How AI Coding Agents Read Code: Inside the Runtime Architecture of Inspection, Memory, and Control

Leandro Damasio

May 2, 2026

Abstract

AI coding agents are often described as "LLMs with better prompts," yet much of their practical capability comes from a surrounding runtime layer: mechanisms outside the model weights that mediate inspection of code, governance of memory under bounded context, and control/approval of actions and tool use. We develop a systems-oriented vocabulary for these mechanisms and propose a measurement-first methodology for studying them in fast-moving, partially closed ecosystems.

Concretely, we contribute (i) a taxonomy of recurring runtime primitives (inspection, memory governance and compaction, persistence/recall, control/approval, and protocol glue), (ii) an instrumentable experimental protocol that standardizes tasks and codebase scales to elicit these primitives, and (iii) a logging and metrics framework that maps primitives to observable event streams (context-length trajectories, tool-call traces, latency, retries, and drift-recovery behavior). The intended outcome is a falsifiable substrate for cross-agent comparison focused on what can be observed and reproduced at the level of measurement, even when implementations and underlying models evolve.

1 Introduction

Large language models (LLMs) are often treated as the "core" of AI coding systems, but in practice many capabilities emerge from the surrounding runtime: mechanisms for inspection, memory governance, and control that mediate what the model can observe, retain, and execute over time. This paper is a *taxonomy, protocol, and measurement-framework* contribution: we aim to make these runtime mechanisms explicit and comparable, rather than to report finalized performance results.

This paper takes a *systems* perspective (OSDI-style): we treat each agent as a runtime that mediates access to state (code, tools, and history), and we define a set of recurring, instrumentable primitives that can be logged and compared across implementations. We separate (i) what we establish here—the taxonomy, observables, and experimental protocol—from (ii) the empirical execution and cross-agent measurements, which are planned as follow-on work using the proposed harness.

Thesis. *Runtime > Prompt.*

1.1 The chain is the architecture

Most people building AI agents optimize the wrong layer: they compare models, rewrite prompts, and tune parameters. Those choices affect reasoning quality, but reasoning is only one stage of the end-to-end system. An agent is not only a model; it is a runtime that governs what the model can see, what it can do, and what it can retain over time.

We call this ordered runtime sequence *the chain*: tool governance, inspection, compaction, persistence, recall, action, and control. Each stage transforms the state handed to the next stage. As with a physical chain, overall performance is constrained by the weakest link, not the strongest one.

This perspective explains frequent failure modes. Even a frontier model will reason poorly when upstream context is polluted or incomplete. Likewise, without durable persistence, an agent can perform well within a session yet forget constraints and decisions between sessions. In both cases, model capability exists, but the runtime chain fails to preserve and deliver task-relevant state.

Operationally, chain quality can be summarized by three questions: what can the agent *perceive*, what can it *do*, and what can it *remember*? Perception is determined by inspection policy and coverage; action is bounded by tool permissions and approval controls; persistent memory resides in external artifacts (files, documentation, commit history, and structured stores), not in model weights alone.

This is why the chain is the architecture. Architecture is not merely a static component diagram; it is the sequence of runtime decisions that governs what the system's most capable reasoning component can inspect, execute, and carry forward. At the meetup, we illustrated this directly with terminal-level mechanisms: `rg` for textual inspection, Tree-sitter for structural parsing, `git` for persistence and recall, and `pytest` for behavioral feedback. The model was not the actor in those steps, yet those steps largely determined outcome quality.

Research question (RQ1). Which "runtime primitives" appear recurrently in modern coding agents?

Scope and evidence types. The motivating artifact set for this paper comes from a February 23, 2026 meetup talk that contrasted *nine* agents across terminal, editor, and agentic-IDE surfaces (Claude Code, Cursor, Codex CLI, Zed, Trae, OpenCode, Gemini CLI, Antigravity, and Robson). In this draft, those systems serve only as motivating examples for vocabulary and observables; our study object is the *runtime layer* and the *measurement methodology*, not a product ranking.

Because many implementations are closed, we structure claims by evidence type: (i) *observed* (screenshots, tool traces, OS-level telemetry, reproducible behaviors), (ii) *inferred* (best-effort hypotheses consistent with evidence), and (iii) *unknown* (unverifiable internal mechanisms). This partition is used throughout as a discipline for scientific honesty and reproducibility.

Contributions.

1. **A taxonomy of runtime primitives** for AI coding agents, with working definitions and a draft coding sheet connecting primitives to observable signals.
2. **A concrete, instrumentable experimental protocol** that standardizes tasks, codebase scales, and stopping rules for observing how primitives manifest in practice.
3. **A measurement and logging framework** that maps primitives to runtime events (tokens/context trajectory, tool-call traces, latency, control/approval events, and drift-recovery traces), enabling future cross-agent comparisons.

