

Dive into Claude Code: The Design Space of Today’s and Future AI Agent Systems

Jiacheng Liu¹, Xiaohan Zhao¹, Xinyi Shang^{1,2}, Zhiqiang Shen^{1,†}

¹VILA Lab, Mohamed bin Zayed University of Artificial Intelligence, ²University College London

†Corresponding author

Claude Code is an agentic coding tool that can run shell commands, edit files, and call external services on behalf of the user. This study describes its comprehensive architecture by analyzing the publicly available TypeScript source code^a and further comparing it with OpenClaw, an independent open-source AI agent system that answers many of the same design questions from a different deployment context. Our analysis identifies five human values, philosophies, and needs that motivate the architecture (human decision authority, safety and security, reliable execution, capability amplification, and contextual adaptability) and traces them through thirteen design principles to specific implementation choices. The core of the system is a simple while-loop that calls the model, runs tools, and repeats. Most of the code, however, lives in the systems around this loop: a permission system with seven modes and an ML-based classifier, a five-layer compaction pipeline for context management, four extensibility mechanisms (MCP, plugins, skills, and hooks), a subagent delegation and orchestration mechanism, and append-oriented session storage. A comparison with OpenClaw, a multi-channel personal assistant gateway, shows that the same recurring design questions produce different architectural answers when the deployment context changes: from per-action safety evaluation to perimeter-level access control, from a single CLI loop to an embedded runtime within a gateway control plane, and from context-window extensions to gateway-wide capability registration. We finally identify six open design directions for future agent systems, grounded in recent empirical, architectural, and policy literature. Our GitHub is available at: <https://github.com/VILA-Lab/Dive-into-Claude-Code>.

Correspondence: Zhiqiang Shen (Zhiqiang.Shen@mbzuai.ac.ae)

^av2.1.88, [link](#). **Disclaimer:** All materials used in this work are obtained from publicly available online sources. We have not used any private, confidential, or unauthorized materials, and we do not intend to infringe any copyright or intellectual property rights. The original intellectual property rights to the source code belong to Anthropic.

1 Introduction

AI-assisted software development has evolved from autocomplete-style tools such as GitHub Copilot (Chen et al., 2021), through IDE-integrated assistants like Cursor (Cursor, 2026), to fully agentic systems that autonomously plan multi-step modifications, execute shell commands, read and write files, and iterate on their own outputs. Claude Code (Anthropic, 2026a) is an agentic coding tool released by Anthropic (Anthropic, 2026c). Its official documentation describes an “agentic loop” that plans and executes actions toward accomplishing a goal and can call tools, evaluate results, and continue until the task is done¹. This shift from suggestion to autonomous action introduces architectural requirements that have no counterpart in completion-based tools. These requirements define a design space, a set of recurring questions spanning topics such as safety, context management, extensibility, and delegation that every coding agent must navigate. This study uses source-level analysis of Claude Code to show how one production system answers these questions.

Despite growing adoption, Anthropic publishes user-facing documentation for Claude Code but not detailed architectural descriptions. This study uses source code analysis to describe architectural design decisions. Anthropic’s internal survey of 132 engineers and researchers (Huang et al., 2025) reports that about 27% of Claude Code-assisted tasks were work that would not have been attempted without the tool, suggesting that the architecture enables qualitatively new workflows rather than merely accelerating existing ones.

¹<https://code.claude.com/docs/en/how-claude-code-works>.

In this work, we first identify five human values/philosophies and thirteen design principles that motivate the architecture (Section 2), then organize the analysis in three parts:

1. **Design-space analysis.** We identify recurring design questions (where reasoning lives, how the iteration loop is structured, what safety posture to adopt, how the extension surface is partitioned, how context is managed, how work is delegated across subagents, and how sessions persist) and analyze Claude Code’s answers through a 7-component high-level structure and a 5-layer subsystem architecture, tracing each choice to specific source files (Section 3). The analysis aims to build a deep understanding of the system mechanism, with the goal of informing the design of better and more powerful agent systems.
2. **Architectural contrast with OpenClaw.** Beyond analyzing Claude Code itself, we also compare its design philosophy with that of open-source agent system OpenClaw (Steinberger and OpenClaw Contributors, 2026), a multi-channel personal assistant gateway, across six design dimensions to show how the same recurring questions produce different answers under different deployment contexts (Section 10), in order to highlight both the common principles and the key differences between commercial and open-source software. This comparison helps reveal how deployment setting, product goals, safety requirements, and user assumptions shape architectural choices in different ways. By examining where these systems converge and where they diverge, our study aims to provide useful guidance and practical insights for the design of future, more capable agent systems.
3. **Open directions for future agent systems.** Building on the design-space analysis and the OpenClaw contrast, Section 12 identifies six open directions spanning the observability-evaluation gap, cross-session persistence, harness boundary evolution, horizon scaling, governance, and the evaluative lens, each drawing on empirical, architectural, and policy literature. Used as an evaluative lens, our study also reveals an open question: while the Claude Code agent system substantially amplifies the short-term capabilities of programmers and end users, it offers limited mechanisms that explicitly support long-term human improvement, deeper understanding, and sustained codebase coherence.

The core agent loop is a while-true cycle with state management. The surrounding subsystems for safety, extensibility, context management, delegation, and persistence make up the bulk of the implementation. Source-level analysis² allows us to identify design choices, subsystem boundaries, and implementation trade-offs directly from the system itself rather than inferring them solely from product descriptions.

Running example. To keep the architecture concrete, we trace the task “Fix the failing test in `auth.test.ts`” through Sections 3 to 9. This example illustrates how a seemingly simple user request activates multiple architectural layers, including tool invocation, permission checks, context selection, iterative repair, delegation, and session persistence.

Paper organization. Section 2 identifies the human values and design principles that motivate the architecture. Section 3 introduces the high-level architecture and the design questions it answers. Sections 4 to 9 each analyze a major subsystem’s design choices. Section 10 contrasts the analysis with OpenClaw, Section 11 provides discussion, and Section 12 surveys open questions for future agent systems. Sections 13 and 14 then cover related work and conclusions. Section B describes the evidence base and methodology.

2 Design Philosophies, Design Principles and Architectural Motivations

Production coding agents are built by humans, for humans, and the architectural decisions they embed reflect what their creators believe matters. This section identifies the human values that motivate Claude Code’s design, traces them through recurring design principles, and frames the design-space questions that organize the analysis in Sections 3 to 9.

Anthropic’s framework for safe agents states a central tension: “Agents must be able to work autonomously; their independent operation is exactly what makes them valuable. But humans should retain control over how their goals are pursued” (Anthropic, 2025a). Claude’s Constitution resolves this not through rigid decision procedures

²Our analysis is grounded primarily in the source code, supplemented by official Anthropic documentation and selected community analysis, Section B details the evidence base and methodology.

but by cultivating “good judgment and sound values that can be applied contextually” (Anthropic, 2026b). These commitments, together with empirical findings about how developers actually use the tool (Huang et al., 2025; McCain et al., 2026), point to five human values that shape the architecture.

2.1 Five Values and Philosophies

Human Decision Authority. The human retains ultimate decision authority over what the system does, organized through a principal hierarchy (Anthropic, then operators, then users) that formalizes who holds authority over what (Anthropic, 2026b). The system is designed so that humans can exercise informed control: they can observe actions in real time, approve or reject proposed operations, interrupt compatible in-progress operations, and audit after the fact. When Anthropic found that users approve 93% of permission prompts (Hughes, 2026), the response was not to add more warnings but to restructure the problem: defined boundaries (sandboxing, auto-mode classifiers) within which the agent can work freely, rather than per-action approvals that users stop reviewing once habituated (Dworken and Weller-Davies, 2025).

Safety, Security, and Privacy. The system protects humans, their code, their data, and their infrastructure from harm, even when the human is inattentive or makes mistakes. This is distinct from Human Decision Authority: where authority is about the human’s *power to choose*, safety is about the system’s *obligation to protect even when that power lapses*. Anthropic’s safe-agents framework separately identifies securing agent interactions and protecting privacy across extended interactions as core commitments (Anthropic, 2025a). The auto-mode threat model (Hughes, 2026) explicitly targets four risk categories: overeager behavior, honest mistakes, prompt injection, and model misalignment.

Reliable Execution. The agent does what the human actually meant, stays coherent over time, and supports verifying its work before declaring success. This value spans both single-turn correctness (did it interpret the request faithfully?) and long-horizon dependability (does it remain coherent across context window boundaries, session resumption, and multi-agent delegation?). Anthropic’s product documentation (Anthropic, 2026d) describes a three-phase loop that the agent repeats until the task is complete: gather context, take action, and verify results. The agent design guidance (Schluntz and Zhang, 2024) further emphasizes that “ground truth from the environment” at each step assesses progress. The harness-design guidance (Rajasekaran, 2026) likewise notes that “agents tend to respond by confidently praising the work,” even when quality is mediocre, motivating separation of generation from evaluation.

Capability Amplification. The system materially increases what the human can accomplish per unit of effort and cost. Anthropic’s internal survey (Huang et al., 2025), discussed in Section 1, suggests that the architecture enables qualitatively new workflows, not merely faster existing ones: approximately 27% of tasks represented work that would not otherwise have been attempted. The system is described by its creators as “a Unix utility rather than a traditional product,” built from the smallest building blocks that are “useful, understandable, and extensible” (Cherny and Wu, 2025). The architecture invests in deterministic infrastructure (context management, tool routing, recovery) rather than decision scaffolding (explicit planners or state graphs), on the premise that increasingly capable models benefit more from a rich operational environment than from frameworks that constrain their choices.

Contextual Adaptability. The system fits the user’s specific context (their project, tools, conventions, and skill level) and the relationship improves over time. The extension architecture (CLAUDE.md, skills, MCP, hooks, plugins) provides configurability at multiple levels of context cost (Sections 6 and 7). Longitudinal data (McCain et al., 2026) shows that the human-agent relationship evolves: auto-approve rates increase from approximately 20% at fewer than 50 sessions to over 40% by 750 sessions. This pattern, described as autonomy that is “co-constructed by the model, the user, and the product,” means the system is designed for trust trajectories rather than fixed trust states. MCP’s donation to the Linux Foundation’s Agentic AI Foundation (The Linux Foundation, 2025) reflects the ecosystem dimension of this value.

Table 1 Design principles, the values they serve, and the design-space question each answers. Principles map to multiple values; implementations appear in the sections indicated.

| Principle | Values Served | Design Question | Sections |
|--|---------------------------------|--|------------|
| Deny-first with human escalation | Authority, Safety | Should unrecognized actions be allowed, blocked, or escalated to the human? | 5, 8, 9 |
| Graduated trust spectrum | Authority, Adaptability | Fixed permission level, or a spectrum users traverse over time? | 5 |
| Defense in depth with layered mechanisms | Safety, Authority, Reliability | Single safety boundary, or multiple overlapping ones using different techniques? | 3, 5 |
| Externalized programmable policy | Safety, Authority, Adaptability | Hardcoded policy, or externalized configs with lifecycle hooks? | 5, 6 |
| Context as scarce resource with progressive management | Reliability, Capability | What is the binding resource constraint, and how to manage it: single-pass truncation or graduated pipeline? | 4, 6, 7, 8 |
| Append-only durable state | Reliability, Authority | Mutable state, checkpoint snapshots, or append-only logs? | 4, 9 |
| Minimal scaffolding, maximal operational harness | Capability, Reliability | Invest in scaffolding-side reasoning, or operational infrastructure that lets the model reason freely? | 3, 4 |
| Values over rules | Capability, Authority | Rigid decision procedures, or contextual judgment backed by deterministic guardrails? | 3, 5, 7 |
| Composable multi-mechanism extensibility | Capability, Adaptability | One unified extension API, or layered mechanisms at different context costs? | 6 |
| Reversibility-weighted risk assessment | Capability, Safety | Same oversight for all actions, or lighter for reversible and read-only ones? | 4, 5, 8 |
| Transparent file-based configuration and memory | Adaptability, Authority | Opaque database, embedding-based retrieval, or user-visible version-controllable files? | 7 |
| Isolated subagent boundaries | Reliability, Safety, Capability | Subagents share the parent’s context and permissions, or operate in isolation? | 8 |
| Graceful recovery and resilience | Reliability, Capability | Fail hard on errors, or silently recover and reserve human attention for unrecoverable situations? | 4, 5 |

2.2 Design Principles

These values are operationalized through thirteen design principles, each answering a recurring question that production coding agents must resolve. Table 1 summarizes the principles; subsequent sections (Section 3–Section 9) trace each through specific implementation choices.

These principles can be read against three major alternative design families. First, *rule-based orchestration*: frameworks such as LangGraph (LangChain, Inc., 2024) encode decision logic as explicit state graphs with typed edges, choosing scaffolding over minimal harness. Second, *container-isolated execution*: SWE-Agent and OpenHands (Yang et al., 2024; Wang et al., 2024b) rely on Docker isolation rather than layered policy enforcement. Third, *version-control-as-safety*: tools like Aider (Gauthier, 2024) use Git rollback as the primary safety mechanism rather than deny-first evaluation. Claude Code’s principle set is distinctive in combining minimal decision scaffolding with layered policy enforcement, values-based judgment with deny-first defaults, and progressive context management with composable extensibility.

2.3 From Values to Architecture

Each value traces through its principles to specific architectural decisions:

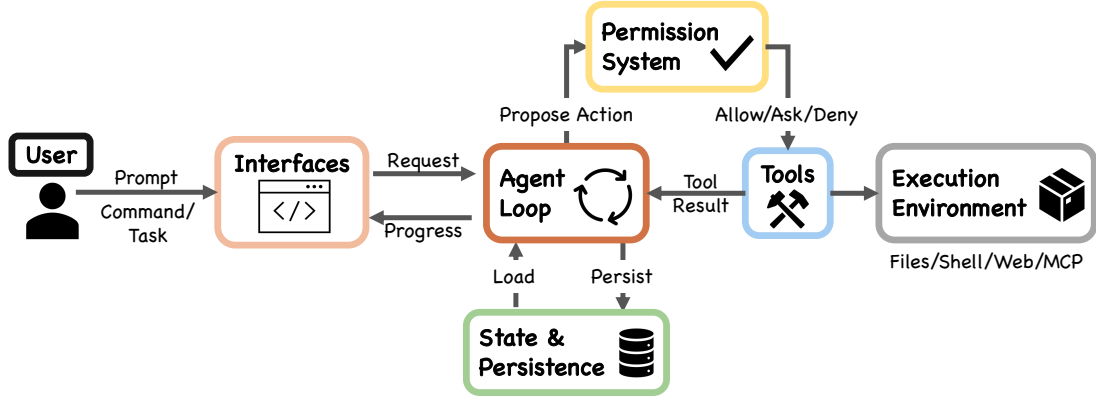


Figure 1 High-level system structure of Claude Code. The system decomposes into seven functional components: user, interfaces, the agent loop, a permission system, tools, state & persistence, and an execution environment. All entry surfaces converge on the same agent loop.

- **Human Decision Authority** motivates deny-first evaluation, the graduated trust spectrum, append-only state (auditable history), externalized programmable policy, and values-over-rules (Sections 5 to 7 and 9).
- **Safety, Security, and Privacy** motivates defense in depth, deny-first defaults, reversibility-weighted assessment, externalized policy, and isolated subagent boundaries (Sections 5 and 8).
- **Reliable Execution** motivates context-as-scarce-resource, append-only durable state, graceful recovery, isolated subagent boundaries, and defense in depth (Sections 4 and 7 to 9).
- **Capability Amplification** motivates minimal scaffolding, composable extensibility, reversibility-weighted risk, context management, and graceful recovery (Sections 4 to 6).
- **Contextual Adaptability** motivates transparent file-based memory, composable extensibility, the graduated trust spectrum, and externalized programmable policy (Sections 5 to 7).

These mappings also reveal what the architecture does *not* do: it does not impose explicit planning graphs on the model’s reasoning, does not provide a single unified extension mechanism, and does not restore all session-scoped trust-related state across resume. These absences are consistent with the principle set above.

2.4 An Evaluative Lens: Long-term Capability Preservation

The five values above describe what the architecture is designed to serve. This paper also applies a sixth concern, whether the architecture preserves long-term human capability, as an evaluative lens. This concern is real: Anthropic’s own study of 132 engineers and researchers (Huang et al., 2025) documents a “paradox of supervision” in which overreliance on AI risks atrophying the skills needed to supervise it, and independent research (Shen and Tamkin, 2026) finds that developers in AI-assisted conditions score 17% lower on comprehension tests. However, this concern is not prominently reflected as a design driver in the architecture or in Anthropic’s stated design values. We therefore treat it not as a co-equal value but as a cross-cutting concern: a question applied across all five values in Section 11, asking whether short-term amplification comes at the cost of long-term human understanding, codebase coherence, and the developer pipeline.

3 Architecture Overview

Building a production coding agent requires answering several recurring design questions: where should reasoning live, how many execution engines are needed, what safety posture to adopt, and what resource to treat as the binding constraint. Claude Code’s architecture can be read as one set of answers to these questions. At the implementation level, the system has seven components connected by a main data flow: a user submits a prompt through one of several interfaces, which feeds into a shared agent loop. The agent loop assembles context, calls the Claude model, receives responses that may include tool-use requests, routes those requests through a permission system, and dispatches approved actions to concrete tools that interact with the execution environment. Throughout this process, state and persistence mechanisms record the conversation

transcript, manage session identity, and support resume, fork, and rewind operations.

3.1 Design Questions and Running Example

The description is organized around four design questions that recur across production coding agents, each grounding one or more of the design principles identified in [Table 1](#). Each question is introduced here with Claude Code’s answer, a note on plausible alternatives, and then demonstrated progressively through [Sections 4 to 9](#).

Where does reasoning live? In Claude Code, the model reasons about what to do; the harness is responsible for executing actions. The model emits `tool_use` blocks as part of its response, and the harness parses them, checks permissions, dispatches them to tool implementations, and collects results (`query.ts`). The model never directly accesses the filesystem, runs shell commands, or makes network requests. This separation has a security consequence: because reasoning and enforcement occupy separate code paths, a compromised or adversarially manipulated model cannot override the sandboxing, permission checks, or deny-first rules implemented in the harness. The model’s only interface to the outside world is the structured `tool_use` protocol, which the harness validates before execution. Community analysis of the extracted source estimates that only about 1.6% of Claude Code’s codebase constitutes AI decision logic, with the remaining 98.4% being operational infrastructure, a ratio that illustrates how thin the core agent reasoning layer is. Alternative designs invest more heavily in scaffolding-side reasoning: Devin maintains explicit planning and task-tracking structures, while LangGraph ([LangChain, Inc., 2024](#)) routes control flow through developer-defined state graphs.

How many execution engines? Claude Code uses a single `queryLoop()` function that executes regardless of whether the user is interacting through an interactive terminal, a headless CLI invocation, the Agent SDK, or an IDE integration (`query.ts`). Only the rendering and user-interaction layer varies. Other systems use mode-specific engines: for example, an IDE integration may follow a different code path than a CLI tool, trading uniformity for surface-specific optimization.

