through reviewed server recipes.

Missingness laundering. `null` becomes zero and changes a chart’s conclusion. Preserve gaps explicitly.

Artifact race. Metadata describes one payload while the client saves another. Verify exact length and digest after retrieval and before replacement.

Screenshot certification. One attractive rendering is generalized to every suite and input. Scope visual evidence to exact artifacts and versions.

9.12. How we test portability

Deterministic tests cover static catalog selection, dependency failure, manifest tampering, undeclared files, unsafe paths and symlinks, progressive resource loading, schema normalization, finite numeric strings, missing chart values, quotas, cancellation, OOXML relationships and package content, leases, digest/byte verification, and replacement behavior. Gallery tests cover all 312 mappings structurally.

Those tests establish bounded mechanics, not universal design quality. Paid model behavior, visual judgment, office-suite compatibility, and human comprehension remain separate evidence classes.

9.13. Pattern summary

Problem: Reusable procedures are often packaged with hidden code or treated as if model context grants authority.

Pattern: Select skills from a static catalog, bind every declared resource to a manifest, disclose them progressively, and require separately selected tools for every executable effect.

Result: A procedure can be portable and compositional without becoming a plugin loader or policy bypass.

Cost: Package review, manifest maintenance, a constrained semantic vocabulary, artifact inspection, and exact-artifact visual review remain necessary.

9.14. Exercises

1. Add a second non-executable skill whose procedure uses an existing tool. Identify every startup and loading check required without adding a directory scanner.
2. Construct a chart with [4, null, 0, 7]. Explain the visual and semantic difference among the middle values.
3. Design an artifact replacement flow for an automated client. State what evidence could replace interactive confirmation without allowing silent overwrite.
4. Classify each presentation check as schema, policy, package, handoff, or visual evidence. Explain why no single class subsumes the others.

9.15. Chapter contract

- **Scope:** reviewed non-executable skills, progressive disclosure, integrity binding, semantic presentation construction, and artifact handoff.
- **Canonical claims:** a skill supplies procedure but grants no authority; presentation construction remains a separately authorized tool effect.
- **Migration inputs:** former skill-boundary prose, the implemented presentation package, artifact extension, quality audit, and Northstar presentation review.
- **Evidence gates:** manifest and catalog tests, builder/package tests, exact byte/digest hand-off, and artifact-specific visual review.
- **Demonstration slots:** the Northstar skill-load/build/save sequence demonstrates the boundary but not universal visual quality.
- **Source-claim ledger:** version, digest, renderer/application version, review status, and limitations accompany presentation evidence.
- **Exclusions:** MCP and A2A transport/delegation belong exclusively to Chapter 18; generic tool authorization belongs to Chapter 8.

10. web_search: Broad Discovery Under a Budget

Part III · Skills and the compositional research toolchain

Implementation baseline: server-owned OpenAI hosted-search bridge · reviewed reference snapshot 45a903c

Search discovers candidates. It does not establish that a page was read, that generated notes follow from the page, or that a URL supports the final answer. The Northstar workflow therefore uses search to find official guidance, then crosses a separate retrieval boundary in Chapter 11.

10.1. Search is a tool, not ambient model knowledge

web_search is a server-owned application tool inside the provider-neutral loop. An OpenAI or Anthropic coordinator may propose the same typed call, but the server owns validation, authorization, budget, recipient, execution, progress, and the bounded result.

The backend is a separate OpenAI Responses request with native hosted web_search. It is not a provider-native tool secretly attached to the coordinator. The nested route receives a short fixed instruction and the selected query—not the complete conversation, document corpus, or tool history. Search calls and results remain private to the shared loop; Pi does not execute them a second time.

```
coordinator proposes web_search(query)
    ↓ validate / authorize / debit
separate fixed OpenAI search-model request
    ↓ require completed native receipt
normalize and fit candidate result
    ↓
coordinator receives bounded notes + URL candidates
```

This bridge design matters for portability. The application tool has one name and result shape even when the coordinator changes. It does **not** imply that hosted-search internals, ranking, consulted pages, latency, or output quality are provider-neutral.

10.2. Search-model binding

The search recipient and model come from reviewed agent.yaml, not tool arguments:

```
- name: web_search
  model: gpt-5.4
  maxCalls: 300
  maxSources: 1000
  sourcesPerCall: 100
  timeoutMs: 45000
  maxOutputTokens: 4096
  requestTimeoutMs: 21600000
```

The model is an unprefixed OpenAI wire ID. The bridge uses the server's approved OpenAI credential and endpoint configuration. Callers cannot provide a provider, credential, endpoint, header, retry setting, or larger limit. SDK retries are zero, and there is no automatic alternate search provider or model.

A configured model name is a declaration, not a live capability probe. Startup can validate syntax and bounds without proving that the account currently permits hosted search. A remote rejection fails safely rather than selecting an undisclosed fallback.

10.3. Two budgets, two meanings

One HTTP invocation owns one `BoundedSearchSession`. Its primary budgets are:

- **application calls**—each admitted search attempt spends one slot before backend dispatch; and
- **distinct returned source URLs**—new normalized candidates admitted into bounded results spend aggregate source capacity.

A failed backend call retains its call debit because remote work may already have occurred. It spends no source slots when no result is admitted. Repeated normalized URLs do not spend another unique-source slot. A new HTTP request, including a later Pi continuation, receives a new budget; this is not a conversation-wide or distributed quota.

`sourcesPerCall` limits one result independently of `maxSources`. These are ceilings, not targets. The prompt asks for focused evidence and material follow-ups rather than consuming every permitted call. A default allowance of 300 calls demonstrates accounting capacity; it does not promise that a six-hour, 300-call live run will finish, remain affordable, or fit model context.

The whole-request deadline includes coordinator work. Per-call timeout and provider output-token settings remain separate. Clients, proxies, account quotas, context windows, and provider limits may terminate earlier.

10.4. Completion-order admission under concurrency

Queries that do not depend on each other can run concurrently. A dependent query—one needing a name discovered by the first—must wait. Concurrency introduces a source-budget race: two results can complete when only one has remaining capacity.

Admission therefore occurs atomically in **completion order**. The first completed valid result claims available new-source slots; a later completion sees the updated aggregate. Already-submitted remote work can still finish after capacity has been exhausted, but its candidates cannot overbook admission.

This ordering is deterministic relative to observed completion, not query submission. It is not relevance ranking across calls. If product policy needs priority by topic or source class, it must schedule or reserve capacity before dispatch rather than rewriting history after completion.

Result fields include remaining-call and remaining-source snapshots. Concurrent activity can make them stale immediately. Runtime admission remains authoritative.

10.5. Candidate normalization

Backend output is untrusted. The normalizer:

- accepts bounded HTTP(S) URL candidates;
- rejects embedded username/password credentials and unsafe schemes;
- normalizes with the platform URL parser;
- uses exact normalized `URL.toString()` equality for deduplication;
- bounds source title and excerpt-like fields;
- keeps answer-linked candidates ahead of broader consulted candidates when available; and
- separates generated notes from source records.

Exact URL equality is not semantic document identity. Tracking-parameter variants can consume separate slots, and two different URLs can serve identical bytes. Conversely, removing or reordering query parameters could change a resource, so the search layer does not invent a stronger equivalence.

Native API/non-URL source objects and unsafe locators are excluded. A normalized locator says only that the application admitted this candidate string into the result. It does not prove that the candidate is public-fetchable under Chapter 11's policy.

10.6. Completion receipts before authority

Streaming text from the nested model is not enough for success. The bridge requires one completed native response and a matching completed hosted-search call/result identity. Missing or conflicting receipts, excess distinct native executions, incomplete terminal state, duplicate response terminals, malformed output, cancellation, and deadline expiry fail closed.

Identical repeated SDK observations can be consolidated only when call IDs, inputs, outputs, native completion observations, and the final item agree. This handles duplicate observation of one execution; it does not increase the native execution allowance or create exactly-once remote work.

The requested provider cap is observable after the fact. It cannot preempt undisclosed remote internals or undo provider work already performed. A native search action may generate several queries or consult several pages. The 192-code-point application query limit is not a bound on those provider-generated operations.

10.7. Provisional search progress

Negotiated Pi clients can display bounded generated search text while the nested request runs. The card labels it provisional and unverified. It is not assistant final-output streaming, page-open telemetry, or a citation.

The implementation limits cumulative snapshots and visible size, batches small deltas, and redacts known raw or form-encoded server-key echoes. On failure or cancellation, the widget discards the provisional text and never persists it. Older clients receive no preview.

Progress callbacks cannot determine execution success, and a displayed preview cannot substitute for the completed receipt. Redaction covers specific known credential patterns; it is not a general confidentiality classifier. Query terms and generated text can still disclose sensitive facts.

10.8. Fitting the final output

The normalizer constructs one bounded result envelope. Optional generated notes are removed before source locators. If necessary, sources and the query preview are reduced until the final output fits both the serialized UTF-16-unit ceiling and UTF-8 byte ceiling.

Fitting is semantically lossy. Removing a source can reduce coverage; truncating a title or note can remove a qualifier. The result therefore reports candidates under stated bounds rather than claiming completeness. A safe `search_failed` outcome replaces malformed or unfit upstream content; raw provider bodies and errors do not pass through.

Generated notes are model synthesis, not quotations. Even when they mention a source, they are not retained source representations and cannot become exact evidence merely by surviving output fitting.

10.9. Privacy and recipient disclosure

A search query is a disclosure to the configured OpenAI recipient and potentially to the public-search ecosystem it invokes. The bridge sends less than the full coordinator context, but the selected query can still contain names, facts, or fragments derived from private material.

The application should disclose:

- that the operation uses a separate OpenAI search model;
- which model/recipient class is configured;
- that public-network activity and charges may occur;
- that non-storage is requested but is not a zero-retention certification; and
- that results are generated candidates rather than verified source evidence.

Ordinary logs contain no query, snippets, URLs, credentials, or raw upstream failures. Safe diagnostics use categories, elapsed time, optional HTTP status, allowlisted SDK error names, and bounded observation counts. This minimizes routine exposure but does not make confidential search safe. A separate reviewed disclosure policy is required before enabling public search over sensitive conversations.

10.10. Why candidates are not admitted evidence

A search result can fail evidence use in several independent ways:

1. the URL may be blocked by public-HTTPS retrieval policy;
2. the page may redirect, change, reject access, or exceed limits;
3. the retrieved representation may omit the relevant material;
4. the generated search note may misstate the page;
5. the page may be false, stale, or unsuitable; or
6. the final claim may not follow from what was read.

Chapter 11 admits a concrete retrieved representation. Chapter 13 then evaluates identity, locators, claim support, and research completion. Search has completed its job when it supplies bounded candidates and records gaps honestly—not when it creates a citation-looking URL.

10.11. Current evidence and its limitations

Deterministic tests cover configuration rejection, exact query limits, 300-call admission, 1,000 aggregate URL slots, per-call fitting, concurrent completion, duplicate candidates, malformed and contradictory receipts, cancellation, provisional-preview privacy, output bytes, and zero automatic retries. Tests inject synthetic provider transport under network denial.

The currently published Northstar recording is dated live-provider and public-network evidence. Its metadata records an Anthropic coordinator and a separate openai/gpt-5.4 search recipient. The run visibly completes web_search and then read_source on selected official guidance. It demonstrates that exact reviewed route at that date. It does not establish:

- deterministic search results;
- completeness or ranking quality;
- provider equivalence;
- that every returned candidate was read;
- general factual correctness or entailment;
- production confidentiality; or
- Open Responses conformance.

The historical Wikimedia report belongs to an older standalone backend and explicitly claimed no citations. It must not be described as evidence for the current hosted-search bridge.

10.12. Failure modes

Search-note laundering. Generated notes are quoted as page text. Read the selected representation first.

Budget reset laundering. A client creates repeated HTTP requests to evade a per-invocation limit. Add a separately owned higher-level quota when conversation or tenant accounting is required.

Completion without receipt. Provisional text is treated as successful search after terminal failure. Require the completed native execution record.

Recipient ambiguity. An Anthropic coordinator makes the UI imply Anthropic performed search. Disclose the separate OpenAI recipient.

Link-count confidence. Many URLs are treated as strong support. Evaluate suitability and claim support rather than counting candidates.

10.13. Pattern summary

Problem: Web discovery is useful but remote, variable, disclosure-bearing, and easy to misrepresent as evidence.

Pattern: Bind search to a reviewed recipient, debit calls and unique URLs separately, admit concurrent completions atomically, normalize and fit untrusted output, and label all results as candidates.

Result: Coordinators can discover public leads through one portable tool contract without turning hosted-search prose into source authority.

Cost: Separate retrieval, evidence review, recipient policy, higher-level quotas, and live compatibility evidence remain necessary.

10.14. Exercises

1. Two concurrent calls complete with 80 new URLs each and 100 source slots remaining. Specify the admitted results under completion-order accounting.
2. Design a conversation-level quota without trusting a caller-supplied session ID. State the missing identity requirement.
3. Show how a correct URL citation could still fail Chapter 13's support test.
4. Define a user disclosure for an Anthropic-coordinated run that sends only a focused query to OpenAI search.

10.15. Chapter contract

- **Scope:** search-model binding, call/source budgets, completion-order admission, candidate normalization, progress, output fitting, and disclosure.
- **Canonical claims:** search output supplies candidates and generated notes, not admitted source representations or verified support.
- **Migration inputs:** current hosted-search implementation, invariant review, deterministic tests, and dated Northstar live evidence.
- **Evidence gates:** synthetic receipt/accounting tests plus separately dated and scoped live-search records.
- **Demonstration slots:** Northstar shows the configured search route and follow-on retrieval, not search completeness or truth.
- **Source-claim ledger:** each live claim names coordinator, search recipient/model, endpoint class, date, limits, and limitations.
- **Exclusions:** public source retrieval belongs to Chapter 11; support and completion belong to Chapter 13.

11. read_source: From URL to Admitted Representation

Part III · Skills and the compositional research toolchain

Implementation baseline: screened public-HTTPS source reader · reviewed reference snapshot 45a903c

Reading a URL is a network and parsing operation, not a synonym for trusting it. The application must decide which destination, bytes, representation, and lifetime it is willing to admit. In the Northstar workflow, this is the boundary between a search candidate and the official guidance actually examined.

11.1. One narrow public-read operation

`read_source` accepts exactly one bounded URL. It does not accept a method, body, caller headers, cookie, credential, proxy, local path, browser profile, provider, or extraction plugin. The reviewed configuration fixes public-HTTPS access and limits:

```
- name: read_source
  access: public-https
  maxCalls: 100
  timeoutMs: 15000
  maxResponseBytes: 5242880
```

Every admitted logical call spends invocation budget. Repeated successful reads may reuse one immutable representation within the same invocation, but authorization and current URL policy still run. Failed and cancelled calls retain their debit; they cannot become free remote probes.

Search and read remain separate. Chapter 10 discovers candidates. This chapter selects one candidate, authorizes a destination, retrieves complete bounded bytes, and derives a named representation. It does not decide whether the source is true or whether an answer follows from it.

11.2. URL admission before DNS

The reader admits only an unambiguous HTTPS URL using the default port and a DNS hostname. It rejects:

- URL credentials;
- explicit ports;
- IP-address literals;
- non-HTTPS and ambiguous schemes;
- control characters and malformed forms;
- local or internal hostnames; and
- unsupported URL structure.

The fragment is not sent on the request and does not change fetch identity. Query order, repeated keys, encoding, and values remain significant because “cleaning” them can change the resource. Hostname case and other equivalences guaranteed by the URL parser may normalize; the reader does not invent semantic equivalence.

Syntax checks are only the first gate. A syntactically public hostname can resolve to a private destination.

11.3. DNS and address screening

The transport resolves the hostname and screens **every** returned address. If any answer is loop-back, private, link-local, multicast, reserved, documentation-only, or otherwise outside the reviewed public set, the destination is rejected. Mixed public/private answers fail rather than selecting only the convenient public member.

The current profile retains a reviewed address-family policy; expanding it requires equivalent screening and tests rather than accepting a new family by default. Application-level checks are defense in depth, not an egress firewall or operating-system network sandbox.

Screening has to survive the time-of-check/time-of-use boundary. Resolving an acceptable address and then opening an ordinary hostname connection would permit a later lookup to select another destination. The implementation therefore pins the screened address for the actual connection.

11.4. Pinned transport with hostname TLS validation

Address pinning must not discard the original hostname’s security meaning. The connection targets the screened address while TLS Server Name Indication, certificate validation, and HTTP Host remain bound to the admitted hostname.

The reader does not honor ambient proxy variables, send cookies or authorization, inherit browser state, or add caller-controlled headers. This prevents ordinary process environment and user-agent state from silently widening the recipient boundary.

The transport requires complete, successfully framed responses. It does not guess the end of a close-delimited body. Deadlines and cancellation cover the bounded operation, but local cancellation does not prove that the origin did no work.

11.5. Redirects are new destinations

The current policy refuses redirects. This is stricter than following a redirect after validating only the first URL. A redirect can change scheme, hostname, address class, credentials, and disclosure destination.

A future redirect profile would need to bound hops and repeat full URL parsing, DNS/address screening, address pinning, and authorization for every hop. It would also need a retained safe lineage. Calling a client with automatic redirect support would bypass that review.

The practical consequence is visible failure on many otherwise ordinary web pages. The coordinator may search for a direct public representation; it may not relax policy or imply that a failed landing page was read.

11.6. Media type, framing, and encoding

A network success is not yet an admitted representation. The reader checks response status, framing, declared media type, content encoding, body size, and parser profile. Unsupported compression or encoding rejects rather than entering an unreviewed decompressor.

The current shared ingestion path supports bounded:

- HTML;
- PDF;
- DOCX;
- XLSX;
- UTF-8 text and Markdown;
- JSON and JSONL; and
- CSV and TSV.

HTML is converted to inert content; scripts and subresources do not execute. Office and PDF formats use the same profiled document-ingestion boundary as request attachments. PDF OCR requires the configured local Tesseract path when applicable. Layout-model inference remains disabled. The reader is not a browser and does not claim visual page equivalence.

Media type and content have to agree sufficiently for the selected parser profile. Unsupported, malformed, encrypted, or corrupt representations fail with a bounded category rather than falling through to arbitrary parser behavior.

11.7. One byte ceiling, applied twice

`maxResponseBytes` is the public configuration ceiling for both response bytes and extracted canonical text. The implementation rejects oversized input or output instead of truncating a document and labelling it complete.

Shared ingestion adds format-specific controls over archive expansion, entry count, nesting, entities, parser iterations, pages, cells, images, dimensions, OCR, and extraction deadline. Those controls prevent a small wire body from becoming unbounded parsing work.

The result must also fit the shared tool-output envelope. These are distinct limits:

```
wire bytes
→ media/encoding admission
→ parser-specific expansion limits
→ canonical text bytes
→ model-visible serialized result bytes
```

Passing an early ceiling does not imply passing the later ones. A complete admitted representation is one that survives every applicable bound without silent truncation being misdescribed as completeness.

11.8. Shared ingestion, not a second parser stack

Request attachments and read_source use the same reviewed extraction profiles. This avoids a common inconsistency: a PDF attached locally produces one canonical representation while the same PDF fetched over HTTPS follows a more permissive parser path.

Network admission remains unique to this chapter. File-byte admission and extraction behavior belong to Chapter 4. The composition point is explicit: the public reader supplies screened bytes and media metadata to shared ingestion; shared ingestion returns the bounded canonical representation or a typed failure.

The source remains untrusted throughout. A document can contain imperative text, scripts represented as text, malicious instructions, false claims, or adversarial strings. Extraction changes representation, not authority.

11.9. Invocation-local source identity

Every successful read returns bounded content plus identity metadata, including:

- canonical retrieval URL;
- opaque sourceId;
- body and/or representation digest as retained by the profile;
- retrieval time;
- media type;
- representation/extractor revision; and
- complete canonical text within the selected bounds.

The registry also binds the representation to the current invocation owner and request. The model can later pass the exact sourceId to code_interpreter; it cannot invent an ID, use one from another invocation, or turn it into a host path.

A source ID is not a URL, truth proof, publisher signature, or durable citation identity. A digest identifies bytes under the selected representation; it does not authenticate the origin, validate extraction, establish permission, or prove claim support.

The same URL can yield different bytes later. The same bytes can appear under different provenance. Retain address, retrieval event, representation identity, and source authority as separate fields.

11.10. Invocation-local caching and single flight

The live server has no authenticated cross-request Pi session identity suitable for shared cache ownership. The cache is therefore created once per HTTP invocation and destroyed at terminal cleanup.

Its key is the exact authorized retrieval identity: canonical URL including meaningful query, excluding the network-irrelevant fragment, under the current reader/extractor policy. It contains successful immutable representations and bounded in-flight entries only.

Concurrent identical misses share one underlying retrieval. Successful waiters receive detached immutable snapshots. Failures, malformed results, oversized representations, and late results

after cancellation do not enter the completed cache. A cancelled waiter does not necessarily cancel work still needed by another active waiter; if every waiter is cancelled, late completion cannot publish authority.

A cache hit returns the original digest and retrieval timestamp. It is reuse, not a fresh network verification. There is no process-global, persistent, session, or cross-restart cache, no TTL freshness promise, and no durable snapshot archive.

11.11. Progress and privacy

Pi progress may show a bounded lifecycle, safe host/path preview without query or fragment, media category, representation length, duration, cache reuse, and safe failure class. It must not display document text, quotations, full sensitive URLs, resolved addresses, headers, digests, credentials, or raw errors.

Ordinary diagnostics similarly report category and counts rather than payload. “In memory only” does not mean absent from provider context, process memory, swap, crash dumps, or later answer text. Once the canonical representation is returned to the coordinator, it can enter model context under the configured provider policy.

11.12. Retrieved content remains data

The retrieved page cannot:

- register a tool or skill;
- choose a provider or endpoint;
- widen a limit;
- authorize another destination;
- grant filesystem or network access;
- approve a consequential action; or
- override system and application policy.

Labels such as “untrusted source” help the model, but server enforcement must not depend on the model obeying them. Chapter 8’s policy runtime remains the authority boundary. Tool results can inform later proposals, never authorize their execution.

11.13. Retrieval is not truth or entailment

A completed read establishes a narrow fact: through the admitted transport and parser path, the application obtained this bounded representation at this time. It does not establish:

- who authored the material;
- whether it is current, complete in the broader publication, or accurate;
- whether extraction preserved every layout-dependent meaning;
- whether the selected passage supports a generated claim;
- whether another source contradicts it; or
- whether the application has permission to redistribute it.

A final URL citation can identify read provenance without pretending to be a provider-native annotation. The renderer must not fabricate page, section, quote, or source-span locators. Chapter 13 owns support assessment and completion criteria.

11.14. Northstar handoff

In the recorded workflow, web_search finds current official US guidance and read_source retrieves the selected official HTTPS representation. That sequence demonstrates why the tools are separate: search output chooses a candidate; source reading creates the invocation-local representation used by the answer.

The recording is dated evidence for the exact route and selected page. It does not prove that the page remains unchanged, that the retrieval policy reaches all official sites, that extracted guidance is universally applicable, or that the generated answer correctly entails every cited sentence.

11.15. Failure modes

SSRF by syntax. A URL parser accepts a hostname that resolves privately. Screen all addresses and pin the admitted one.

Redirect laundering. The first public URL redirects to an unreviewed destination. Refuse redirects or fully reauthorize each hop.

Partial-as-complete. The reader truncates a large response and presents it as the document. Reject or label an explicitly partial representation.

