JEV ENGINEERING FOR CODING AGENTS

The TypeSafe Founder's Blueprint for Building with Jev

A Synthesis for Study · Based on design notes by Diogo Almeida (TypeSafe)
Independently compiled, September 2026. Not affiliated with or endorsed by TypeSafe

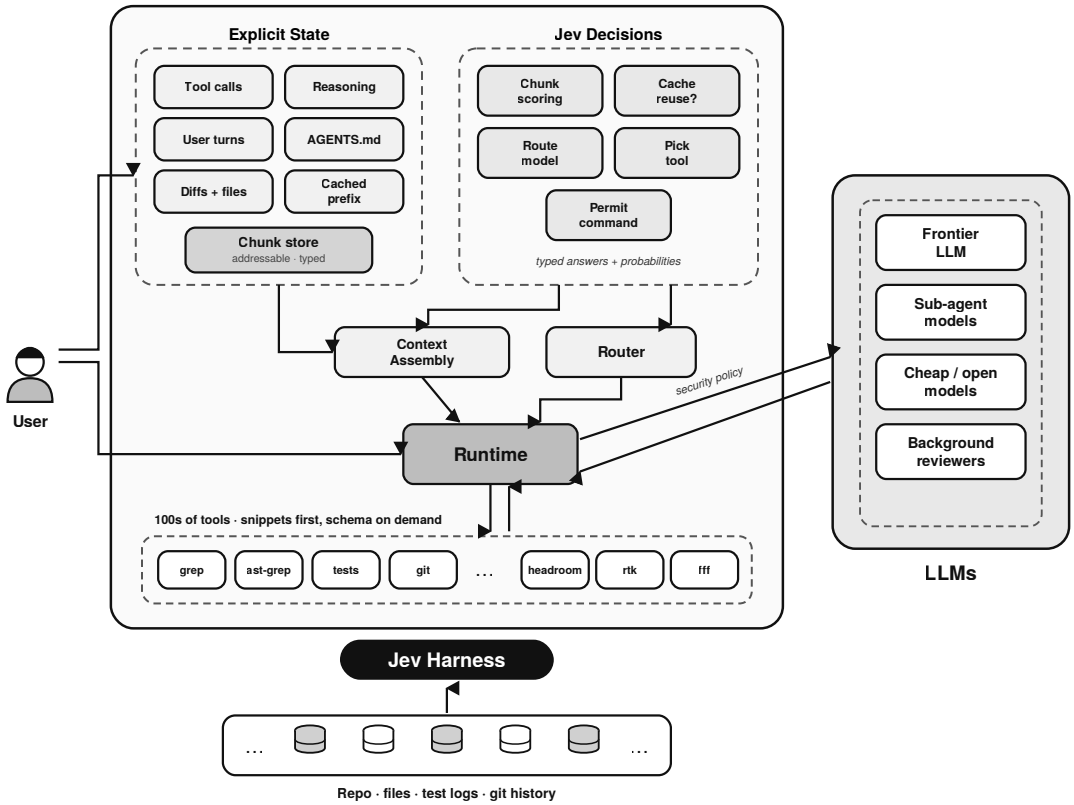

Fig. 1. The *Jev* harness. Explicit state (left) is stored as addressable, typed chunks. *Jev* (right) answers the per-turn questions: which chunks to show, whether to reuse the cache, which model and tool to use, and whether a command may run. Context Assembly and the Router feed the Runtime, which calls hundreds of tools disclosed snippet-first and routes work to frontier, sub-agent, cheap, or background models under a security policy.

***Abstract*—**Coding agents are surprisingly simple. Most are a while loop around a model with a small number of tools, and there has been little meaningful innovation outside the model itself. This note synthesizes design notes by Diogo Almeida, founder of TypeSafe, on building a coding agent around Jev, TypeSafe's decision-making model that converts structured state into typed outputs such as choices, scores, and noul decisions. The notes start from one provocative question: how would you design a coding agent if language models had no KV cache? The question exposes six design choices that current agents inherit without examination: routing that loses money, tool calling that crowds the context, compaction that compresses blind, sub-agents that rarely fire, restarts that discard state, and the batteries debate. We present the alternative architecture the notes propose: a harness built around explicit, typed state, with Jev making the per-turn decisions, where every chunk of context is scored per query, routing is priced including reprocessing cost, tools are disclosed in tiers, instructions load conditionally, and read-only background tasks share one retrieval pass. We work through the routing arith-

metic, the token economics of real agent sessions, and the security case for routing by file sensitivity rather than difficulty alone.