From talk artifacts to falsifiable measurements. The meetup artifacts motivate several recurring qualitative claims (e.g., "compaction helps cost but risks forgetting constraints," "persistent project rules can influence subsequent edits," and "providers can reduce cost/latency via caching"). In the paper, we translate such claims into falsifiable statements by pairing each with: (i) a concrete observable signal, (ii) a measurement procedure, and (iii) a threats-to-validity note that distinguishes correlation from causation.

2 Background and Definitions

2.1 Agent surface

We categorize user-facing surfaces as *terminal*, *editor*, or *agentic IDE*.

2.2 Runtime primitives (working definition)

We use *runtime primitive* to mean a concrete mechanism that (i) observes state (inspection), (ii) transforms state (memory governance/compaction), or (iii) restricts/mediates actions (control), and that is implemented outside the base model weights.

2.3 Memory governance mechanisms: transformation vs. execution optimization

In informal discussions, "memory management" is often used to refer to two distinct classes of mechanisms:

1. **State transformation (semantic, often lossy).** The runtime reduces or reorganizes what is carried forward as execution context, via summarization, eviction, thread rollover, or promotion of artifacts to external storage. We use *compaction* for transformations that primarily

shrink the execution buffer. These mechanisms can change downstream agent behavior (e.g., by removing constraints).

2. **Execution optimization (non-semantic, ideally lossless).** The runtime or provider infrastructure reduces cost/latency via caching and reuse of previous computations or prompt prefixes, without intending to change the semantic content of the task state.

These two classes have different risks and require different observables. For example, compaction introduces risks such as constraint loss and auditability gaps, while caching primarily changes efficiency metrics (latency/cost) and can be studied via cache-hit rates and billing traces.

2.4 A control-theoretic lens on agent runtimes

A useful framing is to treat the agent+runtime loop as a discrete-time feedback system: each step observes a partial state of the world (repo + tools + history), updates an internal state (working memory, summaries, plans), and selects an action whose outcomes feed back into the next step. This is the same conceptual structure emphasized in modern control texts—input/output relations, internal state, dynamics over time, and feedback—even though the "state" here is largely symbolic/semantic and the dynamics are strongly non-linear and stochastic [6].

In this paper we use the control framing as a modeling discipline (what counts as state, input, output, feedback, and stability), not as a claim that classical linear continuous-time techniques apply directly. Our focus stays on *observable* runtime mechanisms (inspection, memory governance/compaction, and control/approval) and on instrumentable traces of the feedback loop.

Control notion	**Agent/runtime analogue**	**What we can log**
Input u_t	prompt, repo state snapshot, tool outputs	prompts, tool results, environment metadata
State x_t	context buffer, external memory, plan/checkpoints	context length, summary/compaction events, state checkpoints
Output y_t	edits, commands, tool calls, stop/continue decision	diffs, tool-call traces, termination events
Feedback	tests, compiler errors, lint, user approvals/denials	exit codes, test results, approval prompts and decisions
Stability / drift	error accumulation, regressions, looping, recovery	retries, rollback events, drift-recovery traces

Table 1: A control-theoretic mapping (discrete, stochastic, symbolic) for agent runtimes. The mapping is used to define observables; it does not assume linear continuous-time models.

2.5 Program-analysis roots of inspection

Many inspection signals we log (file traversal, symbol relationships, and dependency edges) are runtime-level approximations of classic program-analysis representations such as control-flow graphs,

data-flow analysis, call-graph construction, program slicing, and dependence graphs [1, 5, 7, 8, 4]. These representations also motivate composite graphs used in security and code understanding work (e.g., code property graphs) and learning-based code representations that incorporate data-flow structure [**yamaguchi2014cpg**, **guo2021graphcodebert**]. We treat these as prior art on *what* structure exists in code, while our focus is on *how* modern agents (as interactive systems) choose to discover and maintain that structure under tight context and cost constraints.

2.6 Historical trajectory: from textual search to runtime-orchestrated agents

Understanding how coding agents read code benefits from a longer historical trajectory of machine code interpretation. A practical lineage runs from textual search, to syntax- and semantics-aware tooling, to contemporary agent runtimes that orchestrate these capabilities under explicit control policies.

The first layer is textual inspection. Tools such as `grep` (and modern high-throughput descendants such as `ripgrep`) remain foundational for fast repository triage: they answer "where does this pattern appear?" with minimal overhead. This layer is intentionally shallow—it is powerful for narrowing search space, but it does not by itself encode syntax, types, or inter-symbol relationships.

The second layer is structural parsing. Compiler pipelines established the idea that source text should be transformed into syntax trees before higher-order analysis. Abstract syntax trees (ASTs) remove punctuation-level detail while preserving semantically relevant structure (e.g., operator precedence and nesting), enabling transformations, refactoring, and static checks that are difficult to express over raw text alone. In modern editors, incremental parsers (e.g., Tree-sitter-style systems) make this structural layer available interactively at edit time.