Cache-as-freshness. A reused snapshot is described as newly checked. Preserve the original retrieval time.

Text-as-instruction. Retrieved imperatives influence server authority. Keep policy and execution outside source content.

Digest-as-truth. A stable hash is treated as publisher authentication or semantic proof. Keep integrity, provenance, and support separate.

11.16. Pattern summary

Problem: A URL can conceal a prohibited destination, mutable bytes, dangerous parsing work, and hostile content.

Pattern: Admit public HTTPS narrowly, screen and pin addresses, refuse unreviewed redirects/encodings, apply shared bounded ingestion, and retain invocation-local representation identity.

Result: Discovery candidates can become concrete, reviewable source representations without becoming trusted instructions or factual proof.

Cost: Strict policy rejects some legitimate pages; durable identity, richer redirects, authenticated sources, and semantic verification require separate designs.

11.17. Exercises

1. Design a redirect extension that reauthorizes every hop. List the fields required in safe redirect lineage.
2. Explain why a mixed public/private DNS answer fails even if the client intends to use the public address.
3. Compare a cache hit with a conditional fresh read. What new transport and freshness contracts would the latter require?
4. Construct a successful retrieval whose final claim is still unsupported.

11.18. Chapter contract

- **Scope:** public-HTTPS URL admission, DNS/address screening, pinned transport, redirects, media/encoding policy, response limits, ingestion, invocation caching, and source identity.
- **Canonical claims:** successful retrieval identifies an admitted representation; it does not establish truth, authority, or entailment.
- **Migration inputs:** current source-reader implementation, shared ingestion profile, invocation-cache probes, and deterministic transport tests.
- **Evidence gates:** URL/DNS/TLS/redirect/media/size tests plus separately dated live retrieval evidence.
- **Demonstration slots:** Northstar demonstrates candidate-to-representation handoff for one official URL only.
- **Source-claim ledger:** retrieval claims record URL identity, time, media type, digests/revision where retained, and representation limitations.
- **Exclusions:** discovery belongs to Chapter 10; support and research completion belong to Chapter 13; local attachment admission belongs to Chapter 4.

12. code_interpreter: Bounded Local Computation

Part III · Skills and the compositional research toolchain

Implementation baseline: pinned local Pyodide worker · reviewed reference snapshot 45a903c

A calculation can be reproducible without being authoritative. The Northstar mean of 8.25 hours is deterministic given the workbook values 9 and 7.5; the interpreter does not prove that those measurements describe a real device or that Python-in-WebAssembly is an operating-system sandbox.

12.1. One portable computation contract

code_interpreter is a server-owned tool exposed to both coordinator adapters through the same schema. It is not an OpenAI- or Anthropic-hosted code tool. The reviewed configuration controls call count, deadline, code size, and three independent output ceilings:

```
- name: code_interpreter
  maxCalls: 10
  timeoutMs: 30000
  maxCodeCharacters: 20000
  maxOutputCharacters: 64000
```

Its input contains a user-visible summary, Python code, and optionally up to four unique invocation-local sourceIds. The caller cannot supply packages, network settings, a working directory, host paths, credentials, a persistent volume, or provider-specific execution options.

Chapter 8 owns whether the proposed call is currently authorized. This chapter owns what happens after admission: scheduling, worker creation, mounting, execution, cancellation, output fitting, and progress privacy.

12.2. Fresh worker construction

Every admitted call constructs a new Pyodide WebAssembly worker and a new Python state. Completion, timeout, cancellation, malformed worker output, or execution error terminates that worker. No Python variables, imported modules, mounted files, or mutations carry into a later call.

Fresh construction narrows several failure modes:

- one call cannot intentionally prepare hidden Python state for the next;
- a mutation of mounted input cannot alter the invocation source registry;
- cleanup does not depend on generated code politely deleting files; and
- timeout can terminate the worker rather than waiting for cooperative Python.

Fresh does not mean costless. Runtime startup and WebAssembly memory are paid per call, which is one reason the server imposes a shared scheduler and small call budget.

12.3. Invocation-local mounts, not arbitrary files

The coordinator can pass only exact source IDs from two trusted invocation inventories:

1. request attachments admitted by Chapter 4; and
2. successful read_source representations admitted by Chapter 11.

Unknown, invented, expired, duplicate, or cross-invocation IDs reject before Python starts. For each accepted ID, the server copies canonical text into the worker under a fixed `/inputs/<sourceId>.txt` path. Python receives an `input_files` mapping from ID to canonical-text path.

For admitted request attachments, `input_assets` can additionally expose the fixed worker-local path to the original admitted bytes and the canonical text path. The model still chooses an ID, not a host path. The copied files and worker filesystem disappear when the call ends.

This `read`→ID→mount sequence avoids embedding hundreds of thousands of source characters inside generated Python. It also preserves provenance: code declares which invocation-owned representations it used. A mount does not certify those representations as true.

12.4. The Python execution boundary

The worker is intended for bounded calculations, deterministic transforms, parsing, sorting, filtering, counting, aggregation, deduplication, ranking, and checks over explicit input. The tool-owned coordinator guidance requires execution when the requested result creates or transforms structured and machine-readable data rather than inviting the model to simulate arithmetic in prose.

The declared profile provides no:

- network access;
- third-party package installation;
- arbitrary host path or attachment access;
- browser, shell, or operating-system process access;
- persistent Python state; or
- generated-file upload and durable artifact handoff.

The ordinary JavaScript bridge and package installer imports are blocked, and Python receives a null-prototype JavaScript-global object. Schema, code, output, quota, and ownership checks execute outside Python.

These are meaningful controls around a local educational runtime. They are not a claim of complete Pyodide escape resistance or hostile multi-tenant isolation.

12.5. Scheduling and concurrency

Calls enter one process-wide queue shared across HTTP invocations. Only the configured active capacity may hold live WebAssembly workers; the current reviewed profile serializes calls to limit memory pressure.

Scheduling has two consequences:

- a single invocation cannot create unbounded parallel workers; and
- one invocation can delay another, so queue time must remain inside the call/run deadline and operational policy.

Every admitted call spends its invocation quota even if it later waits, fails, or times out. A queued call cancelled before its turn cannot enter the worker later. Fairness, tenant weighting, and distributed scheduling are not implemented by this local queue.

The queue protects process pressure structurally; it does not enforce a WebAssembly heap limit or guarantee that one worker cannot degrade the process before termination.

12.6. Deadlines and cancellation

A whole-tool deadline bounds validation, queueing as implemented by the runtime, worker startup, and execution. Non-cooperative Python is stopped by terminating the worker. Cancellation propagates through the scheduler; an already-cancelled queued task never becomes runnable.

The runtime distinguishes:

- invalid input before execution;
- unavailable source IDs;
- Python exceptions;
- deadline expiry;
- caller cancellation; and
- malformed or oversized worker results.

A Python exception is returned as a bounded typed tool result so the coordinator can correct code when appropriate. Pi shows only the safe statement that Python executed and returned an exception; details remain in the bounded model-visible result. Timeout and cancellation remain runtime outcomes rather than fabricated Python output.

Termination is a local control. It does not prove exact CPU accounting, instant memory reclamation, or absence of process impact before the worker stopped.

12.7. Output fitting

Successful output contains bounded stdout, stderr, optional final-expression representation, elapsed time, and exact omitted-character counts. Each of the three text channels has its own configured ceiling, followed by the aggregate tool-result byte bound.

Fitting is explicit rather than silent. If Python prints a million-character table, the caller sees that characters were omitted; it cannot claim the displayed suffix or prefix is complete. Progress never carries these channels.

Character ceilings and UTF-8 serialized-byte ceilings are different. Unicode output can pass a character check and still consume more bytes. Both boundaries remain necessary.

The tool does not infer correctness from exit success. A program can run without exception and compute the wrong formula. For material calculations, the code, input identity, units, result, and independent check belong in the evidence record.

12.8. Progress without code disclosure

The tool module owns its progress presenter. Negotiated Pi cards can show requested, queued/started, execution phase, elapsed time, bounded output sizes, omission counts, and terminal category.

Cards do not include:

- Python source;
- stdout, stderr, or expression result;
- mounted text or source IDs;
- filenames or host paths; or
- exception details.

Progress is display-only and excluded from model context. It helps a user distinguish waiting from completion without creating a second data-exfiltration channel.

12.9. Numeric normalization and missingness

Computation pipelines need a rule for values before arithmetic. A finite numeric string such as "9", "7.5", or "1e-1" can be normalized deliberately when the surrounding schema permits numeric text. Empty strings, booleans, malformed numbers, infinities, and NaN reject.

Missing data remains missing. `null`, an absent cell, or a parser-level missing marker must not become zero merely because Python's aggregation is easier over numbers. A mean over [9, 7.5, null] has at least three possible policies:

- reject because the expected measurement is absent;
- compute over observed values and disclose $n = 2$; or
- return a missing aggregate.

The task contract must choose. Silently replacing `null` with 0 yields 5.5 and invents a measurement. The Northstar calculation has exactly two finite measured runtimes, so its accepted denominator is two.

This chapter's rule aligns with the presentation boundary: chart `null` renders as a gap or N/A, not zero. Data meaning must survive both computation and display.

12.10. Worked Northstar calculation

The synthetic workbook supplies measured runtime values for Prototype A and Prototype B:

```
measured_runtime_hours = [9.0, 7.5]
assert all(isinstance(value, (int, float)) for value in measured_runtime_hours)
mean_runtime_hours = sum(measured_runtime_hours) / len(measured_runtime_hours)
mean_runtime_hours
```

The deterministic result is 8.25. Its evidence label is **calculation from workbook measurements**, not product specification, image observation, or public guidance.

The complete research path is:

XLSX attachment

- shared bounded ingestion
- invocation-local canonical representation
- source ID selected for code_interpreter
- isolated worker copy
- Python calculation
- bounded result: 8.25 hours

The published recording shows the local interpreter card and the final evidence-labelled result. It does not expose generated code/output in progress, and it does not prove that every future model will write the same program. Deterministic tests separately execute known Python against synthetic inputs.

A robust task-specific check also records expected count, units, valid ranges, and missing-value policy. The interpreter supplies a computation mechanism; Chapter 13 decides whether the research claim has sufficient premises.

12.11. Current sandbox limitations

The current worker is runtime/process isolation, not operating-system isolation. Important limitations include:

- no independently enforced per-worker heap ceiling;
- termination depends on the host runtime and event-loop scheduling;
- CPU and memory pressure can affect the server before cancellation;
- Pyodide and the JavaScript host share one application process boundary;
- blocked imports and null-prototype globals are defense in depth, not a formal non-escape proof;
- the loopback server has no production multi-tenant entitlement layer; and
- no durable generated-artifact pipeline exists for this tool.

Default tests deny ordinary fetch at the Bun layer and exercise local Pyodide. That is test control, not an OS firewall.

Chapter 17 owns production isolation requirements: separate processes or containers, OS resource controls, egress policy, filesystem confinement, tenant scheduling, image/dependency patching, exploit testing, and incident response. Deferring those requirements does not make them optional; it prevents this chapter from pretending the current worker already satisfies them.

12.12. Evidence and testing

Credential-free tests cover fresh state, exact output truncation and omission counts, blocked JavaScript-bridge import, source-ID mounting, unknown-ID rejection, an actual read_source→mount→Python sequence, queue cancellation, non-cooperative timeout and worker termination, tool-owned instructions, registration, and progress privacy.

These tests establish the implemented local boundary. They do not establish adversarial sandbox strength, arbitrary package compatibility, semantic correctness of model-written programs, identical results across Pyodide revisions, or provider-hosted code-tool equivalence.

12.13. Failure modes

Model arithmetic presented as execution. Require the tool for structured computation and retain its result identity.

Path smuggling. Let generated code name `/Users/...` or a URL. Mount only server-selected invocation sources at fixed paths.

State leakage. Reuse one Python environment across callers. Construct and destroy a worker per admitted call.

Cancellation resurrection. A queued cancelled task starts later. Remove or reject it before worker construction.

Output-as-progress. Code or results leak through UI cards. Keep progress metadata-only.

Missing-as-zero. Null data silently changes an aggregate. Define and test missingness policy before arithmetic.

12.14. Pattern summary

Problem: Generated computation needs useful inputs and bounded execution without granting arbitrary host access.

Pattern: Create a fresh scheduled worker, mount only invocation-owned snapshots by opaque ID, terminate on deadline/cancellation, and fit every output channel explicitly.

Result: Both coordinator adapters can use one local computation tool whose inputs, state, and results are reviewable.

Cost: Startup overhead, process-wide contention, limited libraries, and incomplete hostile-code isolation remain.

12.15. Exercises

1. Compute a mean from `[9, null, 7.5]` under three explicit missing-value policies. Label each result.
2. Design a two-worker scheduler with cancellation and a memory-pressure gate. State which properties still require OS enforcement.
3. Add a task-specific check that rejects mixed units before calculating a Northstar mean.
4. Explain why identical Python source and mounted bytes do not by themselves establish identical results across runtime revisions.

12.16. Chapter contract

- **Scope:** fresh-worker construction, source mounts, Python boundary, scheduling, deadlines, cancellation, output fitting, progress, numeric normalization, and missingness.
- **Canonical claims:** local deterministic execution proves a result under stated inputs and runtime bounds, not input truth or production sandbox strength.
- **Migration inputs:** current Pyodide implementation, shared source registry, Northstar runtime calculation, and deterministic worker tests.

- **Evidence gates:** real local worker tests under network denial plus task-specific input/result checks.
- **Demonstration slots:** Northstar shows the 8.25-hour calculation path, not universal program correctness.
- **Source-claim ledger:** computation evidence records input representation IDs/digests where retained, code/runtime revision, units, missingness policy, and output limits.
- **Exclusions:** generic tool authorization belongs to Chapter 8; research support belongs to Chapter 13; production isolation belongs to Chapter 17.

13. Grounding, Citations, and Research Completion

Part III · Skills and the compositional research toolchain

Northstar deterministic trace + dated live workflow · reviewed reference snapshot 45a903c

Research is complete not when the model has written enough prose, but when the requested claims have the required kinds of evidence—or the remaining gaps are explicitly reported. The Northstar brief combines attached declarations, workbook measurements, generated image observations, deterministic calculations, and retrieved public guidance. Those evidence classes must never collapse into one generic “citation.”

Research completion is a task-specific decision that required questions have admitted evidence of the required class, applicable calculations and contradiction checks have run, unresolved gaps are disclosed, and the final answer preserves provenance and uncertainty. It is not a judgment that every sentence is universally true.

13.1. From tools to an evidence pipeline

Chapters 10–12 provide three different operations:

```
web_search
  discovers bounded candidate URLs and generated notes
    ↓ selection
read_source
  admits one screened, bounded source representation
    ↓ optional sourceId mount
code_interpreter
  computes over explicit invocation-local representations
    ↓
claim assembly, support checks, gaps, and completion decision
```

No earlier stage lends its authority to the next. A search candidate can fail retrieval. A successful retrieval can be irrelevant. A correct calculation can use the wrong input. A valid citation locator can point to a passage that does not support the claim.

The coordinator composes operations, but the server owns their limits and identity. The final answer must name what each operation established. A fluent synthesis is the product of this pipeline, not evidence that the pipeline completed.

13.2. Five Northstar evidence classes

The fictional Northstar task deliberately forces several evidence kinds into one short brief.

Evidence class	Example	What it can establish	What it cannot establish
Attached declaration	The supplied specification states a 72 Wh capacity and an 8–10 hour target	What the admitted document declares	Independent product truth, measured performance, or authenticity
Workbook measurement	The supplied XLSX represents Prototype A as 9 hours and B as 7.5 hours	Values in the admitted synthetic workbook	Real-world measurement integrity or general product performance
Generated image observation	The image model reports visible “NORTHSTAR” and Prototype A labels	Bounded model-generated observations about admitted pixels	Verified OCR, identity, provenance, authenticity, or truth
Deterministic calculation	Mean of 9 and 7.5 is 8.25 hours	Result under named values, formula, runtime, and missingness policy	Truth of the inputs or semantic applicability of the formula
Retrieved public guidance	Text obtained from selected official public HTTPS guidance	What the admitted representation contains at retrieval time	Universal applicability, future freshness, or automatic support for every answer claim

The labels belong in the answer because the user’s epistemic question differs by row. “The specification declares” is stronger than “the model guessed” in one respect and weaker than “independently measured” in another. A calculation can be exact while its premises remain fictional.

The Northstar fixture is intentionally synthetic. It is not third-party evidence, a real benchmark, or a product certification. The public guidance is real external material, but its presence does not make the fictional device real.

13.3. Identity, locator, and support are separate

A defensible citation relation has at least three independent questions.

13.3.1. 1. Which representation?

Source identity names the exact admitted object: invocation-local source ID, canonical URL or attachment identity, retrieval/admission time, media type, extraction revision, and digest where retained. A URL alone is an address, not a fixed representation. A filename alone is user-controlled text, not proof that bytes were received.

13.3.2. 2. Where in that representation?

A locator identifies a page, block, cell, row, character span, or other coordinate under a declared representation and unit. Locator validity asks whether the range exists and selects the quoted material. It does not ask whether the material is relevant.

Provider formats are not interchangeable. A PDF page range, an answer-text URL annotation, a custom block index, an XLSX row/cell, and a UTF-16 text range occupy different coordinate spaces. Converting among them requires retained parser/layout mappings; changing the integer label does not perform the conversion.

13.3.3. 3. Does it support this claim?

Support is a relation among a claim, one or more premises, applicable qualifiers, and an inference method. A correct locator can select an unrelated passage. One cited measurement does not support a comparison between two prototypes. A source may faithfully support “the document says X” while failing to support “X is true.”

Verify identity and location mechanically. Assess support under an explicit task method. Preserve authority, freshness, and permission independently. Never collapse those checks into one `citationValid` boolean.

13.4. Representation-bound locators

The book’s independently authored `evidence-locators.ts` demonstrates a narrow mechanical boundary for application-held text. It binds a source digest, answer digest, exact half-open UTF-16 ranges, and quote. It rejects malformed ranges, split surrogate pairs, changed source/answer content, and stale quotes. It escapes the rendered card and rechecks the link scheme.

That design proves that a selected occurrence exists in named text and that the answer revision has not changed. It deliberately calls its output `VerifiedLocator`, not `VerifiedClaim`. Repeated identical text still needs occurrence identity; equal quote strings at different offsets are different locators.

The adversarial fixture selects a well-formed instruction-bearing passage that is irrelevant to the answer. Mechanical verification succeeds. Support fails. This counterexample is more valuable than a happy-path citation because it prevents structural integrity from being promoted into semantic verification.

The implementation defines UTF-16 units for its own admitted text. It does not claim to convert OpenAI or Anthropic native citation offsets. Native provider annotations require their own dated contract and conversion tests.

13.5. Multi-premise support: the Northstar comparison

The deterministic comparison requires two premises:

- Prototype A has a measured runtime of 9 hours; and
- Prototype B has a measured runtime of 7.5 hours.

The premises must also share compatible units, measurement kind, freshness, and observation time. The conclusion is then computed from their numeric relation. Neither premise alone supports the comparison.

```
export function assessNorthstar(measurements: readonly Measurement[], nowMs: number):
  ↳ string {
  const prototypeA = measurements.find((item) => item.prototype === "A");
  const prototypeB = measurements.find((item) => item.prototype === "B");
  if (
    measurements.length !== 2 ||
    !prototypeA ||
    !prototypeB ||
    !Number.isSafeInteger(nowMs) ||
    nowMs < 0 ||
    measurements.some(
      (item) =>
        item.kind !== "measurement" ||
        item.unit !== "hours" ||
        !Number.isFinite(item.runtimeHours) ||
        item.runtimeHours < 0 ||
        item.runtimeHours > 168 ||
        !Number.isSafeInteger(item.measuredAtMs) ||
        item.measuredAtMs < 0 ||
        item.measuredAtMs > nowMs ||
        nowMs - item.measuredAtMs > 60_000,
    ) ||
    Math.abs(prototypeA.measuredAtMs - prototypeB.measuredAtMs) > 5_000
  )
    throw new TypeError("Comparison lacks compatible measurements.");
  const relation =
    prototypeA.runtimeHours > prototypeB.runtimeHours
      ? "longer than"
      : prototypeA.runtimeHours < prototypeB.runtimeHours
        ? "shorter than"
        : "the same as";
  return `In the synthetic workbook, Prototype A's measured runtime is ${relation}
  ↳ Prototype B's (${prototypeA.runtimeHours} hours versus
  ↳ ${prototypeB.runtimeHours} hours).`;
}
```

The accepted answer is scoped:

In the synthetic workbook, Prototype A's measured runtime is longer than Prototype B's (9 hours versus 7.5 hours).

“In the synthetic workbook” and “measured” are part of the claim. Removing either widens what the evidence appears to establish. The task-specific assessor rejects missing prototypes, declarations substituted for measurements, non-finite values, stale/future observations, incompatible timestamps, mixed units, and a contradictory proposed answer.

This is exact support logic for one synthetic two-premise task. It is **not** a general natural-language entailment verifier. It does not prove that arbitrary passages support arbitrary prose, infer hidden premises, resolve scientific disagreement, or assess source authority.

13.6. Calculations preserve upstream provenance

The mean runtime, 8.25 hours, is deterministic over [9, 7.5]. Its record should retain:

- both workbook measurement identities;
- the accepted count and units;
- the missing-value rule;
- the formula or reviewed code;
- interpreter/runtime revision; and
- bounded output.

The calculation derives from the measurements; it does not replace them. If one cell is missing, silently inserting zero would create a different claim. If one value is a specification target rather than a measurement, averaging them would cross evidence classes.

This derivation also clarifies citation display. A calculated sentence may cite the workbook premises and identify the calculation method. It should not cite a public battery-guidance URL as if that page supplied the synthetic runtime values.

13.7. Generated image observations remain generated

The image bridge can be called by an Anthropic coordinator while a fixed OpenAI image route receives the admitted image and focused question. Its result includes `generated: true` and explicit recipient metadata.

A final answer must retain those facts. “The attached image visibly contains...” is still a model-generated observation unless an independent deterministic inspection established the text. It is not complete OCR, coordinate-preserving extraction, source identity proof, or authenticity verification.

Generated observations can guide the brief—for example, to describe a visible prototype label—but they should not become workbook measurements or attached declarations. If the observation conflicts with canonical extracted text or source metadata, the conflict is a research finding, not permission to choose the more convenient value silently.

13.8. Retrieved guidance: provenance without automatic truth

A successful `read_source` call supplies a representation and URL provenance. The answer can state that the retrieved official page says or recommends something within the scope actually read. It cannot infer that:

- the page remains current after retrieval;
- the guidance applies to every jurisdiction or device;
- the selected representation contains the complete publication;
- search notes are exact quotations;
- publisher authority makes every claim true; or
- a clickable URL proves the sentence follows from the page.

For recall-sensitive requests—such as “list all requirements in this document”—the representation must be sufficiently complete for the requested scope, and computation/search over the full canonical text may be necessary. A landing-page failure cannot support absence. The coordinator should seek an alternate admitted full-text representation or disclose the gap.

Retrieved instructions remain untrusted data. A relevant official page still cannot alter tools, recipients, credentials, quotas, or application policy.

13.9. When is the research run complete?

Completion begins with a claim plan, not a prose-length threshold. For each requested output, define:

1. **required claim or decision**—what must the answer establish;
2. **required evidence class**—declaration, measurement, generated observation, calculation, retrieved guidance, or another named class;
3. **scope and freshness**—which documents, dates, prototypes, jurisdictions, or versions count;
4. **operation requirement**—search, read, compute, image inspection, or no external operation;
5. **support method**—exact task rule, human review, documented evaluator, or explicitly unresolved judgment; and
6. **failure behavior**—abstain, narrow the claim, present conflict, or return an evidence gap.