What is the default safety posture? Claude Code’s default safety posture is deny-first with human escalation: deny rules override allow rules, and unrecognized actions are escalated to the user rather than allowed silently (`permissions.ts`). Multiple independent safety layers (permission rules, `PreToolUse` hooks, the auto-mode classifier when enabled, and optional shell sandboxing) apply in parallel, so any one can block an action ([Section 5](#)). This combines the *deny-first with human escalation* and *defense in depth with layered mechanisms* principles from [Table 1](#). Alternative approaches shift the trust boundary elsewhere: SWE-Agent and OpenHands ([Yang et al., 2024](#); [Wang et al., 2024b](#)) rely on container-based isolation to contain arbitrary execution, while Aider ([Gauthier, 2024](#)) uses git-based rollback as its primary safety net.

What is the binding resource constraint? In Claude Code, the context window (200K for older models, 1M for the Claude 4.6 series) is the binding resource constraint. Five distinct context-reduction strategies execute before every model call (`query.ts`), and several other subsystem decisions (lazy loading of instructions, deferred tool schemas, summary-only subagent returns) exist to limit context consumption ([Section 7](#)). The five-layer pipeline exists because no single compaction strategy addresses all types of context pressure. Budget reduction targets individual tool outputs that overflow size limits. Snip handles temporal depth. Microcompact reacts to cache overhead. Context collapse manages very long histories. Auto-compact performs semantic compression as a last resort. Each layer operates at a different cost-benefit tradeoff, and earlier, cheaper layers run before costlier ones. Alternative architectures treat other resources as the primary bottleneck, for instance compute budget (limiting the number of model calls or tool invocations) or working memory (maintaining an explicit scratchpad rather than relying on the conversation history).

Running example. To ground these principles, we thread a single task through [Sections 3 to 9](#): “*Fix the failing test in `auth.test.ts`.*” In this section the user submits the prompt through one of Claude Code’s interfaces. Subsequent sections trace the request through the query loop, permission gate, tool pool, context window, subagent delegation, and session persistence.

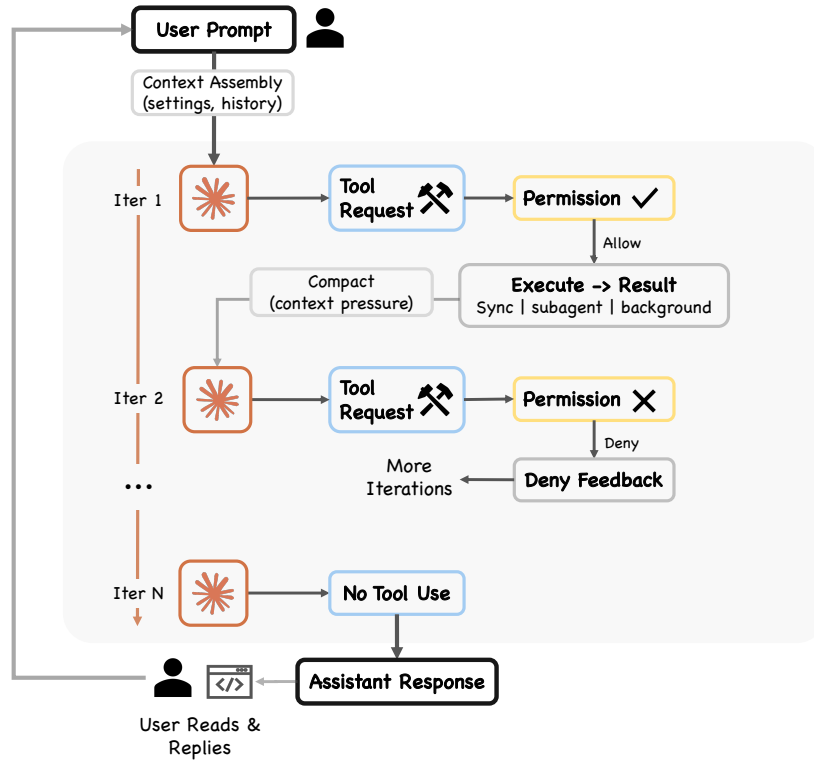


Figure 2 Runtime turn flow showing the end-to-end execution of a single agentic turn: user prompt enters through context assembly, the model is called, tool requests pass through the permission gate, tool results feed back into the loop, and compaction manages context pressure.

3.2 High-Level System Structure

The seven-component model (Figure 1) maps directly to source files:

1. **User:** Submits prompts, approves permissions, reviews output.
2. **Interfaces:** Interactive CLI, headless CLI (`claude -p`), Agent SDK, and IDE/Desktop/Browser. All surfaces feed the same loop.
3. **Agent loop:** The iterative cycle of model call, tool dispatch, and result collection, implemented as the `queryLoop()` async generator in `query.ts`.
4. **Permission system:** Deny-first rule evaluation (`permissions.ts`), the auto-mode ML classifier, and hook-based interception (`types/hooks.ts`).
5. **Tools:** Up to 54 built-in tools (19 unconditional, 35 conditional on feature flags and user type) assembled via `assembleToolPool()` (`tools.ts`), merged with MCP-provided tools. Plugins contribute indirectly through MCP servers and the skill/command registry.
6. **State & persistence:** Mostly append-only JSONL session transcripts (`sessionStorage.ts`), global prompt history (`history.ts`), and subagent sidechain files.
7. **Execution environment:** Shell execution with optional sandboxing (`shouldUseSandbox.ts`), filesystem operations, web fetching, MCP server connections, and remote execution.

The data flow follows a left-to-right spine: the user submits a request through an interface, which enters the agent loop. The loop proposes actions to the permission system; approved actions reach tools, which interact with the execution environment and return `tool_result` messages back to the loop. State and persistence sit alongside the loop, recording transcripts and loading prior session data.

The application entry point `main()` in `main.tsx` initializes security settings (including `NoDefaultCurrentDirectoryInExePath` to prevent Windows PATH hijacking), registers signal handlers for graceful shutdown, and dispatches to the appropriate execution mode.

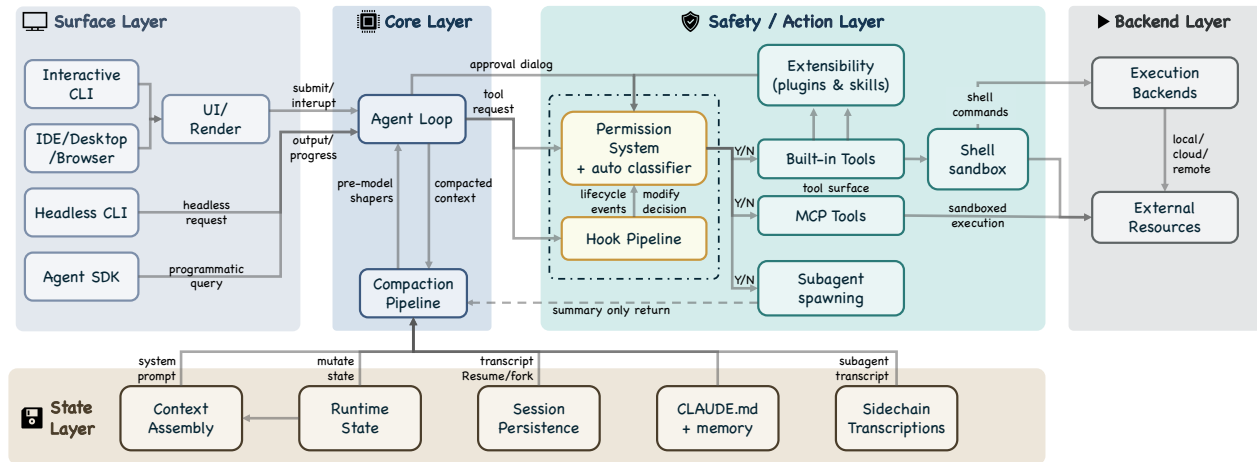


Figure 3 Expanded layered architecture showing five subsystem layers: surface (Interactive CLI, Headless CLI, Agent SDK, IDE/Desktop/Browser, UI/renderer), core (agent loop, compaction pipeline), safety/action (permission system incl. auto-mode classifier, hook pipeline, extensibility, built-in tools, MCP tools, shell sandbox, subagent spawning), state (context assembly, runtime state, session persistence, CLAUDE.md + memory, sidechain transcriptions), and backend (execution backends, external resources).

3.3 Layered Subsystem Decomposition

The five-layer decomposition (Figure 3) expands the seven-component model into a finer-grained view, mapping each layer to specific source directories.

Surface layer (entry points and rendering). The `src/entrypoints/` directory contains startup paths, including the SDK entry with `coreTypes.ts`, `controlSchemas.ts`, and `coreSchemas.ts`. The `src/screens/` directory composes full-screen layouts, and `src/components/` provides terminal UI building blocks via the ink framework. The interactive CLI launches a terminal UI with real-time streaming, permission dialogs, and progress indicators. The headless CLI (`claude -p`) creates a `QueryEngine` instance for single-shot processing. The Agent SDK emits typed events via async generators.

Core layer (agent loop, compaction pipeline). The `queryLoop()` async generator (`query.ts`) implements the iterative agent loop, consuming assembled context from the state layer and dispatching tool requests to the safety/action layer. Before every model call, a *compaction pipeline* of five sequential shapers (`query.ts:365–453`) manages context pressure: budget reduction, snip, microcompact, context collapse, and auto-compact (Sections 4.3 and 7.3).

Safety/action layer (permission system, hooks, extensibility, tools, sandbox, subagents). The *permission system* (`permissions.ts`) implements deny-first rule evaluation with up to seven permission modes (if also counting internal-only `bubble` and feature-gated `auto`) (`types/permissions.ts`) and an integrated *auto-mode ML classifier* (`yoloClassifier.ts`) that provides a two-stage fast-filter and chain-of-thought evaluation of tool safety (Section 5). A *hook pipeline* spanning 27 event types (`coreTypes.ts`; output schemas in `types/hooks.ts`) can block, rewrite, or annotate tool requests; of these, 5 are safety-related while the remaining 22 serve lifecycle and orchestration purposes (Section 6). An *extensibility* subsystem allows plugins and skills to register tools and hooks into the runtime. Tool pool assembly via `assembleToolPool()` (`tools.ts`) merges built-in and MCP-provided tools. Approved shell commands pass through a *shell sandbox* (`shouldUseSandbox.ts`) that restricts filesystem and network access independently of the permission system. *Subagent spawning* via `AgentTool` (`AgentTool.tsx`, `runAgent.ts`) is dispatched through the same `buildTool()` factory as all other tools, re-entering the `queryLoop()` with an isolated context window and returning only a summary to the parent (Section 8).

State layer (context assembly, runtime state, persistence, memory, sidechains). *Context assembly* is a memoized state loader, not a routing hub: `getSystemContext()` (`context.ts`) computes session-level system context including git status, and `getUserContext()` (`context.ts`) loads the CLAUDE.md hierarchy and current date. Both are cached for reuse: system context is appended to the system prompt, while user context is added as a user-context message. The `src/state/` directory manages runtime application state. Session transcripts are stored as mostly append-only JSONL files at project-specific paths (`sessionStorage.ts`). The *CLAUDE.md + memory* subsystem provides a four-level instruction hierarchy (`claudemd.ts`) from managed settings to directory-specific files, plus auto-memory entries that Claude writes during conversations (Section 7.2). *Sidechain transcripts* (`sessionStorage.ts:247`) store each subagent's conversation in a separate file, preventing subagent content from inflating the parent context (Section 8.3). Global prompt history is maintained in `history.jsonl` (`history.ts`). Resume and fork operations reconstruct session state from transcripts (`conversationRecovery.ts`).

Backend layer (execution backends, external resources). Shell command execution with optional sandboxing (`BashTool.tsx`, `PowerShellTool.tsx`), remote execution support (`src/remote/`), MCP server connections across multiple transport variants including stdio, SSE, HTTP, WebSocket, SDK, and IDE-specific adapters (`services/mcp/client.ts`), and 42 tool subdirectories in `src/tools/` implement concrete tool logic.

3.4 QueryEngine: A Clarification

The class documentation at `QueryEngine.ts` states: “QueryEngine owns the query lifecycle and session state for a conversation. It extracts the core logic from `ask()` into a standalone class that can be used by both the headless/SDK path and (in a future phase) the REPL.” The class is a *conversation wrapper* for non-interactive surfaces, not the engine itself. Its constructor accepts a `QueryEngineConfig` with initial messages, an abort controller, a file-state cache, and other per-conversation state. Its `submitMessage()` method is an async generator that orchestrates a single turn. The shared query path lives in `query()` (`query.ts`), which wraps an internal `queryLoop()`; `QueryEngine` delegates to `query()`.

This distinction matters architecturally: the interactive CLI also calls `query()`, bypassing `QueryEngine` entirely. The shared code path is the loop function, not the engine class.

3.5 Permission and Safety Layers

The safety-by-default principle is implemented through seven independent layers. A request must pass through all applicable layers, and any single layer can block it:

1. **Tool pre-filtering** (`tools.ts`): Blanket-denied tools are removed from the model's view before any call, preventing the model from attempting to invoke them.
2. **Deny-first rule evaluation** (`permissions.ts`): Deny rules always take precedence over allow rules, even when the allow rule is more specific.
3. **Permission mode constraints** (`types/permissions.ts`): The active mode determines baseline handling for requests matching no explicit rule.
4. **Auto-mode classifier**: An ML-based classifier evaluates tool safety, potentially denying requests the rule system would allow.
5. **Shell sandboxing** (`shouldUseSandbox.ts`): Approved shell commands may still execute inside a sandbox restricting filesystem and network access.
6. **Not restoring permissions on resume** (`conversationRecovery.ts`): Session-scoped permissions are not restored on resume or fork.
7. **Hook-based interception** (`types/hooks.ts`): `PreToolUse` hooks can modify permission decisions; `PermissionRequest` hooks can resolve decisions asynchronously alongside the user dialog (or before it, in coordinator mode).

These layers are described in detail in Section 5.

3.6 Context as Bottleneck: Beyond Compaction

Beyond the five-layer compaction pipeline (detailed in [Section 7](#)), several other subsystem decisions reflect the context-as-bottleneck constraint:

- **CLAUDE.md lazy loading:** The base CLAUDE.md hierarchy is loaded at session start, but additional nested-directory instruction files and conditional rules are loaded only when the agent reads files in those directories, preventing unused instructions from consuming context.
- **Deferred tool schemas:** When ToolSearch is enabled, some tools include only their names in the initial context; full schemas are loaded on demand.
- **Subagent summary-only return:** Subagents return only summary text to the parent, not their full conversation history ([Section 8](#)).
- **Per-tool-result budget:** Individual tool results are capped at a configurable size, preventing a single verbose output from consuming disproportionate context.

4 Turn Execution: The Agentic Query Loop

When the user submits “Fix the failing test in `auth.test.ts`,” the input enters a reactive loop, one of several possible orchestration patterns for coding agents. This section examines Claude Code’s choice of a simple while-loop architecture and traces one turn of that loop end-to-end, illustrating three design principles from [Table 1](#): *minimal scaffolding with maximal operational harness*, *context as scarce resource with progressive management*, and *graceful recovery and resilience*.

4.1 The Query Pipeline

Each turn follows a fixed sequence ([Figure 2](#), `query.ts`):

1. **Settings resolution.** The `queryLoop()` function destructures immutable parameters including the system prompt, user context, permission callback, and model configuration.
2. **Mutable state initialization.** A single `State` object stores all mutable state across iterations, including messages, tool context, compaction tracking, and recovery counters. The loop’s seven `continue` points (the “continue sites”) each overwrite this object in one whole-object assignment rather than mutating fields individually.
3. **Context assembly.** The function `getMessagesAfterCompactBoundary()` retrieves messages from the last compact boundary forward, ensuring that compacted content is represented by its summary rather than the original messages.
4. **Pre-model context shapers.** Five shapers execute sequentially ([Section 4.3](#)).
5. **Model call.** A `for await` loop over `deps.callModel()` streams the model’s response, passing assembled messages (with user context prepended), the full system prompt, thinking configuration, the available tool set, an abort signal, the current model specification, and additional options including fast-mode settings, effort value, and fallback model.
6. **Tool-use dispatch.** If the response contains `tool_use` blocks, they flow to the tool orchestration layer ([Section 4.2](#)).
7. **Permission gate.** Each tool request passes through the permission system ([Section 5](#)).
8. **Tool execution and result collection.** Tool results are added to the conversation as `tool_result` messages, and the loop continues.
9. **Stop condition.** If the response contains no `tool_use` blocks (text only), the turn is complete.

The `queryLoop()` function is defined as an `AsyncGenerator`, yielding `StreamEvent`, `RequestStartEvent`, `Message`, `TombstoneMessage`, and `ToolUseSummaryMessage` events as it progresses. This generator-based design enables streaming output to the UI layer while maintaining a single synchronous control flow within the loop.

Claude Code’s reactive loop follows the ReAct pattern ([Yao et al., 2022](#)): the model generates reasoning and tool invocations, the harness executes actions, and results feed the next iteration. Alternative orchestration patterns include explicit graph-based routing ([LangChain, Inc., 2024](#)), where control flow is defined as a state machine with typed edges, and tree-search methods ([Zhou et al., 2023](#)) that explore multiple action trajectories

before committing. Anthropic’s own documentation (Schluntz and Zhang, 2024) identifies five composable workflow patterns (prompt chaining, routing, parallelization, orchestrator-workers, and evaluator-optimizer) of which Claude Code primarily uses the orchestrator-workers pattern for subagent delegation (Section 8) while keeping the core loop reactive. The reactive design trades search completeness for simplicity and latency: each turn commits to one action sequence without backtracking.

4.2 Tool Dispatch and Streaming Execution

When the model response contains `tool_use` blocks, the system chooses between two execution paths. The primary path uses `StreamingToolExecutor`, which begins executing tools as they stream in from the model response, reducing latency for multi-tool responses. The fallback path uses `runTools()` in `toolOrchestration.ts`, which iterates over partitions produced by `partitionToolCalls()`. Both paths classify tools as concurrent-safe or exclusive. Read-only operations can execute in parallel, while state-modifying operations like shell commands are serialized.

The `StreamingToolExecutor` (`StreamingToolExecutor.ts`) manages concurrent execution with two coordination mechanisms:

- **Sibling abort controller.** Fires when any Bash tool errors, immediately terminating other in-flight subprocesses rather than letting them run to completion.
- **Progress-available signal.** Wakes up the `getRemainingResults()` consumer when new output is ready.

Results are buffered and emitted in the order tools were received, so output order stays the same even when tools run in parallel. This is important because the model expects tool results in the same order as its tool-use requests. This concurrent-read, serial-write execution model occupies a middle ground between fully serial dispatch and more aggressive speculative approaches such as PASTE (Sui et al., 2026), which speculatively pre-executes predicted future tool calls while the model is still generating, hiding tool latency through speculation.

