

Six Reliability Primitives for LLM Agents: An Artifact Pattern for Stackable, Single-Concern Libraries

Mukunda Rao Katta
Independent Researcher

Abstract

Reliability concerns for large-language-model (LLM) agents are typically addressed inside frameworks that bundle prompting, tool routing, and runtime governance into a single dependency. This paper presents the agent-stack: a set of six small, single-concern reliability libraries published independently to npm, PyPI, and the Model Context Protocol (MCP) registry. The libraries are AgentFit (context-window fitting), AgentGuard (network egress allowlist), AgentSnap (snapshot tests for tool-call traces), AgentVet (validate tool arguments before execution), AgentCast (structured-output enforcer), and AgentBudget (token and dollar caps). Each library is zero-dependency, has a TypeScript implementation with hand-maintained type declarations, a Python port with the same surface, and an MCP-server variant. Repository inspection across the eighteen public packages identifies a recurring artifact pattern: a single class or function as the public surface, a typed error carrying retry-friendly context, and an opt-in automatic adapter for popular provider response shapes. Rather than proposing a benchmark or a unified runtime, the contribution is a documented design pattern for reliability primitives that compose by inclusion rather than by framework lock-in. The paper describes the primitives, the cross-cutting invariants the design enforces, the trade-offs of single-concern packaging, and the operational questions that emerge when reliability is split across many small dependencies.

1. Introduction

LLM agent runtimes are typically distributed as monolithic frameworks. A single dependency provides prompting, tool dispatch, retries, schema validation, observability, budget enforcement, and safety filters. This packaging is convenient when an agent matches the framework's assumptions. It becomes a liability when the agent does not: framework upgrades pull in unrelated changes, opinionated defaults are difficult to override, and the user cannot adopt one reliability concern without adopting the others.

The agent-stack is a deliberate counterpoint. It splits reliability into six independent libraries. Each library solves one problem, has zero runtime dependencies, and can be adopted in isolation. Composition happens by including the libraries the operator needs, not by inheriting framework defaults.

This paper documents the agent-stack as a systems artifact. It does not claim that single-concern libraries outperform monolithic frameworks on benchmark tasks, and it does not propose a new evaluation. It describes the design choices behind a stackable reliability surface, the invariants the public APIs preserve across six unrelated concerns, and the questions the artifact pattern raises for users who want reliability primitives that they can audit, swap, and version independently.

2. Artifact Overview

The agent-stack is published as eighteen public packages across three ecosystems:

- six TypeScript libraries on npm under the @mukundakatta/ scope: agentfit, agentguard, agentsnap, agentvet, agentcast, agentbudget.
- six Python ports on PyPI under the *-py or descriptive name suffixes: agentfit-py, agentguard-firewall, agentsnap-py, agentvet-py, agentcast-py, agentbudget-py.

- six MCP server variants on npm under the `*-mcp` suffix: `agentfit-mcp`, `agentguard-mcp`, `agentsnap-mcp`, `agentvet-mcp`, `agentcast-mcp`, `agentbudget-mcp`.

Each TypeScript package ships ESM source with hand-maintained `.d.ts` declarations rather than emitted-from-build types. Tests use the Node `node:test` runner. None of the six TypeScript packages carries a runtime dependency. The Python ports use `hatchling` with an `src/` layout and `pytest`; their only base dependency is the Python standard library. The MCP variants depend on `@modelcontextprotocol/sdk` and the corresponding stack library.

A landing site at mukundakatta.github.io/agent-stack lists the libraries, their npm and PyPI links, and the conceptual pipeline they are commonly composed into: fit, guard, snap, vet, cast, budget.

3. Design Goals

The agent-stack is organized around five design goals.

First, every library should solve exactly one reliability concern. The boundary is drawn so that a user who needs only one of them does not pay the cognitive cost of the others.

Second, every library should be safe to add late. The libraries do not require the user to restructure an agent or migrate to a new abstraction. Each library wraps a well-defined surface: a tool function, an LLM call, a message list, or a network egress site.

Third, every library should be zero-dependency at runtime. Reliability primitives are infrastructure for the operator's trust in the agent. Pulling third-party transitive dependencies into a primitive that is supposed to enforce safety invariants would expand the trust surface in a way that contradicts the goal.