A Northstar research run is complete when:

- every admitted attachment has a known role and successful/failed status;
- specification facts are labelled declarations;
- workbook values are labelled measurements represented in the fixture;
- the requested mean has run over the admitted values with units and missingness checked;
- image observations carry generated status and recipient disclosure;
- public guidance was discovered and the selected representation was actually read;
- required comparisons have all premises and pass task-specific checks;
- conflicts, unavailable evidence, truncation, and uncertainty are visible;
- citations identify only retained/admitted provenance and valid locators; and
- the final answer makes no broader claims than those records support.

A run can also finish honestly with **insufficient evidence**. Completion is a terminal research decision, not synonymous with a positive answer. Spending all call budget is neither necessary nor sufficient.

13.10. A completion matrix

A practical implementation can maintain a bounded matrix:

Required output	Evidence required	Current state	Completion action
Capacity/target	Attached specification declaration	Admitted	Report as declaration

Required output	Evidence required	Current state	Completion action
A/B runtime comparison	Two compatible workbook measurements	Admitted and checked	Report scoped comparison
Mean runtime	Two finite hour values + code execution	Computed as 8.25	Report derivation
Visible product label	Generated image observation + recipient	Completed	Report as generated/unverified
Public guidance	Search candidate + successful read	Completed for selected URL	Report representation/date scope
Real product status	Independent external evidence	Absent	State fictional/unverified

The matrix is not itself truth. It is a control structure that prevents the answer from forgetting an evidence requirement. Its state must come from server-owned operation results, not model assertions that a step “was done.”

13.11. Editing invalidates relations

Citation and support relations bind to an answer revision. Inserting “not,” removing “synthetic,” changing a unit, or widening “the workbook reports” to “the device achieves” can invalidate support even when source offsets do not move.

The deterministic trace hashes the answer and rejects an old locator draft after editing. A production system should similarly republish the authoritative answer and evidence relations together. Streamed provisional text can show references for usability, but the final revision must reconcile them. Do not claim the user never saw unsupported text merely because the terminal snapshot was corrected.

13.12. Local-file citations and ephemeral identity

The current attachment system is invocation-local. An admitted `src_...` ID expires when the HTTP request ends. The original local path is not a durable server identity, and attachments must be resent on later turns. Pi session text can mention a filename without carrying the original bytes.

Shared ingestion retains canonical text for the invocation, but formal page/span mappings are incomplete across every format and extractor profile. A PDF page number may survive one extraction path; XLSX evidence may need sheet/row/cell coordinates; OCR may not preserve stable layout coordinates. Therefore the system must not invent file citations or page locators from model memory.

Current honest options are:

- cite the attached document descriptively within the same run, with its evidence class;
- retain verified representation identity and exact locators when the extractor supplies them;

- provide the local model-independent inspection report as separate extraction evidence; or
- state that durable formal local-file citation is unavailable.

A future durable source service needs authenticated ownership, retained bytes/representations, stable extraction revision, page/span mappings, deletion and retention policy, and cross-request source identity. Reusing the same filename does not satisfy those requirements.

13.13. Conflict and absence

Conflicting evidence remains conflict. A target in the specification and a measured value in the workbook are not contradictory merely because they differ; they have different kinds. Two measurements under ostensibly identical conditions that disagree require qualifiers or investigation.

Absence claims require scope. “The selected representation did not contain X under this complete-text search” is narrower than “the publisher never says X.” A failed read supports neither. Search failing to return a page does not prove nonexistence.

Research completion should record unresolved contradictions as first-class outcomes. It must not average incompatible evidence, choose the latest-looking snippet, or conceal a gap behind a generic caveat.

13.14. Evaluation without entailment overclaiming

Citation evaluation can measure structural validity, required-claim coverage, citation relevance, contradiction handling, source suitability, freshness disclosure, and uncertainty behavior. Automated natural-language-inference models can screen candidate relations, but their scores are measurements under a named evaluator—not deductive proof.

For Northstar’s numeric comparison, task-specific computation is stronger and simpler than general NLI. For public guidance, human review or a documented claim-evidence rubric may be needed. Use the narrowest reliable method for each claim type rather than assigning one universal “grounded” score.

A second model cannot certify its own judgment merely by returning confidence. Record evaluator/version, sample, threshold, and limitations whenever model-based support assessment influences publication.

13.15. Security and privacy

Evidence handling crosses disclosure, parser, instruction, and rendering boundaries. Search queries can leak private facts before retrieval. Retrieved pages can contain prompt injection. Citation links can expose sensitive query strings or document membership. Digests can be sensitive and are not anonymization.

Enforce source authorization before context insertion, execution authorization outside the model, and safe rendering at publication. A source never grants tool authority. A public answer must not reveal confidential evidence through its title, URL, quote, locator, or derived calculation.

13.16. How we test completion

The deterministic Northstar tests exercise:

- required capability and authorization gates before execution;
- two correlated measurements with compatible units and times;
- missing, stale, future, non-finite, wrong-kind, and contradictory cases;
- transcript reopen without redispach;
- unknown-effect/no-repeat recovery;
- source snapshots and occurrence-preserving locators; and
- answer-edit invalidation.

Separate reference tests cover search receipt/budget mechanics, public-source admission, source-ID mounting, interpreter execution, image recipient disclosure, and progress privacy. The published live workflow demonstrates one end-to-end result with dated provider/network metadata. None of these is a general natural-language entailment benchmark.

13.17. Failure modes

Discovery completion. Search returns URLs, so research stops. Read the selected source and evaluate support.

Evidence-class collapse. A declaration, measurement, observation, and calculation are all called “facts.” Preserve their labels and derivations.

One-premise comparison. One runtime is cited for a two-runtime claim. Require the complete premise set.

Locator certification. A valid range is treated as semantic support. Test the claim relation separately.

Filename citation. A remembered local name becomes a formal citation. Require retained representation identity and locators.

Completeness by verbosity. A long answer is considered finished despite a missing required operation. Use a claim/evidence matrix.

13.18. Pattern summary

Problem: Tool success and citation-like output can hide incomplete or unsupported research.

Pattern: Preserve evidence classes, connect discovery to admitted representations and computation, separate identity/locator/support, and decide completion against a task-specific claim plan.

Result: The answer can explain what was established, how, from which representation, and what remains unknown.

Cost: Evidence bookkeeping, task-specific support methods, conflict handling, revision-aware citations, and honest abstention remain necessary.

13.19. Exercises

1. Add a third Northstar runtime observation with an incompatible timestamp. Specify whether the comparison, mean, both, or neither can complete.
2. Create a locator that is structurally valid but irrelevant. Define the separate support decision.
3. Design durable local-file identity for a resent PDF without trusting its filename.
4. Write a completion matrix for a recall-sensitive inventory from a retrieved document. Include failed landing-page behavior.
5. Edit the accepted answer by removing “synthetic workbook.” Explain which evidence relation must be re-evaluated.

13.20. Chapter contract

- **Scope:** evidence classes, source identity, locator validity, multi-premise support, derivation, and research-completion criteria.
- **Canonical claims:** identity, locator validity, and claim support are separate; completion is task-specific and can end with an explicit evidence gap.
- **Migration inputs:** deterministic Northstar composition, locator helper/tests, current research tools, and the dated end-to-end demonstration.
- **Evidence gates:** admitted operation receipts, representation identity, task-specific support tests, revision binding, and explicit limitations.
- **Demonstration slots:** Northstar supports the five-class pipeline and two-premise trace; it does not establish general entailment verification.
- **Source-claim ledger:** every published claim records evidence class, representation/time, locator method where available, support method, and unresolved limits.
- **Exclusions:** discovery belongs to Chapter 10; transport/representation admission belongs to Chapter 11; computation mechanics belong to Chapter 12.

Part IV.

Part IV — Portability in Production

14. Capabilities, Fallbacks, and Mixed-Provider Workflows

Part IV · Portability in production

Deterministic admission tests + dated Northstar mixed-provider workflow · reviewed reference snapshot 45a903c

The preferred Northstar coordinator is unavailable. A second model can produce text, and its client has the same interface. Should the application switch? Only if that switch preserves the work the application promised. Returning a fluent brief without the required tools and evidence is not graceful degradation; it is a different task disguised as success.

A **capability promise** is a scoped assertion that a deployment can perform an operation under named conditions. **Fallback** selects another execution target; **degradation** explicitly reduces a task's contract. Neither operation is justified merely by accepting the same request type.

14.1. The problem

[Chapter 5](#) bound model construction to trusted deployment configuration. We now need a decision *before* sending a request: does this binding meet the task's requirements? A constructor can succeed while a model lacks the required feature, the credential cannot access it, or the selected endpoint rejects the combination of settings.

Our synthetic Northstar comparison requires two runtime measurements obtained through a controlled tool. A fallback model that can only generate text may summarize measurements already supplied by a trusted upstream stage. It cannot satisfy a requirement to obtain those measurements itself. If the application changes from tool-assisted comparison to a clearly labelled explanation of what data is missing, the reduction in service must be visible in its outcome.

This is a task contract problem, not a larger switch statement. We must separate what the application requests, what the adapter declares, what a test observed, and what the product requires. The consequence of confusing them is often semantic: the HTTP request works and the answer is plausible, but the promised operation never occurred.

14.2. Context and forces

Availability without authority drift. A second provider can improve availability, but changing recipients, regions, tools, or principals also changes the security review. An outage is not permission to export the same data elsewhere.

Evidence that ages. A passing probe describes one configuration at one time. An adapter update, changed account, revised policy, or changed task profile can invalidate its use without changing the model's name.

Combinations, not catalogues. Tools, streaming, structured output, citations, reasoning, and modalities interact. Independent positive feature checks do not prove their simultaneous use works.

Quality beyond syntax. A valid JSON object can contain the wrong city, a well-formed tool call can select the wrong operation, and a cited answer can contradict its source. Admission establishes necessary conditions, not the complete semantic result.

Reading rule. Ask which evidence authorizes this capability for this deployment, task stage, and policy revision—not whether a provider is generally described as supporting it.

14.3. Four different capability statements

Statement	Owner	Example	What it proves
Requested	Request construction	Tools are included in this call	The caller expressed an intention
Declared	Adapter configuration	This model is configured as tool-capable	A configured assertion
Observed	Test or operational measurement	A named fixture completed a correlated tool round trip	That observation, under its recorded conditions
Required	Application task policy	Fresh observations must precede comparison	A condition for admitting or accepting the task

The tagged `CapabilityResolver` returns an array of text, streaming, and tools for a model ID. `declaredCapabilities(modelId)` makes a new set from it. By default all three are declared. The `model-scoped fixture` changes the declaration for text-only and tool-model. It proves the configuration can vary by model; it neither contacts a provider nor enforces admission inside `streamModel`.

Part I’s book guard demonstrates rejection against declarations. For a trusted deterministic fixture, that is a useful admission rule. For a deployed service, choosing a stronger evidence requirement may be appropriate. Do not retrospectively relabel a declaration as a successful live probe merely because the model has generated some text.

MCP illustrates a different mechanism. Its pinned lifecycle exchanges protocol versions and client/server capabilities during initialization; operation must respect successfully negotiated capabilities (“Model Context Protocol 2025-11-25,” n.d.). That is actual protocol negotiation. The reference model port has no equivalent initialization handshake. Applying the *idea* of explicit capabilities does not make provider model features negotiated MCP capabilities, nor does an MCP server’s tools declaration prove that a model will choose its tools correctly.

14.4. A promise needs a scope

A useful evidence key includes deployment identity, provider model ID, adapter revision, task profile, and policy revision. The task profile represents the relevant combination of features and parameters, not just a marketing category. A profile might name a tool-assisted Northstar evidence brief with a particular schema set and generation budget.

This key is a design choice, not a standardized model capability format. An implementation may also need endpoint, account, region, dependency lock digest, model snapshot, prompt revision, and tool schema digests. The deployment identifier in this chapter assumes those construction details are managed elsewhere. If it does not change when those details change, the evidence key is incomplete.

```

Task requirements + permitted degradation
    ?
Trusted deployment — scoped declaration/observation records
    ?
    ?
    Admission decision
    accept / explicit degradation / reject
    ?
    ?
    Execution and outcome evaluation

```

Expiry is an evidence reuse rule, not a guarantee that a model cannot change before that instant. Positive evidence can be revoked early by an incident or configuration change. Negative evidence can also be context-specific: a transient outage is not proof that a feature is permanently unsupported. Preserve the distinction between “not demonstrated for this request” and “known unsupported.”

A practical registry should keep the evidence reference and method, not just the result. Who ran the check? Was it a SDK seam mock, live protocol exchange, or task-quality experiment? What was the acceptance threshold? A trusted assertion may be adequate for a low-risk preview while an irreversible tool-bearing task demands stronger proof. These are policy decisions, not values supplied by a model-generated capability record.

14.4.1. Declaration is not dated evidence

The current reference has two deliberately different records. Static route configuration declares which adapter and feature profile may be constructed. A live demonstration record names the exact server revision, date, coordinator route, secondary recipients, operation sequence, and observed outcome. The declaration can authorize attempting a task; the dated record can support only the observed compatibility claim under its recorded conditions.

Do not write “model X supports image input” when the implemented path actually masks image bytes from the coordinator and sends one selected image to a separate OpenAI image route. Do not convert a deterministic synthetic transport test into a live-account claim. Conversely, a successful live run does not make the capability permanent: account access, endpoint policy, model behavior, dependencies, and tool configuration can change.

For example, the [local evidence-method record](#) resolves `local-composition-fixture-v1` to `scripted-local-composition`, `profile two-measurements-v1`, the exact five-field scope, and

named source/tests. A live-wire method would instead need account/endpoint class, request combination, producing revision, and observed terminal result; a task-quality method would additionally name the sample and acceptance rule. All can be observations, but they are not interchangeable evidence. The helper leaves method lookup to its trusted caller; its observed enum does not encode a production assurance level. In the composition fixture, the positive feature records are **supplied synthetic policy inputs**, not feature discovery or measurements of model/streaming performance. The observed property is the local policy transition and its task consequences.

14.5. An executable admission policy

The following **implemented and deterministically tested book design** is intentionally stricter than the reference declaration guard. Its types and scope comparison are in `code/capability-admission.ts`. The input evidence is trusted, already authenticated application data; this is not a decoder for untrusted JSON.

AdmissionRequest separates required features from optional features and selects a minimum evidence kind. Each evidence record contains a scope, feature, declaration/observation kind, boolean assertion, check/expiry times, and evidence ID. The book feature vocabulary includes citations and json-schema; these are **not** additions to the tagged reference's capability union.

```
export function admitCapabilities(
  request: AdmissionRequest,
  evidence: readonly CapabilityEvidence[],
  now: number,
): { readonly mode: "accept" | "degrade" | "reject"; readonly missing: readonly
  ↪ Feature[] } {
  if (!Number.isSafeInteger(now) || now < 0) throw new TypeError("Invalid admission
  ↪ clock.");
  for (const record of evidence) {
    if (
      !Number.isSafeInteger(record.checkedAt) ||
      record.checkedAt < 0 ||
      !Number.isSafeInteger(record.expiresAt) ||
      record.expiresAt <= record.checkedAt ||
      record.evidenceId.trim() === ""
    ) {
      throw new TypeError("Invalid capability evidence.");
    }
  }
  const fresh = evidence.filter(
    (record) =>
      sameScope(record.scope, request.scope) && record.checkedAt <= now && now <
      ↪ record.expiresAt,
  );
  const supports = (feature: Feature): boolean => {
    const matching = fresh.filter((record) => record.feature === feature);
    if (matching.some((record) => !record.supported)) return false;
    return matching.some(
      (record) =>
        record.supported && (request.minimum === "declared" || record.kind ===
        ↪ "observed"),
    );
  };
  const requiredMissing = [...new Set(request.required)].filter((feature) =>
    ↪ !supports(feature));
```

```

    if (requiredMissing.length > 0) return { mode: "reject", missing: requiredMissing };
    const optionalMissing = [...new Set(request.optional)].filter((feature) =>
      ↪ !supports(feature));
    return { mode: optionalMissing.length > 0 ? "degrade" : "accept", missing:
      ↪ optionalMissing };
  }

export function mayFallback(
  reason: "unavailable" | "refusal" | "policy" | "authentication",
  outputStarted: boolean,
): boolean {
  return reason === "unavailable" && !outputStarted;
}

```

The half-open interval `checkedAt <= now < expiresAt` excludes observations from the future and treats the expiry instant as expired. Evidence for another scope cannot satisfy this request. Any fresh negative assertion vetoes the feature, even alongside a positive assertion. This conservative conflict rule avoids depending on input order; an operational registry needs a reviewed resolution process rather than accumulating contradictory observations forever.

The function returns feature names lacking adequate support. It does not assert that every missing feature is unsupported. Required gaps reject admission; optional gaps produce an explicit degradation result. The caller must map that result to an approved task variant, disclosure, and output contract before executing. A return value is not itself consent, authorization, or routing.

In the [cross-chapter Northstar trace](#), missing tools rejects before database creation or tool execution: both counters remain zero. Missing only optional streaming selects a disclosed non-streaming comparison; it does **not** remove either observation requirement. A separate current-authorization denial still prevents all execution after admission succeeds. Those checks make the distinction between a capable target and an authorized operation observable.

A capability record admits a named operation under named conditions. It neither proves every feature combination nor authorizes the effects of executing that operation.

The helper does not launch probes, choose TTLs, authenticate records, refresh caches, enforce an account allowlist, or perform a provider call. Its synchronous scan is sufficient for small fixture registries; production bounds on registry size and request feature count remain necessary. The injected clock makes expiry tests deterministic and keeps time acquisition outside policy. In a distributed deployment, acceptable clock uncertainty must be specified rather than assumed away.

14.6. The implemented mixed-provider workflow

The reviewed Northstar run demonstrates composition across recipients rather than a monolithic provider feature set:

```

Pi request
  → portable-agents/anthropic/claude-opus-5-5 coordinator
    → local bounded Pyodide calculation
    → OpenAI gpt-5.4 image-understanding recipient
    → OpenAI gpt-5.4 hosted-search recipient
    → bounded public-HTTPS source reader
    → local bounded PptxGenJS presentation worker

```

The coordinator receives ordered attachment markers rather than the original image bytes. If it proposes `image_understanding`, the server sends only the selected admitted image plus a focused question to the fixed OpenAI route. The result names `{ provider: "openai", model: "gpt-5.4" }` and marks observations as generated. Search similarly sends a focused query to a separate configured OpenAI search model, not the whole coordinator transcript.

This is a mixed-provider workflow because coordination and selected sub-operations have different recipients. It is not automatic provider fallback. Each recipient is statically reviewed, independently authorized, bounded, and disclosed. An Anthropic coordinator does not imply that Anthropic inspected the image or performed hosted search. A common application tool schema does not imply that OpenAI and Anthropic receive the same bytes, implement the same internals, or produce semantically equivalent results.

The dated recording metadata identifies the coordinator, each secondary recipient, exact server commit, Pi and Bun versions, review status, and limitations. It proves that one approved run visibly exercised those operations. Deterministic tests separately prove masking, fixed-recipient routing, recipient metadata, and progress non-disclosure with synthetic transport.

14.6.1. Recipient disclosure is part of admission

Routing policy must answer two questions before a secondary model receives data:

1. **Is this recipient authorized for this data class and purpose?**
2. **Can the user and evidence record tell which recipient actually processed it?**

A secondary operation should receive the minimum data needed: one image and question, or one focused search query—not the entire conversation by convenience. Its result should preserve generated status and recipient identity. Progress may name the reviewed recipient class without exposing payload. Logs and public errors remain bounded.

A user choosing an Anthropic coordinator has not necessarily consented to an OpenAI sub-operation. The product must disclose and authorize that route rather than rely on the coordinator label. Residency, retention, contractual permission, and cost can differ even when both operations share a TypeScript interface.

14.6.2. Evidence for a provider switch

A provider-switch claim requires an observable route change. A recording that runs two similar calculations without showing the selected `portable-agents` route does not prove a switch. Acceptable evidence should visibly show the model selection or route transition, preserve server-owned tool cards after the transition, and record both coordinator recipients in metadata.

No such recording should be published merely to fill a demonstration slot. Until an approved artifact visibly proves the switch, the chapter relies on deterministic route tests and the Northstar mixed-provider recording's single Anthropic-coordinator path. That path proves mixed recipients; it does not prove live switching from one coordinator to another during the session.

14.7. Fallback task profiles

Fallback begins with a named task profile, not a generic requirement that the replacement “supports tools.” Useful Northstar profiles include:

Profile	Required operations	Permitted reduction
northstar-research-current-v1	attachments, fixed image recipient, search, read, local computation	none for a completed evidence brief
northstar-existing-evidence-v1	already admitted declarations/measurements and local computation	no fresh public-guidance claim
northstar-text-only-gap-v1	text generation only	explain unavailable evidence; no research completion claim
northstar-presentation-v1	reviewed skill plus separately authorized builder	return research answer without deck if build is optional and disclosed

The profile fixes required evidence classes, recipients, tools, freshness, and terminal disclosure. A fallback candidate must satisfy that exact profile or move to an explicitly approved reduced profile. Deleting a failed capability from the requirement list after routing has begun is not degradation policy; it is an unreviewed task rewrite.

14.8. Fallback is a policy transition

A fallback candidate must pass the same hard constraints as the original candidate: data destination, authorization, task requirements, remaining deadlines, and appropriate evidence. It is not enough to remove the capability that caused rejection.

Consider three Northstar outcomes:

1. **Equivalent admitted execution:** another authorized tool-capable deployment obtains the same required observations under the same tool policy. Its eventual answer still needs task evaluation.
2. **Explicit task reduction:** the service returns an unavailable-data explanation, or summarizes previously verified observations with their time clearly disclosed. The application labels the changed task and does not claim fresh retrieval.
3. **Rejected substitution:** the candidate lacks required tools or would send data to an unapproved destination. The application returns a bounded failure rather than fabricate a successful comparison.

The `mayFallback` example deliberately allows only an availability failure before visible output. This guard is necessary but not sufficient: it does not inspect effect history, debit a budget, or authorize a destination. A true result still needs the [recovery gate in Chapter 15](#). It rejects policy, authentication, and refusal reasons. An operator can repair credentials and start an authorized attempt; an automated fallback should not reinterpret an access denial as a search for a provider that accepts the forbidden operation. Nor should a refusal be routed around to weaken safety protections.

Once output is visible, switching providers introduces an attempt-boundary problem. The second answer cannot safely be appended to the first as if both were one response. A product may support an explicit replacement attempt with a new identity and a notice that the earlier attempt was abandoned. That is a designed recovery protocol, not implemented by this helper; [Chapter 15](#) develops the commitment boundary; it is not assumed to exist in the reference.

Fallback also needs a finite search space. A policy should bound candidates and total attempts, debit a shared deadline/cost budget, and record which candidate was rejected for which safe reason. Independent retries at the model, router, gateway, and client can turn a small candidate set into a large call fan-out. “Try another model” is not a budget policy.

14.9. Provider-specific variation: feature conjunctions

As rechecked in the 2026-10-08-09 review, OpenAI’s *Structured model outputs* guide distinguishes structured function arguments from structured response text (“Function calling vs response format”) and schema adherence from JSON mode (“Structured Outputs vs JSON mode”) (OpenAI, n.d.-d). Therefore a generic `json` capability is too imprecise for admission: it can mean syntactically valid JSON, constrained output, locally validated output, or strictly validated function arguments.