The third layer is project-scale semantic navigation. With language-server architectures, symbol resolution and cross-file relations become first-class operations (definition/reference lookup, type-aware completion, workspace symbol queries). This shifts code understanding from local file parsing to a continuously updated semantic index spanning the repository.

Recent systems add a fourth layer: embedding-based retrieval. Vector search complements lexical and symbolic methods by returning code that is conceptually related to a task even when exact tokens differ. In practice, robust systems treat these methods as complementary rather than substitutive: lexical search for precision and speed, structural/semantic analysis for correctness, and embedding retrieval for conceptual recall.

Modern coding agents are best understood as runtimes that orchestrate all four layers in a closed loop. Instead of loading full repositories into a single prompt, they perform staged perception (inspect, retrieve, prioritize, compress), model reasoning over curated context, then controlled action (edits, commands, tests) with governance constraints. From this perspective, model quality is necessary but insufficient: runtime quality determines what the model can perceive, what actions are admissible, and which state survives across turns.

This historical view motivates our central claim—*runtime* > *prompt* as an explanatory lens for end-to-end agent behavior. It also clarifies why measurement must target runtime traces (inspec-

tion trajectories, compaction events, control decisions, and recovery behavior), not only final task outcomes.

2.7 Connection to cognitive-architecture decompositions

Recent cognitive-architecture proposals for "artificial beings" explicitly separate (i) *observation* (adding sensations/observables to memory), (ii) *coordination* (adding inferences), (iii) *reflection* (altering the system's behavior), and (iv) *consolidation* (compacting memory) [2]. While these works target general agency rather than code, their decomposition aligns closely with the runtime primitives we study in coding agents:

- **Observation → Inspection:** structured and repeated access to external state (code, tools, and environment).
- **Consolidation → Memory governance/compaction:** mechanisms that summarize, compress, and forget to sustain long horizons.
- **Reflection → Control:** meta-control over action selection, safety/approval gates, and mode switching.

We use this alignment as motivation, but focus our empirical results on observable runtime behavior in production coding agents.

3 Taxonomy for RQ1

Table 2 is a draft coding sheet for labeling systems.

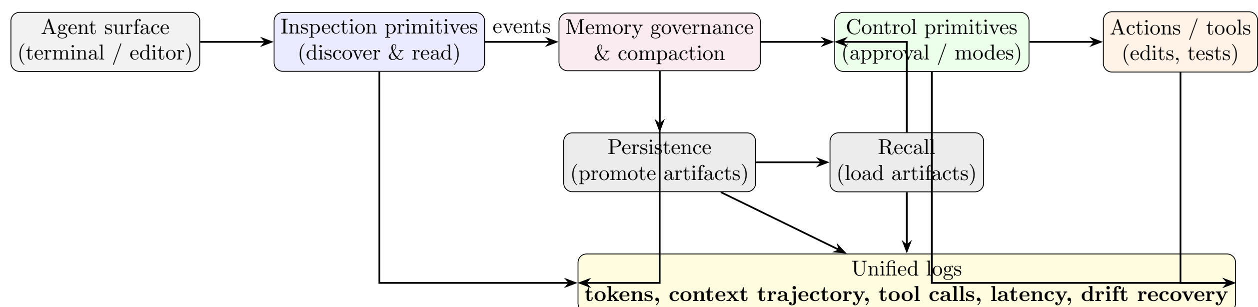

Figure 1: Runtime-primitives dataflow: surface affordances drive inspection; inspection expands state; governance compacts it; control gates actions; all stages emit measurable events.

Primitive	Observable signal	Operationalization (what to log)
Inspection (structure)	directory / file discovery	#paths listed, #files opened, repo-scan steps
Inspection (content)	line-range reads, greps	#reads, #searches, bytes read, coverage
Inspection (relations)	symbol/refs/graph building	#symbol lookups, dependency edges observed
Memory governance	summaries, thread rollover	context length over time; summary events
Compaction	compression / eviction	before/after token counts; retained keys
Persistence	promote working artifacts to archive	file writes/copies; VCS events (e.g., `git add`)
Recall	load archived artifacts into execution	reads of persisted docs before edits; citations in plan
Control/approval	ask-before-edit, modes	edits gated? approvals required?
Protocol glue	MCP/ACP integrations	tool registry; protocol messages; error rates

Table 2: Draft runtime primitives and how we plan to observe them.

4 Hypotheses, operational variables, and minimal metrics

The meetup-derived landscape motivates qualitative statements about runtime behavior. To make them scientifically usable, we express them as falsifiable hypotheses tied to operational variables and a minimal measurement set. This section is intentionally prescriptive: it specifies what to log even when internal implementations are opaque.