Fourth, every library should fail with a typed, descriptive error rather than a string. Reliability primitives often run on the boundary between an agent and a tool or LLM. Their errors are observed by humans during incident investigation and by other agents during retry loops. A typed error carries the cap, the limit, the offending key, the rejected value, and a structured retry hint.

Fifth, every library should expose an automatic adapter for the dominant provider shapes. `AgentBudget` recognizes Anthropic and OpenAI usage shapes by default. `AgentSnap` recognizes the standard tool-call envelope. `AgentCast` recognizes the validation result of `safeParse`-style validators. The default should match the most likely caller; everything else is configurable.

4. The Six Libraries

This section describes each primitive in sequence: the concern it isolates, its public surface, and its principal failure mode.

4.1 AgentFit

`AgentFit` fits a list of messages into a model's context window. It accepts a token-aware budget, a list of messages, and a strategy. The bundled strategies are drop-oldest, drop-middle, and priority-keyed. The output is a fitted message list and a structured trim report describing which messages were dropped and why. The pluggable tokenizer interface allows the library to be used with provider-specific token counters or with character-count approximations when no tokenizer is available. `AgentFit`'s principal failure mode is silent context truncation; it surfaces dropped messages explicitly rather than letting them disappear from the input.

4.2 AgentGuard

AgentGuard installs a declarative network-egress allowlist around agent tools. The user provides a list of allowed domains. Tool functions are wrapped so that any outbound HTTP request to a domain not on the list throws with the offending URL and the active allowlist. AgentGuard's principal failure mode is a tool that fetches a URL the user did not anticipate; the failure is caught at the egress site and the agent's retry loop sees a clear error rather than a silent data exfiltration.

4.3 AgentSnap

AgentSnap captures snapshot tests for tool-call traces. The user wraps tool functions with `traceTool`. During a recorded run the snap library captures the sequence of tool names, argument hashes, and result hashes into a stable JSON envelope. A baseline file is committed alongside the test. Subsequent test runs compare the new trace to the baseline and report drift in a human-readable form. AgentSnap's principal failure mode is undetected behavior change in the tool sequence after a model or prompt revision; the snap diff turns that drift into a CI signal.

4.4 AgentVet

AgentVet validates tool-call arguments before the tool function runs. The user wraps each tool with a schema, a Zod validator, or a small built-in shape spec. Calls that fail validation throw a `ToolArgError` that carries an LLM-friendly retry hint formatted for direct insertion into the next conversation turn. AgentVet's principal failure mode is a model that hallucinates argument types and reaches the tool body; the wrapper catches the hallucination and gives the model a structured chance to correct itself.

4.5 AgentCast

AgentCast enforces structured output on LLM completions. The user supplies a schema, a target type, and an LLM call. The library wraps the call with a validate-and-retry loop: the response is parsed, validation errors are translated into a feedback message, and the LLM is asked to correct itself up to a configurable retry limit. The output is either typed data or an error after the retry budget is exhausted. AgentCast's principal failure mode is a model that returns prose where the application expects a structured object; the loop pushes the model toward the requested shape without inventing data.

4.6 AgentBudget

AgentBudget tracks token and dollar usage across an agent run and refuses calls that would push past a configured cap. The user constructs a `Budget` with caps for input tokens, output tokens, total tokens, dollars, or any combination. After each LLM call the user records usage; the library raises `BudgetExceededError` carrying the offending cap, the limit, the attempted total, and the model name. A built-in pricing table covers the dominant Anthropic and OpenAI models; user-supplied pricing maps override the defaults per key. AgentBudget's principal failure mode is a planner loop that accidentally calls a model thousands of times; the cap turns runaway spend into an immediate error rather than a billing surprise.

5. Cross-Cutting Invariants

Across the six libraries, the public surfaces share three invariants.

The first invariant is that every primitive is a single noun and a single verb. AgentFit is a *fit* over messages. AgentGuard is a *guard* over fetch sites. AgentSnap is a *snap* over tool traces. AgentVet is a *vet* over tool arguments. AgentCast is a *cast* over LLM outputs. AgentBudget is a *record* and a *cap* over LLM usage. The naming convention is part of the artifact. It tells users which library to reach for without consulting a manual.