***Index Terms*—**Jev, TypeSafe, coding agents, harness engineering, KV cache, context engineering, model routing, sub-agents, tool calling, compaction, AGENTS.md, background agents.

I. Why Yet Another Agent

The notes open with four working assumptions. First, coding agents are simple, especially the agentic part: a loop, a model, a handful of tools. Second, the best parts of existing agents can be reused. The frontier models are all available through APIs, open-source projects provide inspiration and even UI components, and only a few areas, such as authentication for MCP servers, carry real complexity. Third, the cost advantage of first-party agents may be shrinking, as usage drifts toward pay-as-you-go API pricing rather than bundled subscriptions. Fourth, and most important, some capabilities can only be built natively. They cannot be delivered as a plugin to someone else's agent, because they require control over how context is assembled on every turn.

That fourth assumption is the thesis. If an agent is just a loop, the loop is not where the leverage is. The leverage is in what the loop feeds the model each time it turns.

A. Where Jev Sits

Jev is not the model that writes the code. It is the decision layer beside it. The harness hands Jev the current application state (goal, context, rules, available actions, previous actions) together with a predefined question, and Jev returns a typed answer: a choice, a score, or a noul decision, each with a probability. Frontier models, sub-agents, tools, and deterministic code then do the actual work. Because the output is typed rather than free-form, the harness can validate it, apply thresholds, and branch on it without parsing prose.

Every native feature in this note is, underneath, a Jev question asked at a high-frequency decision point. Which chunks should the next turn see? Should this subtask leave the frontier model? Which tool matches this intent? Should this command run? Asked thousands of times per session, those small decisions are where the notes locate the leverage.

TABLE I
THE JEV QUESTIONS IN A CODING AGENT

Decision point	Question to Jev	Typed answer
Context	How visible should this chunk be for this query?	choice: hide / short / long / full
Cache	Reuse the cached prefix or rebuild?	noul + probability
Routing	Can this subtask leave the frontier model?	choice + cost estimate
Tools	Which tool fits this intent?	ranked choice, top-k
Permissions	Should this command run?	allow / ask / deny
Security	Which files will this task touch?	sensitivity score

B. The Question That Organizes Everything

The notes describe a favorite question to put to other engineers: *how would you design a coding agent if LLMs had no KV cache?* The KV cache is the reason agents are built as append-only transcripts. Reusing a cached prefix is cheap; changing anything early in the context invalidates the cache and forces the model to reprocess everything after the change. That single economic fact shapes almost every design decision in current agents, usually without anyone stating it.

Imagining the cache away does two things. It permits a state-explicit design built for Jev, where context is assembled rather than accumulated. And it reveals why ideas that feel intuitively correct, like routing easy work to a cheap model, fail in practice. The notes call this the tyranny of the KV cache.

II. SIX SYMPTOMS OF THE KV CACHE

TABLE II
SIX DESIGN CHOICES CURRENT AGENTS INHERIT

Symptom	Why it exists	What it costs
1. Routing fails	Handing back to the large model reprocesses context	Mixed routes cost more than pure frontier
2. Tools crowd context	Schemas must sit in the system message	Tokens spent on irrelevant tools; weaker selection
3. Compaction	One shared state assumed for all future turns	Query-blind compression loses what matters later
4. Subagents are rare	Choosing what context to pass in and merge back is hard	Little automatic parallelism
5. Restarts	Stateful transcripts corrupt over time	Relevant old state discarded with the bad
6. Batteries debate	Every built-in costs permanent context	Forced choice between ease and power