4.1 Operational variables

We define a minimal set of operational variables that can be extracted from runtime traces:

- **Observability/visibility** (V): which runtime events are externally visible (e.g., explicit compaction banners vs. silent behavior).
- **Governance locus** (G): where governance is implemented (client runtime vs. provider infrastructure), inferred from available evidence and interfaces.
- **State-bounding mechanism** (B): how long-horizon behavior is bounded (compaction, thread partitioning, FSM, retrieval).
- **Control surface** (C): approval gates, permission models, budgets/circuit breakers, and their placement relative to tool execution.

4.2 Minimal metrics (cross-agent comparable)

We recommend logging at least:

- **Context trajectory:** tokens (or bytes) sent per model call; before/after values for compaction events; compaction frequency.
- **Tool trajectory:** tool-call count, tool-call latency distribution (p50/p95), exit codes, retries, and backtracks.
- **Inspection efficiency:** file-read coverage proxies (unique paths opened), redundancy (re-reads), and time-to-first-relevant-file (task-defined).
- **Outcome metrics:** task pass/fail (tests), time-to-solution, and number/size of edits (files changed, lines added/deleted).
- **Cost proxies:** billed tokens or provider cost units, and cache-hit indicators when available.

4.3 Example falsifiable hypotheses

H1 (compaction efficiency–quality trade-off). Compaction reduces total cost per completed task by decreasing context size, but overly aggressive compaction increases retries and failures due to loss of constraints. Operationalization: correlate compaction ratio and compaction frequency with (i) total billed tokens/cost, (ii) retries, and (iii) final task success, controlling for task and codebase scale.

H2 (persistent artifacts and subsequent edits). If a runtime persists planning artifacts and later recalls them into the execution buffer, subsequent edits will show higher alignment with the plan than in runs where recall is prevented. Operationalization: use a "plan influence" protocol that compares runs with (A) persist+recall, (B) persist but block recall, and (C) recall a placebo plan; score plan–diff alignment and task success.

H3 (caching primarily affects latency/cost, not behavior). Provider-managed caching reduces latency and billed cost, with minimal effect on semantic outcomes for fixed visible context. Operationalization: compare p50/p95 latency and billed tokens between cache-hit and cache-miss conditions while holding user-visible context constant (as much as interfaces allow).

5 From Meetup Observations to a Scientific Framing

The meetup talk format makes an explicit distinction between (i) what an agent exposes at its surface, (ii) what we can reliably measure, and (iii) what remains hidden by closed-source runtimes and provider-side policies. We can reuse this as a methodological discipline in the paper.

5.1 Concrete motivating artifacts: four "screenshots" and a memory cycle

A useful improvement over purely conceptual discussion is to anchor claims in a small, reproducible artifact set. The meetup talk used four screenshots (two general UI/runtime messages and two Cursor traces) to motivate a concrete, testable sequence:

1. **Inspection expands the explored set:** directory discovery, file reads, and searches increase the number of artifacts touched in a session.
2. **Compaction governs execution context:** when the execution buffer grows too large, a compaction event reduces the active context (often with an explicit "compacting" message and a before/after percentage).
3. **Persistence promotes working memory to an archive:** session-generated artifacts (e.g., a plan stored in a tool-specific directory) are copied into a repository-backed document and staged in version control.
4. **Recall loads archived artifacts before edits:** the agent re-reads the persisted document shortly before applying patches.
5. **Control gates actions:** multi-file edits are presented for review/approval.

This suggests a simple three-layer model of memory that we can operationalize in logs:

- **Execution buffer (Layer 1):** what the model sees in the current request.
- **Working memory (Layer 2):** session-local artifacts and plans stored outside Layer 1.
- **Archive (Layer 3):** repository documents and other durable artifacts that survive sessions.

5.2 Compaction trade-offs (benefits vs. risks)

The same artifacts also motivate a key methodological caution: compaction enables longer sessions and lower token cost, but can drop constraints and reduce auditability. In the measurement plan, this means we should log not only the frequency and magnitude of compaction, but also downstream indicators of harm: retries, regressions, repeated re-inspection, and "goal drift" behaviors.

5.3 Observation discipline: observed vs. inferred vs. unknown

For each claim about an agent's runtime, we label it as:

- **Observed:** directly visible in UI/logs/telemetry (e.g., token banner, explicit "compacting" message).

- **Inferred:** a plausible explanation consistent with observations, but not directly verified.
- **Unknown:** not observable with our instrumentation, or plausibly happening server-side.

This labeling supports fair cross-agent comparisons and constrains over-interpretation.

