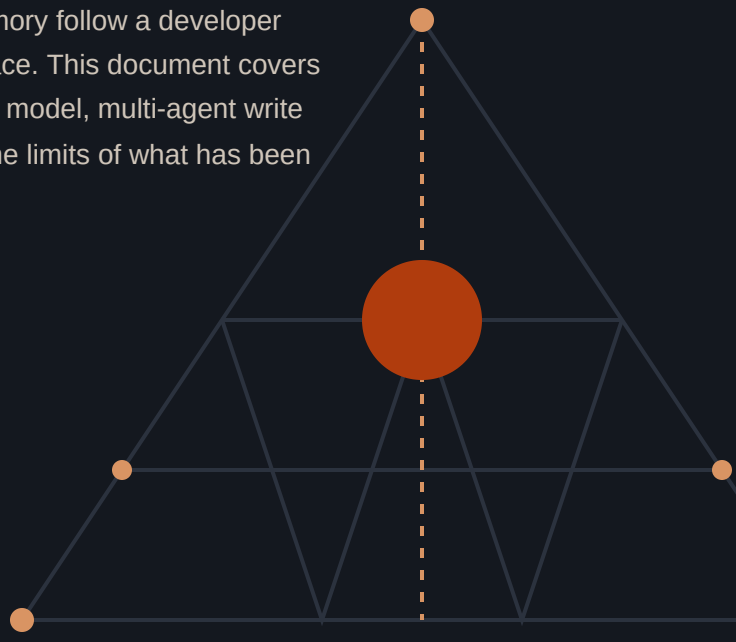
TECHNICAL CASE STUDY · CODEFRONT LABS

Truss

One provider-neutral coding-agent runtime behind six clients: CLI, terminal UI, Neovim, VS Code, desktop and Android.

How Truss separates the agent loop from every interface and every model provider, so the same tools, permissions, plans and workspace memory follow a developer from the shell to the editor to a dedicated desktop workspace. This document covers the architecture, the provider and tool contracts, the safety model, multi-agent write coordination, release engineering across six clients, and the limits of what has been shipped so far.



Releases covered: CLI 0.1.37 · VS Code 0.1.39 · Desktop 0.1.64 · MIT License

Prepared 14 November 2026

Contents

01 Executive summary

02 The problem

03 Design principles and constraints

04 System architecture

05 The provider contract

06 The agent loop, tools and approvals

07 Managed agents and the workspace write lease

08 Clients: one runtime, six surfaces

09 Runtime delivery and release engineering

10 Development process and quality gates

11 By the numbers

12 Limitations and what comes next

13 Lessons learned

Sources: the Truss repository (`truss-agent/truss-harness`, about 500 commits on `master` between 18 July and 14 November 2026), its `AGENTS.md` specification and design plans under `plans/`, and the public website at `truss-agent.com`. Code metrics were measured from the working tree on the preparation date. Capabilities described only in plans or the roadmap are labelled as such; no usage or adoption figures are claimed.

1. Executive summary

Truss is a local-first, open-source (MIT) coding-agent platform. Most coding agents start as a feature of one editor or one hosted service, and their tools, prompts, permissions and memory are tied to that surface. Truss inverts that: it starts with a standalone TypeScript runtime that knows nothing about VS Code, Electron or any model vendor, and treats every user interface as a client of that runtime.

6
clients on one runtime: CLI, TUI, Neovim, VS Code, Desktop, Truss Go

~67k
lines of strict TypeScript across 13 packages

2
members in the ModelProvider interface: id and stream()

150+
merged pull requests, each linked to an issue

A developer brings a local model server (Ollama, LM Studio, llama.cpp or any OpenAI-compatible endpoint) or their own key for one of fifteen catalogued cloud providers. No hosted control plane is involved. Saved provider profiles, tool permissions, conversations, implementation plans and workspace memory live with the workspace and are shared by every client.

Chat, Plan and Edit modes make intent explicit. Several workspace-local agent profiles, each with its own provider, model and approval policy, can run at once; Chat and Plan agents run concurrently while Edit agents are serialised through a workspace write lease so two agents never mutate the same tree.

> **Outcome.** Over roughly four months, a solo-led project went from an empty repository to published npm packages for the CLI and TUI, a signed VS Code extension, Windows and Linux desktop builds, a Neovim plugin and an Android companion, all driven by the same runtime package.