A. Routing Does Not Work

The intuitive plan is to let a frontier model plan, hand execution to a cheaper model, and bring the frontier model back to review. The notes run the numbers using list prices of $5 input and $25 output per million tokens for Opus, and $3 and $15 for Sonnet. Let X be context tokens, Y generated output tokens, and Z additional tokens read during the work, such as command output and file reads.

```
Path 1  pure Opus
  generate  25·Y
  read       5·Z
  total     25Y + 5Z

Path 2  Opus → Sonnet → Opus
  Sonnet loads context        3·X
  Sonnet generates           15·Y
  Sonnet reads                3·Z
  Opus reloads what changed   5·(Y+Z)
  total     3X + 20Y + 8Z
```

Path 2 is more expensive whenever the session is long (large X), whenever the work reads substantially more than it writes, or some combination of the two. Plugging in a plausible session shape of $X = 0.65$, $Y = 0.12$, $Z = 0.23$ gives 4.15 for pure Opus against 6.19 for the routed path. Staying on the frontier model costs roughly two thirds as much as the route that was supposed to save money.

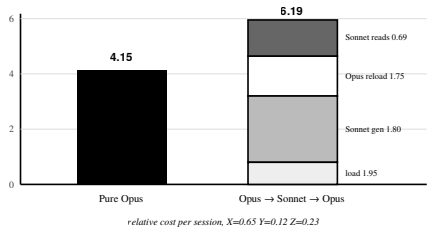

Fig. 2. The routing trap. Delegating execution to the cheaper model adds a context load on the way down and a reprocessing pass on the way back, which together outweigh the per-token discount.

The lesson is not that routing is wrong. It is that routing priced per token, rather than per context rebuild, is wrong. Routing becomes viable only when the harness can hand the cheaper model a small, purpose-built context instead of the full transcript, and when the return trip does not force the frontier model to reread everything the helper produced.

B. Tool Calling Is a Weird Tradeoff

Tools must be declared up front in the system message, with their full argument schemas, whether or not they are relevant to the current turn. This consumes a large share of context and, in the notes' assessment, does not produce especially smart tool selection. The working hypothesis is that models struggle with some combination of high cardinality (many tools at once) and off-policy tool calling (tools whose usage patterns differ from what the model saw in training). This may be part of why skills, which load a short description and defer detail, often outperform raw tool lists and MCP servers.

C. Compaction Exists

Compaction makes perfect sense if every future turn wants the same shared state. The notes question that assumption. Compaction attempts generic compression, which is hard and lossy. Query-aware compression is far easier: if you know what the next question is, you know what to keep. A summary written before the question is known will reliably throw away something the question needed.

D. Sub-Agents Are Meh

Models parallelize less than one would expect. The suspected cause is state management: deciding which parts of the parent context to pass in, and which parts of each sub-agent's findings to merge back. When that decision is expensive and error-prone, the model avoids it.

E. Restarting Exists

Restarting a session is the standard remedy for a transcript that has drifted or become corrupted. It discards the bad state and the good state together. With addressable state, the alternative is to start clean and reload only the old chunks that are still relevant, on demand.

F. Batteries Are Not Included

There is a standing debate over whether agents should ship with built-in capabilities. Today it is a tradeoff between ease of use, where heavily batteried agents sit at one extreme, and power-user tools such as Claude Code and Codex at the other. The tradeoff exists because every battery costs context permanently. Remove that cost and the debate dissolves.

III. Where Tokens Actually Go

Before redesigning the harness, it helps to know which part of a session consumes the budget. The breakdown below estimates the share of total processed tokens by subtask in a typical CLI coding-agent session. It is an input-heavy view, so repeated rereads count each time.

TABLE III
Token Share by Subtask (Input-Heavy View)

Subtask	~Share	Note
Reading file contents	30–40%	Largest bucket; files re-read as context
Searching the codebase	10–18%	grep, glob, listings; noisy output
Command output	10–20%	Stack traces and logs balloon on failure
System prompt, tool schemas, AGENTS.md	5–12%	Fixed overhead paid on every turn
Conversation replay	amplifier	Why everything above is counted repeatedly
Reasoning and planning	5–15%	Higher on hard debugging
Writing and editing code	4–10%	Diffs and str_replace edits are compact
Explaining to the user	2–5%	Terse by design in CLI agents

The striking row is the one near the bottom. Writing code, the thing a coding agent exists to do, is among the smallest line items. Reading and searching dominate. Independent analysis points the same way: Microsoft's fastcontext project reports that in GPT-5.4 trajectories, reading and searching account for 56.2% of all tool-use turns

and 46.5% of the main agent's total tokens. If that generalizes, the largest efficiency gain in a coding agent is not a better model or a better diff format. It is smarter retrieval.