Label	Allowed evidence	Example phrasing
Observed	UI banners, runtime messages, logs, tool traces, token counts, timings	"We observed X in the UI/logs."
Inferred	Explanations consistent with evidence, but not independently confirmed	"It seems consistent with X, but we did not verify."
Unknown	Not measurable; closed-code or provider-side policies	"Internal policy is unknown."

Table 3: A conservative evidence taxonomy for runtime claims.

5.4 Evals and observability: measuring trajectories

Coding agents are not pure functions; they are multi-turn systems whose behavior is a *trajectory* of tool calls, state transitions, and recovery steps. To connect runtime primitives to evaluation, we adopt a simple multi-layer validation view [**lindenberg2026evals**]:

- **Code-based:** verify outcomes with executable checks (tests pass, command exits 0).
- **Model-based:** judge decision quality (search efficiency, plan quality, avoidance of drift).
- **Human:** confirm intent alignment and usefulness.

This paper's contribution is the observability substrate (logs and evidence labels) needed to support all three layers.

From evals to learning signals. When the outcome is verifiable (e.g., tests and build), trajectory logs can be scored and used as feedback signals for improvement (often discussed as reinforcement learning from verifiable rewards, RLVR) [**lindenberg2026evals**].

5.5 Inspection strategies as runtime-level design choices

The talk distinguishes three recurring inspection styles that can be operationalized via tool traces:

- **Command-based (terminal):** explicit filesystem traversal (e.g., directory listing), pattern search, and line-range reads.

- **Mention-based (editor):** referencing artifacts through UI affordances (e.g., file/symbol mentions) that trigger structured retrieval.
- **Retrieval-based (agentic IDE):** semantic and/or graph-guided retrieval with a smaller execution buffer.

This becomes a measurable axis: which inspection actions occur, with what coverage and cost (Table 2).

5.6 A "runtime map" for positioning agents

To make comparisons legible, we can place agents on two coarse dimensions: (i) surface (terminal/editor/agentic IDE), and (ii) observability of memory governance (explicit runtime-managed signals vs. model/provider-native caching vs. state-bounded designs vs. unknown).

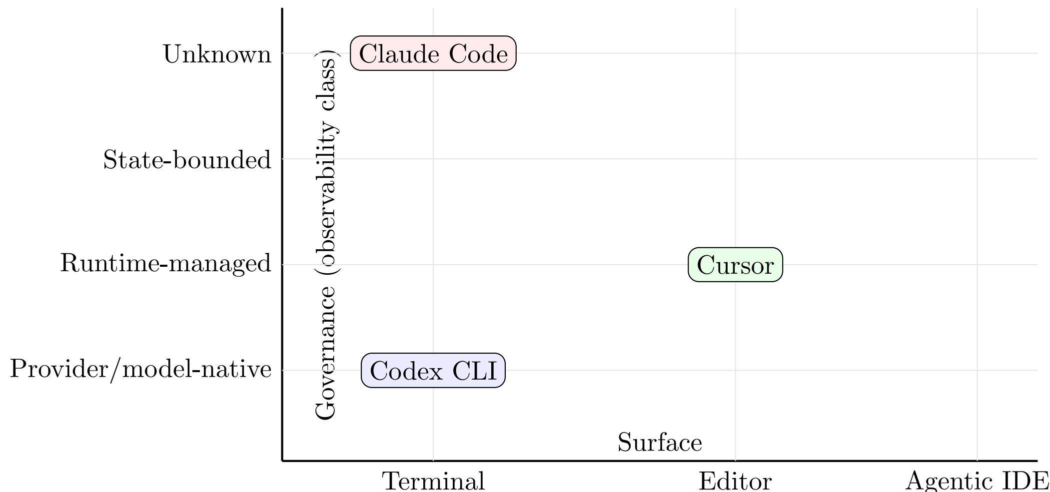

Figure 2: Runtime map (draft): the scientific claim is not the exact placement, but the rule for mapping observable evidence into surface × governance classes.

5.7 A simple cost model tied to governance

The talk motivates governance as a cost-efficiency lever. In the paper we can formalize this as a decomposition of expected cost per task:

$$\text{Cost} \propto \underbrace{T}_{\text{tokens per attempt}} \times \underbrace{A}_{\text{attempts/retries}} \times \underbrace{R}_{\text{repair overhead (re-explaining/re-reading)}}. \quad (1)$$

Memory governance mechanisms act on all three factors by (i) shrinking the execution buffer (lower T), (ii) reducing drift and rework (lower A), and (iii) reducing repeated inspection due to lost context (lower R).

Governance class	Terminal	Editor	Agentic IDE
Unknown (not observable)	Claude Code		
Runtime-managed (observable compaction)	Gemini CLI (if UI/telemetry exposes events)	Cursor	
Model/provider-native (cache signals)	Codex CLI; Trae; OpenCode		
State-bounded (structural limits)	Robson (FSM-style)	Zed (threads/profiles)	Antigravity (retrieval-bounded)

Table 4: A draft runtime map (meetup-derived examples). Cells indicate example placements; the scientific contribution is the mapping rule (evidence + observables), not the labels themselves.