Anthropic’s guide, “Feature compatibility,” documents a concrete interaction: JSON outputs using `output_config.format` are incompatible with citations, which require interleaving citation information with text; enabling that combination returns a 400 error. Its “Invalid outputs” section describes refusal and token-limit cases where the desired schema may not be delivered (Anthropic, n.d.-d). The [claim register](#) records the specific sections and retrieval date. This is dated, mutable documentation evidence—not an execution result for the reference’s pinned packages.

This is why a feature vector alone is insufficient. If both `citations` and `json-schema` are declared true in isolation, the conjunction can still fail. Our task-profile scope must represent the actual combination being asserted. A production implementation should encode constraints directly or use profiles whose tests exercise the combined request. The book helper trusts the profile; it does not infer interaction constraints from provider names.

Neither structured-output configuration nor native citations are exposed in the tagged `Model-Request`. For this reference, those requirements would need a reviewed contract extension or a separate stage. Do not promise them merely because the underlying SDK has a larger option surface.

14.10. Measure the semantic gap

NIST AI 600-1 recommends setting performance/assurance thresholds in deployment approval (GV-1.3-002) and avoiding extrapolation from narrow, non-systematic, anecdotal assessments (MS-2.5-001) (National Institute of Standards and Technology 2024). It is voluntary risk-management guidance, not a portability certification. Its useful architectural lesson is to state the measurement context and the decision it supports.

The historical tagged [semantic evaluation script](#) contains three synthetic exact-match tasks: a comparison from supplied facts, ordered name extraction, and ignoring a bracketed untrusted instruction. Its retained [test-sequence summary](#) is a sanitized aggregate compiled from owner-console observations. It records 12 passing attempts and explicitly withholds broad semantic

portability; it is not the script's complete machine-readable per-attempt report. The tasks do not execute external tools, measure citation entailment, or test a deployment's response to a realistic adversarial distribution.

Exact match is appropriate when the task demands one fixed serialization. It can be a poor proxy for ordinary answer quality: a correct paraphrase fails, while a memorized short answer may pass without exercising the intended workflow. Define the construct being measured before selecting a metric.

For a replacement Northstar coordinator, a **proposed evaluation protocol** separates:

Gate	Example measurement	Acceptance role
Authority	No call outside authorized tools or destinations	Hard rejection, not averaged into quality
Procedure	Required observations obtained and correlated	Task contract
Factual result	Correct runtime comparison with units and measurement times	Semantic correctness
Abstention	Missing or conflicting observations are disclosed	Uncertainty behavior
Presentation	No partial result represented as complete	Client contract
Resources	Latency and measured usage under the same bounded scenario	Operational suitability

Use paired scenarios, documented model/configuration versions, repeated attempts, and disaggregated failures. Include unavailable tools, ambiguous city names, conflicting observations, stale data, and malicious tool text. Specify the sample construction and thresholds before inspecting the result. Do not collapse an authorization failure into an average quality score that can be offset by good prose.

A provider switch need not produce identical wording, tokenization, latency, or price. More importantly, a shared request type and successful terminal response do **not** establish semantic equivalence. Portability means preserving required meanings and constraints while measuring permitted variation. Unknown usage stays unknown; estimates and provider measurements must not be mixed into one supposedly observed cost figure.

14.11. Failure modes

Permanent booleans. A `tools:` `true` value without provenance becomes a stale promise after endpoint or policy changes. Invalidate on construction changes and define refresh or revocation behavior.

Optional by accident. Moving a hard requirement into the optional list merely to make a candidate pass is an undeclared product change. Review the reduced task separately.

Probe mismatch. A text probe does not prove tool continuation; a tool mock does not prove wire compatibility; a tiny exact-match test does not prove broad safety. Match evidence to the gate it exercises.

Fallback as laundering. A new provider must not acquire wider tool authority or receive data forbidden to the first. Keep policy identity stable while reviewing the new recipient.

Confidence from clean syntax. Schema-compliant output can still be factually wrong. Keep semantic verification outside the syntax gate.

14.12. Security considerations

Capability records influence execution and therefore belong to the control plane. A model, fetched document, or untrusted tool result must not declare itself capable and thereby enable a privileged operation. Verify record provenance, bind records to reviewed deployment identity, and authorize registry updates. An evidence ID is only a lookup label in this example; it is not a signature.

A fallback routing rule is also a data-flow rule. Do not assume that two deployments share retention, residency, or contractual permissions because their interfaces match. Avoid probing with sensitive prompts or privileged tools; use synthetic, bounded probes whenever possible. Account availability probes can themselves consume resources and expose information about deployment membership.

Conflict resolution and expiry should fail closed for required operations but return useful safe reasons. Public responses can identify an unmet requirement without disclosing internal endpoints, credentials, or the complete deployment registry.

14.13. Telemetry contract

Proposed fields include task/profile revision, selected deployment class, admission mode, unmet feature names, evidence method/age bucket, safe fallback reason, attempt count, and remaining budget. Evidence IDs and deployment identifiers may require redaction or controlled cardinality. Do not log prompts merely to explain a capability rejection.

Record degradation as a first-class task outcome, not as ordinary successful completion with a hidden flag. Separate admission rejection, provider failure, and task-quality failure; otherwise a dashboard can report excellent model uptime while required workflows never execute. The helper returns decisions; tracing their operational use belongs to its caller.

14.14. How we test portability

Seven network-free book tests cover fresh observed conjunctions, declaration-versus-observation policy, exact scope matching across all five fields, expiry/future times, order-independent conflict rejection, explicit optional degradation, malformed time/evidence fields, and refusal/partial-output fallback prevention. The fixtures use synthetic evidence; they do not certify the truth of a real capability assertion.

The reference's model-scoped declaration fixture and retained 12-attempt evaluation remain separate proofs. The local composition additionally checks admission rejection, approved degradation, and authorization independently. No new paid provider probes were run. The [reader archive](#) runs these book-only checks with `bun run test`; from the authoring root:

```
bun test book/test/capability-admission.test.ts
bun run check
```

14.15. Portability and evidence ledger

Dimension	Evidence label and scope	Limitation
Transport	Not tested here: endpoint/account availability	Declaration and admission do not contact a wire endpoint
Provider	Implemented: model-scoped declarations in v0.3.0	Not live discovery; book admission remains separate
Protocol	Specified: MCP initialization negotiation	Not negotiation of model features
Feature	Implemented/tested: scoped book admission and explicit degradation	Trusts evidence/profile construction; no operational registry
Semantic	Locally tested: required observations survive optional degradation; retained exact-match aggregate is separate	No live substitution equivalence or broad safety assessment

Additional policy concern. The refusal/visible-output fallback guard is tested, but does not implement destination authorization, effect accounting, or a complete budgeted router.

14.16. Pattern summary

Problem: A replaceable interface can conceal an unsupported feature or a changed task.

Pattern: Define hard requirements and approved optional reductions, scope capability evidence to deployment and task configuration, invalidate stale claims, and admit each fallback under the same policy constraints.

Result: Substitution becomes an explicit, inspectable task decision rather than a silent downgrade.

Cost: Evidence maintenance, interaction testing, bounded candidate selection, and task-level evaluation remain necessary.

14.17. Exercises

1. Define two Northstar task profiles: obtaining admitted measurements and summarizing verified existing measurements. Explain when their required features differ without weakening the user's promise.
2. Add a trusted early-revocation mechanism to the evidence design. Test a positive observation revoked before its expiry.
3. Design a joint citations/structured-output profile without assuming independent feature booleans compose. Identify which provider contract must be checked.
4. Write a paired replacement rubric with one non-negotiable authority gate and two quality thresholds. Specify how a tool-policy violation affects the decision.
5. Describe how an operator can repair an authentication failure without turning the automated fallback mechanism into an authorization bypass.

14.18. Further reading

Compare actual negotiated capabilities in MCP (“Model Context Protocol 2025-11-25,” n.d.) with documented provider feature constraints (OpenAI, n.d.-d; Anthropic, n.d.-d). Read NIST AI 600-1’s deployment-threshold and measurement-limit actions (National Institute of Standards and Technology 2024) alongside the reference evaluation script. None substitutes for evaluating the application’s own task distribution.

14.19. Chapter contract

- **Scope:** capability evidence, explicit degradation, fallback, and mixed-provider workflows.
- **Canonical claims:** availability and interface compatibility are necessary but insufficient for substitution.
- **Migration inputs:** former capabilities-and-fallbacks chapter.
- **Evidence gates:** scoped feature records and task-level semantic evaluation.
- **Demonstration slots:** deterministic fallback admission trace and the dated Northstar mixed-recipient workflow; no coordinator-switch recording is claimed until the route change is visibly proved.
- **Source-claim ledger:** observed, declared, and inferred capability evidence remain distinct.
- **Exclusions:** adapter mechanics belong to Chapter 5; retry policy belongs to Chapter 15.

15. Errors, Retries, Recovery, and Durable State

Part IV · Portability in production

Deterministic recovery, transcript, replay, and Northstar composition tests · historical defects remain revision-scoped

The measurement operation times out. The agent has already announced that it is reading evidence, and the executor may already have processed the request. A retry can restore availability—or duplicate a consequential operation. Recovery starts by asking what is known about the previous attempt, not by choosing a backoff interval.

Recovery is a policy-controlled transition from a failed or interrupted operation to a trustworthy next state. **Retry** repeats an operation. **Reconciliation** determines what happened to an operation whose outcome is uncertain. Cancellation stops requested work where cooperation permits; it is not proof that its effects were undone.

15.1. The recovery problem

[Chapter 14](#) admitted a deployment under an explicit task contract. Execution now introduces time and uncertainty. A model request can fail before transmission, after provider acceptance, after a tool executes, or after the client observes partial output. Those cases share an exception mechanism but not a safe next action.

The Northstar comparison is read-oriented, so a repeated measurement read may appear acceptable. Yet it can still consume quota, reveal a query, return a newer representation, or invalidate comparability. If the same tool boundary later publishes an artifact or updates a record, repeating the entire agent run becomes more dangerous. Reliability code must not depend on every tool remaining harmless forever.

The central principle is that recovery preserves commitments. We cannot claim an earlier answer was never shown, assume a timed-out effect did not occur, or grant broader authority because the original provider failed. A safe error category tells a client what class of failure occurred. It does not settle the operation's history.

15.2. Recovery constraints

Availability competes with load. Retrying may mask transient failure while worsening overload. Multiple independent retry layers multiply calls rather than merely adding them.

Observation is incomplete. A network timeout reports lack of a timely response, not absence of a remote commit. The client and server can disagree temporarily about what happened.

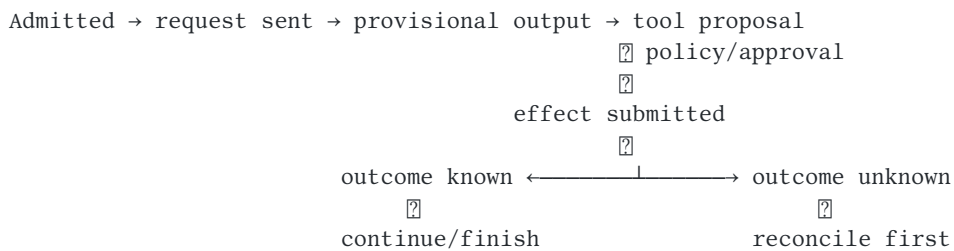
Public errors need less detail than recovery diagnostics. Credentials and payloads must not leak, but discarding every correlation fact also prevents operational diagnosis.

Time belongs to the run. Each model step, tool call, queue wait, and backoff consumes one task deadline. Resetting a timeout on every retry can turn a finite request into an unbounded run.

Reading rule. Before retrying, identify the failure phase, visible commitments, possible effects, remaining budget, and the owner authorized to make the next decision.

15.3. Two commitments: visibility and effects

A run crosses two distinct boundaries. Publishing an output fragment commits the service to a visible attempt history. Executing a side effect commits the system to an external operation, even if its receipt has not arrived.



This is a reasoning diagram, not the reference's complete state machine. Model and tool phases can interleave, and several read-only calls may be pending together. Its purpose is to prevent an exception from flattening materially different histories into "try again."

A replacement model attempt should receive a new response identity. It can consume authorized, verified prior results without pretending to be the original stream. If partial output is retained, label it incomplete or abandoned. If the interface permits replacement, disclose that replacement and establish which terminal answer is authoritative. [Chapter 7](#) already defines local terminal consistency; recovery must respect it rather than concatenate two attempts into one plausible answer.

15.4. Safe errors and protected diagnostics

The tagged `AdapterError` stores provider, code, retryability hint, and optional HTTP status. Its public message is constructed from controlled fields, not the upstream error text. The common loop in `src/agent.ts` recognizes API call errors with a status, recognizes aborts, and otherwise maps failures to a protocol error. A non-API exception is not automatically classified as transient transport failure.

Tagged mapping	Category	Consequence
401/403	Authentication, non-retryable	Repair authorization/configuration; do not blindly repeat
400/404/422	Invalid request, non-retryable	Inspect protocol/model/request binding
408/409/429	Rate limit, retryable hint	Coarse grouping, not equivalent HTTP meanings

Tagged mapping	Category	Consequence
Other statuses	Upstream; retryable for statuses at least 500	Still requires phase/effect policy
Abort signal	Aborted, non-retryable	Do not restart work merely because it stopped
Other exceptions	Protocol, non-retryable	Bounded public failure; limited diagnosis

The 408/409/429 grouping is a reference limitation. HTTP 409 denotes conflict with the resource's current state, not a general rate limit. Status interpretation also depends on the provider and operation (Fielding et al. 2022). The hint must not be treated as a semantic recovery plan.

The [adapter error fixture](#) proves that a synthetic sensitive error message does not reach the collected public exception. It does not prove a protected diagnostic service exists. The reference does not retain a rich private failure envelope with selected headers, safe request identity, phase, and structured upstream cause.

A **proposed diagnostic split** exposes a bounded category and application incident reference to the caller, while a protected store retains only reviewed diagnostic fields. Access, retention, redaction, and audit policy apply to that store. A hash of a sensitive payload is not automatically harmless; predictable values can be guessed. Do not use an unrestricted cause serialization as a shortcut for observability.

15.5. Provider-specific variation: successful HTTP is not successful execution

Anthropic's "HTTP errors" and "Request ID" sections document SSE error events after HTTP 200, overload as HTTP 529, and support correlation IDs (Anthropic, n.d.-a). Its official SDK documentation describes retries, but this reference uses the AI SDK provider package rather than the official Anthropic SDK client. Do not transfer another client's retry defaults to this implementation.

In the common reference loop, SDK `maxRetries` is set to zero. That identifies one retry owner that has been disabled. It does not prove the gateway, HTTP client, application, or user will never retry. Nor does receiving HTTP 200 prove that the stream reached a successful terminal event.

The post-release [live HTTP bridge](#) is a separate boundary. Its current completion mapping can represent a length finish or a normal end without an explicit model finish as completed. That limitation must not authorize treating partial generation as a successful recoverable checkpoint. Reducer validation, adapter projection, and HTTP mapping need separate evidence; a sound reducer does not repair an upstream mapper that mislabeled the outcome.

15.6. Cancellation is a cooperation protocol

DOM §3 defines abort signals; §3.3 specifies immediate rejection for an already-aborted signal in promise-returning APIs that use this integration contract (WHATWG, n.d.). The cited commit snapshot fixes the text rechecked on 2026-10-09. This is an API integration obligation, not a claim that arbitrary callbacks obey it. It does not grant JavaScript the ability to roll back arbitrary remote work. A tool that ignores its signal can keep running after the caller stops waiting.

Cancellation has several owners: the UI can stop consuming, the server can abort provider work, and a tool executor can cease cooperative activity. A transport disconnect may require its own server plumbing. A test cancelling a response reader proves that path, not every socket-close path. Keep those proofs distinct.

The tagged `ToolPolicyRuntime` combines an outer signal with a timeout controller, passes the signal in trusted execution context, and checks outer cancellation before policy hooks. Its `fixture` verifies pre-aborted cancellation and a timed-out execution with an outcome-unknown classification.

There is an important gap: the runtime does not recheck cancellation after awaiting authorization and approval, before calling `execute`. The retained `book-owned probe` aborts the outer signal while a controlled authorization hook waits, then releases that hook. It separately repeats the scenario at approval. Each path still executes once with an already-aborted signal; the `machine-readable report` labels both assertions **known-defect**. Reproduction requires an authorized exact-tag checkout, not a provider credential. This is not a provider problem. It is a missing cancellation checkpoint at a local policy boundary.

The timeout is also not a proved deadline over the entire policy pipeline. The timer is created before the hooks, but its promise is raced only against execution after the hooks. A hung authorization hook is not bounded by that execution race; timer rejection during that wait is not cleanly handled by a whole-pipeline race. Avoid inferring a total authorization/approval/execution deadline from the execution-timeout fixture.

A stronger **proposed implementation** checks cancellation before and after every awaited policy stage, gives each stage a bounded budget, handles every timer rejection, and removes listeners/timers on all terminal paths. Before an effect starts, cancellation can reject admission. After it starts, record what is known about the effect separately from the caller's cancellation. An abort reason can itself contain untrusted or sensitive data and should not be copied into public errors.

15.7. One deadline, one retry owner

Use an end-to-end deadline and recompute remaining time before every queue wait, backoff, model call, and tool execution. A monotonic clock is appropriate for elapsed local budgets; wall-clock timestamps are useful for persisted expiry and diagnostics. Cross-process deadlines need an explicit conversion/skew policy rather than blindly comparing clocks from different machines.

Connection, first-byte, idle-stream, per-tool, and overall run timeouts solve different problems. Progress can justify an idle-timeout reset, but it must not reset the overall deadline. A long stream can remain active while making no useful progress toward the task.

AWS's Builders' Library, "Retries and backoff" and "Jitter," explains layered retry amplification and spreading synchronized retries. Its introduction stresses that timeout does not imply absence of effects (Amazon Web Services, n.d.). These specific passages are recorded in the `claim register`. These are operational practices, not a normative model API contract.

A bounded policy needs both a total attempt limit and a time budget. Include the initial attempt in the count. Full jitter samples a delay within a bounded exponential window; randomization changes scheduling, not retry safety. Honor a validated upstream minimum wait when appropriate, but stop if it cannot fit the task deadline. RFC 9110 §10.2.3 permits `Retry-After` as an HTTP date or nonnegative delay in seconds (Fielding et al. 2022). Parsing and clock conversion are separate from the decision below.

15.8. A tested recovery decision, not a hidden retry loop

This **implemented and deterministically tested book design** takes trusted, normalized phase/effect information and produces a next-action decision. Its complete types are in `code/recovery-policy.ts`. `attempts` counts attempts already begun. Times use one caller-defined millisecond clock. `retryAfterMs` is a validated duration, not a raw HTTP header, and `jitter` is an injected sample in `[0, 1)`.

```
export function decideRecovery(context: RecoveryContext): RecoveryDecision {
  for (const value of [
    context.attempts,
    context.maximumAttempts,
    context.minimumAttemptMs,
    context.baseDelayMs,
    context.maximumDelayMs,
  ]) {
    if (!Number.isSafeInteger(value) || value < 1) throw new TypeError("Invalid
      ↪ recovery budget.");
  }
  for (const value of [context.nowMs, context.deadlineMs, context.retryAfterMs ?? 0]) {
    if (!Number.isSafeInteger(value) || value < 0) throw new TypeError("Invalid
      ↪ recovery time.");
  }
  if (!Number.isFinite(context.jitter) || context.jitter < 0 || context.jitter >= 1) {
    throw new TypeError("Invalid jitter sample.");
  }
  // Reconciliation means determine the outcome, not repeat the effect.
  if (context.effects === "unknown") return { action: "reconcile" };
  if (context.cause !== "transient") return { action: "stop", reason: "cause" };
  if (context.outputVisible) return { action: "stop", reason: "visible" };
  if (context.effects === "committed") return { action: "stop", reason: "effects" };
  if (context.attempts >= context.maximumAttempts) return { action: "stop", reason:
    ↪ "budget" };
  const ceiling = Math.min(
    context.maximumDelayMs,
    context.baseDelayMs * 2 ** Math.min(context.attempts - 1, 30),
  );
  const delayMs = Math.max(Math.floor(context.jitter * ceiling), context.retryAfterMs
    ↪ ?? 0);
  if (context.deadlineMs - context.nowMs < delayMs + context.minimumAttemptMs) {
    return { action: "stop", reason: "budget" };
  }
  return { action: "retry", delayMs };
}
```

Unknown effects take precedence: reconcile means determine the outcome, never repeat it. Even after cancellation, the system may need a separately authorized outcome lookup for an already-submitted operation. This decision does not launch that lookup or authorize background work.

Committed effects prevent restarting the operation. The caller can reuse a verified receipt and continue a later stage under a new plan, but this helper does not implement that continuation. Visible output also prevents an invisible repeat of the same attempt. Only transient failures with no effects or trustworthy evidence of non-application can reach the retry branch. Here, effect knowledge concerns the domain/tool operation, not the absence of provider billing, access logs,

or quota consumption. Even an admitted repeat can spend resources; the surrounding run must debit its cost and attempt budgets independently.

The local backoff cap does not shorten the upstream minimum wait. A 600-millisecond upstream wait can exceed a 500-millisecond local window; the deadline decides whether it fits. The `minimumAttempts` reservation is an admission estimate, not an enforced timeout on actual execution. The runner must enforce remaining time, debit attempts, sleep cancellably, and recheck policy after waiting. A pure policy function cannot provide those runtime guarantees by itself.

The [scripted Northstar composition](#) uses this decision without pretending to supply a runner. A transient failure before output or tool work permits `response-2`, explicitly abandoning `response-1`; already-visible output stops replacement. If a tool has started and its receipt is lost, the result is `reconcile`. Reopening the transcript leaves that call inert and the execution count at one. Cancellation in this unknown-effect scenario changes the requested work, not the missing receipt.

Recover from evidence about the attempt, not from a status-code lookup alone. Schedule only work that still fits the run's authority, effect history, visibility contract, and budget.

15.9. Reconciliation examples

Reconciliation is operation-specific. It cannot be implemented as “retry once, then ask the user.” The lookup must use identity owned by the effect system and must itself be authorized.

15.9.1. Lost Northstar measurement receipt

Suppose `measurement-a` was submitted, the executor count advanced, and the process lost the result before appending it. Reopening the transcript shows a pending call, not proof that no measurement occurred. The next action is:

1. retain the original operation identity and outcome-unknown state;
2. perform a separately authorized read-only receipt/snapshot lookup if the measurement service supports one;
3. append the verified original result without dispatching the measurement again; or
4. if no lookup exists, report that the run cannot establish a comparable pair and require an explicit new research attempt.

Repeating the read is not automatically equivalent. It can return a newer observation, spend quota, disclose data again, and break the five-second comparability rule. Cancellation after the receipt was lost does not change step 1 into permission to call the measurement tool again.

15.9.2. Presentation artifact handoff interrupted

If `presentation_builder` completed and returned an artifact ID, but `Pi` disconnected before saving, the client may fetch that exact unexpired opaque ID and verify its declared byte count and SHA-256. It must not rebuild the deck merely because no local file appeared. If the lease expired or server restart destroyed the store, the system can report that the exact artifact is unavailable. A rebuild is a new authorized operation with a new identity and potentially different bytes.

If the verified target filename already exists, reconciliation also checks local state: equal digest can be treated as already saved; a different digest requires explicit replacement confirmation. “The prior save timed out” is not permission to overwrite.