Fig. 3. Retrieval dominates. Reading, searching, and command output together account for roughly two thirds of processed tokens; writing code is under a tenth.

IV. Basic Layer: Permissions and Tool Routing

Two improvements could be integrated into any existing agent, native or not.

A. Programmable Permissions

Every command an agent runs raises the question of whether it should run at all. Claude's auto mode answers this with a classifier. The notes propose going further: permissions expressed as programmable queries over what is and is not allowed, and deeper inspection where the stakes justify it, for instance reading the contents of a Python or shell file before executing it rather than approving the command name alone.

```
policy "exec":
  deny   if command touches ~/.ssh or .env*
  deny   if script contents contain network
egress
         and task.scope != "deploy"
  ask    if command writes outside repo root
  allow  if command in read_only_set
  allow  if tests/ and exit code is expected
```

B. The Harness as Tool Router

Instead of exposing every tool schema to the model, the harness can sit between intent and invocation. The model describes what it is trying to do in plain text. The harness then uses a sequence of typed Jev calls to select the single best tool, or the top few candidates, and constructs the arguments. The model never has to hold hundreds of schemas in context, and a wrong argument type becomes a validation error rather than a silent failure.

V. Meta-Attention: Context As a Decision

The central proposal removes the idea that context is static. For every user query the harness asks Jev two things. First, how good the previous context is: whether reusing the existing KV cache is the right call or whether rebuilding from scratch will be cheaper and better. This is framed as an explicit, cost-aware decision rather than a default. Second, how to construct a new context that contains everything relevant and nothing else.

In its simplest form this is a noul on every chunk of context: each tool call input, each tool call output, each piece of internal reasoning, and

possibly each exchange with the user. A later version replaces the score with a visibility level.

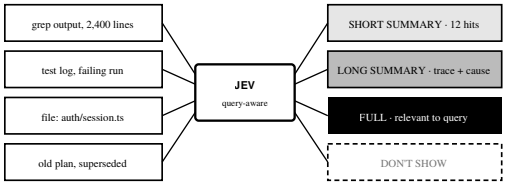

Fig. 4. The visibility ladder. The same chunk can be hidden, summarized briefly, summarized at length, or shown in full depending on the current query. This is query-aware compression, the property compaction lacks.

The payoff is that the idea behind compaction survives while its main flaw disappears. Compaction compresses once, before knowing the question. The ladder compresses per query, after knowing it. A 2,400-line grep result can be twelve relevant hits for one question and invisible for the next, without ever being deleted from state.

The notes add a visual idea worth noting: if the harness can heatmap which part of a grep output is relevant, it can filter that output down by any amount the budget allows. It would also, the notes observe, look cool.

VI. Routing and Sub-Agents Revisited

First-class dynamic contexts are what make routing work again. Once the harness can build a small, relevant context for a subtask, handing that subtask to a cheaper or faster model no longer requires the cheap model to load the whole session, and the result can be merged back as a scored chunk rather than a transcript the frontier model must reread. Everything stays cost-aware and intelligence-aware.

The same mechanism unblocks sub-agents. The notes' guess is that a large share of sub-agent cost today is the work of deciding what context to pass, compared with a simple instruction of the kind a user would type. If building that context becomes cheap and automatic, sub-agents can be used far more often. It also opens a user-facing control: spend more for faster or better results, or run conservatively and spend as little as possible.

A. Extreme Parallelism

If spawning a task becomes cheap, many tasks will run at once, and the harness inherits the problems of concurrent systems: synchronization primitives, inter-agent communication, and write collisions when several agents share state. The notes suggest shared state with locks. Explicit typing of reads versus writes is what makes that tractable, because read-only tasks never contend.

B. Deduplicating Goals

For goal-driven loops such as `/goal`, an open question is whether the agent ever repeats work. One mitigation: before spawning any subtask, register it as a subgoal and deduplicate it against all previous subgoals. Work that has already been done, or is already in flight, is never launched twice.