The tool result collection phase iterates over updates from either the streaming executor or the synchronous `runTools()` generator. Each update may carry a tool result, an attachment, or a progress event. A special check detects `hook_stopped_continuation` attachments: if a `PostToolUse` hook signals that the turn should not continue, a `shouldPreventContinuation` flag is set. Results are normalized for the Anthropic API via `normalizeMessagesForAPI()`, filtering to keep only user-type messages.

4.3 Pre-Model Context Shapers

Five context shapers execute sequentially in `query.ts` before every model call, each operating on the `messagesForQuery` array. The five shapers run in sequence, with earlier steps applying lighter reductions before later steps apply broader compaction.

Budget reduction. (`applyToolResultBudget()`). Enforces per-message size limits on tool results, replacing oversized outputs with content references. Exempt tools (those where `maxResultSizeChars` is not finite) retain their full output. Content replacements are persisted for agent and session query sources to enable reconstruction on resume. Budget reduction runs before `microcompact` because `microcompact` operates purely by `tool_use_id` and never inspects content; the two compose cleanly.

Snip. (`snipCompactIfNeeded()`, gated by `HISTORY_SNIP`). A lightweight trim that removes older history segments, returning `{messages, tokensFreed, boundaryMessage}`. The `snipTokensFreed` value is plumbed to auto-compact because the main token counter derives context size from the `usage` field on the most recent assistant message, and that message survives snip with its pre-snip `input_tokens` still attached; snip’s savings are therefore invisible to the counter unless passed through explicitly.

Microcompact. Fine-grained compression that always runs a time-based path and optionally a cache-aware path (gated by `CACHED_MICROCOMPACT`). When the cached path is enabled, boundary messages are deferred until after the API response so they can use actual `cache_deleted_input_tokens` rather than estimates. Returns `{messages, compactionInfo}` where `compactionInfo` may include `pendingCacheEdits`.

Context collapse. Gated by `CONTEXT_COLLAPSE`. A read-time projection over the conversation history. The source comments explain: “Nothing is yielded; the collapsed view is a read-time projection over the REPL’s full history. Summary messages live in the collapse store, not the REPL array. This is what makes collapses persist across turns.” Unlike the other shapers, context collapse does not mutate the REPL’s stored history; it replaces the `messagesForQuery` array with a projected view via `applyCollapsesIfNeeded()`, so the model sees the collapsed version while the full history remains available for reconstruction.

Auto-compact. The fifth shaper, triggering a full model-generated summary via `compactConversation()` in `compact.ts`. This function executes `PreCompact` hooks, creates a summary request using `getCompactPrompt()`, and calls the model to produce a compressed summary. The result feeds into `buildPostCompactMessages()` (`compact.ts`). Auto-compact fires only when the context still exceeds the pressure threshold after all four previous shapers have run.

4.4 Recovery Mechanisms

The query loop implements several recovery mechanisms for edge cases:

- **Max output tokens escalation:** When the response hits the output token cap, the system can retry with an escalated limit, subject to a `GrowthBook` flag and the absence of an existing override or environment-variable cap. Up to three recovery attempts are allowed per turn (`MAX_OUTPUT_TOKENS_RECOVERY_LIMIT = 3`).
- **Reactive compaction** (gated by `REACTIVE_COMPACT`): When the context is near capacity, reactive compact summarizes just enough to free space. The `hasAttemptedReactiveCompact` flag ensures this fires at most once per turn.
- **Prompt-too-long handling:** If the API returns a `prompt_too_long` error, the loop first attempts context-collapse overflow recovery and reactive compaction. Only after these fail does it terminate with `reason: 'prompt_too_long'`.
- **Streaming fallback:** The `onStreamingFallback` callback handles streaming API issues, allowing the loop to retry with a different strategy.
- **Fallback model:** The `fallbackModel` parameter enables switching to an alternative model if the primary model fails.

4.5 Stop Conditions

Multiple conditions can terminate the loop:

1. **No tool use:** The model produces only text content (the primary stop condition).
2. **Max turns:** The configurable `maxTurns` limit is reached.
3. **Context overflow:** The API returns `prompt_too_long`.
4. **Hook intervention:** A `PostToolUse` hook sets `hook_stopped_continuation`.
5. **Explicit abort:** The `abortController` signal fires.

The turn pipeline determines *how* tool requests are orchestrated and recovered. The next section examines the gate that determines *whether* each request is permitted to execute at all.

5 Tool Authorization and Control Boundaries

Production coding agents adopt different safety architectures: layered policy enforcement, OS-level sandboxing, or version-control-based rollback. Claude Code combines the first two, implementing four design principles from [Table 1](#): *deny-first with human escalation*, *graduated trust spectrum*, *defense in depth with layered mechanisms*, and *reversibility-weighted risk assessment*.

When Claude decides to execute a tool (for example, running `npm test` via `BashTool` to reproduce the auth test failure), the request enters the permission pipeline shown in [Figure 4](#). Every tool invocation passes through the permission system, and the default behavior is to deny or ask rather than allow silently. This default is motivated by a documented behavioral pattern: Anthropic’s auto-mode analysis ([Hughes, 2026](#))

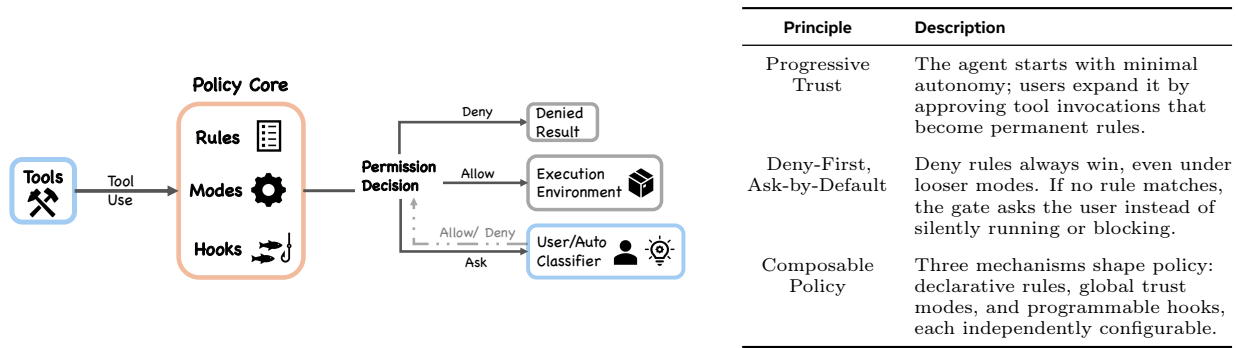


Figure 4 Permission gate overview and design principles.

found that users approve approximately 93% of permission prompts, indicating that approval fatigue renders interactive confirmation behaviorally unreliable as a sole safety mechanism. Because users habitually approve without careful review, the system must maintain safety independently of human vigilance. This motivates the architectural commitment to deny-first evaluation, blanket-deny pre-filtering, and sandboxing as independent layers that operate regardless of user attentiveness.

5.1 Permission Modes and Rule Evaluation

Seven permission modes exist across the type definitions (5 external modes at `types/permissions.ts`; auto added conditionally; bubble in the type union):

1. **plan**: The model must create a plan; execution proceeds only after user approval.
2. **default**: Standard interactive use. Most operations require user approval.
3. **acceptEdits**: Edits within the working directory and certain filesystem shell commands (`mkdir`, `rmdir`, `touch`, `rm`, `mv`, `cp`, `sed`) are auto-approved; other shell commands require approval.
4. **auto**: An ML-based classifier evaluates requests that do not pass fast-path checks (gated by `TRANSCRIPT_CLASSIFIER`).
5. **dontAsk**: No prompting, but deny rules are still enforced.
6. **bypassPermissions**: Skips most permission prompts, but safety-critical checks and bypass-immune rules still apply.
7. **bubble**: Internal-only mode for subagent permission escalation to the parent terminal.

The five externally visible modes (**acceptEdits**, **bypassPermissions**, **default**, **dontAsk**, **plan**) are defined in the `EXTERNAL_PERMISSION_MODES` array. The **auto** mode is conditionally included only when the `TRANSCRIPT_CLASSIFIER` feature flag is active. The **bubble** mode exists in the type union but not in either mode array; it is used internally for subagent permission escalation (Section 8).

Permission rules are evaluated in deny-first order (`permissions.ts`). The `toolMatchesRule()` function checks deny rules first: a deny rule always takes precedence over an allow rule, even when the allow rule is more specific. A broad deny (“deny all shell commands”) cannot be overridden by a narrow allow (“allow `npm test`”). The rule system supports tool-level matching (by tool name) and content-level matching (matching specific tool input patterns, such as `Bash(prefix: npm)`).

The seven modes span a graduated autonomy spectrum, from **plan** (user approves all plans before execution) through **default** and **acceptEdits** to **bypassPermissions** (minimal prompting). This gradient reflects a recurring design tension: as autonomy increases, the system must shift from interactive approval to automated safety checks. Other agent systems resolve this tension differently. SWE-Agent and OpenHands (Yang et al., 2024; Wang et al., 2024b) use Docker container isolation, sandboxing the agent’s entire execution environment rather than evaluating individual tool invocations. Aider (Gauthier, 2024) relies on Git as a safety net, making all changes reversible through version control. Claude Code’s approach layers multiple policy-enforcement mechanisms on top of optional container sandboxing, trading simplicity for fine-grained control over individual actions.

5.2 The Authorization Pipeline

The full authorization pipeline proceeds through several stages:

Pre-filtering. Before any tool request reaches runtime evaluation, `filterToolsByDenyRules()` (`tools.ts`) strips blanket-denied tools from the model’s view entirely at tool pool assembly time. The documentation states: “Uses the same matcher as the runtime permission check, so MCP server-prefix rules like `mcp__server` strip all tools from that server before the model sees them.” This prevents the model from attempting to invoke forbidden tools, so the model does not waste calls on them.

PreToolUse hook. Registered hooks fire as part of the permission pipeline. A `PreToolUse` hook can return a `permissionDecision` to deny or ask, or an `updatedInput` that modifies the tool’s input parameters (`types/hooks.ts`). A hook `allow` does not bypass subsequent rule-based denials or safety checks. In the interactive path, the user dialog is queued first and hooks run asynchronously; coordinator and similar background-agent paths await automated checks before showing the dialog.

Rule evaluation. The deny-first rule engine evaluates the request. MCP tools are matched by their fully qualified `mcp__server__tool` name, and server-level rules match all tools from that server.

Permission handler. The handler in `useCanUseTool.tsx` branches into one of four paths based on runtime context:

1. **Coordinator:** For multi-agent coordination mode. Attempts automated resolution (classifier, hooks, rules) before falling back to user interaction.
2. **Swarm worker:** Handles worker agents in a multi-agent swarm with their own resolution logic.
3. **Speculative classifier:** When `BASH_CLASSIFIER` is enabled and the tool is `BashTool`, a speculative classifier races a pre-started classification result against a timeout. If the classifier returns with high confidence, the tool is approved instantly without user interaction.
4. **Interactive:** The fallback path. Presents the standard user approval dialog through the terminal UI.

In coordinator and some background paths, automated resolution is attempted before user interaction. In the standard interactive path, the dialog can appear first while hooks or classifier checks continue in parallel. When the classifier or a deny rule blocks an action, the system treats the denial as a routing signal rather than a hard stop: the model receives the denial reason, revises its approach, and attempts a safer alternative in the next loop iteration. The `PermissionDenied` hook event ([Section 6](#)) enables external code to observe and respond to these denials programmatically. This recovery-oriented design means that permission enforcement shapes the agent’s behavior rather than simply halting it.

5.3 Auto-Mode Classifier and Hook Lifecycle

The auto-mode classifier (`yoloClassifier.ts`) participates in permission decisions when enabled. When `TRANSCRIPT_CLASSIFIER` is enabled, the classifier loads three prompt resources:

- A base system prompt.
- An external permissions template.
- For Anthropic-internal users, a separate internal template.

The classifier evaluates the proposed tool invocation against the conversation transcript and the permission template, producing an `allow`, `deny`, or `request for manual approval`. The function `isUsingExternalPermissions()` checks `USER_TYPE` and a `forceExternalPermissions` config flag to select the appropriate template.

Of the 27 hook events defined in the source (`coreTypes.ts`), five participate directly in the permission flow, each with a specific Zod-validated output schema (`types/hooks.ts`):

- **PreToolUse:** Can return `permissionDecision` (deny or ask, but `allow` does not bypass subsequent checks), `permissionDecisionReason`, and `updatedInput` (modify parameters).
- **PostToolUse:** Can inject additional `Context` and, for MCP tools, return `updatedMCPToolOutput` to modify results before they enter the context.

- **PostToolUseFailure:** Can inject `additionalContext` for error-specific guidance.
- **PermissionDenied:** Can provide `retry` guidance after auto-mode denials.
- **PermissionRequest:** Can return a `decision` of `allow` or `deny`. In coordinator and similar paths, this can resolve before the user dialog. In the standard interactive path, it can also run alongside the dialog.

For non-MCP tools, the `tool_result` is emitted before the `PostToolUse` hook fires. For MCP tools, the result is delayed until after post hooks have run, enabling `updatedMCPToolOutput` to take effect.

5.4 Shell Sandboxing

Shell sandboxing provides an additional layer of protection for Bash and PowerShell commands (`shouldUseSandbox.ts`). The `shouldUseSandbox()` function checks whether sandboxing is globally enabled, whether the invocation has opted out, and whether the command matches any exclusion patterns.

When active, the sandbox provides filesystem and network isolation independent of the application-level permission model. A command can be permission-approved but still sandboxed, or permission-denied and never reach the sandbox check. The two systems operate on different axes: authorization versus isolation.

The layered safety architecture rests on an independence assumption: if one layer fails, others catch the violation. However, several layers share common performance constraints. Security researchers ([Adversa.ai, 2026](#)) have documented that commands with more than 50 subcommands fall back to a single generic approval prompt instead of running per-subcommand deny-rule checks, because per-subcommand parsing caused UI freezes. This example demonstrates that defense-in-depth can degrade when its layers share failure modes, a structural tension between safety and performance analyzed further in [Section 11.3](#).

The permission pipeline governs whether a tool request executes. The next section examines what determines which tools exist in the first place: the extensibility architecture that assembles the model’s action surface.

6 Extensibility: MCP, Plugins, Skills, and Hooks

A recurring design question for coding agents is how to structure the extension surface: a single unified mechanism, a small number of specialized mechanisms, or a layered stack with different context costs. The analysis here illustrates two design principles from [Table 1](#): *composable multi-mechanism extensibility* and *externalized programmable policy*. Returning to the running example, once Claude is trying to repair `auth.test.ts` and the earlier `npm test` request has been mediated by the permission system ([Section 5](#)), the next question is what extension-enabled action surface is available for the repair. When a turn begins in Claude Code, the model sees not just built-in tools like `BashTool` and `FileReadTool`, but also database query tools from an MCP server, a custom lint skill from `.claude/skills/`, and tools contributed by an installed plugin. These arrive through four mechanisms that extend the agent at different points of the loop: MCP servers provide external tool integration, plugins package and distribute bundles of components, skills inject domain-specific instructions, and hooks intercept the tool execution lifecycle. Anthropic’s documentation ([Anthropic, 2026d](#)) presents a broader view that includes `CLAUDE.md` ([Section 7](#)) and subagents ([Section 8](#)) alongside the four mechanisms analyzed here. We treat `CLAUDE.md` and subagents in their own sections because they operate in different subsystems (context construction and delegation, respectively), but the context-cost ordering is architecturally significant: it reveals how each extension point trades off expressiveness against the bounded context window.

6.1 Four Extension Mechanisms

The mechanisms are implemented in distinct source directories ([Figure 5](#)) and serve different integration patterns:

MCP servers. The Model Context Protocol is the primary external tool integration path. MCP servers are configured from multiple scopes: project, user, local, and enterprise, with additional plugin and `claude.ai` servers merged at runtime (`services/mcp/config.ts`). The MCP client (`services/mcp/client.ts`) supports multiple transport types: stdio, SSE, HTTP, WebSocket, SDK, plus IDE-specific variants (`sse-ide`, `ws-ide`) and

```

# one turn of Claude Code's agent loop
while not stopped:
    # (a) assemble -- build what the model sees
    context = assemble(
        system_prompt,    # instructions header
        tool_schemas,    # callable tool signatures
        history,          # prior turn messages
        hook_additions,   # pushed in by hooks
    )
    # (b) model -- pick the next action
    action = model(context, tools) # flat tool pool
    if action.is_text_only():
        stopped = run_stop_hooks(action) # may veto
        continue
    # (c) execute -- gate and run the tool call
    if not permitted(action):         # permission
        continue
    action = run_pre_tool_hooks(action) # block/rewrite
    result = execute(action)          # tool runs here
    result = run_post_tool_hooks(result) # mutate/annotate
    history.append(action, result)

```

(a) assemble(): what the model sees

| Element | What it does |
|-------------------------|---|
| CLAUDE.md files | Loaded into context; files above the working directory load at startup, and subdirectory files load on demand |
| Skill descriptions | Advertises skills so the model calls <code>SkillTool</code> |
| MCP resources & prompts | Non-tool content an MCP server pushes |
| Output style | Replaces the response-formatting system block |
| UserPromptSubmit hook | Inject context, or block, on every user turn |
| SessionStart hook | One-shot context injection at session start |

(b) model(): what the model can reach

| Element | What it does |
|----------------|--|
| Built-in tools | Read / Edit / Bash / ... shipped with the CLI |
| MCP tools | Tools from any MCP server, in the same flat pool |
| SkillTool | Meta-tool that launches a skill by name |
| AgentTool | Meta-tool that spawns a sub-agent recursively |

(c) execute(): whether / how an action runs

| Element | What it does |
|-------------------|--|
| Permission rules | Declarative <code>allow</code> / <code>deny</code> / <code>ask</code> per call |
| PreToolUse hook | Approve / block / rewrite a tool call |
| PostToolUse hook | Mutate output or inject context after a call |
| Stop hook | Force the loop to keep going at model stop |
| SubagentStop hook | Same, for sub-agents spawned via <code>AgentTool</code> |
| Notification hook | External side effects on user notifications |

Figure 5 Where Claude Code’s extension mechanisms plug into the agent loop. The pseudocode on the left is a zoom-in of the **Agent Loop** block in Figure 1. Every agent loop has three injection points: (a) `assemble()` controls what the model sees, (b) `model()` controls what it can reach, and (c) `execute()` controls whether and how an action actually runs.

an internal `claudeai-proxy`. Each connected server contributes tool definitions as `MCPTool` objects. Dedicated built-in tools `ListMcpResourcesTool` and `ReadMcpResourceTool` provide access to MCP resources.

Plugins. Plugins serve a dual role: they are both a packaging format and a distribution mechanism. The `PluginManifestSchema` (`utils/plugins/schemas.ts`) accepts ten component types: commands, agents, skills, hooks, MCP servers, LSP servers, output styles, channels, settings, and user configuration. The plugin loader (`utils/plugins/pluginLoader.ts`) validates manifests and routes each component to its respective registry: commands and skills surface through the `SkillTool` meta-tool, agents appear in definitions consumed by `AgentTool`, hooks merge into the hook registry, MCP and LSP servers fold into their standard configurations, and output styles modify response formatting. A single plugin package can therefore extend Claude Code across multiple component types simultaneously, making plugins the primary distribution vehicle for third-party extensions.