2. The problem

By mid-2026, developers had a rich choice of coding agents, but each came bundled with decisions that were hard to undo:

- **Interface lock-in.** An agent built into one editor does nothing in a terminal session, a CI script or a different editor. Switching surfaces means switching agents and re-learning their behaviour.
- **Provider lock-in.** Many tools assume one vendor's API shape. Teams that want local models for privacy or cost, or want to compare providers, end up maintaining parallel setups.
- **Fragmented state.** Each client keeps its own conversation history, approval settings and memory, so context does not carry from the shell to the editor.
- **Opaque control.** When an agent can write files and run commands, the user needs to see and approve sensitive actions in whatever surface they happen to be using.

> **Goal.** An agent "operating system" in which models, tools, memory, planning and interfaces are replaceable parts behind stable interfaces, so a capability such as a new provider, tool or client can be added without modifying the core loop.

The project's own specification states it plainly: *"This is not a VSCode extension with AI features. This is an agent operating system/runtime that can power many different agent experiences. The VSCode extension is only the first client."*

3. Design principles and constraints

The repository's `AGENTS.md` was written before most of the code and acts as an enforceable design contract. Its rules are short and absolute:

Always

- TypeScript only, strict typing
- Dependency injection; no singleton services
- Interfaces between every subsystem
- Event-driven and plugin-based
- Every component independently testable
- ESLint, Prettier, Vitest and CI from day one

Never

- Assume OpenAI or hardcode a provider
- Hardcode prompts
- Couple business logic to VS Code
- Let tools call each other directly
- Put planning logic in UI components
- Let the runtime know which client hosts it

A measurable target

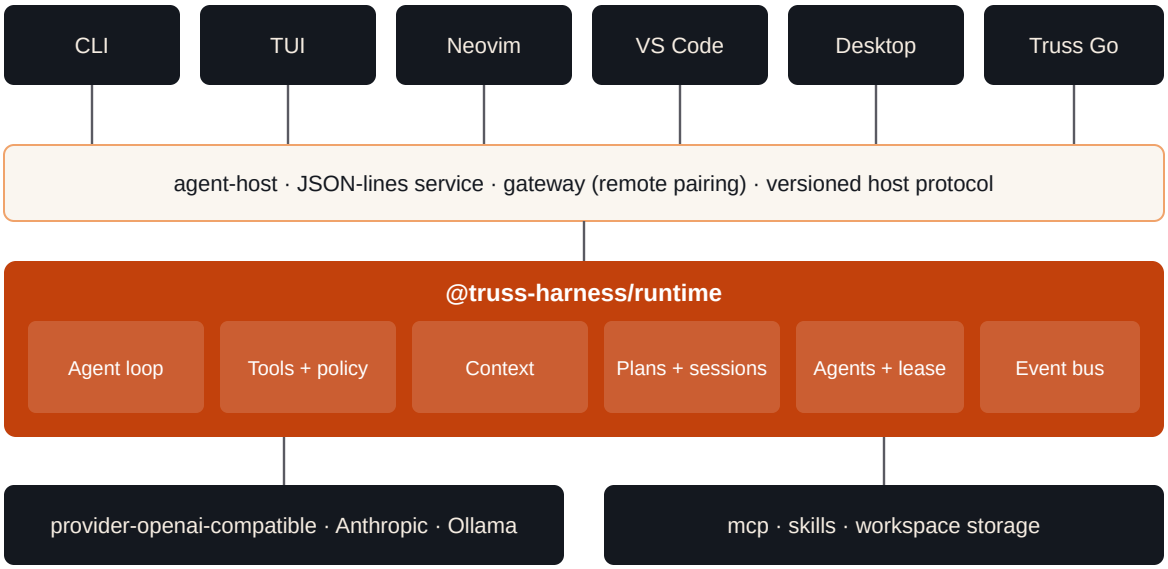
One rule doubles as an acceptance test for the provider abstraction: **adding a provider should take under 200 lines of code.** That forces all wire-format, retry and discovery logic into shared code, leaving a new provider as little more than an endpoint, an authentication scheme and a mapping of stream events.

Responsibility split

Runtime owns	Clients own
Planning, context building, memory, tool execution, streaming, retries, interruption, checkpoint recovery	Rendering, authentication UI, settings screens, diff previews, approval dialogs, status bars, notifications

4. System architecture

Truss is an npm-workspaces monorepo of thirteen packages. The dependency direction is strictly inward: clients depend on the runtime; the runtime depends only on its own contracts.



Package layering. Clients never import provider code directly; the runtime receives providers and tools by injection.