VII. Tools and Skills from First Principles

The notes argue for a layer between today's always-loaded tools and on-demand skills. A model cannot propose an action it does not know exists, so it needs short snippets describing what is available, similar to skill descriptions. But those snippets need not live in the system message; they can load dynamically when relevant. Behind them sits the ability to dump the full schema of available actions when needed, similar to a tool-search tool. The requirement that ties it together is that none of this corrupts the context once it is no longer needed.

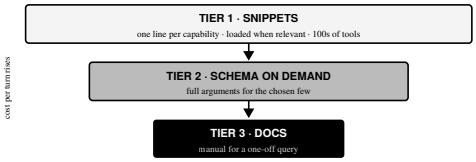

Fig. 5. Tiered disclosure. The model sees a cheap map of everything, pays for detail only on what it selects, and drops the detail from context when it is done.

If this works, the batteries debate from Section II disappears. When a built-in costs almost nothing until it is used, an agent can ship with hundreds of tools and thousands of documentation pages. The notes point out a second benefit: near-zero-cost integrations are a strong co-marketing channel, and they let things simply work.

A. Better Batteries

Many community tools promise to help agents and in practice do not. The notes cite a tool-output compressor as an example and guess at the cause: models do not natively understand these tools. A native harness can ship first-party prompts that teach the model how to use each one, effectively a built-in sub-agent or skill per tool, and its clean context means the tool's custom logic does not poison the rest of the session. There is a marketing angle here too: shipping integrations for whatever tools are currently popular keeps the agent in the conversation.

VIII. Conditional Instructions

AGENTS.md files load in full today. The notes propose loading sections conditionally. Working on front-end code loads the style guide. Working inside a particular subdirectory loads that directory's footguns file, and the notes suggest every subdirectory should have one.

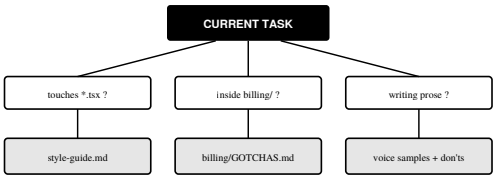

Fig. 6. Conditional AGENTS.md. Instructions attach to conditions rather than to the session, and a conditional fragment is pinned so compaction cannot summarize it away.

This resembles skills, but the notes draw a distinction. Skills tend to mean *do this now*. Conditional instructions mean *keep this in memory somewhere*. The second kind also needs a property skills lack: immunity to compaction. A skill loaded early in a long session will eventually be compacted or summarized out. A condition-bound instruction is reloaded whenever its condition holds.

The notes observe the same need in ordinary chat. A request for a summary should pull in a preferred format. A request for writing in one's own style should pull in samples and a list of things to avoid. A request for code should pull in a style guide and a note not to add a million assertions.

A. Structured Skills

Depending on how programmable the harness becomes, skills could carry behavioral changes, not just instructions, similar to Claude Code's skill hooks but more powerful. One limitation the notes flag in current hook systems is that once added, a hook stays in the session permanently. Structured skills would attach and detach behavior with the condition that triggered them.

B. Recursive Language Models

A related direction is the recursive language model approach, which treats more of the agent's state as explicit variables rather than transcript text. A world where state is held in named variables could be considerably cleaner than one where state is whatever happens to remain in the context window.

IX. Security-Aware Routing

Routing today is framed around difficulty and cost. The notes add a third axis: trust. Some open-weight models served through low-cost providers are dramatically cheaper than frontier

APIs, and the notes raise the concern that data passing through some of those endpoints may not stay private. The proposal is to give each subtask an estimate of which kinds of files it is likely to touch, attach policies to file types, and route subtasks accordingly.

TABLE IV
ROUTING BY DATA SENSITIVITY

Files likely touched	Policy	Eligible models
Public docs, open-source deps	open	any, cheapest first
Application code	standard	vetted providers
Secrets, env, infra config	restricted	first-party frontier only
Proprietary research code	custom	excludes named vendors

The last row reflects a broader point in the notes: difficulty and cost are not the only reasons to route. A team might avoid one vendor's models for its own model research, or avoid certain providers for safety-sensitive work. Once routing is policy-driven, those preferences become configuration rather than discipline.