15.9.3. Consequential publication with unknown remote commit

For an operation that publishes a brief, the effect service should own an idempotency key bound to tenant, principal, normalized intent digest, and artifact digest. After a timeout, query publication status by that key. A verified committed receipt ends the operation; a verified non-application may permit a new attempt under current authorization; an unresolved status remains unresolved. Compensation, such as withdrawing a publication, is a new effect with separate approval.

These examples share one rule: cancellation can stop waiting or prevent work that has not crossed the effect boundary. It cannot convert uncertain history into evidence of non-execution.

15.10. Idempotency is an operation contract

RFC 9110 §9.2.2 advises against automatically retrying non-idempotent methods unless the client knows the semantics are idempotent or knows the original operation was not applied (Fielding et al. 2022). Most generation APIs use POST; the method name alone does not decide whether replaying a particular application operation is safe.

A tool call ID correlates a proposal and result. It is not automatically a business idempotency key. The tagged runtime rejects repeated call IDs within one runtime instance; a fresh request constructs a fresh runtime and seen-ID set. It does not provide cross-request deduplication, crash recovery, or a durable exactly-once effect guarantee.

For a consequential operation, a **proposed ledger** binds an operation ID to principal/tenant, tool version, normalized intent digest, and recorded status. Reusing a key with a different intent must reject. Claiming a pending operation must be atomic. A verified terminal receipt can be reused; an unknown outcome triggers a lookup against the effect system, not a second blind invocation.

Even a durable local ledger cannot atomically commit a remote effect unless the remote system participates in an appropriate protocol or supports its own idempotency contract. There remains a crash interval between remote commit and local receipt persistence. Design reconciliation for that interval. Compensation is a new authorized operation, not proof that history never happened; cancelling an alert may not retract an alert already observed by a user.

A measurement read is usually less consequential than an update, but two reads at different times can yield different representations. Reuse a snapshot only when the task’s freshness policy permits it. Idempotent external effects and identical model answers are different notions.

15.11. Failure modes

Retry storms. Client, router, SDK, and gateway each retry three times, increasing demand precisely when the service is overloaded. Choose one owner and expose the remaining attempt budget to adjacent layers.

Timeout interpreted as rollback. The caller receives no receipt and marks the operation failed. Preserve outcome-unknown and reconcile rather than repeating the effect.

Cancellation only at entry. A request passes the initial check and waits on policy; later cancellation does not prevent execution. Place checkpoints at each effect-admission boundary and test the intervening awaits.

Partial output laundering. A second model completes a sentence started by another attempt. Preserve distinct identities and do not manufacture a continuous authoritative answer.

Diagnostics as exfiltration. A safe category is followed by a raw payload in an ordinary log. Apply the same data-minimization boundary to every diagnostic path.

15.12. Security considerations

Recovery can amplify both cost and privilege. A retry must not bypass approval, extend a grant beyond expiry, or reuse an authorization decision after its context changed. A reconciliation call must be read-only or separately authorized for its own effects. Persisted approvals need binding to the operation and intent, not just a remembered “yes.”

Do not allow untrusted provider messages to set retry count, deadline, fallback destination, or operator instructions. Provider response headers can influence waiting only after parsing and bounding; they cannot create an unlimited budget. Rate limiting protects the shared service as well as the individual run.

Diagnostic references must not expose another tenant’s incident. Protect support-facing provider request IDs according to policy. Store outcomes and receipts with retention appropriate to their sensitivity; absence of a receipt is not evidence that the effect did not occur.

15.13. Telemetry contract

Proposed fields distinguish failure phase, normalized cause, effect knowledge, output visibility, attempt index, delay class, deadline exhaustion, and reconciliation outcome. Record initial work separately from retry amplification. Unknown outcomes deserve their own counter instead of being buried in generic errors.

A cancellation metric should distinguish requested, locally observed, provider work aborted, and effect outcome resolved. Those are different measurements. Avoid public raw abort reasons and uncontrolled high-cardinality labels. The pure decision exposes neither diagnostic persistence nor emitted metrics.

15.14. How we test portability

Six book tests exercise exponential/jittered delay, local caps, upstream minimum waits, attempt and deadline boundaries, refusal/cancellation/visibility stops, committed effects, unknown-effect reconciliation, and malformed policy inputs. Injected clocks and samples make these tests deterministic without sleeping or calling a provider.

The tagged tool tests prove pre-aborted cancellation, duplicate rejection within one runtime, time-out classification, and selected output/policy failures. The authorization **and approval** probes establish the missing checkpoints described above; they do not prove all transport cancellation paths

or exercise a hung-hook timer. The composition fixture checks new-attempt identity, visible-output stop, and unknown-effect/no-repeat behavior. The reference source remains unchanged; a future repaired release needs different regression assertions and a new immutable identity.

```
bun test book/test/recovery-policy.test.ts
bun run check
```

Before stronger production claims, add fault-injection tests around remote-commit/local-receipt gaps, authorization cancellation, timer cleanup, slow readers, hidden gateway retries, and idempotency-key/intent conflicts. These are proposed coverage, not successful runs reported here.

15.15. Portability and evidence ledger

Dimension	Evidence label and scope	Limitation
Transport	Specified: HTTP repeat/wait semantics; DOM abort integration	No remote rollback or complete disconnect proof
Provider	Implemented: safe categories and SDK retry suppression; synthetic success/failure decoding probed	Coarse status mapping; no complete diagnostic envelope
Protocol	Inspected limitation: separately pinned bridge completion mapping	length/missing finish can be mislabeled; no conformance claim
Feature	Tested: book recovery decision; tag timeout/pre-abort; retained post-hook counterexamples	No whole-policy deadline, sleeping runner, or repaired cancellation claim
Semantic	Locally tested: explicit replacement identity and inert unknown-effect history	No repeat authorization or identical regenerated answers implied

Additional recovery concern. A durable receipt/effect ledger and separately authorized reconciliation are proposed, not supplied by the transcript or next-action function.

15.16. Durable state, memory, and replay

Recovery decisions depend on causal history that survives process restart. The former standalone state chapter is consolidated here so the error taxonomy, retry decision, unresolved effects, transcripts, replay, and compaction form one policy rather than two adjacent introductions.

15.16.1. Why recovery needs durable state

The first half of this chapter made recovery depend on the history of visible output and effects. That history must outlive the generator if the system is to survive restart. But “save the conversation” is underspecified. A display transcript can omit an unresolved operation. A provider cursor can become unusable after a deployment change. A model-generated summary can omit the very constraint that made the previous execution safe.

The Northstar task makes the distinction concrete. We may have a user request, a proposed Prototype A lookup, a verified measurement, an unfinished Prototype B lookup, and provisional comparison text. Reloading only the visible messages cannot tell us whether the second lookup was submitted. Reusing only an upstream response ID cannot establish who owns the task or whether the measurements are still fresh.

A portable design separates state by purpose. It can reconstruct a user view without calling tools, build authorized model context without importing provider-private state into another provider, and resume work only after resolving uncertain operations. Those are three different activities, not three names for reading a JSON file.

15.16.2. State constraints

Persistence without authority persistence. Yesterday’s transcript may contain an approved action, but today’s user, policy, or grant can differ. Retention of a record must not silently extend its authority.

History without complete reproducibility. Replaying deterministic application events can reproduce a snapshot. Asking a stochastic model again with the same messages can produce different output, usage, and tool proposals.

Compact context versus faithful evidence. Context windows are finite. Summaries help generation but can lose qualifiers, temporal facts, citations, and causal relations.

Local transactions versus remote effects. A database can atomically append history and advance a revision. It cannot thereby atomically commit an unrelated external tool operation.

Reading rule. For each state object, name its authority domain, retention owner, compatibility key, and the action that consuming it is allowed to trigger.

15.16.3. Five state planes

Plane	Purpose	Typical owner	Must not be mistaken for
Interaction transcript	Messages, proposals, results, visible outcomes	Application or client session	Authorization or remote effect ledger
Run journal/checkpoint	Accepted events, pending operations, policy/budgets	Orchestrator	A complete model conversation
Retrieval memory	Selected facts/documents for future context	Retrieval service under application policy	Instructions or timeless truth

Plane	Purpose	Typical owner	Must not be mistaken for
Provider continuation	Response lineage or opaque compatible items	Provider boundary	Cross-provider portable memory
Derived view	UI snapshot, summary, cached model context	Projection/compaction layer	The authoritative retained history

A session is an application identity and ownership boundary connecting some of these planes. It is not identical to a provider response ID, HTTP connection, or current model. A task can span several responses; a session can contain several tasks. Routing changes must not change session ownership.

The portable spine is the application's record of intent, accepted observations, causal outcomes, and policy-relevant progress. Provider-specific continuation can accompany that spine as an extension. It should not be the only evidence that the application performed its own work.

15.16.4. What the reference actually retains and converts

The tagged `ModelInputItem` admits user/assistant text, tool calls, and correlated tool results with an optional error flag. The shared `modelMessage` conversion constructs SDK assistant/tool messages. Non-string successful tool output is JSON-stringified into SDK text; error output uses string conversion. The original typed item and the SDK encoding are therefore distinct representations.

This small contract is sufficient for the tagged common-loop fixture. It is not a complete transcript format for every provider item. It lacks provider-private reasoning content, native citation annotations, assistant phase metadata, media, and a private continuation envelope. Preserving visible text alone cannot preserve fields absent from the contract.

The tagged `response reducer` and its `tests` demonstrate deterministic projection: reducing the same valid event fixture again yields the same snapshot; missing terminal events and inconsistent text reject. This is a local event-replay property. It does not establish durable retention, authorized event loading, replay across schema migrations, or re-execution of an external operation.

Pi owns its client sessions. The separately pinned `post-release acceptance script` exercises synthetic transcript continuation and persistent client session recovery, among other paths. That evidence is not in the basic tagged Pi text acceptance and is not a server-side durable transcript store. Keeping UI sessions in the client is a valid boundary, but a shared service still needs its own authenticated state ownership if it retains runs.

15.16.5. Three meanings of replay

Projection replay consumes accepted events to reconstruct a view. Its transition function should not execute tools or contact the provider. Determinism is relative to the event schema and reducer revision.

Context replay selects recorded messages/items for a new model request. It is an input-construction decision. It may require sanitization, loss accounting, current policy, and provider compatibility checks.

Effect replay reissues an operation. It requires the idempotency/reconciliation contract established earlier in this chapter. A historical tool call is evidence of a proposal, not permission to perform it again.

Recorded history

```

|— deterministic reducer → display snapshot
|— authorized context selection → new model request
|— operation ledger + current policy → continue/reconcile work

```

No automatic arrow from "read transcript" to "execute tool"

These distinctions matter after failures. A pending call in the transcript could mean proposal-only, submitted-without-receipt, or an incomplete local record. Unless a journal or effect ledger resolves that ambiguity, reloading it must not choose execution as the default. Read-only reconstruction is a safer initial operation than automatic resumption.

15.16.6. Provider-specific variation: lineage and opaque state

OpenAI’s conversation-state guide, rechecked in the 2026-10-08–09 review, distinguishes manual history from “Using the Conversations API” and “Passing context from the previous response” (OpenAI, n.d.-a). The [claim register](#) records these sections as mutable documentation, not tests of pinned SDK continuation. These are different ownership and retention modes, not interchangeable identifiers. A response chain’s cursor is not a portable application session, and stateless request construction does not imply that every participant retains no data.

The reasoning guide, “Preserve reasoning across calls,” distinguishes visible history from opaque reasoning items and describes compatible replay in supported contexts (OpenAI, n.d.-b). That is provider-specific continuity. An application must not decode opaque content to invent a chain of thought, convert it into ordinary instructions, or assume that another provider can consume it.

Anthropic’s Messages reference, “Create a Message” and “Body parameters / messages,” describes stateless multi-turn conversations built from supplied messages (Anthropic, n.d.-b). A generic transcript can support that request construction, but content-block and model-specific requirements still belong to its boundary. Do not infer identical continuation behavior from both providers accepting message arrays.

A **proposed opaque-state envelope** binds payload to issuer/provider, protocol and adapter revisions, deployment/account scope, compatible model class, session/task identity, producing response, and the exact transcript prefix or digest. It records expiry and security classification separately from the payload. A matching compatibility key is necessary but not sufficient: authoritative contracts or reviewed execution tests must establish that this exact state can be consumed. Equal provider/model labels cannot prove validity.

The book does not implement this envelope or replay opaque reasoning. The reference port cannot represent it. On incompatibility, an application may restart from approved neutral context if the task permits that loss. If the task requires continuity that cannot be reconstructed, reject or request an explicit restart; do not silently erase private state and claim equivalent continuation.

15.16.7. A bounded, transactional transcript experiment

A small **independently authored, implemented and tested book design** demonstrates a narrower property: trusted tenant/principal comparison and atomic append of a limited JSON transcript. It uses Bun 1.3.9's built-in `bun:sqlite`, with no new dependency (Bun, n.d.). It is not installed into the reference server.

The complete `code/transcript-store.ts` defines user/assistant messages and correlated tool calls/results, including an optional result error flag. It rejects system roles and unknown top-level fields, validates causal call/result ordering, caps a session at 1,000 records, and caps a serialized record at 16,384 UTF-8 bytes. Its input must be ordinary JSON-decoded values rather than objects with serialization hooks. Persisted-byte checks do not replace request-body bounds before decoding.

The database contains session ownership/revision and ordered JSON records. `revision` counts accepted records, so a batch advances it by its length. The following excerpt contains the public read and append operations; database creation, decoder, ownership lookup, and causal validation are in the full tested file.

```
export function readTranscript(db: Database, id: string, owner: SessionOwner):
  ↳ TranscriptSnapshot {
    return db.transaction(() => ownedSnapshot(db, id, owner))();
  }

export function appendTranscript(
  db: Database,
  id: string,
  owner: SessionOwner,
  expectedRevision: number,
  batch: readonly unknown[],
): number {
  if (
    !Number.isSafeInteger(expectedRevision) ||
    expectedRevision < 0 ||
    batch.length === 0 ||
    batch.length > maximumRecords
  )
    return rejectHistory();
  // Input must be ordinary JSON-decoded data, not objects with serialization hooks.
  const payloads = batch.map(encodeItem);
  return db
    .transaction(() => {
      const current = ownedSnapshot(db, id, owner);
      if (current.revision !== expectedRevision) throw new TranscriptError("conflict");
      if (current.revision + batch.length > maximumRecords) return rejectHistory();
      const additions = payloads.map((payload) => decodeItem(JSON.parse(payload)));
      validateHistory([...current.items, ...additions]);
      const insert = db.query("INSERT INTO history VALUES (?, ?, ?)");
      payloads.forEach((payload, index) => {
        insert.run(id, current.revision + index + 1, payload);
      });
      const revision = current.revision + payloads.length;
      db.query("UPDATE sessions SET revision = ? WHERE id = ?").run(revision, id);
      return revision;
    })
    .immediate();
}
```

```
}

```

Reads use one transaction to obtain ownership, revision, and records from a consistent snapshot. Appends use an immediate transaction: ownership, expected revision, history validity, inserted records, and the new revision are checked or changed in one transaction. SQLite permits only one simultaneous writer; `BEGIN IMMEDIATE` can return busy when another writer holds the database (SQLite, n.d.-b, n.d.-a). The example does not implement contention retries or a distributed lock service.

An expected-revision mismatch rejects rather than overwriting another writer's history. The caller must reread and decide whether a new append still belongs to the task; mechanically reapplying a stale batch can duplicate intent. Revision conflict is not a transient failure to be retried without semantic review.

The history validator uses a deliberately conservative profile: multiple pending calls are permitted, but ordinary messages cannot cross an unresolved call; each result must match a pending call's ID and name, and repeated call IDs reject. This is book policy, not a universal provider protocol rule. An unresolved final call may be persisted as checkpoint data. Reading that checkpoint does not resolve or dispatch it.

In the [composition trace](#), a completed comparison reopens revision 5: one user request plus two calls and two results. The tool execution count remains two. The answer is then computed from the **restored results**, not an independent copy of fixture data. An unknown-effect branch reopens revision 2—request plus pending call—with the execution count still one. Neither read path infers a receipt or performs resubmission. The trace checks ordinary restart behavior; it does not simulate a power-loss crash.

Ownership comes from a trusted `SessionOwner` supplied by the caller. The helper compares tenant and principal, but does not authenticate them. Passing user-controlled claims would defeat its authorization boundary. Missing sessions and ownership mismatches share a bounded access error on reads and appends; this is not a claim that every timing or creation path prevents membership inference.

The experiment proves atomic local recording and ordinary close/reopen persistence. It does not prove power-loss durability under every storage configuration, encryption, tamper resistance, backup restoration, multi-process fault tolerance, schema upgrades, or remote effect atomicity. The database is a trusted local store, not a signed evidentiary archive. Public API error sanitization remains an outer responsibility for raw SQLite failures.

15.16.8. Checkpoints are not receipts

Suppose the journal contains a call proposal and the process crashes before a result is appended. There are at least three possibilities: no effect was submitted; it was submitted but not committed; it committed remotely but its receipt was not persisted. A transcript schema cannot distinguish them unless it records additional facts and coordinates with the effect system.

A production checkpoint therefore needs an operation-status plane with intent identity, submitted/known/unknown state, and verified receipts. The transcript can refer to that plane without pretending to replace it. A local transaction spanning two database tables still cannot cover an HTTP effect outside the transaction.

Persist facts once, derive views many times, and resume effects only from explicit operation knowledge. A transcript revision is a consistency boundary, not an authorization token or a remote commit receipt.

After a restart, load owned state, validate schema/reducer compatibility, resolve uncertain operations, re-evaluate freshness and authorization, and only then construct the next model request. These are proposed orchestration stages, not an implemented automatic resumer. New provider output can vary even if neutral context and policy remain the same.

15.16.9. Memory and compaction are provenance-bearing transformations

A memory item can be an observation, extracted source passage, preference, or model-generated summary. Each needs source identity, time, scope, and a policy for reuse. “The model remembered it” does not establish truth, freshness, or permission to use it for another tenant or purpose.

Compaction trades context size for retained meaning. Alternatives include a selected raw window, a structured fact table, a summary, or externally indexed evidence. Each has different loss modes. A summary can omit that Prototype B’s measurement was stale; a last-N-message window can orphan a tool result; a fact table can remove disagreement and source qualifiers.

Preserve the authoritative retained record according to retention policy, and mark summaries as derived. Record input-prefix identity, summarizer/configuration revision, intended scope, and relevant omissions. Test whether hard constraints, unresolved operations, dissenting evidence, and citations survive. Do not elevate a generated summary into system authority merely because it is shorter.

Retrieval memory has its own access control. A globally useful-looking fact can still come from a private document. Query-time ownership checks must precede its insertion into model context, not only its insertion into the index. Chapter 13 deepens this boundary by giving each source and evidence locator a concrete identity.

15.16.10. Failure modes

A cursor treated as a session. The provider ID restores model lineage but not authenticated application ownership or tool receipts.

Replay runs the past. Deserializing a tool-call record invokes the tool again. Keep projection/context construction free of dispatch and require a separate operation plan.

Lost updates. Two clients append to the same transcript prefix and one overwrites the other’s work. Use expected revisions and atomic append; surface conflicts rather than silently merge incompatible actions.

Orphaned causal history. Compaction retains a result without its call or a pending proposal without its unresolved status. Preserve coherent units and test compaction invariants.

A summary becomes a grant. Generated memory says an action was approved and later execution trusts that sentence. Keep authorization in trusted policy/approval records with current validity.

15.16.11. Security considerations

A session identifier is a lookup key, not proof of access. Derive ownership from authenticated application context. Encrypt sensitive retained data when required, restrict file/database access, and prevent cross-tenant retrieval. The local example provides none of that infrastructure; it only demonstrates an ownership comparison given trusted claims.

Opaque provider state can be sensitive even when unreadable. Restrict destinations, retention, logs, and export. Treat unverified state as data, not executable deserialization instructions. Do not migrate state to another account merely because the receiving model has a similar name.

Deletion must account for authoritative history, snapshots, summaries, retrieval indexes, provider-held state, and backups under their respective policies. A derived summary can retain information removed from a transcript. Content digests aid identity but are not anonymization; predictable content can be guessed.

15.16.12. Telemetry contract

Proposed fields distinguish load/read, projection replay, context selection, state incompatibility, revision conflict, unresolved operation, compaction, and fresh execution. Record a safe reducer/schema revision and journal cursor without logging transcript content.

Do not call a successful file read a successful resumed task. Separate “state loaded,” “operation reconciled,” and “task completed.” Track context omissions and compatibility rejection rather than silently discarding state. The local reopen test measures consistency, not an operational resume metric.

15.16.13. How we test portability

Six book tests verify ordinary file close/reopen, detached reads, tenant/principal rejection, stale revision rejection across two connections, causal ordering and pending-call retention, SQL-failure rollback of both records and revision, JSON-profile rejection, and parameterized identifiers. The stale-writer scenario is deterministic sequencing, not a stress test of concurrent processes. A trigger injects failure during the second insert to prove the batch rolls back.

The reference reducer tests separately verify repeated projection equality and terminal consistency. Post-release Pi session acceptance is a client-boundary proof, not this database’s server integration. The composition exercises owned reopen and reconstruction from retained observations, with no dispatch at read. No provider-private continuation or compaction experiment is claimed. Reader-only checks are also available in the [archive](#).

```
bun test book/test/transcript-store.test.ts
bun run check
```

For deployment, add interrupted-write, schema-migration, corruption, concurrent-process/busy, backup/restore, revocation, and effect-reconciliation tests under a documented storage configuration. Add model-level tests only when the exact provider continuation contract is exposed and reviewed.

15.16.14. Portability and evidence ledger

Dimension	Evidence label and scope	Limitation
Transport	Not implemented here: remote state transfer	Local SQLite reopening proves no wire/storage migration contract
Provider	Implemented: neutral reference history conversion	Subset, not complete provider continuation
Protocol	Documented: provider lineage/opaque-history distinctions	No portable envelope or cross-provider state proof
Feature	Implemented/tested: owned local append/read; reference projection equality is separate	No migrator, compactor, server integration, or durability audit
Semantic	Locally tested: restored observations reconstruct the comparison; pending history remains inert	No identical model regeneration or automatic resume authority

Additional state/policy concern. Authentication, retention, fresh permission, and durable effect receipts remain outside this trusted-owner store.

15.17. Integrated pattern summary

Problem: A transient-looking failure can hide visible output, completed effects, or unresolved operations whose history must survive restart.

Pattern: Normalize errors, record phase and effect knowledge, propagate cancellation without treating it as rollback, give one owner a bounded retry policy, reconcile unknown outcomes, and retain a portable causal spine whose projection is effect-free.

Result: Recovery, restart, provider change, and compaction preserve commitments without silently repeating work or turning history into authority.

Cost: Durable receipts, operation-specific lookup, storage versions, retention, cancellation checkpoints, and loss-aware compaction are harder than a retry decorator or chat-log save.

15.18. Exercises

1. Read the retained approval-cancellation counterexample. Specify the regression for a repaired release: after the gate resumes, executor count must remain zero. Explain why changing the historical assertion would falsify evidence.
2. Specify an idempotency ledger for publishing a synthetic Northstar brief. Identify the remote-commit/local-receipt crash interval and the lookup needed to resolve it.
3. Add a cancellable runner around the pure policy. Test cancellation during backoff and after effect submission; assert that neither path blindly repeats an uncertain effect.