Skills. Each skill is defined by a `SKILL.md` file with YAML frontmatter. The `parseSkillFrontmatterFields()` function (`loadSkillsDir.ts`) parses 15+ fields including display name, description, allowed tools (granting the skill access to additional tools), argument hints, model overrides, execution context (`‘fork’` for isolated execution), associated agent definitions, effort levels, and shell configuration. Skills can define their own hooks, which register dynamically on invocation. Bundled skills are registered in-memory at startup. When invoked, the `SkillTool` meta-tool injects the skill’s instructions into the context.

Hooks. The source code defines 27 hook events spanning tool authorization (`PreToolUse`, `PostToolUse`, `PostToolUseFailure`, `PermissionRequest`, `PermissionDenied`), session lifecycle (`SessionStart`, `SessionEnd`, `Setup`, `Stop`, `StopFailure`), user interaction (`UserPromptSubmit`, `Elicitation`, `ElicitationResult`), subagent coordination (`SubagentStart`, `SubagentStop`, `TeammateIdle`, `TaskCreated`, `TaskCompleted`), context management (`PreCompact`, `PostCompact`, `InstructionsLoaded`, `ConfigChange`), workspace events (`CwdChanged`, `FileChanged`, `WorktreeCreate`, `WorktreeRemove`), and notifications (`coreTypes.ts`, `coreSchemas.ts`). Of these, 15 have event-specific output schemas with rich fields supporting permission decisions, context injection, input modification, MCP result transformation, and retry control (`types/hooks.ts`). Persisted hook commands configured via settings and plugins use four command types: shell commands (`type: command`), LLM prompt hooks (`type: prompt`), HTTP hooks (`type: http`), and agentic verifier hooks (`type: agent`) (`schemas/hooks.ts`). The runtime additionally supports non-persistable callback hooks (`type: callback`) used by the SDK and internal instrumentation (`types/hooks.ts`). Hook sources include `settings.json`, plugins, and managed policy at startup; skill hooks register dynamically on invocation (`utils/hooks.ts`). The five tool-authorization events are detailed in [Section 5.3](#).

6.2 Tool Pool Assembly

The `assembleToolPool()` function at `tools.ts` is documented as “the single source of truth for combining built-in tools with MCP tools.” The assembly follows a five-step pipeline:

1. **Base tool enumeration.** `getAllBaseTools()` (`tools.ts`) returns an array of up to 54 tools: 19 are always included (such as `BashTool`, `FileReadTool`, `AgentTool`, `SkillTool`), and 35 more are conditionally included based on feature flags, environment variables, and user type. Anthropic-internal users get additional internal tools. Worktree mode enables `EnterWorktreeTool` and `ExitWorktreeTool`. Agent swarms enable team tools. When embedded search tools are available in the Bun binary, dedicated `GlobTool` and `GrepTool` are omitted.
2. **Mode filtering.** `getTools()` (`tools.ts`) applies mode-specific filtering. In `CLAUDE_CODE_SIMPLE` mode, only Bash, Read, and Edit are available (or `REPLTool` in the REPL branch; plus coordinator tools if applicable). Each tool’s `isEnabled()` method is called for runtime availability checks.
3. **Deny rule pre-filtering.** `filterToolsByDenyRules()` (`tools.ts`) strips blanket-denied tools from the model’s view before any call.
4. **MCP tool integration.** MCP tools from `appState.mcp.tools` are filtered by deny rules and merged with built-in tools.
5. **Deduplication.** Tools are deduplicated by name, with built-in tools taking precedence over MCP tools.

Both `REPL.tsx` (via the `useMergedTools` hook) and `AgentTool.tsx` (when building the worker tool set) invoke this function, ensuring consistent assembly across all execution paths. At request time, deferred tools may be hidden from the model’s context until explicitly queried via `ToolSearch` (`tools.ts`).

Agent-based extension (custom agent definitions via `.claude/agents/*.md` and plugin-contributed agents) is covered in [Section 8](#), because agents differ fundamentally from the four mechanisms above: they create new, isolated context windows rather than extending the current one.

6.3 Why Four Mechanisms?

Given that each additional extension mechanism increases the surface area developers must learn, a natural question is why Claude Code uses four distinct mechanisms rather than consolidating into one or two. The answer lies in the observation that different kinds of extensibility impose different costs on the context window, and a single mechanism cannot span the full range from zero-context lifecycle hooks to schema-heavy tool servers without forcing unnecessary trade-offs on extension authors.

As [Table 2](#) summarizes, each mechanism trades deployment complexity for a different kind of extensibility. MCP servers provide runtime tool integration (the model gains new callable tools) at the cost of server management overhead and context budget consumed by tool schemas. Skills shape *how* the agent thinks (not just what tools it has) at minimal context cost, since only frontmatter descriptions (not full content) stay in the prompt. Hooks provide cross-cutting lifecycle control (blocking, rewriting, or annotating tool calls) with no context footprint by default, though hooks can opt into injecting additional context. Plugins bundle any

Table 2 What each extension mechanism uniquely provides. Context cost refers to how much of the bounded context window the mechanism consumes when active.

| Mechanism | Unique Capability | Context Cost | Insertion Point |
|-------------|---|-------------------------|---|
| MCP servers | External service integration (multi-transport) | High (tool schemas) | <code>model():tool pool</code> |
| Plugins | Multi-component packaging + distribution | Medium (varies) | All three points |
| Skills | Domain-specific instructions + meta-tool invocation | Low (descriptions only) | <code>assemble():context injection</code> |
| Hooks | Lifecycle interception + event-driven automation | Zero by default | <code>execute():pre/post tool</code> |

combination of the other three into distributable packages, acting as the packaging and distribution layer rather than a distinct runtime primitive. The graduated context-cost ordering (zero for hooks, low for skills, medium for plugins, high for MCP) means that cheap extensions can scale widely without exhausting the context window, while expensive ones are reserved for cases that genuinely require new tool surfaces.

Some agent frameworks provide a single extension mechanism, typically a tool-only API where all customization arrives as additional callable tools. Others use two tiers, separating tools from configuration or instruction injection. Claude Code’s four-mechanism approach can accommodate a broader range of extension patterns, from zero-context event handlers to full external service integrations, but it increases the learning curve developers face when deciding which mechanism to use for a given integration task.

7 Context Construction and Memory

How an agent manages its context window and persists user instructions is a central design choice, with different systems choosing between file-based transparency, database-backed retrieval, and opaque learned representations. The design choices here implement two principles from [Table 1](#): *context as scarce resource with progressive management* and *transparent file-based configuration and memory*.

By this point in the running example, the task has accumulated state: the original request, the `npm test` permission outcome, the tool pool assembled in [Section 6](#), and any file reads or command outputs gathered so far. This section asks how that growing state is packed into Claude Code’s bounded context window before the next model call.

Before the model is called, the agent loop assembles a context window from the tool pool ([Section 6](#)), CLAUDE.md files, auto memory, and conversation history. The following subsections cover the assembly order, the CLAUDE.md hierarchy, and the multi-step compaction pipeline.

7.1 Context Window Assembly

The context window ([Figure 6](#)) is assembled from the following sources, some at initial assembly and others injected late during the turn:

1. **System prompt**, incorporating output style modifications and any `--append-system-prompt` flag content.
2. **Environment info** via `getSystemContext()` (`context.ts`): git status (skipped in remote mode or when git instructions are disabled) and an optional cache-breaking injection for internal builds (gated by `BREAK_CACHE_COMMAND`). Memoized once per session.
3. **CLAUDE.md hierarchy** via `getUserContext()` (`context.ts`): four-level instruction file hierarchy ([Section 7.2](#)). Also memoized.
4. **Path-scoped rules**: conditional and directory-matched rules that load lazily when the agent reads files in matching directories.
5. **Auto memory**: contextually relevant memory entries prefetched asynchronously.
6. **Tool metadata**: skill descriptions, MCP tool names, and deferred tool definitions (via `ToolSearch`, on demand).
7. **Conversation history**: carried forward, subject to compaction.
8. **Tool results**: file reads, command outputs, subagent summaries.
9. **Compact summaries**: replacing older history segments.

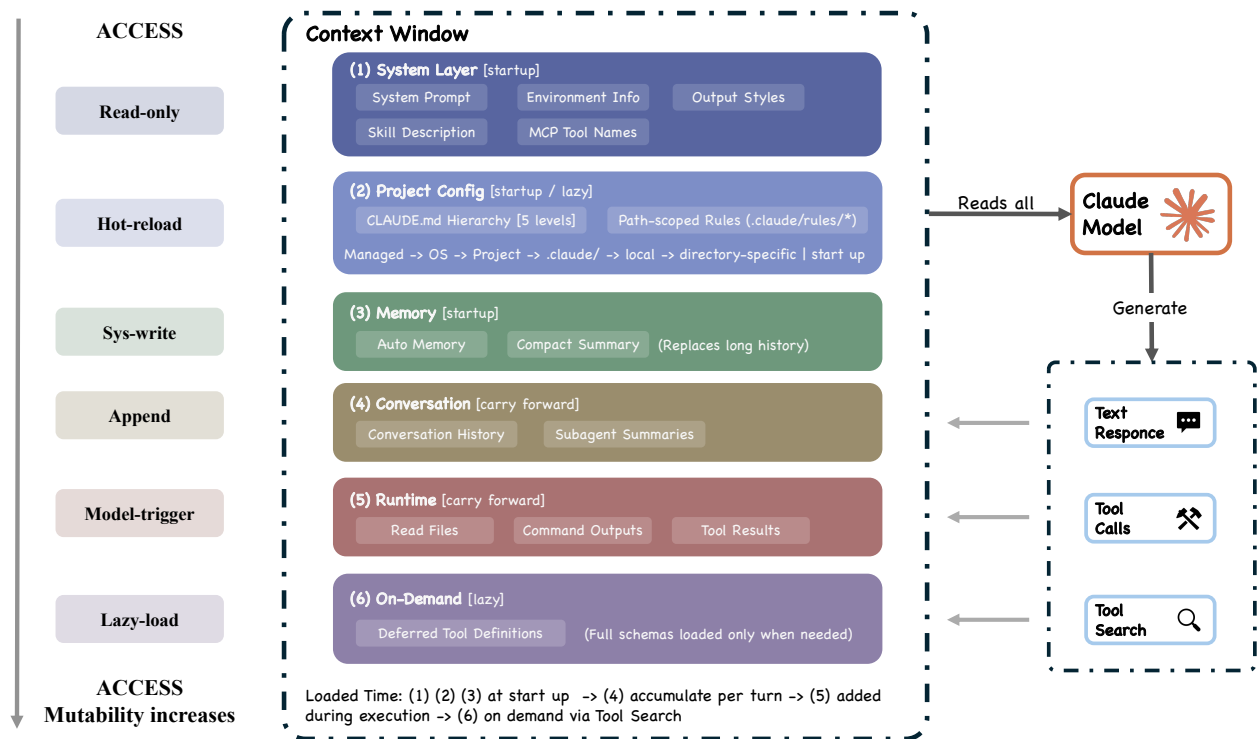


Figure 6 Context construction and memory hierarchy. Sources converging on the context window include system prompt, output styles, environment info, the CLAUDE.md hierarchy (managed through directory-specific), auto memory, path-scoped rules, MCP tool names, deferred tool definitions via ToolSearch, conversation history, file reads, command outputs, tool results, subagent summaries, and compact summaries.

The system prompt assembly at `query.ts` combines system context with the base prompt via `asSystemPrompt(appendSystemContext(systemPrompt, systemContext))()`. User context (CLAUDE.md and date) is prepended to the message array via `prependUserContext()`. This separation means CLAUDE.md content occupies a different structural position in the API request than the system prompt, potentially affecting model attention patterns.

Several context sources are injected late, after the main window is constructed: relevant-memory prefetch (`query.ts`), MCP instructions deltas (only new or changed server instructions), agent listing deltas, and background agent task notifications. The context window is therefore not static at assembly time but can grow during the turn.

7.2 CLAUDE.md Hierarchy and Auto Memory

A design principle shapes the memory system: stored context should be inspectable and editable by the user. CLAUDE.md files are plain-text Markdown rather than structured configuration or opaque database entries. This transparency choice trades expressiveness for auditability: users can read, edit, version-control, and delete any instruction the agent sees (MindStudio Team, 2026). Alternative memory architectures illustrate the trade-off. Retrieval-augmented approaches use embedding-based lookup to surface relevant prior context, gaining flexibility at the cost of inspectability: the user cannot easily see or edit what the retrieval system considers relevant. Database-backed memory offers structured querying but requires additional infrastructure and is opaque to version control. Claude Code’s file-based approach makes every instruction the agent sees directly readable, editable, and committable alongside the codebase. The system does not use embeddings or a vector similarity index for memory retrieval; instead it uses an LLM-based scan of memory-file headers to select up to five relevant files on demand, surfacing them at file granularity rather than entry granularity. Embedding-based systems can retrieve individual entries more selectively, at the cost of inspectability and the infrastructure needed to maintain an index.

CLAUDE.md files follow a multi-level loading hierarchy. The source header (`claudemd.ts`) defines four memory types:

1. **Managed memory** (e.g. `/etc/claude-code/CLAUDE.md` on Linux): OS-level policy for all users.
2. **User memory** (`~/.claude/CLAUDE.md`): private global instructions.
3. **Project memory** (`CLAUDE.md`, `.claude/CLAUDE.md`, and `.claude/rules/*.md` in project roots): instructions checked into the codebase.
4. **Local memory** (`CLAUDE.local.md` in project roots): gitignored, for private project-specific instructions.

File discovery traverses from the current directory up to root, checking for all project and local memory files in each directory. Files closer to the current directory have higher priority (loaded later).

Files load in “reverse order of priority”: later-loaded files receive more model attention. For root-to-CWD directories, unconditional rules from `.claude/rules/*.md` load eagerly at startup. For nested directories below CWD, even unconditional rules are loaded lazily when the agent reads files in matching directories. This means the model’s instruction set can evolve during a conversation as new parts of the codebase are explored.

CLAUDE.md content is delivered as user context (a user message), not as system prompt content (`context.ts`). This architectural choice has a significant implication: because CLAUDE.md content is delivered as conversational context rather than system-level instructions, model compliance with these instructions is probabilistic rather than guaranteed. Permission rules evaluated in deny-first order (Section 5) provide the deterministic enforcement layer. This creates a deliberate separation between guidance (CLAUDE.md, probabilistic) and enforcement (permission rules, deterministic). The function calls `setCachedClaudeMdContent()` to cache the loaded content for the auto-mode classifier, to avoid an import cycle between the CLAUDE.md loader and the permission system.

Memory files support an `@include` directive for modular instruction sets (`processMemoryFile()` at `claudemd.ts`). Syntax variants include `@path`, `@./relative`, `@~/home`, and `@/absolute`. The directive works in leaf text nodes only (not inside code blocks). In the implementation, the including file is pushed first and included files are appended after it, circular references are prevented by tracking processed paths, and non-existent files are silently ignored.

7.3 Compaction Pipeline

The five-layer compaction pipeline (Section 4.3) implements the “context as bottleneck” principle through graduated compression (`query.ts`). Rather than a single strategy, Claude Code applies five layers in sequence, each with increasing aggressiveness (three are gated by feature flags; budget reduction is always active, while auto-compact is user-configurable). This graduated approach contrasts with simpler alternatives: many agent frameworks use single-pass truncation (dropping the oldest messages) or a single summarization step. The graduated design reflects a lazy-degradation principle: apply the least disruptive compression first, escalating only when cheaper strategies prove insufficient. The cost of this approach is complexity. Five interacting compression layers, several gated by feature flags, create behavior that is difficult for users to fully predict. Auto-compact produces a visible summary in the transcript, and microcompact emits a boundary marker, but context collapse operates without user-visible output. Simpler single-pass approaches sacrifice information but are easier to reason about.

1. **Budget reduction** (always active): per-tool-result size limits.
2. **Snip** (`HISTORY_SNIP`): lightweight older-history trimming.
3. **Microcompact** (`CACHED_MICROCOMPACT`): fine-grained cache-aware compression.
4. **Context collapse** (`CONTEXT_COLLAPSE`): read-time virtual projection over history.
5. **Auto-compact** (enabled by default, can be disabled): full model-generated summary.

The `buildPostCompactMessages()` function (`compact.ts`) returns the following compacted output structure: `[boundaryMarker, ...summaryMessages, ...messagesToKeep, ...attachments, ...hookResults]`. The boundary marker is annotated with preserved-segment metadata via `annotateBoundaryWithPreservedSegment()`, recording `headUuid`, `anchorUuid`, and `tailUuid` to enable read-time chain patching. This mostly-append design means compaction never modifies or deletes previously written transcript lines; it only appends new boundary and summary events.

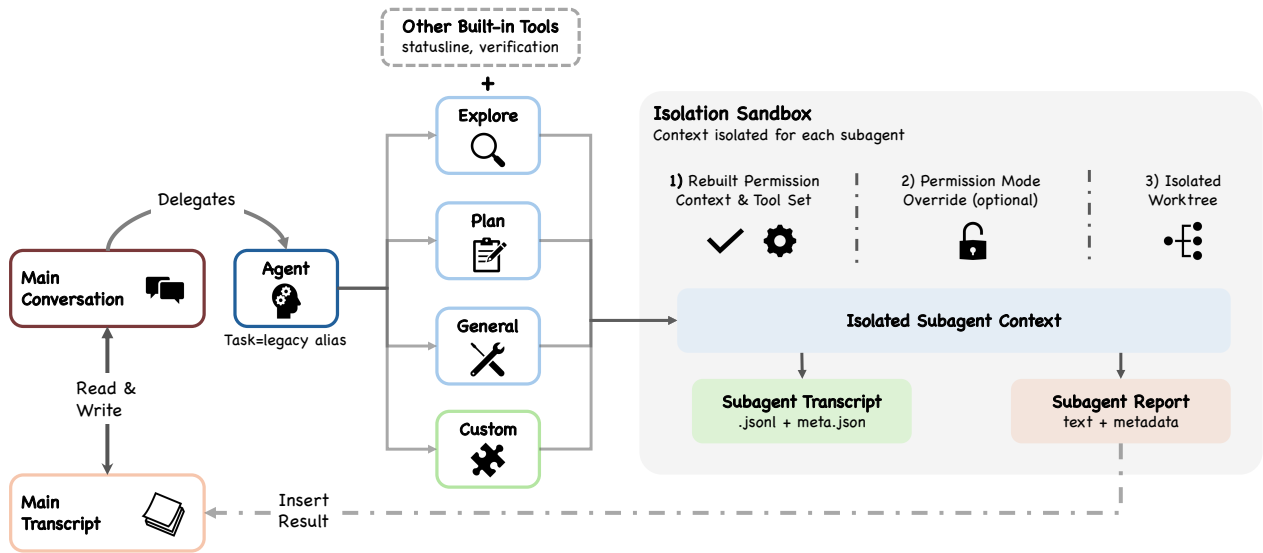


Figure 7 Subagent isolation and delegation architecture. The Agent tool dispatches to built-in subagents (Explore, Plan, general-purpose) or custom subagents, each running in an isolated context with rebuilt permission context and independent tool sets. The Agent tool dispatches along three axes: routing (teammate), isolation (remote, worktree), and lifecycle (async, sync).