6 From Meetup Observations to a Scientific Framing

The meetup talk format makes an explicit distinction between (i) what an agent exposes at its surface, (ii) what we can reliably measure, and (iii) what remains hidden by closed-source runtimes and provider-side policies. We can reuse this as a methodological discipline in the paper.

6.1 Observation discipline: observed vs. inferred vs. unknown

For each claim about an agent's runtime, we label it as:

- **Observed:** directly visible in UI/logs/telemetry (e.g., token banner, explicit "compacting" message).
- **Inferred:** a plausible explanation consistent with observations, but not directly verified.
- **Unknown:** not observable with our instrumentation, or plausibly happening server-side.

This labeling supports fair cross-agent comparisons and constrains over-interpretation.

Label	Allowed evidence	Example phrasing
Observed	UI banners, runtime messages, logs, tool traces, token counts, timings	"We observed X in the UI/logs."
Inferred	Explanations consistent with evidence, but not independently confirmed	"It seems consistent with X, but we did not verify."
Unknown	Not measurable; closed-code or provider-side policies	"Internal policy is unknown."

Table 5: A conservative evidence taxonomy for runtime claims.

6.2 Inspection strategies as runtime-level design choices

The talk distinguishes three recurring inspection styles that can be operationalized via tool traces:

- **Command-based (terminal):** explicit filesystem traversal (e.g., directory listing), pattern search, and line-range reads.
- **Mention-based (editor):** referencing artifacts through UI affordances (e.g., file/symbol mentions) that trigger structured retrieval.
- **Retrieval-based (agentic IDE):** semantic and/or graph-guided retrieval with a smaller execution buffer.

This becomes a measurable axis: which inspection actions occur, with what coverage and cost (Table 2).

6.3 A "runtime map" for positioning agents

To make comparisons legible, we can place agents on two coarse dimensions: (i) surface (terminal/editor/agentic IDE), and (ii) observability of memory governance (explicit runtime-managed signals vs. model/provider-native caching vs. state-bounded designs vs. unknown).

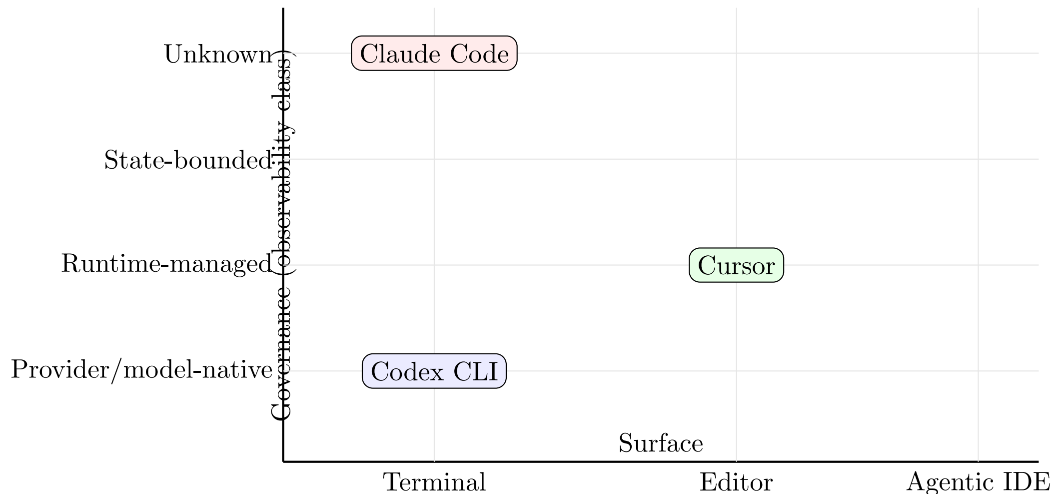

Figure 3: Runtime map (draft): the scientific claim is not the exact placement, but the rule for mapping observable evidence into surface × governance classes.

Governance class	Terminal	Editor	Agentic IDE
Unknown	Claude Code		
Runtime-managed (observable compaction)	Codex-like CLI wrappers (if exposed)	Cursor	
Model/provider-native (cache signals)	Codex CLI (prefix caching)		
State-bounded (structural limits)		Zed (threads/profiles)	Retrieval-centric systems

Table 6: A draft runtime map. Cells indicate example placements; the scientific contribution is the mapping rule (evidence + observables), not the labels themselves.