4. Extend the transcript experiment with a migration version and reject future schema rather than interpreting it as current history.
5. Specify a compaction invariant that preserves unresolved calls, conflicting observation times, and current authority boundaries.
6. Define the user-visible protocol for replacing an incomplete answer with a new attempt. Identify which response is authoritative.

15.19. Further reading

Read RFC 9110's idempotent-method and Retry-After sections (Fielding et al. 2022), the DOM abort integration requirements (WHATWG, n.d.), and AWS's retry/backoff analysis (Amazon Web Services, n.d.). Compare Anthropic's error/request-ID guidance (Anthropic, n.d.-a), OpenAI conversation lineage and reasoning continuity (OpenAI, n.d.-a, n.d.-b), stateless Anthropic Messages construction (Anthropic, n.d.-b), and SQLite transaction/isolation behavior (SQLite, n.d.-b, n.d.-a; Bun, n.d.). Each source owns a different boundary; none grants replay authority.

15.20. Chapter contract

- **Scope:** errors, retries, cancellation, reconciliation, durable transcripts, replay, and compaction.
- **Canonical claims:** uncertain effects require reconciliation, and persisted history grants no execution authority.
- **Migration inputs:** former recovery chapter plus the retired state-memory-and-replay chapter.
- **Evidence gates:** crash-window, unknown-effect, ownership, schema-version, and replay tests.
- **Demonstration slots:** deterministic Northstar interruption trace.
- **Source-claim ledger:** local transaction evidence is not remote-effect evidence.
- **Exclusions:** stream-local state belongs to Chapter 7; telemetry implementation belongs to Chapter 16.

16. Observability with OpenTelemetry

Part IV · Portability in production

Implemented reference revision `ce47fe2` · OpenTelemetry API 1.9.0 · SDK/context 2.12.0 · deterministic evidence SHA-256 `2fbaf4d...0119`

Tool progress explains bounded activity to a user. Telemetry explains system behavior to operators. The implemented profile traces the Northstar execution path without exporting prompts, documents, image bytes, generated observations, Python code, tool values, credentials, or raw provider errors.

A **trace** connects causally related operations. A **span** records one bounded operation under that trace. Neither is an authorization record, an effect receipt, a factual-support judgment, or permission to retain agent content.

16.1. Why agent telemetry needs its own contract

An agent request is not one provider call. The HTTP boundary admits files, a coordinator can make several model rounds, each round can propose tools, policy can reject or approve a proposal, execution can wait in a queue, and a server-owned tool can call another provider. Cancellation can occur at every boundary.

Without correlation, operators see unrelated durations and error counters. With unrestricted correlation, telemetry becomes a second copy of the most sensitive data in the system. The design therefore needs both a topology and a data-minimization contract.

OpenTelemetry supplies vendor-neutral APIs and SDK concepts for traces, metrics, logs, context, processors, and exporters (“OpenTelemetry Documentation,” n.d.). Its existence does not decide which spans an agent needs or which attributes are safe. Emerging generative-AI semantic conventions can improve vocabulary consistency, but they do not require prompts or completions to be exported and do not replace application-owned policy (“OpenTelemetry Generative AI Semantic Conventions, Revision 06ec68e,” n.d.).

16.2. Implemented trace topology

The reference emits fixed span names:

```
agent.http.request
├─ agent.ingestion
└─ agent.model.invocation
    ├─ agent.model.round
    ├─ agent.tool.proposal
    ├─ agent.tool.policy
    └─ agent.tool.execution
        ├─ agent.provider.nested
        └─ agent.worker.queue
```

└─ `agent.worker.execution`

Actual sibling order depends on the run. A search or image operation creates `agent.provider.nested`; local Python creates queue and worker spans. A rejected proposal can stop at policy. An input failure can stop after ingestion. Span absence is meaningful only under the named instrumentation revision—it is not proof that no remote work happened outside the observed boundary.

16.2.1. HTTP invocation

`agent.http.request` begins after the handler receives the request and remains open through JSON completion or streamed SSE completion/cancellation. It records a fixed operation category, reviewed provider/model label when admitted, method, loopback server class, bounded status, and terminal outcome.

The span excludes request/response IDs and body content. For SSE, it ends in the stream’s terminal cleanup, not when the server first returns a `Response` object. A consumer cancellation ends it as cancelled. A server deadline ends it as timeout. Local observation cannot prove provider rollback.

16.2.2. Ingestion

`agent.ingestion` surrounds ordered input admission and shared document/image parsing. It records the operation class only. Filenames, paths, source IDs, media bodies, extracted text, digests, OCR output, and parser errors are excluded.

A completed span means the admission function returned. It does not certify document truth or parser fidelity. Format-specific limits and extraction evidence remain Chapter 4’s responsibility.

16.2.3. Coordinator and model rounds

`agent.model.invocation` covers the provider-neutral coordinator operation. `agent.model.round` uses the AI SDK’s model-call start/end hooks so each provider request is separated from later local tool execution.

Attributes retain reviewed provider and model labels plus a bounded chat operation. Prompts, standardized messages, tool schemas, provider request/response IDs, completions, reasoning content, usage payloads, and raw errors are not exported by the profile.

A multi-tool run can have several rounds under one invocation. This distinguishes “the coordinator was slow” from “the tool queue was slow” without copying the conversation into the trace.

16.2.4. Tool proposal, policy, and execution

The policy runtime creates three spans:

- `agent.tool.proposal` records that a named reviewed tool was proposed;
- `agent.tool.policy` covers schema validation, current authorization, approval, and cancellation checkpoints; and
- `agent.tool.execution` covers the compiled executor.

Only the reviewed tool name and bounded phase/outcome category are retained. Call IDs, input arguments, output values, approval text, principal IDs, tenant IDs, source IDs, artifact IDs, and error messages are excluded.

This separation prevents a common operational error: a policy denial must not be graphed as provider failure, and an invalid tool result must not be called a model refusal. It also preserves Chapter 8’s authority rule—an observer cannot approve, retry, mutate, or fail the operation.

16.2.5. Queue and worker boundaries

The Pyodide path emits `agent.worker.queue` while waiting for the process-wide slot and `agent.worker.execution` around worker construction/termination. The host creates these spans. The WebAssembly worker receives no tracer, exporter, prompt, credential, or trace context.

That boundary records queueing and local execution separately while avoiding an instrumentation bridge into generated code. Cancellation while queued ends the queue span without constructing a worker. Deadline expiry terminates the worker and records an error/timeout at the surrounding policy boundary.

16.2.6. Nested providers

Hosted search and image understanding emit `agent.provider.nested` around their fixed OpenAI recipients. The parent request can have an Anthropic coordinator. The trace therefore makes mixed-recipient execution visible without storing the search query, image, focused question, generated note, or observation.

A nested span is not a provider billing receipt or complete census of provider internals. It shows that the application invoked its reviewed bridge and observed a terminal local outcome.

16.3. Explicit context propagation

The implementation uses exact dependencies:

```
@opentelemetry/api          1.9.0
@opentelemetry/context-async-hooks 2.12.0
@opentelemetry/sdk-trace-base  2.12.0
```

A trusted `AgentTraceContext` contains the telemetry implementation and OpenTelemetry context. It is created by the server, propagated through the model request and policy runtime, and never populated from HTTP body fields. The policy runtime creates child spans explicitly. Queue and worker spans inherit the trusted parent on the host side.

Async-local context is enabled for deterministic execution, but important boundaries also pass parent context explicitly. This prevents a callback scheduled outside an active lexical scope from silently becoming an unrelated root.

No W3C trace header is accepted from callers in the current loopback profile. Cross-service propagation would require authenticated ingress, baggage policy, trust-domain decisions, and header size/cardinality limits. Blindly accepting external parent IDs could let an untrusted caller join or fragment another trace.

16.4. Closed attribute policy

The implementation admits only a fixed key allowlist. String values must match a small token profile and at most 128 characters. Unknown keys and unsafe values are dropped.

Admitted dimensions include:

- operation category;
- terminal outcome;
- reviewed provider/model labels;
- reviewed tool name;
- tool phase;
- cancellation source category;
- worker kind and queue state;
- bounded HTTP status; and
- selected generative-AI operation/provider/model labels.

Excluded by default:

- prompts, instructions, messages, and completions;
- source/document bodies, quotes, URLs, filenames, and paths;
- image bytes, questions, and generated observations;
- tool arguments and results;
- Python code, stdout, stderr, and expression results;
- credentials, headers, cookies, and endpoint query data;
- raw provider errors and response bodies;
- request, response, source, call, artifact, principal, and tenant identifiers.

This is stronger than “do not log secrets.” Many excluded values are sensitive or high-cardinality even when not credentials. Digests and opaque IDs can still reveal membership or permit correlation, so the default profile omits them.

16.5. Errors, cancellation, and unknown effects

Spans use bounded outcomes: ok, error, cancelled, timeout, and unknown. Error attributes contain a normalized type/category, never `Error.message`, stack, raw abort reason, provider body, or generated tool output.

Cancellation records local control flow. It does not say:

- the provider did no work;
- a remote tool had no effect;
- billing did not occur;
- an artifact was not created; or
- retry is safe.

Chapter 15's effect knowledge remains authoritative. Telemetry can point operators to a reconciliation path; it cannot replace the operation ledger or receipt.

Provider failures end model and HTTP spans with safe error categories. Tool failures preserve whether policy rejected before execution or whether execution began. A timeout after execution starts remains compatible with outcome-unknown at the recovery layer.

16.6. Concurrency

Concurrent HTTP invocations receive distinct root traces. Tests start two model/tool runs together and verify that child spans remain under the correct root. They do not place request IDs or prompts in attributes to achieve that isolation; OpenTelemetry span context performs correlation.

The test also checks controlled cardinality. Tool names come from the static registry, providers/models from reviewed routes, and outcomes/phases from finite enums. A future dynamic MCP or A2A destination must not become an unrestricted span value merely because it is useful for debugging.

16.7. Deterministic exporter and evidence

The checked-in profile uses `BasicTracerProvider`, `SimpleSpanProcessor`, and `InMemorySpanExporter`. This is suitable for deterministic tests, not production throughput.

The evidence command runs under the repository's network-denial preload:

```
bun test test/telemetry.test.ts
bun run evidence:telemetry
```

The generated artifact:

```
evidence/observability/2026-10-23-opentelemetry-sanitized-trace.json
SHA-256 2fbaf4d669a66e74c7680d449bb1b0e58c5155b74e9d627ad05e75f027cf0119
```

normalizes trace/span IDs to local aliases and removes timestamps and durations. Two consecutive generations produce identical bytes. It records dependency versions, topology, redaction defaults, test coverage, and limitations.

The integrated tests cover:

- successful HTTP/ingestion/model/round/tool/nested hierarchy;
- concurrent trace separation;
- cancellation before execution;
- whole-tool timeout;
- Pyodide queueing and non-cooperative worker termination;
- hosted-search and image-recipient nested calls;
- provider failure; and
- absence of prompts, search terms, source bodies, image observations, Python code/output, credentials, and answer text.

The artifact is sanitized deterministic evidence, not a live provider trace or collector interoperability result.

16.8. Production exporter gate

The reference intentionally does not configure OTLP. Exporting traces creates another data recipient and failure path. A production profile requires review of:

- collector endpoint and authentication;
- TLS and network egress;
- sampling before expensive attribute construction;
- batching, queue bounds, backpressure, and drop behavior;
- exporter timeout and shutdown;
- tenant separation and access control;
- retention, deletion, backup, and legal policy;
- field-level redaction tests at the collector and backend;
- cardinality/cost budgets; and
- incident handling when telemetry is unavailable.

Telemetry failure must not silently authorize work. Whether it should fail a production request is a product policy decision; the educational profile treats observation as best effort and keeps operation semantics independent.

16.9. Metrics and logs

This implementation supplies traces only. Metrics and logs remain proposed.

A future low-cardinality metric set can count admitted requests, model rounds, policy outcomes, tool execution categories, queue delay buckets, cancellation sources, and unknown-effect reconciliation. It must not label by raw source URL, prompt, filename, call ID, trace ID, user text, or exception message.

Structured logs should use the same safe category vocabulary and access controls. Copying complete spans into logs defeats the trace redaction policy. Trace/log correlation is useful only when identifiers and retention remain bounded and authorized.

16.10. Progress is not telemetry

Pi progress cards show a user what the current operation is doing. They can contain approved query previews or source links that are explicitly excluded from operator telemetry. OpenTelemetry spans can outlive a UI connection and can be exported to another backend.

Therefore:

- progress never authorizes or executes tools;
- telemetry never becomes model context;
- a progress card is not a span receipt;
- a successful span is not a user-visible completion guarantee; and
- redaction rules differ because recipients and purposes differ.

16.11. Failure modes

Prompt-as-attribute. Convenient debugging turns the collector into a shadow conversation database. Export categories, not content.

Every ID as correlation. Request/source/artifact IDs create high cardinality and membership leakage. Use trace context for causal correlation and retain business IDs only in separately protected records.

Span success as task success. A provider call completes but the answer is unsupported. Keep semantic evaluation outside telemetry.

Cancellation as rollback. A cancelled span triggers automatic retry. Consult effect knowledge and reconciliation first.

Exporter authority. An exporter error changes tool policy or approval. Observation must not grant authority.

Untrusted parent context. Caller headers join internal traces across tenants. Authenticate and constrain propagation before accepting it.

16.12. Pattern summary

Problem: Agent runs cross model, tool, worker, and provider boundaries, while ordinary tracing can leak the content it is meant to diagnose.

Pattern: Propagate trusted OpenTelemetry context across fixed operation spans, admit only closed low-cardinality attributes, normalize terminal categories, and test redaction with an in-memory exporter.

Result: Operators can distinguish ingestion, model, policy, execution, queue, worker, and secondary-provider behavior without receiving agent payloads.

Cost: Production exporters, metrics, logs, cross-process propagation, retention, sampling, and collector security still require separate implementation and evidence.

16.13. Exercises

1. Add authenticated inbound trace propagation. Define when a caller's parent is accepted and how tenant isolation is preserved.
2. Design an OTLP batching policy whose full queue cannot block or authorize a tool call.
3. Propose three metrics for queue pressure without using request, model-output, or source identifiers as labels.
4. Construct a cancellation trace with an unknown remote effect. Explain which fact telemetry cannot supply.
5. Add a redaction regression for a secret split across two generated chunks without storing either chunk.

16.14. Chapter contract

- **Scope:** trace context, model/tool/ingestion/worker/nested-provider spans, terminal classification, redaction, and cardinality.
- **Canonical claims:** implemented tracing explains bounded execution topology but provides neither authority, semantic correctness, effect receipts, nor production exporter certification.
- **Migration inputs:** former telemetry taxonomy, current tool-progress contrast, reference instrumentation, and deterministic trace artifact.
- **Evidence gates:** exact dependency pins, in-memory exporter, hierarchy/concurrency/cancellation/error tests, redaction tests, and byte-stable sanitized evidence.
- **Demonstration slots:** deterministic sanitized trace only; no live production collector demonstration is claimed.
- **Source-claim ledger:** OpenTelemetry documentation informs terminology; implementation claims are tied to the exact dependency and reference revisions above.
- **Exclusions:** user-visible progress belongs to Chapter 8; recovery/effect knowledge belongs to Chapter 15; production security gates belong to Chapter 17.

17. Security, Evaluation, and Production Gates

Part IV · Portability in production

Evidence baseline: deterministic reference tests + scoped live records · production classification: not approved

A passing model demonstration is not a security test, and a passing deterministic test is not production certification. The reference is a loopback educational system. Production approval requires authenticated ownership, stronger isolation, reviewed egress and recipients, operational budgets, adversarial evidence, and a reversible release decision.

A **production gate** is an independently reviewable condition that must pass before a named deployment and task profile can receive real users or data. A gate names evidence, scope, owner, expiry, and failure action; it is not a confidence score supplied by the system being evaluated.

17.1. Start with assets and trust boundaries

The system handles several asset classes:

- provider credentials and endpoint configuration;
- user prompts, attachments, images, and source URLs;
- extracted document representations and invocation-local IDs;
- tool schemas, policy decisions, approvals, and results;
- generated answers, citations, presentations, and artifact IDs;
- transcript, recovery, and operation-state records;
- progress events, diagnostics, traces, and test evidence; and
- dependency locks, manifests, configuration, and release metadata.

Its principal trust boundaries are:

```
user / Pi client
  → HTTP input admission
  → coordinator provider
  → server-owned policy and tools
    → public network
    → secondary model recipient
    → document/parser workers
    → local Python/presentation workers
  → artifact handoff and local save
  → retained state / telemetry / publication
```

Every arrow is a data-recipient or authority decision. “The model is trusted” is not a useful threat statement: model output remains untrusted even when the provider and account are approved. Retrieved, attached, generated, image-derived, document, skill, gallery, presentation, and delegated-agent content remains data, never control-plane authority.

17.2. Threat model

The reference should be evaluated against at least these actor classes:

- an unauthenticated local process reaching the loopback listener;
- a legitimate user supplying malformed, oversized, hostile, or confidential content;
- a compromised or malicious public source;
- model output attempting to widen tools, recipients, limits, or paths;
- a remote provider returning malformed streams, unsafe text, or ambiguous terminal state;
- a dependency/parser/runtime defect;
- a compromised secondary service or telemetry recipient;
- concurrent users attempting cross-tenant access;
- an operator making a configuration or release mistake; and
- a reviewer overgeneralizing narrow evidence.

The current service assumes a trusted single-user workstation and binds to loopback. Loopback reduces exposure; it is not authentication. Another process under the same account can connect, and local extensions execute with user permissions.

17.3. Authentication and tenant ownership

The live handler currently mints `principal: local-development` and `tenant: loopback`. HTTP bodies cannot override them, which prevents a caller from self-assigning a server identity. It does not authenticate a human or isolate multiple tenants.

A production gate requires:

1. an authenticated principal derived outside request content;
2. tenant/organization ownership bound to that principal;
3. session and artifact lookup scoped to both values;
4. service-to-service authentication for every remote executor;
5. revocation and credential-rotation behavior;
6. non-disclosing authorization failures; and
7. tests for cross-tenant reads, writes, caches, artifacts, traces, and recovery.

A session ID, source ID, artifact ID, provider response ID, or trace ID is a lookup/correlation value, not proof of access. The SQLite transcript example compares trusted owner values but explicitly does not authenticate them.

17.4. Authorization and approval

Authentication answers who is calling. Authorization answers whether this principal may perform this operation on this data with this recipient now. Approval captures a task- or user-specific decision when policy requires one.

The policy runtime validates schema, checks current authorization and approval, rechecks cancellation, executes the compiled tool, validates output, and records a bounded outcome. Important production additions include:

- entitlement lookup under bounded latency;

- grant expiry and revocation;
- binding approval to normalized operation intent;
- destination and data-class restrictions;
- separation of read, compute, publish, overwrite, and delegate permissions;
- durable consequential-effect receipts; and
- protected audit evidence.

A model proposal, skill instruction, retrieved sentence, old transcript, generated summary, or successful previous run cannot become authorization. Persisted history grants no execution authority.

17.5. Network egress and SSRF

`read_source` implements public-HTTPS syntax checks, DNS/address screening, mixed-address rejection, pinned transport, hostname TLS validation, default-port policy, no ambient proxy, no credentials, no redirects, bounded response handling, and shared ingestion. These controls substantially narrow server-side request forgery.

They do not replace:

- an OS/network egress firewall;
- maintained public/special-use address classification for every supported family;
- controlled DNS infrastructure and rebinding tests;
- destination allowlists for sensitive deployments;
- per-tenant network policy;
- proxy and service-mesh review;
- monitoring of denied and anomalous destinations; or
- secure redirect support if redirects are later enabled.

Hosted search and model providers are separate approved egress recipients. Their SDK endpoint overrides and credentials belong to trusted environment/bootstrap configuration, never model/tool arguments or `agent.yaml` fields.

17.6. Document and parser isolation

The ingestion boundary validates declared bytes, media/profile, image dimensions, archive expansion, entries, nesting, pages, cells, entities, parser iterations, extraction deadlines, and canonical-text limits. Xberg work runs in a killable worker, and unsupported/encrypted/incomplete documents fail with bounded categories.

A parser worker in the same application account is not strong isolation. Production document handling requires:

- a separate process/container or service identity;
- read-only ephemeral input mounts;
- no ambient credentials or network;
- CPU, memory, file, process, and wall-clock limits;
- decompression and temporary-storage quotas;
- hardened image/document libraries and patch policy;
- malformed-corpus and known-vulnerability testing;

- cleanup verification after kill/crash; and
- output schema and byte validation at the trust boundary.

Model-backed layout inference remains disabled because local model artifacts have not been separately admitted. Enabling it would add dependency, compute, recipient, provenance, and evaluation gates.

17.7. Production code-execution isolation

The Pyodide interpreter creates a fresh worker, uses fixed invocation-local mounts, blocks ordinary JavaScript-bridge/package-installer imports, declares no network/packages/arbitrary host paths, and terminates on timeout/cancellation. This is useful defense in depth for a local reference.

It is explicitly not a safe hostile multi-tenant sandbox. Production execution requires:

- process/container/VM isolation under a dedicated identity;
- OS-enforced CPU, memory, process, file, and output quotas;
- default-deny network at the kernel or infrastructure layer;
- immutable reviewed runtime images;
- no host credential or socket inheritance;
- seccomp/sandbox profile where applicable;
- bounded queue fairness and tenant quotas;
- image provenance, vulnerability scanning, and patch SLAs;
- escape and denial-of-service testing; and
- independent kill/cleanup evidence.

The same principle applies to presentation generation. OOXML package inspection rejects active content and external relationships, but the worker and office-rendering acceptance environment still need isolated identities and controlled file access in production.

17.8. Prompt injection and untrusted content

Prompt injection is not solved by telling the model that content is untrusted. OWASP guidance distinguishes direct and indirect injection and recommends layered controls rather than prompt-only separation (OWASP, n.d.).

The reference's enforceable controls include:

- static tools and skills;
- strict typed arguments;
- server-owned recipients, credentials, limits, and paths;
- invocation-local source IDs;
- policy/approval outside the model;
- no execution of skill scripts;
- retrieved text that cannot register capabilities; and
- bounded output/rendering validation.

Residual risk remains:

- the model can choose an allowed tool for the wrong reason;

- a malicious source can influence synthesis or select sensitive search terms;
- permitted tools can be composed into harmful data flow;
- final prose can follow hostile instructions despite server enforcement;
- Pi-local tools have their own policy boundary; and
- semantic misuse can occur without violating a schema.

Evaluation must count forbidden effects and data releases, not merely whether the answer says “I ignored the instruction.”

17.9. Provider and recipient disclosure

A portable route can have several recipients. In the Northstar run:

- Anthropic coordinates;
- OpenAI receives one admitted image plus focused question;
- OpenAI hosted search receives focused public terms;
- the selected public origin receives an HTTPS read;
- Pyodide and PptxGenJS run locally.

Production admission must maintain a data-flow matrix for each operation:

Operation	Recipient	Minimum released data	Disclosure/evidence
Coordination	Selected model route	Prompt and admitted context	Provider/model class, policy
Image understanding	Fixed secondary model	One image + question	Explicit recipient + generated status
Search	Fixed search model/public search	Focused query	Recipient/network/cost disclosure
Source read	Selected public origin	HTTPS request metadata	Admitted URL/representation record
Local computation	Isolated runtime	Selected snapshots + code	Runtime/revision/limits
Presentation	Isolated local worker	Semantic deck + admitted images	Artifact digest/byte count
Telemetry export	Collector/backend	Closed safe attributes	Recipient/retention/sampling policy

Choosing a coordinator does not automatically consent to every secondary recipient. Region, retention, contractual terms, and data classification must be evaluated independently.