The second invariant is that every primitive throws. There is no silent fallback path. Reliability primitives that fall back silently undermine their reason for existence: the user adopted them to make a specific failure observable.

The libraries refuse to pretend a violated invariant is fine.

The third invariant is that every error type is a real class with named fields, not a string.

`BudgetExceededError` carries `cap`, `limit`, `attempted`, `overshoot`, and `model`. `ToolArgError` carries the tool name, the validation error, the rejected arguments, and a `toLLMFeedback()` method. Errors are designed to be programmatically dispatched and human-readable in incident logs.

6. Cross-Language Surface Symmetry

Each library exists in three implementations: TypeScript, Python, and an MCP server. The TypeScript implementation is the reference. The Python port mirrors the TypeScript surface with snake-case names. The MCP server exposes the same primitives over the Model Context Protocol so that any MCP-aware client can adopt the reliability surface declaratively.

Symmetry across implementations is intentional. A user who learns the TypeScript surface can drop into the Python port without re-learning the API. The MCP variant lets an operator configure budgets, allowlists, and validation declaratively from a client configuration file rather than from agent code. The cost of the symmetry is that each fix must be applied three times. The benefit is that the design pattern, not the code, is the thing being distributed.

7. Composition

The six primitives are designed to compose by inclusion. The conceptual pipeline runs left to right: fit messages into context (`AgentFit`), guard tool egress (`AgentGuard`), snap tool traces against a baseline (`AgentSnap`), vet tool arguments before execution (`AgentVet`), cast LLM responses to typed data (`AgentCast`), and cap token and dollar usage (`AgentBudget`). The pipeline is illustrative, not enforced. Each library is independent. A user may adopt only `AgentBudget` if dollar caps are the only concern, or only `AgentVet` if the agent's tool surface is unstable.

Because the libraries are zero-dependency, composing them does not introduce transitive constraints. A user can pin `AgentBudget` at one major version and `AgentSnap` at a different major version without conflict resolution. This is a deliberate trade against the convenience of a single-import framework: the user pays a small cognitive cost in exchange for a trust surface they can audit one library at a time.

8. Operational Considerations

Splitting reliability into six libraries surfaces three operational considerations.

First, the user must decide which primitive owns which failure mode. Token caps, dollar caps, and rate limits overlap. `AgentBudget` owns the first two. Rate limiting is intentionally out of scope for the agent-stack and would belong in a separate primitive. Operators who try to enforce a rate limit by tightening a token cap will misattribute the underlying failure.

Second, errors propagate across primitives. A tool wrapped with both `AgentGuard` and `AgentVet` may throw two different error types depending on which check runs first. Users who handle errors must inspect the type, not the message, to dispatch correctly.

Third, observability is a per-library responsibility. The libraries do not ship a unified logging surface. Operators who want a single trace must thread their own logger through the libraries or wrap them at the call site. The deliberate omission of a shared logger is part of the zero-dependency invariant.

9. Evaluation Path

This paper does not present a benchmark score. Instead, it proposes an evaluation path for reliability primitives like the ones in the agent-stack.

The first layer is artifact-level evaluation: package builds reproduce, type declarations match the source, tests pass on a fresh checkout, and zero-runtime-dependency invariants hold under `npm ls --omit=dev` and `pip show audits`.

The second layer is integration-level evaluation: each primitive composes with realistic provider SDK shapes (Anthropic, OpenAI, Groq) without surprise, and an operator who composes three primitives at once does not encounter conflicting error types or duplicated wrappers.

The third layer is incident-level evaluation: when a primitive fires in production, the structured error carries enough context for the operator to fix the underlying cause without reading the library's source.

The fourth layer is longitudinal evaluation: across model and provider revisions, the primitives remain useful because the surfaces they wrap (token usage, tool call traces, structured output, network egress) are stable concerns even as model behavior drifts.

10. Discussion

The agent-stack is a small artifact, but it argues for a particular position in the design space of LLM tooling.