6.4 A simple cost model tied to governance

The talk motivates governance as a cost-efficiency lever. In the paper we can formalize this as a decomposition of expected cost per task:

$$\text{Cost} \propto \underbrace{T}_{\text{tokens per attempt}} \times \underbrace{A}_{\text{attempts/retries}} \times \underbrace{R}_{\text{repair overhead (re-explaining/re-reading)}}. \tag{2}$$

Memory governance mechanisms act on all three factors by (i) shrinking the execution buffer (lower T), (ii) reducing drift and rework (lower A), and (iii) reducing repeated inspection due to lost context (lower R).

7 Experimental Design and Evaluation Plan (OSDI-style)

7.1 Study objects (agents)

We evaluate three agents:

- **Claude Code** (terminal surface)
- **Codex CLI** (terminal surface)
- **Cursor** (editor surface)

For each run we record: agent version/build, model selection (if configurable), configuration flags, approval/safety mode, and operating environment.

7.2 Tasks

We use a fixed suite of tasks designed to exercise different runtime primitives:

1. **Bug fix:** identify root cause and produce a minimal patch.
2. **Refactor:** restructure code while preserving tests.
3. **Security review:** identify issues and propose mitigations.
4. **Feature addition:** implement a small feature with tests.

Each task has (i) a time budget, (ii) a pass/fail criterion, and (iii) a stopping rule (e.g., tests pass, or explicit "cannot complete").

7.3 Codebases

We run tasks on at least three repo sizes to induce different inspection/memory regimes:

- **Small:** ∼1–5k LOC
- **Medium:** ∼20–100k LOC
- **Large:** ≥500k LOC (or monorepo slice)

7.4 Instrumentation (what we log)

We log a unified event stream per run:

- **Model I/O:** input/output tokens; context length trajectory.
- **Tooling:** tool calls (type, arguments metadata, duration, exit code).
- **Inspection:** file opens; line-range reads; searches; symbol lookups.
- **Memory governance:** summary/compaction events; thread rollover; checkpoints.
- **Control:** approval prompts; denied actions; sandbox constraints.
- **Outcomes:** task success, retries, test results, and drift recovery events.

7.5 Metrics

We report:

- **Capability:** task success rate and time-to-success.
- **Efficiency:** tokens per task; tool calls per task; wall-clock latency.
- **Governance overhead:** compaction frequency; summary size; reset events.
- **Stability:** retry count; drift recovery frequency; catastrophic failures.

7.6 Analysis plan

For each agent we label which runtime primitives are present (Table 2) and compute descriptive statistics and cross-agent comparisons. We additionally present qualitative traces for representative runs to illustrate how primitives manifest (e.g., inspection growth leading to compaction).

7.7 Artifact and reproducibility plan

We will publish (i) task specifications, (ii) repo selection criteria, (iii) the logging schema, and (iv) anonymized run traces where licensing permits.

8 Related Work

Castrillo et al. survey the design space of autonomous LLM agents by decomposing agent architectures into perception, reasoning, memory, and execution subsystems, and by cataloging representative techniques and benchmarks across these layers [3]. This taxonomy is useful as a shared vocabulary for situating mechanisms such as planning/reflection, external memory, and tool/function calling within an end-to-end agent loop. However, the survey largely stops short of operationalizing the runtime control plane required for production-grade autonomy: explicit state representations and synchronization policies, measurable stopping rules and budget enforcement, auditability via standardized event schemas, and enforceable tool contracts that model pre/post-conditions, idempotency, and side effects. Our work complements this component-level view by treating autonomy and reliability as properties of the runtime-controlled execution loop, not of "adding memory and tools." We define a taxonomy of runtime primitives (inspection, memory governance/compaction, persistence/recall, and control/approval) and map them to instrumentable event streams and metrics (context trajectories, tool-call traces, retries, drift-recovery), enabling reproducible evaluation under fixed budgets and explicit safety/control policies.

9 Threats to Validity

- **Version drift:** agent behavior changes rapidly across releases.
- **Hidden policies:** provider-side caching/compaction may be opaque.
- **Task dependence:** primitives may appear only for certain workloads.

10 Conclusion

This paper reframes coding-agent capability as an outcome of runtime primitives—inspection, memory governance, control/approval, and protocol glue—and contributes a taxonomy plus an

instrumentable protocol and measurement framework for studying them. The next step is empirical execution: implementing the logging harness, running the standardized task suite across agents and codebase scales, and using the resulting traces to validate and refine the taxonomy and metrics.

A Task Suite Specification (Draft)

This appendix makes the task suite more operational by specifying (i) artifacts provided to the agent, (ii) pass/fail criteria, (iii) scoring (when applicable), and (iv) stopping rules.

A.1 Bug-fix task

Artifacts: a repository snapshot with at least one failing test (or a provided reproducer) and a target command (e.g., `pytest -q`).

Pass/fail: **pass** if the agent produces a patch such that all tests pass and no new tests fail under the same command; **fail** otherwise.