Package	Role
`runtime`	Contracts, agent loop, tool executor, edit policy, plans, sessions, memory, managed agents, retry and resource limits (48 source files)
`provider-openai-compatible`	OpenAI-compatible, Anthropic and Ollama adapters, wire parsing, model discovery and the cloud provider catalog
`agent-host`	Hosts the runtime behind a stable, versioned protocol so clients do not bundle a specific runtime build
`cli` / `tui`	Shell commands, setup and a JSON-lines service used by editor clients; full-screen terminal workspace
`vscode` / `desktop` / `neovim`	Editor extension, Electron workspace, Lua plugin that talks to the CLI service
`mobile` / `gateway`	Truss Go Android app and the pairing gateway that lets it continue a trusted host session
`mcp`, `branding`, `docs`	Model Context Protocol integration, shared brand assets, the Next.js documentation website

5. The provider contract

The entire model abstraction is one interface:

```
// packages/runtime/src/contracts.ts
export interface ModelProvider {
  readonly id: string;
  stream(request: ModelRequest): AsyncIterable<ModelStreamEvent>;
}
```

Every provider, local or cloud, turns a `ModelRequest` (messages, tool definitions, attachments, an abort signal) into an async stream of normalised events: text deltas, tool-call fragments, usage reports and completion. Because the runtime only ever consumes this stream, streaming, interruption and token accounting work identically regardless of vendor.

Why so small

- **Streaming is the only mode.** Non-streaming responses are a degenerate stream, so there is one code path to test.
- **Cancellation is structural.** An `AbortSignal` rides on the request; "interrupt generation" is simply aborting the iterator.
- **Wire formats stay out of the core.** SSE parsing, tool-call reassembly and error normalisation live in the provider package, with dedicated tests for the wire parser and stream completion.

Coverage

Local: Ollama, LM Studio, llama.cpp server and any OpenAI-compatible endpoint, with automatic detection of running local servers. Cloud (bring your own key): Anthropic, OpenAI, Gemini, OpenRouter, DeepSeek, Mistral, Groq, xAI, Together, Fireworks, Perplexity, NVIDIA NIM, Ollama Cloud and others, fifteen catalog entries in all. Most share the OpenAI-compatible adapter with a catalog entry; Anthropic and Ollama have native adapters. Credentials stay on the host and never travel to remote clients.

> **Design consequence.** Model routing, benchmarking and fallback can be built later as a `ModelProvider` that wraps other providers, with no change to the agent loop.

6. The agent loop, tools and approvals

The runtime implements the iterative cycle the specification calls for: goal → plan → execute tool → observe → update plan → continue. Each run emits typed events (`run_started`, `text_delta`, `progress_delta`, `tool_call_requested`, `usage` and others) on an event bus, and clients render whatever they subscribe to.

Tools

A tool is a schema, a description and an `execute()` function, registered rather than imported. The built-in set:

Tool	Purpose
`read_file`, `list_directory`, `search_files`	Bounded, workspace-scoped reading and search
`write_file`, `replace_in_file`	Edits, subject to edit policy and approval, with diff previews in clients
`run_terminal`	Command execution through a process runner with timeouts, output caps and drain handling
`update_plan`	Durable implementation plans that survive across sessions and clients

Tools never call each other; only the loop orchestrates. MCP servers and skills extend the registry without core changes.

Safety boundaries

- **Workspace boundary.** Path resolution rejects escapes from the workspace root; a dedicated test suite covers the boundary.
- **Edit policy by mode.** Chat cannot write. Plan can only update plans. Edit can write, subject to its approval policy.
- **Approval context.** Sensitive actions are paused and raised to whichever client is attached, with enough context (diff, command, path) to decide.
- **Resource limits.** Output, file size and runtime ceilings keep a misbehaving model from exhausting the host.
- **Recovery.** Malformed or failed tool calls are normalised into errors the model can see and correct, with retry policies for transient provider failures.

7. Managed agents and the workspace write lease

Truss supports several workspace-local agent profiles, each with its own provider, endpoint, model, mode and approval policy. A common pattern is a cheap local model for exploratory Chat, a stronger cloud model for Plan, and a single Edit agent executing the plan.