The compaction function `compactConversation()` (`compact.ts`) includes several design choices. Pre-compact hooks fire first, allowing hook-injected custom instructions. A GrowthBook feature flag controls whether the compaction path reuses the main conversation’s prompt cache (a code comment documents a January 2026 experiment: “false path is 98% cache miss, costs $\sim 0.76\%$ of fleet `cache_creation`”). After compaction, attachment builders re-announce runtime state (plans, skills, and async agents) from live app state, since compaction discards prior attachment messages but not the underlying state.

Context isolation becomes more critical when the system delegates work to subagents, each operating in its own bounded context window.

8 Subagent Delegation and Orchestration

Multi-agent orchestration is a key design dimension for coding agents, with choices spanning parent-child hierarchies, peer-based conversation frameworks (Wu et al., 2024), and graph-structured workflow engines (LangChain, Inc., 2024). Claude Code’s delegation architecture implements the *isolated subagent boundaries* principle from Table 1, together with aspects of *deny-first with human escalation* (permission override) and *reversibility-weighted risk assessment* (subagent tool restrictions).

When Claude determines that the auth test fix requires first exploring the authentication module’s structure, it can delegate this exploration to a subagent. The delegation mechanism is the **Agent** tool (`AgentTool.tsx`), with `Task` retained as a legacy alias. The model invokes **Agent** with a structured input including the delegation prompt, an optional subagent type, and configuration for isolation mode, permission overrides, and working directory.

8.1 The Agent Tool and Delegation Criteria

The Agent tool input schema (Figure 7) uses feature-gated fields, omitting optional parameters when their backing features are disabled. The `isolation` field offers [`worktree`, `remote`] for internal users and [`worktree`] for external users, determined at build time. The `cwd` field is gated by a feature flag. The `run_in_background` field is omitted when background tasks are disabled or when fork-subagent mode is enabled.

Claude Code provides up to six built-in subagent types, depending on feature flags and endpoint:

- **Explore:** primarily read/search-oriented investigation, with write and edit tools in its deny-list.
- **Plan:** creates structured plans; execution proceeds through the standard permission model.
- **General-purpose:** broadly capable, used when explicitly requested (note: omitting the type may route to the fork-subagent path instead).
- **Claude Code Guide:** onboarding and documentation assistance, with its own `permissionMode` override.
- **Verification:** runs validation checks (test suites, linting).
- **Statusline-setup:** specialized for terminal status line configuration.

Beyond built-ins, users define custom subagents via `.claude/agents/*.md` files, and plugins contribute agent definitions via `loadPluginAgents.ts`. The markdown body of each file serves as the agent’s system prompt, and YAML frontmatter specifies configuration fields including `description`, `tools` (allowlist), `disallowedTools`, `model`, `effort`, `permissionMode`, `mcpServers`, `hooks`, `maxTurns`, `skills`, `memory` scope, `background` flag, and `isolation` mode. JSON-formatted agent definitions support the same fields plus `prompt` as an explicit field (`loadAgentsDir.ts`). This means a custom agent can be a fully configured, isolated sub-system with its own tools, model, permissions, hooks, memory scope, and isolation mode. `AgentTool` sits alongside `SkillTool` in the base tool pool as a meta-tool that dispatches to these definitions, but the two differ fundamentally: `SkillTool` injects instructions into the current context window, while `AgentTool` spawns a new, isolated one. The tradeoff is that most subagent invocations require a self-contained prompt, because the default path does not inherit the parent’s conversation history (the fork-subagent path is an exception). Conversation-based frameworks that share full transcript histories avoid this cost but risk context explosion as the number of agents grows.

8.2 Isolation Architecture

Subagent isolation supports multiple modes (`AgentTool.tsx`):

- **Worktree:** Creates a temporary git worktree, giving the subagent its own copy of the repository to modify without affecting the parent’s working tree.
- **Remote** (internal-only): Launches in a remote Claude Code Remote environment, always running in the background.
- **In-process** (default): Shares the filesystem with the parent but operates in an isolated conversation context.

The permission override logic for subagents (`runAgent.ts`) involves several specific rules. When a subagent defines a `permissionMode`, the override is applied unless the parent is already in `bypassPermissions`, `acceptEdits`, or `auto` mode, since those modes always take precedence because they represent explicit user decisions about the safety/autonomy trade-off. For async agents, the system determines whether to avoid prompts through a cascade: explicit `canShowPermissionPrompts` first, then `bubble` mode (always show, since they escalate to the parent terminal), then the default (sync agents show prompts, async agents do not). Background agents that can show prompts set `awaitAutomatedChecksBeforeDialog`: `true`, ensuring the classifier and hooks resolve before interrupting the user.

These isolation modes occupy different points in a design space. Container-based isolation (used by SWE-Agent and OpenHands (Yang et al., 2024; Wang et al., 2024b)) provides stronger resource boundaries but requires container infrastructure. Context-only isolation (used by conversation-based frameworks like AutoGen (Wu et al., 2024)) shares the filesystem but separates conversation histories. Claude Code’s worktree-based isolation provides filesystem-level separation with zero external dependencies, leveraging Git’s built-in mechanism rather than introducing container orchestration.

When `allowedTools` is explicitly provided to `runAgent()` (`runAgent.ts`), a two-tier permission scoping model applies. SDK-level permissions from `--allowedTools` are preserved: “explicit permissions from the SDK consumer that should apply to all agents.” But session-level rules are replaced with the subagent’s declared `allowedTools`. When `allowedTools` is not provided (the common `AgentTool` path), the parent’s session-level rules are inherited without replacement.

8.3 Sidechain Transcripts

Each subagent writes its own transcript as a separate `.jsonl` file with a `.meta.json` metadata file (`sessionStorage.ts`, `runAgent.ts`). This sidechain design means subagent histories are preserved for debugging and auditing but do not inflate the parent’s session file. Only the subagent’s final response text and metadata return to the parent conversation context; the full subagent history never enters the parent’s context window, respecting the “context as bottleneck” principle.

The `runAgent()` function accepts 21 parameters covering agent definition, prompts, permissions, tools, model settings, isolation, and callbacks.

The summary-only return model is a deliberate context-conservation choice: conversation-based frameworks that share full transcript histories between agents risk context explosion as the number of agents grows. Even isolated-context parallelism carries substantial cost. Claude Code’s agent teams consume approximately $7\times$ the tokens of a standard session in plan mode (Anthropic, 2025b), which makes summary-only return more critical when subagents are also in isolated contexts.

For multi-instance coordination in agent teams, the harness uses file locking rather than a message broker or distributed coordination service (Anthropic, 2025b). Tasks are claimed from a shared list via lock-file-based mutual exclusion, with lock files stored at predictable filesystem paths. This trades throughput for two properties: zero-dependency deployment (no external infrastructure required) and full debuggability (any agent’s state can be inspected by reading plain-text JSON files).

9 Session Persistence and Recovery

Session persistence in coding agents involves a design choice between append-only logs, structured databases, checkpoint-based snapshots, and stateless architectures, each with different trade-offs in auditability, query power, and deployment complexity. Claude Code’s persistence design implements the *append-only durable state* principle from Table 1. Session-scoped permissions live in memory only and are not serialized to the transcript, so resume rebuilds the permission context from CLI args and disk settings; requests the rebuilt context does not recognize fall back to deny-first prompting.

By the time the auth-test task reaches this section, the session contains the original prompt, tool invocations and results, compact boundaries, and the subagent summary from exploring the authentication module (Section 8). This section asks which of those artifacts are durably recorded and what can be recovered later without carrying forward the session’s old permission grants.

Claude Code’s persistence mechanisms write the conversation (messages, tool results, and compact boundaries) to disk as events occur.

9.1 Transcript Model

Session transcripts are stored as mostly append-only JSONL files at a project-specific path (with explicit cleanup rewrites as an exception) (Figure 8). The `getTranscriptPath()` function (`sessionStorage.ts`) computes this as `join(projectDir, ${getSessionId()}.jsonl)`, where `projectDir` is determined by first checking `getSessionProjectDir()` (set by `switchSession()` during resume/branch) and falling back to `getProjectDir(getOriginalCwd())()`.

Three persistence channels operate independently:

1. **Session transcripts:** Conversation records including user, assistant, attachment, and system messages, plus compaction and other metadata events. Project-scoped, one file per session.
2. **Global prompt history:** User prompts only, stored in `history.jsonl` at the Claude configuration home directory (`history.ts`). The `makeHistoryReader()` generator yields entries in reverse order via `readLinesReverse()`, supporting Up-arrow and `ctrl+r` navigation.
3. **Subagent sidechains:** Separate `.jsonl` + `.meta.json` files per subagent (Section 8.3).

Session transcripts store several kinds of events beyond simple messages, including compaction markers, file-history snapshots, attribution snapshots, and content-replacement records. The append-only JSONL format

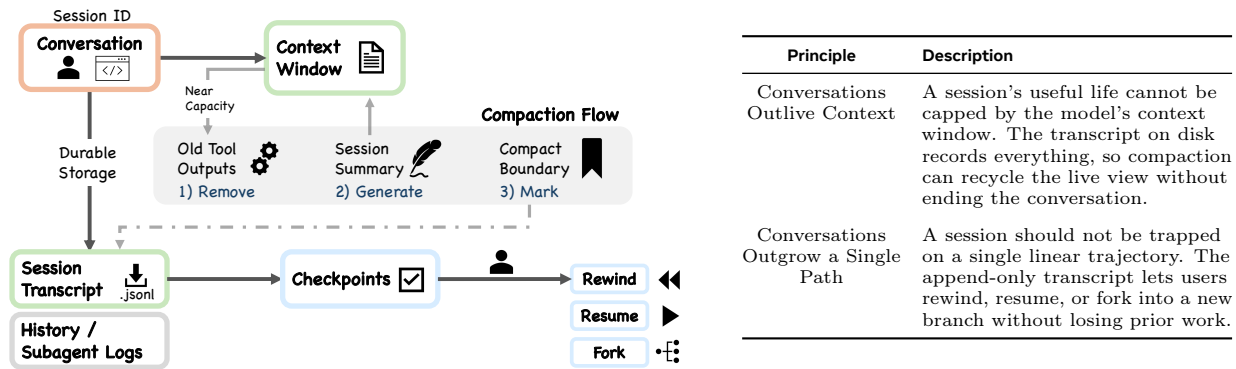


Figure 8 Session persistence and context compaction. The diagram separates live session state (context window, compaction) from durable storage (session transcripts, history.jsonl, subagent sidechains, checkpoints). Resume and fork restore messages but not session-scoped permissions.

is a deliberate choice favoring auditability and simplicity over query power. Every event is human-readable, version-controllable, and reconstructable without specialized tooling. Database-backed alternatives would enable richer queries over session history but introduce deployment dependencies and reduce transparency.

The session identity system pairs `sessionId` with `sessionProjectDir`, set together during resume or branch. The transcript path must use the same project directory that was active when messages were written, to avoid hooks looking in the wrong directory.

9.2 Resume, Fork, and Not Restoring Permissions

The `--resume` flag rebuilds the conversation by replaying the transcript (`conversationRecovery.ts`). Fork creates a new session from an existing one (`commands/branch/branch.ts`). However, resume and fork do not restore session-scoped permissions; users must grant them again in the new session. This is a deliberate safety-conservative design choice: sessions are treated as isolated trust domains. Restoring previously granted permissions on resume would create a convenience benefit but risk carrying stale trust decisions into a changed context. The architecture opts for re-granting over implicit persistence, accepting user friction as the cost of maintaining the safety invariant that trust is always established in the current session.

The `compact_boundary` marker is carefully designed to work with persistence. The `annotateBoundaryWithPreservedSegment()` function (`compact.ts`) records `headUuid`, `anchorUuid`, and `tailUuid` in the boundary event. These UUIDs enable the session loader to patch the message chain at read time: preserved messages keep their original `parentUuids` on disk, and the loader uses boundary metadata to link them correctly. This mostly-append design means compaction never modifies or deletes previously written transcript lines.

The “checkpoints” in Claude Code are file-history checkpoints for `--rewind-files`, stored at `~/.claude/file-history/<sessionId>/`. These are file-level snapshots for reverting filesystem changes, not a generic checkpoint store.

The preceding sections have documented Claude Code’s answers to recurring design questions. The next section contrasts Claude Code’s design choices with those of an architecturally independent AI agent system.

10 Comparative Analysis: Claude Code and OpenClaw

The preceding sections documented Claude Code’s answers to recurring design questions about loop architecture, safety, extensibility, context management, delegation, and persistence. To calibrate these findings, this section compares Claude Code with OpenClaw, an independent open-source AI agent system that answers many of the same design questions from a fundamentally different starting point. OpenClaw is a local-first WebSocket gateway that connects roughly two dozen messaging surfaces (WhatsApp, Telegram, Slack, Discord, Signal, and others) to an embedded agent runtime, with companion apps on macOS, iOS, and Android (Steinberger

Table 3 Architectural comparison: Claude Code vs. OpenClaw across six design dimensions. Each row captures a recurring design question and the different answers the two systems provide.

| Dimension | Claude Code | OpenClaw |
|-------------------------|--|--|
| System scope | CLI/IDE coding harness, ephemeral per-session process | Persistent WS gateway daemon, multi-channel control plane |
| Trust model | Deny-first per-action rule evaluation with hooks and optional ML classifier; 7 permission modes; graduated trust spectrum | Single trusted operator per gateway; DM pairing and allowlists for inbound channels; opt-in sandboxing with configurable scope (per-agent, per-session, or shared) and multiple backends |
| Agent runtime | Iterative async generator (<code>queryLoop()</code>) as system center | Pi-agent runner embedded inside gateway RPC dispatch; per-session queue serialization (with optional global lane) |
| Extension architecture | 4 mechanisms at graduated context costs: MCP, plugins, skills, hooks | Manifest-first plugin system with 12 capability types and central registry; separate skills layer; built-in MCP via <code>openclaw mcp</code> (server and outbound client registry) |
| Memory and context | CLAUDE.md 4-level hierarchy; 5-layer compaction pipeline; LLM-based memory scan | workspace bootstrap files (AGENTS.md, SOUL.md, TOOLS.md, IDENTITY.md, USER.md, plus conditionally BOOTSTRAP.md, HEARTBEAT.md, and MEMORY.md); separate memory system (MEMORY.md, daily notes, optional DREAMS.md); auto-compaction with pluggable providers; optional hybrid search (vector + keyword, conditional on embedding provider); experimental dreaming for long-term promotion |
| Multi-agent and routing | Task-delegating subagents (e.g., Explore, Plan, general-purpose); worktree isolation; final response text returned to parent | Two separate concerns: (a) multi-agent routing with isolated agents, distinct workspaces, and binding-based channel dispatch; (b) sub-agent delegation with configurable nesting depth (max 5, default 1, recommended 2) and thread-bound sessions |

and OpenClaw Contributors, 2026). Where Claude Code is a CLI coding harness bound to a single repository session, OpenClaw is a persistent control plane for multi-channel personal assistance. The two systems occupy different regions of the agent design space. The value of the comparison lies in showing how the same recurring questions produce different architectural answers when the deployment context changes.

10.1 Six Comparison Dimensions

Table 3 summarizes the comparison across six dimensions. Each dimension corresponds to a design question that both systems must answer.

System scope and deployment model. Claude Code runs as an ephemeral CLI process bound to a single repository. Each session starts and ends with the terminal. OpenClaw runs as a persistent daemon (default port 18789, loopback-only) that owns all messaging surface connections and coordinates clients, tools, and device nodes over a typed WebSocket protocol. This difference in system scope is the most fundamental architectural divergence: it determines how every other design question is framed. A compositional relationship also exists: OpenClaw can host Claude Code, OpenAI Codex, and Gemini CLI as external coding harnesses through its ACP (Agent Client Protocol) integration, making the two systems stackable rather than purely alternative.

Trust model and security architecture. The systems address different threat models. Claude Code assumes an untrusted model operating within a trusted developer’s machine: the deny-first permission system (Section 5) evaluates every tool invocation, the ML classifier provides automated safety assessment, and seven permission modes create a graduated autonomy spectrum. OpenClaw assumes a single trusted operator per gateway instance. Its security architecture begins with identity and access control (DM pairing codes, sender allowlists, gateway authentication) rather than per-action safety classification. Tool policy uses configurable allow/deny lists per agent rather than a centralized classifier. Sandboxing is available as an opt-in feature with multiple backends (Docker, SSH, or OpenShell) and configurable scope (per-agent, per-session, or shared); a **non-main** mode can sandbox all non-main sessions when enabled, though sandboxing is not active by default. The OpenClaw security documentation explicitly states that hostile multi-tenant isolation on a shared gateway is not a supported security boundary. This difference reflects a design choice about where the trust boundary sits: Claude Code places it between the model and the execution environment; OpenClaw places it at the gateway perimeter.

Agent runtime and tool orchestration. Both systems implement agentic loops, but these loops occupy different positions in their respective architectures. In Claude Code, the `queryLoop()` async generator (Section 4) is the system’s center: all interfaces feed into it, and it directly manages context assembly, model calls, tool dispatch, and recovery. In OpenClaw, the agent runtime (an embedded Pi-agent core) sits inside a larger gateway dispatch layer. The gateway’s **agent** RPC validates parameters, resolves sessions, and returns immediately; the embedded runner then executes the agentic loop while emitting lifecycle and stream events back through the gateway protocol. Runs are serialized through per-session queues and an optional global lane, preventing tool and session races across the multi-channel surface. Both systems follow the ReAct pattern (Yao et al., 2022), but OpenClaw’s loop is a component within a control plane rather than the control plane itself.

Extension architecture. Claude Code’s four extension mechanisms (MCP, plugins, skills, hooks) are organized by context cost (Section 6): hooks consume zero context, skills consume low context, and MCP servers consume high context. All four extend a single agent’s context window and tool surface. OpenClaw uses a manifest-first plugin system with four architectural layers (discovery, enablement, runtime loading, surface consumption) and twelve capability types including text inference, speech, media understanding, image/music/video generation, web search, and messaging channels. Plugins register capabilities into a central registry; the gateway reads the registry to expose tools, channels, provider setup, hooks, HTTP routes, CLI commands, and services. OpenClaw also has a separate skills layer with multiple sources (workspace, project-level, personal, managed, bundled, and extra directories, with workspace skills taking highest precedence) plus a public registry (ClawHub) and supports MCP through built-in `openclaw mcp` commands (server and outbound client registry). The key architectural difference is that Claude Code’s extensions modify one agent’s action surface, while OpenClaw’s plugins extend the gateway’s capability surface across all agents.