Monolithic frameworks make adoption easy at the cost of opinionated defaults. Stackable primitives make adoption granular at the cost of more decisions for the operator. The agent-stack does not claim that the granular path is correct in all settings. It claims that the granular path is currently underrepresented in published artifacts. Six libraries, each with a single class or function as the public surface, each with zero runtime dependencies, are a usable example of the pattern even if the pattern is not the right answer for every team.

The cross-language and MCP variants demonstrate a related point: the contribution of a reliability primitive is the design, not the implementation. A `Budget` class in TypeScript and a `Budget` class in Python are the same artifact under two languages. An MCP server exposing the budget over a JSON-RPC tool surface is a third presentation of the same design. Distributing a reliability primitive in three forms makes the design portable and tells the user that the primitive is a contract, not a library.

11. Limitations

Several limitations are explicit.

The agent-stack does not include a primitive for rate limiting. Rate limiting interacts with provider headers and retry policies that are outside the scope of the six current libraries. A future addition to the stack could close that gap.

The agent-stack does not provide a unified observability surface. Operators who want a single trace must compose one. The omission keeps each library zero-dependency but pushes the integration cost onto the user.

The agent-stack does not include a primitive for memory or session safety. Memory is a concern that overlaps with personal-assistant architecture (the karna project, in a separate paper) rather than reliability primitives. Splitting the two intentionally.

The agent-stack does not yet have published benchmark results. Those would be valuable additions and are tracked as future work.

12. Related Work

The pattern of zero-dependency single-concern reliability libraries echoes earlier work in the JavaScript ecosystem (small modules, deliberately unscoped) and in the Python ecosystem (the `tenacity`, `pydantic`, and `httpx` design pattern of typed errors and explicit configuration). The MCP variants build on the Model Context Protocol's design of capability-scoped tools.

Related industrial frameworks include LangChain, LlamaIndex, AutoGen, and the OpenAI Agents SDK, all of which bundle reliability concerns into a larger runtime. The agent-stack does not compete with these frameworks; it offers an alternative for users who want reliability primitives separately from agent runtime decisions.

13. Future Work

Three near-term directions are tracked.

First, AgentBudget will gain a per-cache-tier pricing variant covering Anthropic prompt caching and OpenAI batch tier rates.

Second, AgentSnap will gain a structured-diff format for trace comparison so that diffs can be machine-consumed by review tools as well as human reviewers.

Third, the MCP variants will be reviewed against the MCP registry's guidelines for tool-server hygiene so that the primitives can be listed in client-side directories without manual configuration.

14. Availability

The agent-stack is available under the MIT license. The TypeScript packages are published to npm under the `@mukundakatta/` scope. The Python ports are published to PyPI. The MCP variants are published to npm under the `-mcp` suffix. Source repositories are listed at github.com/MukundaKatta under the `Agent*` repository names. The landing page is at mukundakatta.github.io/agent-stack.

15. Conclusion

The agent-stack is one realization of a design pattern: split reliability into independent, single-concern, zero-dependency primitives, distribute them across the languages and protocols where agents are built, and let composition emerge by inclusion rather than by framework adoption. The artifact described in this paper is small. The design pattern it argues for is general. Six libraries are not a proof; they are an example. Users who want reliability primitives that they can audit, swap, and version independently can adopt the pattern by using the agent-stack or by replacing it with their own implementations of the same six concerns.

References

agent-stack landing page. <https://mukundakatta.github.io/agent-stack/>

agent-stack reliability primitives paper repository.

<https://github.com/MukundaKatta/agent-stack-reliability-primitives-paper>

Author npm scope. <https://www.npmjs.com/~mukundakatta>

Author GitHub profile. <https://github.com/MukundaKatta>

Karna chat-native assistant paper (companion). <https://github.com/MukundaKatta/karna>

Model Context Protocol. <https://modelcontextprotocol.io/>

LangChain. <https://www.langchain.com/>

LlamaIndex. <https://www.llamaindex.ai/>

AutoGen. <https://microsoft.github.io/autogen/>

OpenAI Agents SDK. <https://openai.github.io/openai-agents-python/>

tenacity. <https://github.com/jd/tenacity>

pydantic. <https://docs.pydantic.dev/>

httpx. <https://www.python-httpx.org/>

hatchling. <https://hatch.pypa.io/latest/>

Node.js test runner. <https://nodejs.org/api/test.html>