Stopping rules: stop on first passing run; stop on time budget; stop if the agent explicitly declares inability to reproduce or fix.

A.2 Refactor task

Artifacts: a repository snapshot, a refactor objective (e.g., "extract module X and remove duplication"), and an executable test suite (unit + snapshot tests when available).

Pass/fail: **pass** if (i) all tests pass, and (ii) the diff satisfies a syntactic constraint (e.g., no functional changes outside the target module set) or a behavioral constraint (e.g., snapshot tests unchanged). **fail** if tests regress or if constraints are violated.

Stopping rules: stop on first passing run; stop on time budget; stop if the agent proposes a large, non-reviewable change (threshold defined per task).

A.3 Security review task (seeded issues)

Artifacts: a repository snapshot with a known set of seeded vulnerabilities V (documented privately by the experimenter), and a rubric describing the expected categories (e.g., injection, authn/authz, insecure deserialization).

Output format: a structured report with one issue per entry, each containing: location (file/line range), category, impact, exploit sketch, and mitigation.

Scoring: treat each reported issue as a prediction. Compute precision/recall against V under a matching rule (category + location overlap). Optionally weight by severity.

Stopping rules: stop on time budget; stop after k issues reported; stop if the agent begins repeating the same issue with no new evidence.

A.4 Feature addition task

Artifacts: a repository snapshot, a precise acceptance test (or an executable spec) describing the desired behavior, and the test command.

Pass/fail: **pass** if the acceptance test and the full test suite pass; **fail** otherwise.

Stopping rules: stop on first passing run; stop on time budget; stop after n consecutive failing attempts without diff improvement (as measured by reduced failing-test count).

B Logging Schema (JSON) (Draft)

We represent each run as a stream of events. The schema below is an illustrative minimal core; implementations may add fields.

```
{
  "run": {
    "run_id": "uuid",
    "timestamp_utc": "2026-02-08T21:35:19Z",
    "agent": {
      "name": "Cursor",
      "version": "<build/version>",
      "model": "<model id>",
      "config_fingerprint": "<hash of flags + settings>",
      "approval_mode": "ask-before-edit|auto|custom"
    },
    "environment": {
      "os": "<os>",
      "repo_id": "<repo+commit>",
      "task_id": "bugfix/refactor/security/feature",
      "codebase_scale": "small|medium|large"
    }
  },
  "events": [
    {
      "t": "2026-02-08T21:36:02Z",
      "type": "inspection.read",
```

```
      "payload": {
        "path": "src/foo.py",
        "range": {"start_line": 10, "end_line": 80},
        "bytes": 2140
      }
    },
    {
      "t": "2026-02-08T21:36:10Z",
      "type": "tool.call",
      "payload": {
        "tool": "pytest",
        "args": ["-q"],
        "duration_ms": 18432,
        "exit_code": 1
      }
    },
    {
      "t": "2026-02-08T21:36:15Z",
      "type": "memory.compaction",
      "payload": {
        "before_tokens": 93214,
        "after_tokens": 24102,
        "strategy": "summary+eviction",
        "retained_keys": ["task", "constraints", "current_hypothesis"]
      }
    },
    {
      "t": "2026-02-08T21:36:40Z",
      "type": "control.approval",
      "payload": {
        "action": "apply_patch",
        "granted": true,
        "reason": "user approved"
      }
    },
    {
      "t": "2026-02-08T21:37:05Z",
      "type": "edit.diff",
      "payload": {
        "files_changed": ["src/foo.py"],
        "lines_added": 12,
        "lines_deleted": 3
      }
    }
  ]
}
```

References

[1] Frances E. Allen. “Control Flow Analysis”. In: *ACM SIGPLAN Notices* 5.7 (1970), pp. 1–19.

[2] Micah Blumberg and Michael Miller. *Building Sentient Beings*. The Self Aware Networks Institute / SubThought Corporation. 2025. DOI: `10.5281/zenodo.15522356`.

[3] Alejandro Castrillo et al. *Fundamentals of Building Autonomous LLM Agents*. 2025. arXiv: `2510.09244 [cs.AI]`.

[4] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The Program Dependence Graph and Its Use in Optimization”. In: *ACM Transactions on Programming Languages and Systems* 9.3 (1987), pp. 319–349.

[5] Gary A. Kildall. “A Unified Approach to Global Program Optimization”. In: *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 1973.

[6] Katsuhiko Ogata. *Modern Control Engineering*. 5th ed. Prentice Hall, 2010.

[7] Barbara G. Ryder. “Constructing the Call Graph of a Program”. In: *IEEE Transactions on Software Engineering* SE-5.3 (1979).

[8] Mark Weiser. “Program Slicing”. In: *Proceedings of the International Conference on Software Engineering (ICSE)*. 1981, pp. 439–449.