Concurrent reading is safe; concurrent writing is not. Rather than locking files or merging edits, Truss uses a deliberately simple primitive:

```
// packages/runtime/src/agents/write-lease.ts
/** One holder at a time; a host may replace this with a durable or distributed lease. */
export interface WorkspaceWriteLease {
  holder(): AgentRunId | undefined;
  tryAcquire(runId: AgentRunId): boolean;
  release(runId: AgentRunId): boolean;
}
```

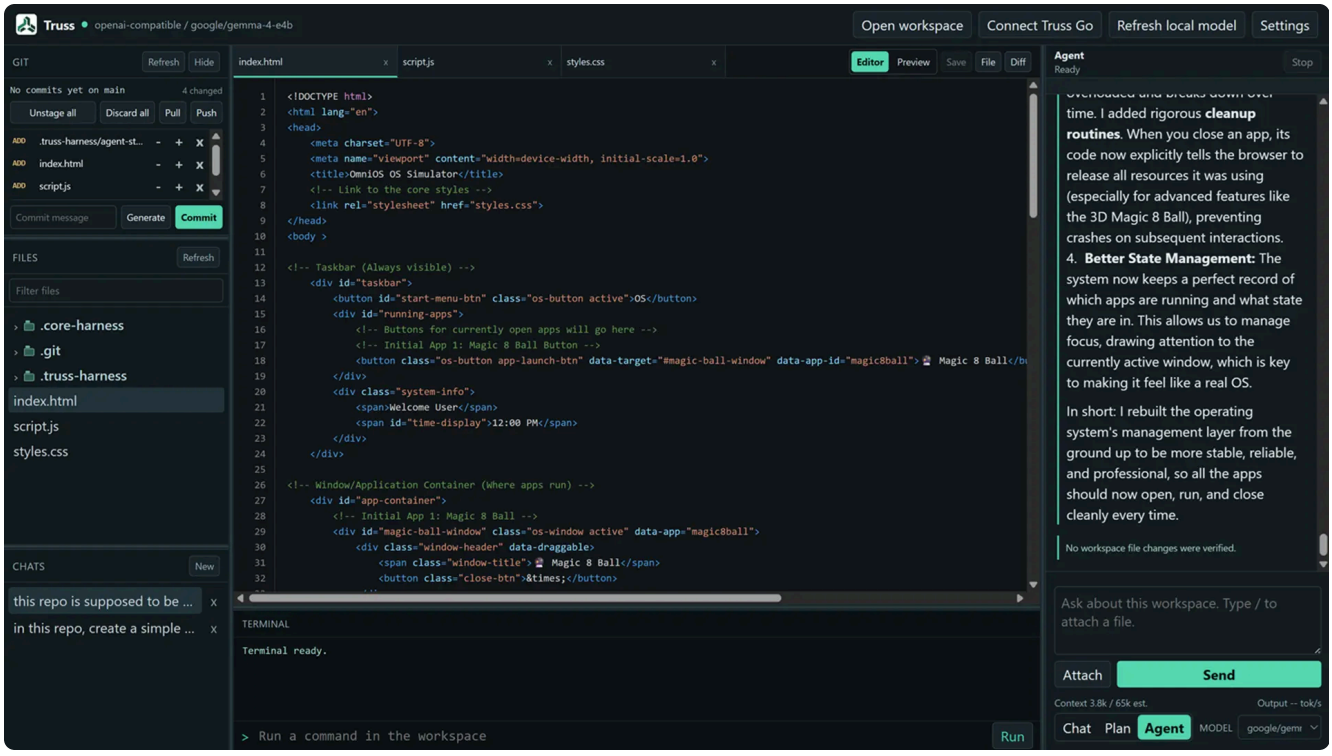
Edit runs must acquire the lease before starting; Chat and Plan runs never need it. The coordinator handles cancellation draining so a cancelled Edit run releases the lease only after in-flight tool calls finish, and run history records what each agent did.

> **Why an interface.** The default lease is in-memory, which is correct for one host. Because it sits behind an interface, a future remote-execution or distributed setup can supply a durable lease without touching the coordinator.

8. Clients: one runtime, six surfaces

Client	Form	Distribution
CLI	Direct tasks, scripts, automation, JSON-lines service for editors	npm `@truss-harness/cli`
Terminal UI	Keyboard-first workspace: files, editor, chat, Git diffs, shell, approvals	npm `@truss-harness/tui`
Neovim	Native Chat, Plan and Edit with bounded editor context and `:checkhealth`	Generated `neovim-release` branch
VS Code	Activity Bar chat, file context, inline completions, approvals	Signed VSIX
Desktop	Electron workspace with files, diffs, terminal, Git and chat	Windows (x64, arm64) and Linux builds
Truss Go	Continue a trusted Desktop or VS Code session from Android	Android release workflow

The Neovim plugin is the clearest proof of the split: model requests, credentials, tools and policy stay in `truss-cli`, and the Lua code owns only the editor experience. Truss Go goes further, pairing through the gateway with a host that keeps all credentials; the phone sends intents and receives events, never keys.