Memory, context, and knowledge management. Both systems use transparent file-based memory rather than opaque databases. Claude Code loads a four-level CLAUDE.md hierarchy and manages context pressure through a five-layer compaction pipeline (Section 7). Memory retrieval uses an LLM-based scan of file headers. OpenClaw injects workspace bootstrap files into the system prompt at session start: five core files (AGENTS.md, SOUL.md, TOOLS.md, IDENTITY.md, USER.md) plus conditionally BOOTSTRAP.md, HEARTBEAT.md, and MEMORY.md, with large files truncated. Separately, the memory system manages three file types: MEMORY.md for long-term durable facts, date-stamped daily notes (`memory/YYYY-MM-DD.md`), and an optional DREAMS.md for dreaming sweep summaries. When an embedding provider is configured, memory search uses hybrid retrieval combining vector similarity with keyword matching. An experimental dreaming system performs background consolidation, scoring candidates and promoting only qualified items from short-term recall into long-term memory. Before compaction, OpenClaw automatically reminds the agent to save important notes to memory files, preventing context loss. Both systems share the design commitment to user-visible, editable memory. OpenClaw invests more heavily in structured long-term memory promotion (dreaming, daily notes, memory search), while Claude Code invests more in graduated context compression (five layers with cache awareness). OpenClaw also supports pluggable compaction providers and session pruning, but its compaction pipeline is less graduated than Claude Code’s five-layer system.

Multi-agent architecture and routing. This dimension reveals the starkest architectural difference. Claude Code’s multi-agent model is task delegation: the parent spawns subagents (Explore, Plan, general-purpose, and custom types) that operate in isolated context windows with restricted tool sets and return summary-only results (Section 8). Worktree isolation provides filesystem-level separation. OpenClaw separates two distinct concerns. First, multi-agent routing: a single gateway can host multiple fully isolated agents, each with its own workspace, authentication profiles, session store, and model configuration, routed to specific channels or senders via deterministic binding rules. Second, sub-agent delegation: within a single agent, background runs can be spawned with configurable nesting depth (maximum 5, default 1, recommended 2), thread-bound sessions on supported channels, and configurable tool policy by depth. OpenClaw’s project vision explicitly rejects agent-hierarchy frameworks as a default architecture. The distinction matters because Claude Code’s subagents are subordinate workers within one user’s coding session, while OpenClaw’s multi-agent routing creates genuinely independent agent instances serving different users or purposes through different channels.

10.2 What the Contrast Reveals

The comparison surfaces three observations about the design space of AI agent systems.

First, the recurring design questions identified in Section 3.1 (where reasoning lives, what safety posture to adopt, how to manage context, how to structure extensibility) apply beyond coding agents. OpenClaw answers every one of these questions, but from the starting point of a multi-channel personal assistant rather than a repository-bound coding tool. The questions are stable; the answers vary with deployment context.

Second, the systems make opposite bets on several dimensions. Claude Code invests in graduated per-action safety evaluation; OpenClaw invests in perimeter-level identity and access control. Claude Code treats the agent loop as the architectural center; OpenClaw treats the gateway control plane as the center and embeds the agent loop as one component. Claude Code’s extensions modify a single context window; OpenClaw’s plugins extend a shared gateway surface. These inversions are not arbitrary: they follow from the different trust models and deployment topologies.

Third, the compositional relationship between the two systems is architecturally significant. OpenClaw can host Claude Code as an external coding harness via ACP, meaning the two systems are composable rather than exclusive alternatives. This suggests that the design space of AI agents is not a flat taxonomy but a layered one, where gateway-level systems and task-level harnesses can compose.

11 Discussion

The analysis in the preceding sections documented how Claude Code answers recurring design questions about loop architecture, safety posture, extensibility, context management, delegation, and persistence. Each answer reflects a position in a design space with real alternatives and measurable trade-offs. This section examines what those answers reveal when read together: the design philosophy they reflect (Section 11.1), the value tensions they create (Section 11.2), the architectural trade-offs they entail (Section 11.3), the empirical predictions they generate (Section 11.4), and the cross-cutting commitments that recur across subsystems (Section 11.7). The five-value framework from Section 2.1 serves as the organizing lens throughout.

11.1 Design Philosophy

The values and design principles introduced in Section 2 predict an architecture that invests in operational infrastructure rather than decision scaffolding. The implementation confirms this: the architecture documented in Sections 3 to 9 is overwhelmingly deterministic infrastructure (permission gates, tool routing, context management, recovery logic), with the LLM invoked as a stateless completion endpoint. An estimated 1.6% of the codebase constitutes decision logic, the remaining 98.4% is the operational harness. This ratio is not accidental.

The design principles documented in Section 2.2 underpin this approach: the harness creates conditions under which the model can decide well, rather than constraining its choices.

Table 4 Tensions between values, with supporting evidence. Each tension demonstrates that the two values capture genuinely distinct concerns.

| Value Pair | Tension | Evidence |
|----------------------------------|----------------------------------|--|
| Authority \times Safety | Approval fatigue vs. protection | 93% approval rate undermines human vigilance (Hughes, 2026); safety must compensate via classifier and sandboxing |
| Safety \times Capability | Performance vs. defense depth | >50-subcommand fallback skips per-subcommand deny checks due to parsing overhead (Adversa.ai, 2026); safety layers share performance constraints |
| Adaptability \times Safety | Extensibility vs. attack surface | Multiple CVEs exploit pre-trust initialization of hooks and MCP servers (Donenfeld and Vanunu, 2026) |
| Capability \times Adaptability | Proactivity vs. disruption | 12 to 18% more tasks but preference drops at high frequencies (Chen et al., 2025) |
| Capability \times Reliability | Velocity vs. coherence | Bounded context prevents full codebase awareness (Section 7); subagent isolation limits cross-agent consistency (Section 8); complexity increases observed in adjacent tools (He et al., 2025) |

This design runs counter to the dominant pattern in agent engineering, where frameworks such as LangGraph route model outputs through explicit graph nodes with typed edges, and systems like Devin pair multi-step planners with heavy operational infrastructure. Claude Code instead gives the model maximum decision latitude within a rich operational harness. The engineering complexity exists not to constrain the model’s decisions but to enable them. This layered architecture, where the model reasons and the harness enforces, raises the question of whether agentic coding tools are converging toward operating-system-like abstractions in which the core loop serves as the kernel and everything else constitutes the OS.

The design gains additional significance as frontier models converge in practical capability for coding tasks: the quality of the surrounding operational harness becomes the principal differentiator, validating an architecture that invests in infrastructure over decision scaffolding. For agent builders, the implication is that investing in deterministic infrastructure such as context management, safety layering, and recovery mechanisms may yield greater reliability gains than adding planning scaffolding around increasingly capable models.

Taken together, the preceding sections show that production coding agents face recurring design choices: where reasoning lives relative to the harness, how the iteration loop is structured, what safety posture to adopt by default, how the extension surface is partitioned, how context is assembled and compressed, how subagents are delegated and orchestrated, and how sessions persist across boundaries. Claude Code’s answers to these questions form a coherent design point that privileges model autonomy within a rich operational harness.

This philosophy assumes that rich deterministic infrastructure can adequately support unconstrained model judgment. The following subsections examine where this assumption is tested.

11.2 Value Tensions

The five values identified in Section 2.1 generate tensions where pursuing one value constrains another (Table 4). These tensions are not design failures; they are structural consequences of pursuing multiple values simultaneously. We report the tensions with the strongest supporting evidence, not the full combinatorial set.

Two additional tensions surface through the evaluative lens of long-term capability preservation (Section 2.4). A randomized controlled trial of 16 experienced developers across 246 tasks (Becker et al., 2025) found that AI tools made developers 19% slower, despite a perceived 20% improvement. A causal analysis of Cursor adoption across 807 repositories (He et al., 2025) found that code complexity increased by 40.7%. An EEG study of 54 participants (Kosmyna et al., 2025) found that LLM users showed weakened neural connectivity that persisted after AI was removed. Researchers have proposed protocols for measuring cognitive offloading in AI-assisted programming, motivated by concerns that students using AI produce applications without understanding

the underlying logic (Aiersilan, 2026). These findings, combined with a 25% decline in entry-level tech hiring between 2023 and 2024 (Rak, 2025), suggest that the tension between capability amplification and long-term sustainability extends beyond individual productivity to the broader developer pipeline. This evidence motivates the evaluative lens but does not target Claude Code’s architecture specifically; it applies to any agent system with bounded context and tool-use loops.

11.3 Architectural Trade-offs

The tensions in Table 4 manifest as concrete architectural trade-offs in four areas. The long-term sustainability concerns documented in the evaluative lens paragraph above surface in the empirical predictions of Section 11.4.

Safety vs. autonomy. The permission modes (five always present, plus `auto` when the classifier feature flag is active, and the internal `bubble` mode) create a gradient from `plan` (user approves all plans) through `default`, `acceptEdits`, `auto` (ML classifier), `dontAsk`, to `bypassPermissions` (skips most prompts but safety-critical checks remain). The progression represents a monotonically decreasing safety gradient with increasing autonomy. Not restoring permissions on resume reflects a deliberate choice to err toward safety: security state does not persist implicitly across session boundaries.

The safety-autonomy gradient is shaped not only by architectural design but by user behavior. Anthropic’s auto-mode analysis (Hughes, 2026) found that users approve approximately 93% of permission prompts, indicating that approval fatigue renders interactive confirmation behaviorally unreliable. Longitudinal usage data (McCain et al., 2026) shows that auto-approve rates increase from approximately 20% at fewer than 50 sessions to over 40% by 750 sessions, with substantial increases in session duration. These patterns suggest that the gradient is navigated not by deliberate mode selection but by gradual habituation. Sandboxing reduced the frequency of permission prompts by an estimated 84% (Dworken and Weller-Davies, 2025), reframing the problem as a human-factors concern: the architectural response to unreliable human approval is to reduce the number of decisions humans must make.

More fundamentally, the defense-in-depth architecture described in Section 5 rests on an independence assumption: if one safety layer fails, others catch the violation. But Claude Code’s safety layers share common performance and economic constraints. The auto-mode classifier is a separate LLM call with direct token cost. The `bashSecurity.ts` module performs sequential AST-based checks with parsing latency. The deny-first rule evaluation operates on command structure. When performance pressure pushes toward reducing these costs, layers can degrade simultaneously. Security researchers (Adversa.ai, 2026) have documented that commands with more than 50 subcommands fall back to a single generic approval prompt instead of running per-subcommand deny-rule checks, because per-subcommand parsing caused UI freezes, demonstrating that defense-in-depth fails when the independence assumption is violated.

This tension is structural. Any LLM-based agent system that uses the model itself for safety evaluation faces it. The relevant evaluation criterion is not whether any individual layer can be bypassed, but how many independent layers must fail simultaneously and whether they share failure modes.

Permission model under adversarial conditions. Independent security research provides empirical validation of the permission architecture, specifically by revealing a temporal ordering property not captured in Figure 4. Two independently verified vulnerabilities share a root cause in pre-trust initialization ordering: code executing during project initialization (hooks, MCP server connections, and settings file resolution) runs before the interactive trust dialog is presented to the user³. This pre-trust execution window falls outside the deny-first evaluation pipeline (`permissions.ts`), creating a structurally privileged phase where the safety guarantees documented in Section 5 do not yet apply.

This pattern reveals that the permission pipeline depicts a spatial ordering of safety checks but does not capture the temporal dimension: specifically, when during session initialization each mechanism becomes active. The initialization sequence (extension loading, then trust dialog, then permission enforcement) creates

³The two pre-trust ordering vulnerabilities are CVE-2025-59536 (CVSS 8.7) and CVE-2026-21852 (CVSS 5.3) (Donenfeld and Vanunu, 2026), discovered by Check Point Research. CVE-2025-54794 and CVE-2025-54795 (Beber, 2025) exploit path validation and command parsing flaws elsewhere in the permission pipeline, separately. All four were patched within weeks of disclosure.

a window where the extensibility architecture (Section 6) operates before the safety architecture (Section 5) is fully engaged. This finding refines the extensibility-versus-simplicity tension by adding a security dimension: extensibility creates attack surface not only through combinatorial complexity but through initialization ordering.

Context efficiency vs. transparency. The five-layer compaction pipeline achieves effective context management, but compression is largely invisible to the user. When budget reduction replaces a long tool output with a reference, when context collapse substitutes messages with a summary (described in the source as “a read-time projection over the REPL’s full history”), or when snip trims older history, the user has no easy way to inspect what was lost. The cache-aware behavior of microcompact adds further opacity, as compression decisions are influenced by prompt caching in ways not visible to the user.

Simplicity vs. extensibility. The four extension mechanisms enable rich customization but create combinatorial interactions. A plugin contributes a PreToolUse hook that modifies tool inputs. The auto-mode classifier reads cached CLAUDE.md content. Path-scoped rules load lazily when new directories are read, potentially changing classifier behavior mid-conversation. The permission handler’s four branches interact with the hook pipeline at multiple points. These cross-cutting concerns create emergent behaviors difficult to predict from any single configuration file.

11.4 Empirical Predictions and Early Signals

The architectural properties documented in this paper generate testable predictions about code quality outcomes not derivable from the source code alone. The bounded context window (Section 7) prevents the agent from maintaining simultaneous awareness of the full codebase: the five-layer compaction pipeline preserves useful information but introduces lossy compression at each stage. This makes it architecturally predicted that agent-generated code will exhibit higher rates of pattern duplication and convention violation than code produced with full codebase visibility. Subagent isolation (Section 8), where each subagent operates in its own context window with an independently assembled tool pool, compounds the effect: parallel agents can independently re-implement solutions that already exist elsewhere. The design philosophy of Section 11.1 trusts the model to make good local decisions, but good local decisions can produce poor global outcomes when the model lacks global context.

Published empirical work on architecturally similar tools provides data consistent with these predictions. A causal analysis of Cursor adoption across 807 repositories (He et al., 2025) found a statistically significant increase in code complexity, with an initial velocity spike that dissipated to baseline by month three; rising complexity was associated with a proportional decrease in future development velocity, suggesting that the gains are self-cancelling⁴. A large-scale audit of 304,000 AI-authored commits across 6,275 repositories (Liu et al., 2026) found measurable technical debt, with approximately one-quarter of AI-introduced issues persisting to the latest revision and security-related issues persisting at a substantially higher rate. While these studies target adjacent systems, the architectural parallels (bounded context, tool-use loops, single-pass generation) suggest the findings are relevant to the design analyzed here.

Claude Code’s context management pipeline is specifically designed to mitigate these effects: graduated compression preserves the most recent and most relevant context, cache-aware compaction avoids invalidating prompt caches during compression, read-time projection maintains full history for reconstruction while presenting a compressed view to the model, and subagent summary isolation prevents exploratory noise from accumulating in the parent context. Whether these mechanisms are sufficient to overcome the structural limitations of bounded context is a directly measurable empirical question that the source-level analysis in this paper cannot resolve.

11.5 Limitations

Beyond the methodological limitations in Section B.3, several analytical constraints apply. The memoized context assembly functions (`getSystemContext()` and `getUserContext()`) both use `lodash memoize` at con-

⁴Complexity +40.7% ($p < 0.001$); velocity spike +281% in month one, baseline by month three.

`text.ts`) mean that git status and CLAUDE.md content are cached rather than recomputed on every turn. Dynamic changes during a conversation may not be reflected immediately, though compaction can clear caches and lazy-loaded path-scoped rules provide a partial counter-mechanism.

Feature flags create build-time variability. In a build where `TRANSCRIPT_CLASSIFIER` is false, the entire auto-mode classifier is eliminated. Feature-gated modules use dynamic `require()` rather than static `import` (e.g., `query.ts` for context collapse), because `feature()` only works in if/ternary conditions due to a bun:bundle tree-shaking constraint. Different build targets may produce functionally different applications.

11.6 Emerging Directions

Several aspects of the implementation relate to broader design questions. Longer context windows would reduce compaction pressure, potentially simplifying the graduated pipeline. Multi-modal tools (screenshots, diagrams, UI previews) would expand the tool surface and create new context challenges. Formal verification of permission properties (for example, proving that deny rules always take precedence, that sandboxed commands cannot escape isolation, or that resumed sessions cannot inherit stale permissions) would provide stronger safety guarantees.

Architectural decoupling. The tightly coupled local architecture analyzed here is one point on a spectrum that is already evolving. Anthropic’s own Managed Agents work (Martin et al., 2026) describes virtualizing the components of an agent (session, harness, sandbox) so that “each became an interface that made few assumptions about the others, and each could fail or be replaced independently”, drawing an explicit analogy to how operating systems virtualized hardware into processes and files. The Harness Design essay (Rajasekaran, 2026) makes a similar point from a different angle, observing that “the space of interesting harness combinations doesn’t shrink as models improve”; instead, “it moves”. The architecture documented in this paper should therefore be read as a snapshot of a co-evolving system rather than a fixed optimum.

Memory as a first-class subsystem. The memory survey of Hu et al. (2025) argues that agent memory is becoming a distinct cognitive substrate rather than a side effect of context window management, and identifies automated memory management, RL-driven memory, and trustworthy memory (privacy, explainability, and hallucination robustness) as open frontiers. Claude Code today exposes the factual tier (CLAUDE.md, auto memory) and the working tier (the conversation window); the experiential tier (accumulated, automatically curated playbooks of strategies learned from past sessions) is the natural next step, and the context-engineering literature (Zhang et al., 2025a) has started to provide mechanisms for that accumulation.

Observability and silent failure. Industry surveys suggest that the dominant failure mode of deployed agents is not crashes but silent mistakes. Bessemer’s 2026 infrastructure report (Wade et al., 2026) estimates that “78% of AI failures are invisible”, while LangChain’s 1,340-respondent state-of-agent-engineering survey (LangChain, 2026) identifies quality, not cost, as the top barrier to production use and finds a wide gap between observability (nearly 89% adoption) and offline evaluation (52.4%). The architecture analyzed here gives operators visibility into tool calls, hooks, and session transcripts; closing the evaluation gap likely requires additional scaffolding (generator-evaluator separation, sprint contracts, and post-hoc checks of the kind discussed in Rajasekaran (2026)) rather than model improvements alone.

Governance. Broader governance trends will constrain the design space as agents become more autonomous. The International AI Safety Report (Bengio et al., 2026) warns that “AI agents pose heightened risks because they act autonomously, making it harder for humans to intervene before failures cause harm,” and the MIT AI Agent Index (Staufer et al., 2026) finds that only 13.3% of indexed agentic systems publish agent-specific safety cards. Emerging regulatory frameworks, notably the EU AI Act (fully applicable August 2026) and evolving copyright jurisprudence around AI-generated code, may impose external constraints on logging, transparency, and human oversight that shape how coding agent architectures evolve.

Proactive architectures. The feature-gated KAIROS system illustrates how this architecture may evolve beyond reactive tool use. KAIROS implements a persistent background agent with tick-based heartbeats: when no user messages are pending, the system injects periodic `<tick>` prompts, and the model decides

whether to act or sleep. The design directly addresses a documented tension: proactive AI assistants increase task completion by 12 to 18% but reduce user preference at high frequencies (Chen et al., 2025). KAIROS resolves this through terminal focus awareness (maximizing autonomous action when the user is away, increasing collaboration when present) and economic throttling via **SleepTool** (each wake-up costs an API call; the prompt cache expires after five minutes of inactivity, making sleep/wake an explicit cost optimization). This binding of proactivity to both user presence and token economics is uncommon among production agent systems, though KAIROS cannot be confirmed as active in production builds.