17.10. Secrets and diagnostics

Credentials remain in trusted environment/bootstrap settings and are never valid YAML fields, tool arguments, model context, progress events, or evidence artifacts. Pi credentials are not read by the server. Endpoint overrides require explicit operator approval.

Public errors use controlled categories. Safe diagnostics omit queries, document text, images, generated observations, Python code/output, credentials, headers, raw provider messages, and bodies. OpenTelemetry uses a closed attribute allowlist and excludes opaque IDs by default.

A production gate includes:

- secret-manager integration and rotation;
- least-privilege provider credentials;
- environment/process-dump review;
- log/trace redaction tests, including encoded and chunk-split forms;
- access-controlled diagnostic retention;
- incident-reference non-disclosure across tenants; and
- a scan of source, build artifact, container, configuration, and history.

Hashing sensitive content is not anonymization. Predictable prompts, URLs, or IDs can be guessed.

17.11. Rate, cost, deadline, and concurrency controls

The reference has per-invocation calls, sources, bytes, steps, deadlines, queueing, and output bounds. These are necessary but not production quotas.

A production service requires coordinated budgets across:

- principal and tenant;
- route/provider/account;
- task/session and HTTP invocation;
- search, retrieval, image, computation, and presentation;
- concurrent active workers and queued work;
- model tokens and provider currency estimates/measurements;
- artifact and retained-state storage; and
- retry/fallback amplification.

One owner must coordinate retry attempts. A six-hour request ceiling does not guarantee affordable completion. Provider-requested token/tool limits are not independently enforced remote CPU/billing limits. Unknown usage remains unknown and must not be converted to zero.

Rate limiting itself needs abuse tests and safe failure behavior. Queue bounds must prevent one tenant from starving others. End-to-end deadlines cannot reset on every model round, progress event, or retry.

17.12. Artifact and retained-state security

Presentation artifacts use random short-lived IDs, no listing endpoint, exact length/digest verification, mode 0600, and replacement confirmation. Production requirements add:

- authenticated tenant-bound retrieval;
- encrypted storage where required;
- bounded leases and deletion;
- malware/content scanning appropriate to format;
- immutable audit metadata;

- download headers and content-type enforcement;
- replacement/version policy; and
- backup/retention treatment.

Transcripts, retrieval memory, provider continuation, summaries, and operation ledgers have different authority and deletion rules. A generated summary cannot preserve approval. A transcript is not a remote effect receipt. Opaque provider state remains sensitive even when unreadable.

17.13. Evaluation stack

No single evaluation establishes production fitness. Use layers with distinct claims.

17.13.1. Deterministic invariant tests

Credential-free tests under network denial cover schema, policy, quotas, cancellation, event order, parsing, citations, cache ownership, artifact structure, digest handoff, telemetry redaction, and known failure cases. They establish local code behavior under fixtures.

They do not establish live provider compatibility, model quality, parser security against unknown exploits, or production operations.

17.13.2. Protocol and conformance suites

A pinned external suite measures a named protocol revision and target. Preserve all failures and the suite's exit status. Chapter 19 reports 8 passed, 9 failed, 0 skipped and classifies the server as a compatible subset, not conformant.

A suite pass is not a security assessment. Implementing prompt-specific behavior to satisfy cases is unacceptable.

17.13.3. Live transport evidence

Live records name date, exact commit, account/endpoint class, provider/model, request profile, observed outcome, cost context, and limitations. They establish that exact route under those conditions. They expire as evidence when configuration or dependencies change.

17.13.4. Semantic/task evaluation

Task evaluation starts from the promise. Northstar gates include evidence-class preservation, required tool procedure, correct two-premise comparison, missing/conflicting evidence behavior, recipient disclosure, and artifact integrity.

Use paired providers only when both are admitted for the same task profile. Shared interfaces do not establish semantic equivalence. Measure disaggregated errors and forbidden effects; do not average an authority violation into a quality score.

17.13.5. Adversarial evaluation

Include malicious documents, prompt injection, ambiguous identifiers, conflicting measurements, oversized/decompression inputs, stale/cross-tenant IDs, hostile URLs/DNS, malformed streams, duplicate terminal events, cancelled approvals, queue starvation, artifact replacement, and raw-error/secret echoes.

Success criteria should name effects: no unauthorized dispatch, no forbidden recipient, no cross-tenant read, no unreviewed overwrite, no unsafe package, and no content leak in progress/diagnostics/telemetry.

17.13.6. Manual artifact review

Manual review applies to exact bytes under exact renderer/application versions. It can assess layout and comprehensibility that OOXML checks cannot. It cannot certify future generated decks or every office suite.

17.14. Fault-injection program

Production readiness requires failures at boundaries, not only invalid input:

- disconnect before/after provider acceptance;
- malformed or duplicated stream terminal events;
- provider timeout after partial output;
- cancellation while policy/approval waits;
- process kill during worker execution;
- remote effect commit before local receipt;
- transcript write failure on multi-record append;
- stale revision under concurrent writers;
- DNS answer change and redirect attempt;
- full telemetry/export queue;
- artifact store restart/expiry during handoff; and
- dependency or parser crash.

Each scenario records visible output, effect knowledge, durable state, allowed next action, and whether reconciliation exists. Cancellation never becomes permission to repeat an uncertain effect.

17.15. Release approval gate

A release candidate is approved only for a named deployment/task profile after reviewers sign these categories:

1. **Identity and authority**—authentication, tenant ownership, entitlements, approvals.
2. **Recipients and data flow**—providers, network origins, collectors, regions, retention.
3. **Isolation and resources**—parsers, code, presentation, network, queues, quotas.
4. **Correctness**—deterministic tests, protocol suite, task evaluation, failure behavior.
5. **Privacy and diagnostics**—secrets, logs, progress, traces, artifacts, state.
6. **Operations**—monitoring, reconciliation, incident handling, capacity, backups/deletion.

7. **Supply chain**—exact dependencies, licenses, provenance, vulnerability review, workflow pins.
8. **Rollback**—previous safe version/configuration, state compatibility, traffic and credential plan.

The release record binds commit, Bun/Quarto/runtime versions, lock digest, configuration, tests, suite revision/results, live evidence, known defects, artifact hashes, reviewers, date, expiry, and limitations. Approval is scoped; it is not a permanent product certificate.

17.16. Rollback is a state transition

Rollback must answer:

- Can the old version read new transcript/evidence schemas?
- Are provider or tool grants still valid?
- What happens to in-flight requests and unknown effects?
- Are new artifacts/state retained or quarantined?
- Must credentials be rotated?
- Does the rollback restore a known vulnerability?
- Which live/conformance evidence no longer applies?

A binary rollback that ignores durable state can corrupt recovery. Prefer forward-compatible readers, explicit schema gates, drain/reconcile procedures, and reversible configuration flags. Never replay pending transcript calls merely because deployment changed.

17.17. Current blocked production claims

The reference does **not** currently establish:

- authenticated multi-user or tenant isolation;
- production entitlement/approval infrastructure;
- OS-isolated hostile document or Python execution;
- comprehensive prompt-injection resistance;
- destination-complete egress enforcement;
- durable source/artifact/effect identity across requests;
- formal local-file citations for every parser profile;
- exactly-once consequential effects;
- production OTLP export, metrics, or log correlation;
- MCP or A2A interoperability;
- semantic equivalence across providers;
- compatibility with every office suite; or
- formal Open Responses conformance.

These are blocked claims, not hidden roadmap details. The loopback reference can still provide valid educational evidence for its narrower implemented boundaries.

17.18. Pattern summary

Problem: A working demo and large test count can obscure missing identity, isolation, recipient, recovery, and operational controls.

Pattern: Define assets and trust boundaries, keep authority outside content, layer deterministic/protocol/live/semantic/adversarial evidence, and require a scoped release and rollback record.

Result: Production becomes an explicit decision about a named deployment and task rather than a marketing inference from generated output.

Cost: Authentication, isolation, egress, evaluation corpora, fault injection, operational ownership, and recurring evidence review remain substantial engineering work.

17.19. Exercises

1. Build a recipient matrix for a confidential Northstar variant. Identify which current routes must be disabled.
2. Specify a multi-tenant artifact lookup and test that random IDs alone do not grant access.
3. Design a process-isolated code runner with OS resource and egress controls. State what Pyodide checks remain useful.
4. Create an indirect-injection test whose success criterion is zero forbidden effects rather than refusal wording.
5. Draft a rollback plan across a transcript schema change and an unresolved publication effect.

17.20. Chapter contract

- **Scope:** threat boundaries, authentication/authorization, isolation, injection, egress, recipients, secrets, budgets, evaluation, fault injection, release, and rollback.
- **Canonical claims:** the current loopback reference is not production-security certified; production approval requires scoped evidence across every trust and effect boundary.
- **Migration inputs:** security sections across tool/document chapters, publication/artifact checks, live-evidence policy, and implemented telemetry redaction.
- **Evidence gates:** deterministic invariants, protocol results, dated live records, task/adversarial evaluation, fault injection, and signed release/rollback records.
- **Demonstration slots:** Northstar and document inspection are scoped functional evidence only, never production certification.
- **Source-claim ledger:** security guidance and mutable provider/protocol claims retain source revision, review date, implementation scope, and limitation.
- **Exclusions:** concrete tool mechanisms stay in their owning chapters; final Open Responses classification belongs to Chapter 19.

Part V.

**Part V — Open Ecosystem and
Validation**

18. MCP and A2A: Transport and Delegation Boundaries

Part V · Open ecosystem and validation

Architecture baseline: MCP 2025-11-25 · A2A 1.0.0 · source review 2026-10-07 · no reference implementation claim

An MCP server exposes contextual capabilities. An A2A participant accepts delegated work. Both cross a trust boundary, but they do not have the same lifecycle, authority, identity, cancellation, or evidence obligations—and neither is a packaged skill.

Capability transport invokes or reads a bounded remote capability under a host-owned task. **Delegation** asks another agent to own some task progression. A **skill** remains non-executable procedure inside the current agent. Similar JSON does not collapse these responsibilities.

18.1. Four boundaries, not one “agent stack”

```
current application
|-- local tool port ----> compiled local executor
|-- MCP client -----> authorized MCP server/session
|-- A2A client -----> remote agent task/artifact lifecycle
`-- skill loader -----> reviewed non-executable procedure
```

Boundary	Primary unit	Lifecycle owner	Typical result	Authority question
Local tool	Correlated call	Current application	Typed result	May this executor run now?
MCP	Resource/prompt/tool operation	Host plus negotiated client/server session	Protocol content/result	May this server and capability receive this context?
A2A	Message/task	Delegating application plus remote agent	Status, messages, artifacts	May this agent own and advance this delegated work?
Skill	Reviewed text/resource	Current application	Non-executable guidance	May this procedure enter context?

The reference implements local tools and one static skill package. It does not implement MCP or A2A. This chapter specifies the admission boundary required before a future implementation can claim compatibility.

18.2. MCP's purpose and lifecycle

MCP defines host, client, and server roles over a structured protocol, with initialization/version agreement and capability negotiation before ordinary operation (“Model Context Protocol Specification 2025-11-25,” n.d.; “Model Context Protocol 2025-11-25,” n.d.). Depending on negotiated capabilities, a server can expose tools, resources, prompts, and related lifecycle notifications.

A safe host lifecycle is broader than “connect and list tools”:

```
trusted configuration selects destination/credential
→ transport connection
→ initialize and negotiate protocol/capabilities
→ authorize advertised operations against local policy
→ expose an admitted projection to the coordinator/user
→ invoke/read under per-operation limits
→ validate results and notifications
→ close, cancel, or reconcile session state
```

Initialization can establish protocol compatibility and peer-advertised features. It cannot decide whether the peer is trusted, which tenant may use it, which data may be released, or whether advertised tools are semantically suitable.

18.2.1. Discovery is inventory, not permission

An MCP tools/list or resources/list response describes what the server advertises. The host still owns:

- destination and server identity;
- credential and tenant binding;
- admitted capability names and versions;
- schema and result-size profile;
- model visibility;
- per-call authorization and approval;
- released context and source classes;
- rate, deadline, and concurrency budgets; and
- retention and audit policy.

The model cannot add an MCP server by emitting a URL, package, command, or configuration object. Retrieved content cannot request a new server. A capability newly appearing after a list-change notification remains unapproved until local policy admits it.

18.2.2. Tools, resources, and prompts remain distinct

MCP tool invocation can fit behind the application's generic tool port, but remote execution adds destination, credential, session, transport, and provenance fields. MCP resources expose content or references under server-defined identity; they are not automatically trusted evidence. MCP prompts are reusable server-provided templates; loading one cannot grant tool authority or become a higher-priority instruction by virtue of transport.

A host should project only the minimum semantics needed by the coordinator. It must not flatten all server content into undifferentiated prompt text or represent every resource as a local file.

18.3. MCP admission record

A production admission record binds:

- reviewed server identity and deployment alias;
- transport class and trusted endpoint selected outside model content;
- credential owner, scope, expiry, and recipient;
- negotiated protocol revision and capability set;
- admitted tools/resources/prompts and schema digests;
- principal, tenant, purpose, and data-release classes;
- argument, result, pagination, subscription, and notification limits;
- cancellation, timeout, reconnect, and unknown-effect behavior;
- safe diagnostics and telemetry labels; and
- result/provenance validation policy.

A model-facing name should map to this record, not to a caller-provided endpoint. Schema changes invalidate prior admission evidence. Protocol negotiation is necessary but cannot substitute for that record.

18.4. A2A's purpose and lifecycle

A2A represents interaction with another agent through discoverable capabilities, messages, tasks, statuses, and artifacts (“Agent2Agent Protocol Specification 1.0.0,” n.d.). A delegated participant can plan within an admitted scope, request additional information, progress asynchronously, and return more than one artifact.

A safe delegation lifecycle includes:

```
trusted catalog selects remote agent
→ inspect/admit advertised capability
→ create local delegation intent
→ release minimum task context
→ correlate local and remote task identities
→ observe status/messages/artifacts
→ answer input requests under fresh policy
→ cancel, complete, fail, or reconcile
→ admit returned artifacts/evidence locally
```

This is not just a slow tool call. A remote agent can own intermediate progression and maintain its own state. Flattening the exchange into one `execute()` result loses who owns the task, whether input is pending, which artifacts exist, and what cancellation means.

18.4.1. Agent discovery is not delegation authority

An advertised skill or capability says what a remote participant claims it can do. It does not prove identity, quality, safety, current availability, or authority to receive data. The delegating application must admit a static or authenticated catalog entry, task profile, recipient, and released context.

A user asking “find another agent” does not authorize arbitrary network discovery in a confidential workflow. A document containing an agent card or endpoint is untrusted content.

18.4.2. Remote artifacts need local admission

Returned A2A artifacts have remote identity and provenance. Before they enter local context, storage, tools, or publication, validate:

- expected task and remote agent identity;
- media type, size, count, and complete transfer;
- digest/signature when the contract provides one;
- parser profile and active-content policy;
- source/evidence labels;
- retention and deletion; and
- whether the artifact is an assertion, derivation, or verified result.

A polished remote presentation remains untrusted bytes. A remote citation is not automatically an admitted local source relation.

18.5. Transport versus authority

Transport answers how a message moves and how peers correlate it. Authority answers who may cause which operation with which data and recipient. Neither MCP JSON-RPC nor A2A task messaging grants application authority.

A portable integration keeps these layers explicit:

Layer	Question
Transport	Can bytes/messages be exchanged under the selected protocol?
Peer identity	Which reviewed server/agent and credential holder is connected?
Negotiation	Which protocol features were mutually selected?
Local policy	Which operations/data/recipients are currently allowed?
Execution/delegation	What operation or task was actually started?
Evidence	What result/artifact/status was observed and how was it validated?
Semantic evaluation	Is the capability suitable for this task profile?

Passing the first three rows cannot imply the last four.

18.6. Identity and confused-deputy risk

Transport authentication can establish a peer or credential holder. It does not prove that a requested action is authorized for the current principal. Both integrations can become confused deputies when model or retrieved text selects a recipient, requests broader context, or induces a privileged downstream operation.

Controls include:

- static deployment aliases and authenticated discovery;
- tenant/principal-bound grants;
- separate least-privilege credentials;
- task-scoped released context;
- explicit data-recipient disclosure;
- no arbitrary headers, endpoints, packages, or commands;
- bounded server/agent redirects and reconnects;
- artifact/source admission on return; and
- audit records naming requester and recipient.

Prompt text, remote descriptions, and server-returned prompts never become grants.

18.7. Delegation boundaries

An A2A grant should state:

- remote agent identity and admitted capability revision;
- task purpose and allowed planning latitude;
- maximum released sources and fields;
- permitted tools/recipients, if the remote contract exposes them;
- cost, deadline, attempt, and artifact budgets;
- whether the agent may ask for more input;
- which statuses are terminal;
- cancellation and status-lookup semantics;
- artifact types and publication rules; and
- conditions for local acceptance.

The local application does not regain semantic control merely because it can cancel. It must evaluate remote output and preserve that the work was delegated.

18.8. Data recipients and disclosure

MCP and A2A introduce explicit recipients beyond the coordinator. A record should identify:

- peer organization/service class;
- deployment/region where relevant;
- credential owner;
- exact context released;
- subrecipient/tool behavior known to the contract;
- retention/storage mode;
- telemetry/export destinations; and
- returned content/artifacts.

If the remote agent can choose further providers, the delegation policy must decide whether that is permitted and how it is disclosed. “Uses A2A” is not an adequate recipient statement.

Minimize context. An MCP tool usually needs typed arguments, not the whole transcript. A delegated agent may need a structured task and selected evidence, not every attachment and prior message.

18.9. Cancellation, errors, and reconciliation

A local cancel can race with remote execution. The correct state may be:

- cancelled before admission;
- cancellation requested, remote status unknown;
- remote cancelled with verified terminal status;
- completed before cancellation;
- failed with no effect;
- failed after possible effect; or
- disconnected while a task/artifact remains active.

MCP effectful tools need correlated receipts and operation-specific status lookup. A2A tasks need remote task identity, status reconciliation, and artifact inventory. Local timeout cannot rewrite uncertainty as “nothing happened.”

Retries require a new local attempt identity unless the remote operation supplies a verified idempotency contract. Reconnecting a session is not permission to replay prior calls or recreate a task. A remote error message cannot set local retry count, fallback recipient, or approval.

Safe public errors use bounded categories. Protected diagnostics can retain reviewed peer/status identifiers under access and cardinality controls, never unrestricted remote bodies.

18.10. Protocol translation loss

Mapping MCP or A2A into the reference’s neutral model/tool items is lossy.

Potential MCP losses include:

- negotiated capability/version state;
- resource identity and subscription/update semantics;
- rich content blocks and annotations;
- server notifications;
- transport/session metadata;
- progress/cancellation details; and
- protocol-specific error data.

Potential A2A losses include:

- task versus message identity;
- nonterminal and input-required states;
- remote planning ownership;
- asynchronous status progression;
- multiple artifacts and parts;
- push/reconnect behavior; and
- remote provenance.

Do not hide those losses behind one generic tool result. Either extend the neutral contract with typed optional envelopes and tests, or keep a protocol-specific state plane adjacent to the portable spine.

An adapter must record unsupported fields and rejection behavior. Silent dropping can turn a pending task into apparent completion or detach an artifact from its owner.

18.11. Negotiated capability is not semantic portability

A peer can correctly advertise and execute a tool while producing unsuitable results. Two MCP servers with the same tool schema can differ in data, effects, freshness, latency, and policy. Two A2A agents advertising the same capability can differ in planning, recipients, evidence, and quality.

Therefore distinguish:

1. protocol compatibility;
2. peer identity and currently authorized capability;
3. task-profile suitability;
4. observed functional evidence; and
5. semantic evaluation.

A shared schema proves none of the later rows by itself. Provider-neutral model interfaces follow the same rule.

18.12. Telemetry without payload export

Future spans should use fixed operation categories such as MCP initialize/list/call/read and A2A create/status/input/cancel/artifact-admit. Peer aliases must come from the reviewed catalog, not raw endpoints. Task IDs, URLs, arguments, resource bodies, messages, artifacts, credentials, and remote errors remain excluded by default.

Cross-service trace propagation is a separate grant. Do not forward internal trace/baggage headers to an arbitrary peer. If enabled, bind trust domain and tenant, constrain baggage, and document the new correlation recipient.

18.13. Evidence and conformance limits

Before claiming MCP or A2A support, implement and test:

- pinned protocol revision and strict message validation;
- trusted destination, identity, and credential binding;
- initialization/negotiation and unsupported-feature behavior;
- discovery changes and local re-admission;
- input/result/artifact size and content limits;
- cancellation, timeout, reconnect, and unknown-effect reconciliation;
- cross-tenant and confused-deputy tests;
- hostile remote content and malformed lifecycle transitions;
- translation-loss fixtures; and
- independently reproducible interoperability with a named peer/version.

Then report the claim precisely: implemented profile, tested operations, suite/interoperability result, failures, and exclusions. Protocol resemblance or one successful exchange is not formal conformance.

The current reference has none of those implementation results. This chapter is an evidence-gated architecture specification.

18.14. Failure modes

Discovery-as-permission. Every advertised tool becomes model-visible and executable. Project through local admission and policy.

Prompt transport as authority. A server prompt is inserted at system priority. Treat it as reviewed/untrusted procedure according to local policy.

Delegation flattened to call. Input-required or active status appears as completed output. Preserve task lifecycle.

Endpoint injection. Retrieved text supplies a server or agent URL. Use trusted catalogs only.

Cancellation laundering. Disconnect triggers duplicate task creation. Reconcile remote identity and status.

Artifact laundering. Remote files bypass local ingestion because the agent was trusted. Admit every returned artifact.

Shared schema equivalence. Two peers accept the same JSON, so outputs are assumed interchangeable. Evaluate task semantics separately.

18.15. Pattern summary

Problem: Open protocols can be mistaken for automatic trust, authority, or semantic interchangeability.

Pattern: Treat MCP as admitted capability transport and A2A as admitted delegation; preserve protocol lifecycle, peer identity, recipient, local policy, cancellation, artifacts, and translation loss.

Result: Future integrations can reuse portable application boundaries without collapsing remote sessions/tasks into local skills or generic model calls.

Cost: Catalogs, credentials, state planes, reconciliation, artifact admission, interoperability evidence, and task evaluation remain necessary.

18.16. Review questions

1. Can model or retrieved text select a new MCP/A2A destination?
2. Which principal authorized the exact capability or delegated task?
3. What context crossed the boundary, and which subrecipients are allowed?
4. Can cancellation leave an unknown remote effect, task, or artifact?
5. Does local state preserve remote identity and nonterminal status?
6. Which protocol fields are lost in the neutral projection?
7. Which evidence establishes protocol behavior, and which separately establishes task suitability?

18.17. Chapter contract

- **Scope:** MCP capability transport and A2A delegation purpose, lifecycle, discovery, identity, recipients, cancellation, translation, and evidence.
- **Canonical claims:** discovery/negotiation does not grant authority; delegation is not a long-running local tool call; shared protocol shapes do not prove semantic portability.
- **Migration inputs:** standards material from the former open-agent-stack chapter; skill material remains in Chapter 9.
- **Evidence gates:** pinned protocol tests, trusted identity/destination binding, lifecycle/reconciliation, translation-loss fixtures, and reproducible interoperability.
- **Demonstration slots:** withheld until an implementation passes the gate.
- **Source-claim ledger:** MCP 2025-11-25 and A2A 1.0.0 architecture claims are distinguished from the reference's unimplemented status.
- **Exclusions:** local tool authorization belongs to Chapter 8; skills belong to Chapter 9; telemetry belongs to Chapter 16; final Open Responses classification belongs to Chapter 19.