The desktop client: files, Git changes, editor, terminal and agent conversation in one workspace.

9. Runtime delivery and release engineering

Shipping six clients created a problem the architecture did not solve on its own. Clients imported the runtime at an exact version and several bundled it into native artifacts, so publishing a fixed runtime to npm did nothing for an installed desktop app, VSIX or APK. Loosening to semver ranges would have made installs non-reproducible.

The runtime delivery contract

The fix (tracked as issue #234) inserts a versioned host protocol and a compatibility gate between client and runtime. Each client carries an embedded fallback runtime and can load a separately versioned, verified runtime host when one is compatible. After a one-time bootstrap release, runtime fixes ship independently of UI releases, and an update that cannot be verified falls back safely.

Pipelines

Seven GitHub Actions workflows cover pull-request checks, dependency audits and per-client releases: desktop, VS Code, Neovim, runtime host and Android. Client versions move independently (CLI 0.1.37, VS Code 0.1.39, Desktop 0.1.64 at the time of writing) and releases are tagged only from merged `master` commits.

10. Development process and quality gates

Truss was built largely with AI coding agents under a strict, written workflow, and its process is part of the case study:

- **Issue first.** Every change starts as an issue from a template, on a fresh branch, and merges through a templated PR with a closing reference. The history shows more than 140 issues and 150 merged PRs.
- **Architecture as tests.** The runtime includes tests that enforce boundaries rather than behaviour: `architecture.test.ts`, `public-api.test.ts`, `workspace-boundary.test.ts`, `resource-boundaries.test.ts` and `web-boundaries.test.ts`. A dependency that breaks the layering fails CI.
- **Regression audits.** An `audit-regressions` suite locks in fixes from review findings.
- **Plans in the repo.** Larger changes (runtime delivery, maintainability refactor, master-prompt templates, tool-call validation, mobile remote client) get a written plan under `plans/` before code.
- **No hardcoded prompts.** System prompts are templates with tests, so they can be versioned and replaced.

Timeline

Period	Commits	Focus
July 2026	228	Contracts, runtime loop, providers, tools, CLI and VS Code client
August 2026	193	TUI, desktop, approvals, plans, memory, provider catalog
September 2026	6	Maintenance
October 2026	67	Managed agents, Neovim, Truss Go, runtime delivery, website
November 2026	2+	CLI terminal experience (bottom composer, diff display)

11. By the numbers

~500 commits, 18 Jul – 14 Nov 2026	**33** distinct days with commits	**13** workspace packages	**~67k** lines of TypeScript
137 test files (Vitest)	**15** cloud catalog entries + local servers	**7** CI and release workflows	**6** shipped clients

Line and file counts measured from the working tree on 14 November 2026, excluding `node_modules` and build output. Commit and PR counts from Git and GitHub. No download, user or benchmark figures are claimed.

12. Limitations and what comes next

Current limits

- Models are not interchangeable: tool-calling quality varies widely, especially among small local models, and the runtime cannot hide that.
- Context selection is bounded and explicit; vector memory, tree-sitter indexing and LSP-driven symbol context are on the roadmap, not shipped.
- The write lease is single-host; remote and distributed execution are planned.
- Desktop builds target Windows and Linux; there is no macOS build yet.
- Truss moved from a custom source-available license to the MIT License in November 2026; package versions published before that keep the license they shipped with.

Roadmap items enabled by the architecture

Model routing and automatic provider benchmarking (as wrapping providers), Docker sandboxes and remote execution (as alternative process runners), a plugin and tool marketplace (as registry sources), long-term memory, browser automation and session replay (from the event stream).

13. Lessons learned

1. **Write the contract before the code.** A short, absolute `AGENTS.md` kept dozens of agent-written PRs pointed in one direction. The "never" list did more work than the feature list.
2. **Make the core interface embarrassingly small.** A two-member `ModelProvider` made fifteen providers cheap and made streaming and cancellation uniform.
3. **Enforce architecture with tests.** Boundary tests catch coupling that code review misses, especially when much of the code is generated.
4. **Prefer simple coordination primitives.** A single write lease behind an interface solved multi-agent safety without merge logic.
5. **Distribution is architecture too.** A clean runtime split was not enough once runtimes were bundled into installed apps; delivery needed its own versioned protocol.
6. **Client count tests the abstraction.** Each new surface, Neovim and Android especially, exposed any logic that had leaked into a UI.