11.7 Recurring Design Choices

Reading the six subsystem analyses together reveals three cross-cutting design commitments that recur across otherwise independent components.

Graduated layering over monolithic mechanisms. Safety, context management, and extensibility all use graduated stacks of independent mechanisms rather than single integrated solutions. The permission architecture layers seven stages from tool pre-filtering through deny-first rules, permission modes, the auto-mode classifier, shell sandboxing, non-restoration on resume, and hook interception. Context management layers five compaction stages, lazy-loaded CLAUDE.md files, deferred tool schemas, and summary-only subagent returns. Extensibility layers four mechanisms (MCP servers, plugins, skills, and hooks) at different context costs (Section 6). In each case, the design trades simplicity and debuggability for defense in depth, accepting that the interaction between layers can produce emergent behaviors difficult to predict from any single configuration.

Append-only designs that favor auditability over query power. Session transcripts are append-only JSONL files with read-time chain patching; permissions are not restored across session boundaries; context compaction applies read-time projections over a full history rather than destructive edits. This commitment recurs because it preserves the ability to resume, fork, and audit sessions without modifying previously written state. The cost is that richer structured queries (“show me all tool calls that modified file X across sessions”) require post-hoc reconstruction rather than direct lookup.

Model judgment within a deterministic harness. Across all subsystems, the architecture trusts the model’s judgment within a rich deterministic harness rather than constraining its choices. The estimated 1.6% decision-logic ratio captures this quantitatively: the harness creates conditions (tool routing, permission enforcement, context assembly, recovery logic) under which the model can decide well. Hierarchical permissions preserve safety invariants across agent boundaries, and `assembleToolPool()` merges built-in and MCP tools into a single unified interface, but the model retains full latitude over which tools to invoke and in what order. The trade-off is that good local decisions can produce poor global outcomes when bounded context prevents global awareness, as the empirical predictions of Section 11.4 document.

12 Future Directions

Section 11 read the architecture documented in Sections 3 to 9 as a coherent design point and surfaced the tensions, trade-offs, and near-horizon directions that design point implies. This section steps beyond the architecture itself to record six open questions that Section 11.6 partially names and that a growing external literature has sharpened enough to state concretely. The six span the paper’s five-value framework (Section 2.1) and its evaluative lens (Section 2.4): external governance constraints on the Authority hierarchy (Section 12.5); the observability–evaluation gap on the Safety side (Section 12.1); cross-session persistence of state and relationship on the Reliability side (Section 12.2); four extensions of the Capability frontier (Section 12.3); horizon scaling as a distinct axis of Reliable Execution beyond cross-session continuity (Section 12.4); and the evaluative lens of Section 2.4 reframed as a design question rather than a diagnostic one (Section 12.6). Consistent with Section 11.6’s framing, each question is posed in the form *whether/how/which*; specific mechanism choices are named when the cited sources name them and left open otherwise.

12.1 Silent Failure and the Observability–Evaluation Gap

Whether the observability–evaluation adoption gap reported in [Section 11.6](#) reflects a missing tooling layer, a missing evaluation interface inside the harness, or a model-capability ceiling is not resolved by the sources cited there. How the silent-mistake failure mode noted in that paragraph should be surfaced is therefore an architectural question for the harness rather than a capability question for the model. Recent empirical work characterises the gap at several resolutions. [Cemri et al. \(2025\)](#) catalogue fourteen failure modes spanning system-design issues, inter-agent misalignment, and task verification; [Pathak et al. \(2025\)](#) build a benchmark of agent trajectories specifically for anomaly detection in traces; [Yao et al. \(2024\)](#) expose consistency gaps via the pass^k metric (the probability that all k independent trials succeed); and [Kapoor et al. \(2024\)](#) argue that current agent benchmarks lack holdouts and cost controls, limiting what observability can actually diagnose.

Against the permission pipeline and tool-orchestration layers analysed in [Sections 4](#) and [5](#), two architectural questions remain open. First, whether the scaffolding the paper cites from [Rajasekaran \(2026\)](#) (generator–evaluator separation, sprint contracts, post-hoc checks, building on [Madaan et al. \(2023\)](#)’s self-refine pattern) belongs inside the harness (e.g., as additional hook events alongside the 27 documented in [Section 6](#)) or outside it as a separate evaluation layer is not settled by the cited sources. Second, whether the existing hook pipeline of [Section 6](#) can host such scaffolding within its current context-cost envelope is a further open question. The observation that closing this gap “likely requires additional scaffolding ... rather than model improvements alone” ([Section 11.6](#)) locates the open work at the harness layer.

12.2 Persistence: Memory and Longitudinal Colleague Relationships

Whether agent state and the human–agent working relationship should persist across sessions, and in what form, is treated by the paper at two distinct layers today. [Section 7](#) documents the four-level CLAUDE.md hierarchy and auto memory; [Section 9](#) documents mostly-append-only JSONL transcripts (with explicit cleanup rewrites as an exception) whose session-scoped permissions resume does not restore. What belongs between these two layers (durable state that is neither a static instruction nor a single session’s transcript) is an open design question. [Hu et al. \(2025\)](#) and [Zhang et al. \(2025a\)](#), already cited in [Section 11.6](#), motivate an accumulating layer. [Packer et al. \(2023\)](#) reframes the LLM as an operating system with paged memory; [Chhikara et al. \(2025\)](#) builds a production-oriented memory store that survives restarts, while [Xu et al. \(2025\)](#) proposes a research agentic-memory design; [Wang et al. \(2024c\)](#) captures reusable procedural traces; [Shinn et al. \(2023\)](#) accumulates self-reflection traces via verbal reinforcement across attempts; and surveys by [Zhang et al. \(2025b\)](#) and [Huang et al. \(2026\)](#) map candidate mechanisms.

The same persistence question recurs on the human side. [Section 11.6](#) already cites longitudinal autonomy evidence ([Huang et al. \(2025\)](#), [McCain et al. \(2026\)](#)); [Dell’Acqua et al. \(2025\)](#)’s field experiment with 776 Procter & Gamble professionals, together with longitudinal and organisational studies of Copilot rollouts ([Stray et al., 2025](#)) and AI-teamwork trajectories ([Xiao et al., 2025](#)), report shifts in human–AI work dynamics as collaboration accumulates. [Wang et al. \(2023\)](#) illustrates an embodied agent that accumulates a skill library across tasks; [Mollick \(2024\)](#) frames the human–AI working relationship as co-intelligence.

Whether a single substrate can carry both a user’s personal instruction hierarchy and a shared organisational context while preserving the file-based transparency of CLAUDE.md that [Section 7](#) documents is an open architectural question. How session-scoped permissions interact with such a substrate, without reintroducing the resume-restoration concern that [Section 9](#) closes as a deliberate safety choice, is a further open question.

12.3 Harness Boundary Evolution: Where, When, What, and with Whom the Agent Acts

[Section 11.6](#) cites [Rajasekaran \(2026\)](#)’s observation that “the space of interesting harness combinations doesn’t shrink as models improve; it moves.” Whether that movement will be most pronounced in *where* the harness runs, *when* it acts, *what* it acts on, or *with whom* it coordinates is not resolved by the source-level analysis in [Sections 3](#) to [9](#). Each of the four has an active research literature that the paper touches only in passing.

Where. [Martin et al. \(2026\)](#)’s Managed Agents design virtualizes session, harness, and sandbox into independently replaceable interfaces, extending the virtual-memory analogy that [Packer et al. \(2023\)](#) applies

to context-window management and that Karpathy (2023) popularizes more broadly; Khattab et al. (2023) treats the harness itself as a compile target.

When. Section 11.6 already introduces KAIROS as a feature-gated illustration, motivated by the +12%–18% task-pass gain that Chen et al. (2025) report and the sharp preference penalty (47% vs. 80–90%) restricted to the high-frequency *Persistent Suggest* variant. Liu et al. (2025), Pu et al. (2025), and Lee et al. (2025) extend the proactivity design space across programming and ambient-interface settings; Pasternak et al. (2025) and Sun et al. (2025) introduce benchmarks and training regimes aimed at sharpening it, and Deng et al. (2025) surveys the broader landscape.

What. Vision-language-action work extends the harness beyond textual tool returns: Brohan et al. (2024) and Black et al. (2024) train VLA policies that execute physical actions, and Ahn et al. (2022) grounds plans in robot affordances; industry systems such as Figure AI (2025) and Bjorck et al. (2025) push similar ideas into humanoid control. These systems face the reversibility-weighted risk principle (Table 1) at a cost asymmetry that the principle names but does not quantify for non-textual actions. *With whom.* Role-differentiated multi-agent systems (Hong et al. (2023), Li et al. (2023), Chen et al. (2023), Qian et al. (2024)) compose agents with distinct responsibilities; multi-agent debate (Du et al., 2024; Liang et al., 2024) and graph-structured workflows (Zhuge et al., 2024) explore alternatives to the parent/subagent pattern of Section 8; Guo et al. (2024) surveys this space.

Whether a single harness architecture can span all four extensions, or whether the “harness combinations” Rajasekaran (2026) describes will fragment into specialised stacks, is an open design question. The *when*-extension directly continues the Capability-versus-Adaptability tension in Table 4. The *with-whom*-extension partially maps onto Capability-versus-Reliability but raises cross-agent consistency concerns that Table 4 does not itself cover. The *where*- and *what*-extensions raise further questions the paper’s current subsystem boundaries do not cover: which governance obligations attach when harness components become hosted services (Section 12.5), and how reversibility-weighted risk (Table 1) scales to physical rather than textual effects. How these extensions compose across axes, rather than within any one, is not something the paper’s single-subsystem analyses can resolve.

12.4 Horizon Scaling: From Session to Scientific Program

Section 2.1 defines Reliable Execution as spanning “both single-turn correctness and long-horizon dependability.” How the architecture documented in Sections 3, 4 and 7 to 9 (whose primary units are the turn, the session, and the sub-agent) continues to support long-horizon dependability as autonomous work extends beyond a single session is an open question. A growing literature targets this regime. Lu et al. (2024) present an end-to-end autonomous research pipeline producing draft manuscripts; Beel et al. (2025) provide an independent SIGIR Forum evaluation of that pipeline, characterising what “autonomous research” currently delivers and where it falls short. Gottweis et al. (2025) develop a multi-agent hypothesis-generation system that runs across days rather than turns, and Novikov et al. (2025) pursue algorithmic discovery over timescales that previously took human experts weeks. Kwa et al.’s METR study measures the task duration at which frontier agents succeed with fixed reliability (the 50%-time horizon) and how that horizon has evolved across model generations, giving an empirical frame for this scaling question.

Against the paper’s analysis, long-horizon deployment tests whether the context-management pipeline of Section 7, the last-assistant-text return policy of Section 8, and the append-only persistence of Section 9 remain sufficient when sessions compose into multi-session programs. Section 11.4 already frames this as “a directly measurable empirical question” that source-level analysis cannot resolve. Horizon scaling restates that question at the scale of weeks: whether the harness layer alone closes the gap, whether a cross-session memory substrate (Section 12.2) is required, or whether horizon-scale work demands coordination primitives beyond session, sub-agent, and memory, is not something the paper’s session-scoped analyses can settle.

12.5 Governance and Oversight at Scale

Emerging AI regulation adds an external constraint on the architectures that implement the Authority hierarchy of Anthropic, operators, and users documented in Section 2.1. Which logging, transparency, and

human-oversight affordances coding-agent architectures should expose under that external constraint remains an open design question. The European Commission’s GPAI Code of Practice (European Commission, 2025a) and implementation guidelines (European Commission, 2025b) detail the general-purpose AI obligations that accompany the EU AI Act’s full applicability in August 2026; the MIT AI Agent Index (Staufer et al., 2026) and the International AI Safety Report (Bengio et al., 2026), already cited in Section 11.6, motivate the disclosure and oversight side of this constraint. The Bartz v. Anthropic ruling (bar, 2025) adds an input-side constraint on training-data sourcing (lawful acquisition of copyrighted works), distinct from the output-side copyright questions about AI-generated code that emerging cases address separately. An OECD report on AI governance frameworks (OECD, 2025) and an early analysis of compliance obligations for agent providers by Nannini et al. (2026) sketch what regulator-facing interfaces might look like without prescribing specifics.

Read against the permission pipeline analyzed in Section 5, two properties of the current architecture are open under this constraint. First, the deny-first evaluation the paper documents is internally auditable through session transcripts (Section 9) but not yet externally auditable in the forms that emerging frameworks such as the GPAI Code of Practice (European Commission, 2025a) contemplate. Second, whether the *values-over-rules* principle, which the paper pairs with deterministic guardrails, admits the kind of explicit rule articulation that compliance review may call for is a further open question. Both properties lie within the harness rather than the model, which is where future architectures may need to expose new interfaces.

12.6 The Evaluative Lens Revisited: Long-Term Human Capability

Section 2.4 introduces long-term human-capability preservation as an analytical lens rather than a co-equal design value; Sections 11.2 and 11.4 extend the lens with external evidence (perceived-versus-measured productivity, comprehension loss, complexity accrual, technical-debt persistence, neural-connectivity persistence, early-career hiring decline), and Section 14 pivots: “Future systems could treat that sustainability gap as a first-class design problem, not a downstream evaluation metric.” Whether that pivot is possible, and what architectural mechanisms a first-class treatment would require, is the last of the open questions this section records.

Two sub-questions separate the measurement gap from the design gap. First, whether the empirical claims that motivate the lens are measurable at session granularity. The existing citations operate at session to multi-month scales (Becker et al. (2025)’s 16-developer RCT, Shen and Tamkin (2026)’s comprehension-test comparison, Kosmyrna et al. (2025)’s EEG study, He et al. (2025)’s 807-repository causal analysis, Liu et al. (2026)’s 304,000-commit audit, Rak (2025)’s hiring series), but the harness documented in Sections 3, 4 and 7 exposes no per-session signal for comprehension or convention drift. Related work on programmer interaction modes (Barke et al., 2023) and AI-induced code-security regressions (Perry et al., 2023) sketches session-granularity measurement, and Aiersilan (2026) proposes a protocol for session-level cognitive-offloading probes. Second, whether architecture can respond to such measurements once they exist (an analogue of the generator–evaluator separation (Rajasekaran, 2026) applied to the human loop, comprehension-preserving surfaces, or mechanisms not yet named) is the design-gap question Section 14 poses. The paper takes no position on which mechanism class is appropriate, and whether the harness documented here is even the right locus for that action (as opposed to the IDE, the organisation, or the human development loop) is a question the architectural analysis cannot adjudicate; the related work surveyed in Section 13 and the sustainability pivot of Section 14 mark where this paper leaves the question.

13 Related Work

13.1 Coding Agent Taxonomy

AI coding tools can be organized by the degree of autonomous action they support (Table 5). Inline completion tools such as GitHub Copilot (Chen et al., 2021) suggest code fragments within the editor without autonomous action. Chat-integrated products including Cursor and Windsurf add conversational interaction and multi-file edits but remain coupled to the IDE environment. Agentic CLI tools, including Claude Code, OpenAI’s Codex CLI, and Aider (Gauthier, 2024), operate from the command line and can autonomously execute shell commands, read and write files, and iterate on outputs within a single request. Fully autonomous systems

Table 5 AI coding tool categories by degree of autonomous action.

| Category | Examples | Pattern |
|-------------------|-------------------------------|---------------------|
| Inline completion | Copilot, Tabnine | Editor plugin |
| Chat-integrated | Cursor, Windsurf, Cody | IDE-coupled product |
| Agentic CLI | Claude Code, Codex CLI, Aider | Tool-use loop |
| Fully autonomous | Devin, SWE-Agent, OpenHands | Sandbox + planning |

like Devin, SWE-Agent (Yang et al., 2024), and OpenHands (Wang et al., 2024b) aim for minimal human supervision, often in sandboxed cloud environments.

Claude Code shares features with higher-autonomy agents (auto-mode classifier, background agent execution, remote environments) but retains interactive approval by default. Evaluation benchmarks such as SWE-Bench (Jimenez et al., 2023) and HumanEval (Chen et al., 2021) have driven much of the academic focus on coding agents. This paper examines Claude Code’s internal architecture from source code.

13.2 Agent Architecture Patterns

Claude Code’s core loop follows the ReAct pattern (Yao et al., 2022): the model generates reasoning and tool invocations, the harness executes actions, and results feed the next iteration. Toolformer (Schick et al., 2023) demonstrated that language models can learn to use tools; Claude Code uses up to 54 built-in tools and a layered permission system. The broader design space has been mapped by several surveys. Weng (2023) offered the now-standard decomposition into planning, memory, and tool use, and Wang et al. (2024a) catalogued early autonomous-agent work. Xu (2026) frames the field around three recurring trade-offs (autonomy vs. controllability, latency vs. accuracy, capability vs. reliability) that recur throughout our analysis, and Hu et al. (2024) casts agent design itself as a search problem over components, algorithms, and evaluation functions. This paper characterizes one specific point in that space.

Multi-agent orchestration frameworks such as AutoGen (Wu et al., 2024), LangChain, and CrewAI provide conversation-based agent coordination. Claude Code’s subagent delegation (Section 8) includes permission override precedence, two-level permission scoping, and separate transcript files for each subagent. LATS (Zhou et al., 2023) unifies reasoning, acting, and planning in a tree-search framework; Claude Code’s plan permission mode implements a simpler plan-then-execute approach.

Practitioner writing has converged on a handful of recurring patterns that Claude Code’s architecture instantiates. Anthropic’s own “Building Effective Agents” (Schluntz and Zhang, 2024) distinguishes agents from workflows and argues for simple composable patterns over heavy frameworks. Martin (2026) synthesizes seven patterns observed in production systems, including giving agents filesystem and shell access as a general-purpose action layer, and discovering actions on demand rather than loading every tool schema upfront. Chase (2025) observes that Claude Code’s planning tool is “basically a no-op” whose value lies in keeping the agent on track rather than in performing any external computation. Wang (2025) argues that authority is the element academic frameworks most often leave out, calling trust “the most overlooked element” in production agent design, a gap the permission analysis in Section 5 attempts to close. Huyen (2025) makes the compound-error concern concrete: at 95% per-step accuracy, a 100-step task succeeds only 0.6% of the time, which motivates the per-step verification patterns we trace in Section 4 and Section 5.

Context management. Table 6 presents a design-space taxonomy of context management approaches. Claude Code’s five-layer compaction pipeline applies multiple strategies at different granularities before escalating, with cache-aware compression and virtual-view-on-read semantics. Zhang et al. (2025a) characterizes two failure modes that this design mitigates (summarization that drops domain details, and detail loss from iterative context rewriting), and instead proposes treating context as an “evolving playbook” that accumulates strategies over time. Claude Code’s approach is consistent with that framing, since the CLAUDE.md hierarchy accumulates structured instructions rather than repeatedly summarizing them. Hu et al. (2025) distinguishes context engineering from agent memory: context engineering handles transient assembly, while memory covers persistent factual knowledge and experiential traces. Claude Code’s architecture separates the two in the

Table 6 Design space of context management approaches in LLM-based tools.

| Approach | Mechanism | Granularity |
|----------------------|----------------------------|-------------|
| Simple truncation | Drop oldest messages | Coarse |
| Sliding window | Fixed-size recent history | Medium |
| RAG | Retrieve relevant snippets | Fine |
| Single summarization | One-pass compress | Coarse |
| Graduated compaction | Multi-layer pipeline | Very fine |

same way, pairing a compaction pipeline with a file-based memory hierarchy.