X. BACKGROUND PROCESSING

The notes identify an emerging pattern across several popular agent workflows: building HTML pages that update in parallel as work proceeds; the argument that understanding, not generation, is the new bottleneck; generating evals in the background; an ELI5 skill that explains a system with a few big pictures and very few words; and the habit of having an agent maintain a small deployed progress page, with screenshots and notes, that can be checked from a phone during a long run. A related production pattern mirrors live traffic to a candidate model and generates evals automatically for roughly a day before any switch is made.

What these share is that they run in the background, as extensions to the normal coding workflow, and they are read-only functions of the current codebase state. Cross-model review, such as having one vendor's agent review another's work, fits the same mold.

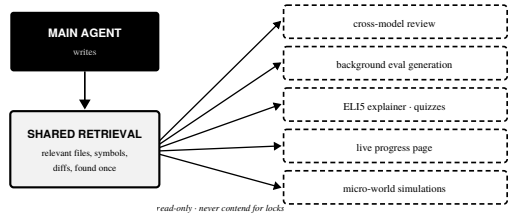

Fig. 7. Background processing on explicit state. The expensive work of finding what is relevant to a change is done once and shared by every read-only background task.

This is where the Jev thesis pays off. A Jev-centric harness has to be precise about what is in context and whether each operation reads or writes. Finding the information relevant to a code change is non-trivial work. If that retrieval is shared across all background tasks rather than re-

peated by each one, running them becomes much cheaper, and it becomes economical to run many more of them. Given Section III's finding that retrieval dominates the token budget, sharing it is the largest single saving available. Being explicit about reads versus writes, in the notes' words, should eventually give the agent superpowers.

XI. The Batteries Shortlist

The notes close with open-source projects that could be integrated natively, each with a design note on how a Jev-centric harness would use it.

TABLE V
Candidate Built-In Tools

Project	Role	Native angle
headroom	context compressor	classifier checks the compression kept the needed facts
rtk	tool-output compressor	first-party prompts so the model understands it
ast-grep	structural search	load manual once, generate N queries, filter by relevance
ast-outline	structural outline	hierarchical calling: pick the subtree to inspect
fast-context	repo-exploration sub-agent	route to it, or replace its search with structure
fff	path and content search	in-memory index, frequency-ranked, faster than ripgrep in long sessions

XII. Conclusion

The design notes make an argument that is easy to state and hard to act on. Coding agents are simple loops, and the loop is not where the leverage is. The leverage is in what the harness puts in front of the model on every turn, and today that decision is made by default, by an append-only transcript shaped around KV cache economics.

Take the cache away as a thought experiment and six familiar behaviors look different. Routing fails because context is reprocessed, not because cheap models are weak. Tools crowd the window because they must be declared up front. Compaction loses information because it compresses before the question is known. Sub-agents are rare because passing state is hard. Restarts throw away good state with bad. And the batteries debate exists only because every built-in costs context forever.

The proposed harness addresses all six with one move: make state explicit and typed, and let Jev decide context, routing, tools, and permissions per query. Chunks are scored on a visibility ladder. Routing is priced per context rebuild. Tools are disclosed in tiers. Instructions attach to conditions. Background tasks share one retrieval pass. None of this requires a better model. It requires treating the context window as something assembled on purpose rather than something that accumulates by accident.

Sources

Independent synthesis for study, not affiliated with or endorsed by TypeSafe. Jev is described from TypeSafe materials as a decision model returning typed choices, scores, and noul decisions with probabilities. Core arguments, routing arithmetic, the six symptoms, and the proposed features are from design notes by **Diogo Almeida**, founder of TypeSafe, as provided to the compiler. Reading and searching share figures (56.2% of tool-use turns, 46.5% of main-agent tokens in GPT-5.4 trajectories) are as reported by Microsoft's fastcontext project. The token-share table is an illustrative estimate of CLI coding-agent sessions. Recursive language models: A. Zhang, 2025. Tool references: headroom, rtk, ast-grep, ast-outline, fastcontext, fff (GitHub). Model list prices are as used in the notes and may have changed. All diagrams are original.