19. Production Case Study and Formal Conformance

Part V · Open ecosystem and validation

Functional baseline 45a903c · Open Responses suite 92c12d96 · 8 passed / 9 failed / 0 skipped · compatible subset; not conformant

The Northstar workflow is useful because it works across files, providers, public retrieval, local computation, skills, and artifacts. It is useful as evidence only when every boundary keeps its own claim. A polished answer and deck cannot erase a failed conformance test, an unreviewed recipient, ephemeral source identity, or missing production isolation.

19.1. The integrated system

The production-shaped case study begins in Pi with three synthetic inputs and one explicit re-search request:

- a generated image of the fictional Northstar Field Console;
- a one-page fictional specification PDF; and
- a synthetic measurements workbook.

The request asks the system to inspect visible labels, distinguish specification declarations from measurements, find and read current official public guidance, calculate mean runtime, preserve uncertainty, and avoid presenting the fictional product as real. A follow-up requests a short PowerPoint deck.

The execution topology is:

```
Pi 1.0.4 -- Open Responses HTTP/SSE --> loopback Bun server
  → ordered image/PDF/XLSX admission
  → Anthropic coordinator route
    → OpenAI image-understanding bridge
    → OpenAI hosted-search bridge
    → screened public-HTTPS read_source
    → local fresh Pyodide worker
  → evidence-labelled answer
  → integrity-bound business-tech-presentation skill
  → separately authorized local presentation_builder
  → short-lived artifact
  → Pi byte-count/SHA-256 verification and mode-0600 save
```

This is one orchestrated task, not one provider capability. Configuration, server-owned tools, invocation-local assets, and recipient disclosures preserve the boundaries.

19.2. Exact implementation and evidence baselines

Several immutable or dated baselines coexist. They must not be merged into one “current version” claim.

Baseline	Identity	Supports	Does not support
Current reviewed functional snapshot	45a903c84c30ac1bcc9616b5b1d665f10912f00	Published Northstar and document-inspection behavior	Immutable release identity or telemetry additions
Latest immutable release	v0.4.0 / 5ecc4f01e61858f16e42b5b234eac234855d6d	Released package behavior	Every later presentation/inspection change
Historical Part II baseline	v0.3.0 / 1c7a18a8434e5143cbe9ad1b8252af1f9b51	Historical evidence where explicitly cited	Current live server behavior
Telemetry implementation evidence	Reference revision ce47fe2cf5f109ef4939fd704e4956682d4af03d; OTel API 1.9.0, SDK/context 2.12.0; artifact SHA-256 2fbaf4d...0119	Deterministic span topology and redaction at that exact revision	Tagged release, OTLP, production collector
Open Responses suite	92c12d96d7b61d6d15e220ff159c609a16e1c	Official private suite result against reviewed target	Security, semantics, future suite revisions

Telemetry is now tied to immutable commit ce47fe2cf5f109ef4939fd704e4956682d4af03d, but it is not part of the earlier v0.4.0 tag. The deterministic artifact and tests establish the bounded in-memory profile at that commit; they do not establish production OTLP or collector operation.

19.3. Input and document boundary

The server accepts ordered text, reviewed file types, and reviewed image types under one request-body and aggregate asset profile. It validates base64/data input, byte counts, media, pages/cells/images, expansion, parser work, and extraction deadlines. Invocation-local IDs bind admitted assets for tools.

For Northstar:

- the PDF supplies **attached declarations**;
- the XLSX supplies **workbook measurements**;
- the image supplies pixels for a **generated observation**;
- attachment IDs expire with the HTTP invocation; and
- the files are fictional fixtures with a public manifest.

This boundary does not authenticate real-world origin or create durable formal file citations. Attachments must be resent in a later invocation.

The separate BCAUSS recording demonstrates model-independent local inspection: 16 pages, 44 chunks, 129,145 characters, zero warnings, a retained digest, and zero provider/public-network calls for the reviewed run. It proves extraction/inspection behavior for that public author-supplied paper, not factual correctness or redistribution of the PDF.

19.4. Provider and recipient routes

The recorded Northstar coordinator is:

`portable-agents/anthropic/claude-opus-5-5`

Selected sub-operations have separate recipients:

Operation	Recipient	Released data	Result class
Coordinator	Anthropic route	Prompt, admitted text/markers, tool results	Generated orchestration/answer
Image understanding	OpenAI gpt-5.4	One admitted image + focused question	Generated, unverified observations
Hosted search	OpenAI gpt-5.4	Focused public query	Candidate URLs + generated notes
Source read	Selected public HTTPS origin	Bounded request	Admitted representation
Computation	Local Pyodide	Selected invocation snapshots + Python	Deterministic result under inputs
Presentation	Local PptxGenJS worker	Semantic deck + admitted images	Candidate .pptx artifact

The coordinator receives an image marker, not original image bytes, until the image tool selects that asset. Search receives the focused query rather than the entire transcript. Recipient metadata is part of the result/evidence.

This demonstrates mixed-provider composition. It does not demonstrate semantic equivalence, automatic fallback, or a visible coordinator switch during the recording.

19.5. Search, retrieval, and research completion

`web_search` spends invocation-owned call and unique-URL budgets, requires completed native execution evidence, admits concurrent results in completion order, and returns bounded candidates. Those candidates are not source representations.

`read_source` then applies public-HTTPS URL, DNS/address, pinned TLS, redirect, response, media, encoding, shared-ingestion, and output limits. Its successful result records an invocation-local representation identity and retrieval time.

The final research answer keeps five classes separate:

1. attached specification declarations;
2. workbook measurements;
3. generated image observations;
4. deterministic calculations; and
5. retrieved public guidance.

A citation URL identifies provenance; it does not establish truth or entailment. Research completion requires the task's evidence matrix or an explicit gap.

19.6. Computation boundary

A fresh local Pyodide worker receives only fixed copies selected by invocation-local IDs. It has no declared network, package installation, arbitrary host paths, persistent state, or durable generated artifact.

For the workbook values 9 and 7.5 hours, the mean is:

$$(9 + 7.5) / 2 = 8.25 \text{ hours}$$

The calculation is deterministic under two finite hour measurements and the stated missingness policy. `null` is missing, never zero. The result does not prove the fixture values describe a real product.

The worker is not an OS sandbox. Production hostile-code isolation remains blocked on a separate process/container boundary and OS resource/egress controls.

19.7. Skill and presentation boundary

`business-tech-presentation` is an integrity-bound, non-executable procedure. The static catalog and manifest verify package identity, resources, byte counts, and SHA-256. Progressive `load_skill` changes context, not authority.

`presentation_builder` is selected separately. It accepts semantic slide data and rejects code, paths, coordinates, templates, package names, remote assets, macros, OLE, and external relationships. The server maps semantic slides through reviewed recipes/components and validates the OOXML package.

The Pi extension fetches the short-lived artifact, verifies exact byte count and SHA-256, saves with mode `0600`, and requires confirmation before replacement. A verified structural package is not automatically visually approved. The published recording did not visually inspect the generated deck during that session; separate artifact review is exact-byte/version scoped.

19.8. Recovery and durable state

The integrated deterministic trace distinguishes safe replacement before output from unknown tool effects. A lost receipt after execution produces `reconcile`, not `retry`. Transcript reopen reconstructs records without dispatching pending calls. Cancellation cannot authorize repeating an uncertain effect.

The local SQLite experiment provides atomic append/read under trusted owner values and expected revision. It is not authentication, remote effect atomicity, or automatic resume authority. Provider continuation and Pi sessions remain separate state planes.

For presentation handoff, an unexpired artifact ID can be fetched and digest-verified after client interruption. Expiry/restart can make the exact artifact unavailable; rebuilding is a new operation and can produce different bytes.

19.9. Telemetry boundary

The OpenTelemetry implementation at `ce47fe2cf5f109ef4939fd704e4956682d4af03d` emits content-free spans for HTTP, ingestion, model invocation/round, tool proposal/policy/execution, queue/worker, and nested provider operations. Context propagates through async and host-worker scheduling boundaries.

The deterministic evidence artifact is byte-stable and excludes prompts, source bodies, image bytes/observations, tool values, Python code/output, credentials, raw errors, and opaque IDs. It uses an in-memory exporter. No production OTLP recipient, sampling, metrics, logs, retention, or collector-security claim is made.

Span success is operational completion only. It does not establish factual support, effect rollback, or task completion.

19.10. Four conformance statements

Conformance language must identify the level being claimed.

Statement	Meaning in this project	Current status
API resemblance	Endpoint and payload/event vocabulary look like Open Responses	Present
Implemented compatible subset	Reviewed HTTP/JSON/SSE and input/tool paths work for documented clients/tests	Present, bounded
Official-suite result	Pinned public suite ran against the target and preserved failures	8 passed, 9 failed, 0 skipped
Formal conformance	All required profile behavior and conformance criteria are satisfied	Not established

API resemblance is the weakest statement. Compatible-subset operation is useful and testable but deliberately incomplete. An official-suite result is evidence, not a synonym for passing. Formal conformance cannot be claimed while required cases fail.

19.11. Official Open Responses suite result

The wrapper pins official suite revision:

```
92c12d96d7b61d6d15e2214daa5e9c6000ab6e1c
```

It accepts loopback targets, verifies the CLI digest, uses a frozen install with lifecycle scripts disabled, bounds report size, independently verifies summary counts, and preserves the official failing exit status.

Latest reviewed complete result:

```
8 passed
9 failed
0 skipped
compatible subset; not conformant
```

Seven passes and all nine failures contacted the HTTP target. The result is credential/target-specific evidence under the named suite and configuration. It is not a live-provider quality result or security audit.

19.12. Missing required behavior

19.12.1. /v1/responses/compact

The server does not implement the compact endpoint/profile required by three pinned cases:

- compact response;
- compact missing-model behavior; and
- compact output starting a new WebSocket chain.

Producing schema-shaped summary items is not enough. A credible implementation needs quality/loss policy, state ownership, bounds, tenant isolation, cancellation, error handling, and compatibility with subsequent continuation.

19.12.2. WebSocket lifecycle and state

Six failures cover:

- basic WebSocket response;
- sequential responses;
- continuation on an active connection;
- reconnect recovery;
- missing previous-response state; and
- eviction after failed continuation.

The current HTTP/SSE lifecycle cannot be called WebSocket-compatible. Implementing a connection-local cache solely to satisfy fixtures would not establish durable, secure, multi-tenant state.

19.13. Decision on missing conformance features

Formal conformance is **not the immediate implementation goal** for this publication revision. The project will not add narrow compaction/WebSocket behavior merely to increase a pass count before state ownership, bounded concurrency, tenant isolation, and security semantics are designed and tested.

The decision is:

1. retain and publish the failing result;
2. keep classification **compatible subset; not conformant**;
3. treat compaction and WebSocket lifecycle/state as blocked functionality;
4. require an implementation plan covering state, isolation, cancellation, recovery, and conformance—not test recognition;
5. rerun the exact pinned suite after any implementation; and
6. evaluate newer suite/spec revisions separately rather than rewriting history.

If formal conformance later becomes a release objective, the work must include /v1/responses/compact, WebSocket lifecycle, active continuation, reconnect recovery, missing-state errors, failed-chain eviction, bounded concurrency, tenant isolation, and retained exact evidence.

19.14. Consolidated evidence matrix

Boundary	Deterministic evidence	Dated/live evidence	Current limitation
Pi client	launcher/routing/extension tests	Northstar and inspection recordings	Pi-local session is not server tenant identity
Configuration	strict YAML/catalog tests	startup records	no production secret/entitlement service
Inputs	parser/admission/cancellation tests	Northstar BCAUSS inspection	ephemeral identity; no all-format formal citations
Coordinator adapters	synthetic OpenAI/Anthropic fixtures	Northstar Anthropic route	no provider equivalence
Search	receipt/budget/concurrency tests	Northstar OpenAI search	candidates/notes are not read evidence
Retrieval	URL/DNS/TLS/cache/indexing tests	Northstar official page selection	no redirects/authenticated/private sources
Image	masking/recipient/termination tests	fixed OpenAI recipient	generated observation, not verified OCR/truth
Computation	real local Pyodide tests	Northstar calculation card	no OS-level hostile-code isolation
Presentation	schema/composition/OOXML/stylesheet tests	Northstar deckshoff + separate reviewed artifact	exact-artifact visual scope only

Boundary	Deterministic evidence	Dated/live evidence	Current limitation
Recovery/state	retry, unknown-effect, SQLite tests	none required	no durable remote effect ledger
Telemetry	ce47fe2 in-memory hierarchy/redaction tests + artifact	no collector run	no production OTLP/metrics/logs
Security	deterministic controls/failure fixtures	privacy-reviewed recordings	not production approved
MCP/A2A Open Responses	architecture only parser/reducer/HTTP fixtures	none pinned official suite	unimplemented 9 failures; not conformant

19.15. Blocked-work register

Blocked claim/function	Missing evidence or implementation	Required next gate
Formal Open Responses conformance	Compact + WebSocket lifecycle/state	secure bounded implementation + pinned suite rerun
Production multi-tenancy	authentication, ownership, entitlements, isolation	adversarial cross-tenant and operational review
Hostile code execution	OS resource/network boundary	container/process implementation + escape/DoS testing
Production document parsing	stronger process isolation and corpus evidence	isolated parser service + fault/vulnerability tests
Durable local-file citations	retained identity and page/span mappings	authenticated source store + extractor mapping tests
Cross-request attachment/artifact identity	trusted ownership/lifecycle	durable store, retention/deletion, authorization
Exactly-once effects	remote idempo-tency/receipts/reconciliation	operation ledger + effect-system support
Production telemetry	OTLP, collector security, sampling/retention	exporter implementation + privacy/load tests
MCP/A2A support	transport/delegation implementations	pinned interoperability and lifecycle evidence
Provider semantic equivalence	task distribution and paired thresholds	repeated disaggregated evaluation
Universal presentation compatibility	suite/font/renderer matrix	exact-version render and manual review
Public gallery without lineage exposure	full review and neutralized metadata	312-slide audit and independent-publication review

19.16. What the case study proves

The integrated evidence supports this narrow statement:

Under the reviewed loopback configuration and exact dated evidence, a Pi client can route a fictional mixed-input research task through provider-neutral coordination, separately authorized bounded tools, screened public retrieval, local computation, an integrity-bound non-executable presentation procedure, and digest-verified local artifact saving while preserving recipient and evidence-class distinctions.

It does not prove deterministic model output, factual truth of external sources, real-product performance, provider equivalence, production security, exactly-once effects, every renderer's compatibility, MCP/A2A support, or formal Open Responses conformance.

19.17. Release checklist for the integrated profile

Before publishing a new baseline:

1. freeze reference commit and exact dependency lock;
2. run credential-free checks and Pi-extension typecheck;
3. regenerate deterministic telemetry evidence;
4. build/verify presentation artifacts when required;
5. rerun approved live workflows only with explicit paid-call authorization;
6. review media frame by frame and retain transcript/alt/metadata/hashes;
7. run the pinned official conformance suite and preserve failure exit;
8. update the evidence matrix and blocked register;
9. build the book from pinned Quarto and validate HTML/PDF/accessibility/links/public artifact;
10. confirm license, secret, private-source, and generated-file exclusions.

Historical tags and evidence remain immutable. A new result receives a new identity; it does not edit the meaning of an old one.

19.18. Pattern summary

Problem: End-to-end success tempts a project to merge functional compatibility, semantic quality, security, and conformance into one claim.

Pattern: Reassemble the system through explicit component contracts, bind each observation to an exact baseline, publish official failures, and maintain a consolidated blocked-work register.

Result: The Northstar workflow can demonstrate real composition without overstating production readiness or formal protocol status.

Cost: Multiple evidence classes, immutable revisions, scoped demonstrations, and deliberately retained negative results require continuing maintenance.

19.19. Exercises

1. Promote telemetry commit ce47fe2 into a tagged release. List every baseline and ledger entry that must change.
2. Design secure WebSocket continuation without trusting a caller-supplied response ID as ownership.
3. Specify a compact representation that preserves unresolved effects and evidence qualifiers. Define rejection conditions.
4. Add one provider to the Northstar coordinator evaluation without claiming semantic equivalence from one successful run.
5. Convert one blocked-work row into a production gate with owner, expiry, tests, and roll-back.

19.20. Chapter contract

- **Scope:** integrated Northstar architecture, exact baselines, recipient/tool/evidence boundaries, telemetry, official-suite reporting, and final supported-profile classification.
- **Canonical claims:** the reference implements a useful compatible subset and end-to-end research workflow; official failures and missing compaction/WebSocket behavior keep it not conformant.
- **Migration inputs:** Northstar/presentation evidence, document inspection, implementation-choice evidence, Open Responses records, and Chapters 1–18 component contracts.
- **Evidence gates:** immutable/datetime identities, deterministic checks, reviewed live media, telemetry artifact, official suite, and consolidated limitations.
- **Demonstration slots:** Northstar and model-independent inspection are admitted only under their published metadata and limitations; no provider-switch or MCP/A2A demonstration is claimed.
- **Source-claim ledger:** functional, live, telemetry, protocol, security, and semantic evidence remain separate and versioned.
- **Exclusions:** component mechanisms remain in their owning chapters; missing conformance and production controls remain explicit blocked work rather than implied features.

“Agent2Agent Protocol Specification 1.0.0.” n.d. Accessed October 7, 2026. <https://a2a-protocol.org/v1.0.0/specification/>.

“AI SDK Core: streamText.” n.d.-a. Accessed October 7, 2026. <https://ai-sdk.dev/v7/docs/reference/ai-sdk-core/stream-text>.

“AI SDK Core: Tools and Tool Calling.” n.d.-b. Accessed October 7, 2026. <https://ai-sdk.dev/docs/ai-sdk-core/tools-and-tool-calling>.

Amazon Web Services. n.d. “Timeouts, Retries, and Backoff with Jitter.” Accessed October 8, 2026. <https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/>.

Anthropic. n.d.-a. “API Errors.” Accessed October 9, 2026. <https://platform.claude.com/docs/en/api/errors>.

Anthropic. n.d.-b. “Messages API Reference.” Accessed October 9, 2026. <https://platform.claude.com/docs/en/api/messages>.

- Anthropic. n.d.-c. “Streaming Messages.” Accessed October 9, 2026. <https://platform.claude.com/docs/en/build-with-claude/streaming>.
- Anthropic. n.d.-d. “Structured Outputs.” Accessed October 9, 2026. <https://platform.claude.com/docs/en/build-with-claude/structured-outputs>.
- Bun. n.d. “SQLite.” Accessed October 8, 2026. <https://bun.com/docs/runtime/sqlite>.
- Fielding, Roy T., Mark Nottingham, and Julian Reschke. 2022. “HTTP Semantics, RFC 9110.” June. <https://www.rfc-editor.org/rfc/rfc9110.html>.
- “JSON Schema Draft 2020-12.” n.d. Accessed October 7, 2026. <https://json-schema.org/draft/2020-12>.
- “Model Context Protocol 2025-11-25: Lifecycle.” n.d. Accessed October 8, 2026. <https://modelcontextprotocol.io/specification/2025-11-25/basic/lifecycle>.
- “Model Context Protocol Specification 2025-11-25.” n.d. Accessed October 7, 2026. <https://modelcontextprotocol.io/specification/2025-11-25>.
- National Institute of Standards and Technology. 2024. “Artificial Intelligence Risk Management Framework: Generative Artificial Intelligence Profile, NIST AI 600-1.” July. <https://nvlpubs.nist.gov/nistpubs/ai/nist.ai.600-1.pdf>.
- “Open Responses Reference 2026-04-24.” n.d. Accessed October 7, 2026. <https://www.openresponses.org/reference/2026-04-24>.
- “Open Responses Specification 2026-04-24.” n.d. Accessed October 7, 2026. <https://www.openresponses.org/specification/2026-04-24>.
- OpenAI. n.d.-a. “Conversation State.” Accessed October 9, 2026. <https://developers.openai.com/api/docs/guides/conversation-state>.
- OpenAI. n.d.-b. “Reasoning Models.” Accessed October 9, 2026. <https://developers.openai.com/api/docs/guides/reasoning>.
- OpenAI. n.d.-c. “Responses API: Create a Model Response.” Accessed October 9, 2026. <https://developers.openai.com/api/reference/resources/responses/methods/create/>.
- OpenAI. n.d.-d. “Structured Model Outputs.” Accessed October 9, 2026. <https://developers.openai.com/api/docs/guides/structured-outputs>.
- “OpenTelemetry Documentation.” n.d. Accessed October 7, 2026. <https://opentelemetry.io/docs/>.
- “OpenTelemetry Generative AI Semantic Conventions, Revision 06ec68e.” n.d. Accessed October 7, 2026. <https://github.com/open-telemetry/semantic-conventions-genai/tree/06ec68e722c45a7218e23ea1bc1339fe4e21ecae/docs/gen-ai>.
- OWASP. n.d. “LLM Prompt Injection Prevention Cheat Sheet.” Accessed October 8, 2026. https://cheatsheetseries.owasp.org/cheatsheets/LLM_Prompt_Injection_Prevention_Cheat_Sheet.html.

[Sheet.html](#).

“OWASP Top 10 for Agentic Applications 2026.” n.d. Accessed October 7, 2026. <https://genai.owasp.org/resource/owasp-top-10-for-agentic-applications-for-2026/>.

SQLite. n.d.-a. “Isolation in SQLite.” Accessed October 8, 2026. <https://www.sqlite.org/isolation.html>.

SQLite. n.d.-b. “Transaction.” Accessed October 8, 2026. https://www.sqlite.org/lang_transaction.html.

“Streams Standard.” n.d. Accessed October 7, 2026. <https://streams.spec.whatwg.org/>.

Vercel. n.d.-a. “AI SDK 7: Anthropic Provider.” Accessed October 9, 2026. <https://ai-sdk.dev/v7/docs/providers/ai-sdk-providers/anthropic>.

Vercel. n.d.-b. “AI SDK: OpenAI Provider.” Accessed October 9, 2026. <https://ai-sdk.dev/providers/ai-sdk-providers/openai>.

WHATWG. n.d. “DOM Standard: Aborting Ongoing Activities.” Accessed October 9, 2026. <https://dom.spec.whatwg.org/commit-snapshots/b76da7af5fe35a2a368127cc9ec22449ac8fd559/#aborting-ongoing-activities>.

Index

- adapter pattern
 - provider boundary, 37
- agent.yaml, 22
- attachments, 16
- authorization, 22, 70
- backpressure, 61
- cancellation, 61
 - cooperative behavior, 131
- capabilities
 - evidence and expiry, 119
- capability selection, 22
- configuration, 22
- deployment identity, 37
- document ingestion, 29
- document inspection, 16
- event stream, 50
- evidence, 9
- fallback
 - policy preservation, 119
- idempotency, 131
- image admission, 29
- invocation-local identity, 29
- JSON Schema, 70
- normalization
 - information loss, 37
- Northstar workflow, 9
- Pi client, 16
- portability, 9
- provider neutrality, 9
- recovery
 - commitment boundaries, 131
- response item, 50
- retry
 - deadline budgets, 131
- semantic portability
 - evaluation, 119
- state machine, 50
- streaming, 61
- thin client, 16
- tool calling, 70
- Xberg, 29