Safety and permissions. Production coding agents adopt safety architectures that vary along three axes: *approval model* (per-action prompting, classifier-mediated automation, or no prompting with post-hoc review), *isolation boundary* (OS-level container, filesystem sandbox, permission-scoped tool pool, or none), and *recovery mechanism* (version-control rollback, session-scoped permission reset, or checkpoint-based rewind). SWE-Agent and OpenHands (Yang et al., 2024; Wang et al., 2024b) rely primarily on Docker container isolation, providing environment-level sandboxing that constrains all agent actions. Codex CLI supports sandbox modes and approval policies for shell commands. Aider (Gauthier, 2024) uses Git as its primary safety mechanism, making all changes reversible through version control. Claude Code combines per-action deny-first rules, an ML-based classifier for automated approval, optional shell sandboxing, and session-scoped permission non-restoration, layering multiple mechanisms rather than relying on a single isolation boundary.

Protocols and extensibility. The Model Context Protocol that Claude Code uses as its primary external tool integration has become a de facto standard with a substantial ecosystem and a corresponding attack surface. Hou et al. (2025) catalogues thousands of community-developed MCP servers across 26 major directories and organizes MCP-specific threats into four attacker categories and sixteen scenarios, including tool poisoning, rug pulls, and cross-server shadowing. The permission and deny-rule machinery analyzed in Section 5 and the pre-filtering step in Section 6.2 can be read as the runtime side of the mitigations that survey calls for.

Software architecture. Layered architecture patterns (Garlan et al., 1993) inform our five-layer decomposition. Role-based access control models (Sandhu et al., 2002) provide theory for the permission mode system. Browser sandboxing (Reis and Gribble, 2009) is a similar per-process isolation approach. Multi-agent system theory (Wooldridge, 2009) helps explain subagent delegation.

Positioning. Prior work on coding agents has focused on benchmarks (how well agents solve tasks), frameworks (how to compose agents), and products (what users can do). This paper contributes a source-grounded design-space analysis of a production coding agent, using source-level analysis and architectural comparison to surface design choices and trade-offs. It draws on the software architecture case study tradition (Garlan et al., 1993) but applies it to an LLM-based agent by systematically identifying design questions, mapping alternatives, and contrasting Claude Code’s choices with those of OpenClaw, an independent AI agent system operating from a different deployment context.

14 Conclusion

This paper shows that production coding agents can be understood as answers to a recurring set of design questions: where reasoning sits relative to the harness, how execution, safety, extensibility, context, delegation, and persistence are organized, and which trade-offs those choices encode. Claude Code occupies a clear design point within that space. It gives the model broad local autonomy while surrounding it with a dense deterministic harness for permissioning, tool routing, context compaction, extensibility, and session recovery. Read through the five values and thirteen design principles identified in Section 2, these choices are coherent rather than ad hoc: the system consistently prioritizes human decision authority, safety, reliable execution, capability amplification, and contextual adaptability.

The OpenClaw comparison sharpens the main architectural finding by showing that the same design questions recur in different agent systems but produce different answers. Where Claude Code invests in per-action safety classification and graduated context compression within a CLI harness, OpenClaw invests in perimeter-level access control and structured long-term memory within a multi-channel gateway. The two systems can even compose: OpenClaw hosts Claude Code as an external harness via ACP. For agent builders, the most consequential open question is therefore not how to add more autonomy, but how to preserve human capability over time. As the evaluative lens in [Section 2.4](#), the analysis in [Section 11](#), and the open questions surveyed in [Section 12](#) document, the architecture provides limited mechanisms that explicitly preserve long-term human understanding, codebase coherence, or the developer pipeline. Future systems could treat that sustainability gap as a first-class design problem, not a downstream evaluation metric.

References

- Bartz v. Anthropic PBC, no. 3:24-cv-05417-WHA. U.S. District Court for the Northern District of California, Order on Motion for Summary Judgment (June 23, 2025), Alsup, J. Court docket: <https://www.courtlistener.com/docket/69058235/bartz-v-anthropic-pbc/>, 2025.
- Adversa.ai. Critical Claude Code vulnerability: Deny rules silently bypassed because security checks cost too many tokens. <https://adversa.ai/blog/claude-code-security-bypass-deny-rules-disabled/>, 2026.
- Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.
- Aizierjiang Aiersilan. The vibe-check protocol: Quantifying cognitive offloading in ai programming. *arXiv preprint arXiv:2601.02410*, 2026.
- Anthropic. Our framework for developing safe and trustworthy agents. <https://www.anthropic.com/news/our-framework-for-developing-safe-and-trustworthy-agents>, 2025a.
- Anthropic. Orchestrate teams of Claude Code sessions. <https://code.claude.com/docs/en/agent-teams>, 2025b.
- Anthropic. Claude code overview. <https://code.claude.com/docs>, 2026a. Official Claude Code documentation. Accessed April 12, 2026.
- Anthropic. Claude’s constitution. <https://anthropic.com/constitution>, 2026b.
- Anthropic. Anthropic on github. <https://github.com/anthropics>, 2026c. Verified GitHub organization page. Accessed April 12, 2026.
- Anthropic. How Claude Code works. <https://code.claude.com/docs/en/how-claude-code-works>, 2026d.
- Shraddha Barke, Michael B James, and Nadia Polikarpova. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):85–111, 2023.
- Elad Beber. InversePrompt: Turning claude against itself, one prompt at a time. <https://cymulate.com/blog/cve-2025-547954-54795-claude-inverseprompt/>, 2025. CVE-2025-54794, CVE-2025-54795; updated April 6, 2026.
- Joel Becker, Nate Rush, Elizabeth Barnes, and David Rein. Measuring the impact of early-2025 ai on experienced open-source developer productivity. *arXiv preprint arXiv:2507.09089*, 2025.
- Joeran Beel, Min-Yen Kan, and Moritz Baumgart. Evaluating sakana’s ai scientist: Bold claims, mixed results, and a promising future? In *ACM SIGIR Forum*, volume 59, pages 1–20. ACM New York, NY, USA, 2025.
- Yoshua Bengio, Stephen Clare, Carina Prunkl, Maksym Andriushchenko, Ben Bucknall, Malcolm Murray, Rishi Bommasani, Stephen Casper, Tom Davidson, Raymond Douglas, et al. International ai safety report 2026. *arXiv preprint arXiv:2602.21012*, 2026.
- Johan Bjorck, Fernando Castañeda, Nikita Cherniadev, Xingye Da, Runyu Ding, Linxi Fan, Yu Fang, Dieter Fox, Fengyuan Hu, Spencer Huang, et al. Gr00t n1: An open foundation model for generalist humanoid robots. *arXiv preprint arXiv:2503.14734*, 2025.
- Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, et al. π_0 : A vision-language-action flow model for general robot control. *arXiv preprint arXiv:2410.24164*, 2024.

- Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control, 2023. URL <https://arxiv.org/abs/2307.15818>, 1:2, 2024.
- Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent llm systems fail? *arXiv preprint arXiv:2503.13657*, 2025.
- Harrison Chase. Deep agents. LangChain Blog, <https://blog.langchain.com/deep-agents/>, 2025.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- Valerie Chen, Alan Zhu, Sebastian Zhao, Hussein Mozannar, David Sontag, and Ameet Talwalkar. Need help? designing proactive ai assistants for programming. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, pages 1–18, 2025.
- Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *The Twelfth International Conference on Learning Representations*, 2023.
- Boris Cherny and Cat Wu. Claude code: Anthropic’s agent in your terminal. Latent Space podcast, <https://www.latent.space/p/claude-code>, 2025.
- Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.
- Cursor. Cursor: The best way to code with AI. <https://cursor.com/>, 2026. Official product website. Accessed April 12, 2026.
- Fabrizio Dell’Acqua, Charles Ayoubi, Hila Lifshitz, Raffaella Sadun, Ethan Mollick, Lilach Mollick, Yi Han, Jeff Goldman, Hari Nair, Stewart Taub, et al. The cybernetic teammate: A field experiment on generative ai reshaping teamwork and expertise. Technical report, National Bureau of Economic Research, 2025.
- Yang Deng, Lizi Liao, Wenqiang Lei, Grace Hui Yang, Wai Lam, and Tat-Seng Chua. Proactive conversational ai: A comprehensive survey of advancements and opportunities. *ACM Transactions on Information Systems*, 43(3):1–45, 2025.
- Aviv Donenfeld and Oded Vanunu. Caught in the hook: RCE and API token exfiltration through Claude Code project files. <https://research.checkpoint.com/2026/rce-and-api-token-exfiltration-through-claude-code-project-files-cve-2025-59536/>, 2026. CVE-2025-59536 (CVSS 8.7), CVE-2026-21852 (CVSS 5.3).
- Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *Forty-first international conference on machine learning*, 2024.
- David Dworken and Oliver Weller-Davies. Beyond permission prompts: Making Claude Code more secure and autonomous. Anthropic Engineering, <https://www.anthropic.com/engineering/claude-code-sandboxing>, 2025.
- European Commission. General-purpose AI code of practice. <https://digital-strategy.ec.europa.eu/en/policies/contents-code-gpai>, 2025a. Official EU Commission publication, 10 July 2025.
- European Commission. Guidelines on the scope of obligations for providers of general-purpose AI models under the AI act. <https://digital-strategy.ec.europa.eu/en/library/guidelines-scope-obligations-providers-general-purpose-ai-models-under-ai-act>, 2025b. Official EU Commission guideline document.
- Figure AI. Helix: A vision-language-action model for generalist humanoid control. <https://www.figure.ai/news/helix>, 2025. Figure AI technical blog.
- David Garlan, Mary Shaw, et al. An introduction to software architecture. *Advances in software engineering and knowledge engineering*, 1(3.4), 1993.
- Paul Gauthier. Aider: AI pair programming in your terminal, 2024. <https://github.com/Aider-AI/aider>. Open-source software, <https://aider.chat>.

- Juraj Gottweis, Wei-Hung Weng, Alexander Daryin, Tao Tu, Anil Palepu, Petar Sirkovic, Artiom Myaskovsky, Felix Weissenberger, Keran Rong, Ryutaro Tanno, et al. Towards an ai co-scientist. *arXiv preprint arXiv:2502.18864*, 2025.
- Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680*, 2024.
- Hao He, Courtney Miller, Shyam Agarwal, Christian Kästner, and Bogdan Vasilescu. Speed at the cost of quality: How cursor ai increases short-term velocity and long-term complexity in open-source projects. *arXiv preprint arXiv:2511.04427*, 2025.
- Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*, 2023.
- Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. Model context protocol (mcp): Landscape, security threats, and future research directions. *ACM Transactions on Software Engineering and Methodology*, 2025.
- Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.
- Yuyang Hu, Shichun Liu, Yanwei Yue, Guibin Zhang, Boyang Liu, Fangyi Zhu, Jiahang Lin, Honglin Guo, Shihan Dou, Zhiheng Xi, et al. Memory in the age of ai agents. *arXiv preprint arXiv:2512.13564*, 2025.
- Saffron Huang, Bryan Seethor, Esin Durmus, Kunal Handa, Miles McCain, Michael Stern, and Deep Ganguli. How AI is transforming work at Anthropic. Anthropic Research Blog, <https://anthropic.com/research/how-ai-is-transforming-work-at-anthropic>, 2025.
- Wei-Chieh Huang, Weizhi Zhang, Yueqing Liang, Yuanchen Bei, Yankai Chen, Tao Feng, Xinyu Pan, Zhen Tan, Yu Wang, Tianxin Wei, et al. Rethinking memory mechanisms of foundation agents in the second half. *arXiv preprint arXiv:2602.06052*, 2026.
- John Hughes. Claude Code auto mode: A safer way to skip permissions. Anthropic Engineering, <https://www.anthropic.com/engineering/claude-code-auto-mode>, 2026.
- Chip Huyen. Agents. <https://huyenchip.com/2025/01/07/agents.html>, 2025.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.
- Sayash Kapoor, Benedikt Stroebel, Zachary S Siegel, Nitya Nadgir, and Arvind Narayanan. Ai agents that matter. *arXiv preprint arXiv:2407.01502*, 2024.
- Andrej Karpathy. [1hr talk] intro to large language models. YouTube talk, https://www.youtube.com/watch?v=zjkBMFhNj_g, 2023. November 2023; popularizes the LLM-as-OS framing.
- Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.
- Nataliya Kosmyna, Eugene Hauptmann, Ye Tong Yuan, Jessica Situ, Xian-Hao Liao, Ashly Vivian Beresnitzky, Iris Braundstein, and Pattie Maes. Your brain on chatgpt: Accumulation of cognitive debt when using an ai assistant for essay writing task. *arXiv preprint arXiv:2506.08872*, 4, 2025.
- Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, et al. Measuring ai ability to complete long software tasks. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.
- LangChain. State of agent engineering. <https://www.langchain.com/state-of-agent-engineering>, 2026. Survey of 1,340 respondents conducted Nov-Dec 2025.
- LangChain, Inc. LangGraph: Build resilient language agents as graphs, 2024. <https://github.com/langchain-ai/langgraph>. GitHub repository.
- Geonsun Lee, Min Xia, Nels Numan, Xun Qian, David Li, Yanhe Chen, Achin Kulshrestha, Ishan Chatterjee, Yinda Zhang, Dinesh Manocha, et al. Sensible agent: A framework for unobtrusive interaction with proactive ar agents. In *Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology*, pages 1–22, 2025.

- Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for "mind" exploration of large language model society. *Advances in neural information processing systems*, 36: 51991–52008, 2023.
- Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. Encouraging divergent thinking in large language models through multi-agent debate. In *Proceedings of the 2024 conference on empirical methods in natural language processing*, pages 17889–17904, 2024.
- Xingyu Bruce Liu, Shitao Fang, Weiyan Shi, Chien-Sheng Wu, Takeo Igarashi, and Xiang'Anthony' Chen. Proactive conversational agents with inner thoughts. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, pages 1–19, 2025.
- Yue Liu, Ratnadira Widyasari, Yanjie Zhao, Ivana Clairine Irsan, and David Lo. Debt behind the ai boom: A large-scale empirical study of ai-generated code in the wild. *arXiv preprint arXiv:2603.28592*, 2026.
- Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhunoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.
- Lance Martin. Agent design patterns. https://rlancemartin.github.io/2026/01/09/agent_design/, 2026.
- Lance Martin, Gabe Cema, and Michael Cohen. Scaling managed agents: Decoupling the brain from the hands. Anthropic Engineering Blog, <https://www.anthropic.com/engineering/managed-agents>, 2026.
- Miles McCain, Thomas Millar, Saffron Huang, Jake Eaton, Kunal Handa, Michael Stern, Alex Tamkin, Matt Kearney, Esin Durmus, Judy Shen, Jerry Hong, Brian Calvert, Jun Shern Chan, Francesco Mosconi, David Saunders, Tyler Neylon, Gabriel Nicholas, Sarah Pollack, Jack Clark, and Deep Ganguli. Measuring AI agent autonomy in practice. Anthropic Research Blog, <https://anthropic.com/research/measuring-agent-autonomy>, 2026.
- MindStudio Team. What is the anthropic Claude Code source code leak? three-layer memory architecture explained. <https://www.mindstudio.ai/blog/claude-code-source-leak-three-layer-memory-architecture>, 2026.
- Ethan Mollick. *Co-intelligence: Living and working with AI*. Penguin, 2024.
- Luca Nannini, Adam Leon Smith, Michele Joshua Maggini, Enrico Panai, Sandra Feliciano, Aleksandr Tiulkanov, Elena Maran, James Gealy, and Piercosma Bisconti. Ai agents under eu law. *arXiv preprint arXiv:2604.04604*, 2026.
- Alexander Novikov, Ngan Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.
- OECD. Governing with artificial intelligence: The state of play and way forward in core government functions. https://www.oecd.org/en/publications/governing-with-artificial-intelligence_795de142-en/full-report.html, 2025. Official OECD Public Governance Committee report, 18 September 2025.
- Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. Memgpt: towards llms as operating systems. 2023.
- Gil Pasternak, Dheeraj Rajagopal, Julia White, Dhruv Atreja, Matthew Thomas, George Hurn-Maloney, and Ash Lewis. Beyond reactivity: Measuring proactive problem solving in llm agents. *arXiv preprint arXiv:2510.19771*, 2025.
- Divya Pathak, Harshit Kumar, Anuska Roy, Felix George, Mudit Verma, and Pratibha Moogi. Detecting silent failures in multi-agentic ai trajectories. *arXiv preprint arXiv:2511.04032*, 2025.
- Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do users write more insecure code with ai assistants? In *Proceedings of the 2023 ACM SIGSAC conference on computer and communications security*, pages 2785–2799, 2023.
- Kevin Pu, Daniel Lazaro, Ian Arawjo, Haijun Xia, Ziang Xiao, Tovi Grossman, and Yan Chen. Assistance or disruption? exploring and evaluating the design and trade-offs of proactive ai programming support. In *Proceedings of the 2025 CHI conference on human factors in computing systems*, pages 1–21, 2025.

- Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pages 15174–15186, 2024.
- Prithvi Rajasekaran. Harness design for long-running application development. Anthropic Engineering Blog, <https://anthropic.com/engineering/harness-design-long-running-apps>, 2026.
- Gwendolyn Rak. How to stay ahead of AI as an early-career engineer. *IEEE Spectrum*, 2025. <https://spectrum.ieee.org/ai-effect-entry-level-jobs>.
- Charles Reis and Steven D Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 219–232, 2009.
- Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-based access control models. *Computer*, 29(2):38–47, 2002.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.
- Erik Schluntz and Barry Zhang. Building effective agents. Anthropic Research, <https://www.anthropic.com/research/building-effective-agents>, 2024.
- Judy Hanwen Shen and Alex Tamkin. How ai impacts skill formation. *arXiv preprint arXiv:2601.20245*, 2026.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.
- Leon Staufer, Kevin Feng, Kevin Wei, Luke Bailey, Yawen Duan, Mick Yang, A Pinar Ozisik, Stephen Casper, and Noam Kolt. The 2025 ai agent index: Documenting technical and safety features of deployed agentic ai systems. *arXiv preprint arXiv:2602.17753*, 2026.
- Peter Steinberger and OpenClaw Contributors. OpenClaw: Personal AI assistant. <https://github.com/openclaw/openclaw>, 2026. Open-source multi-channel AI assistant gateway. MIT License.
- Viktoria Stray, Elias Goldmann Brandtzaeg, Viggo Tellefsen Wivestad, Astri Barbala, and Nils Brede Moe. Developer productivity with and without github copilot: A longitudinal mixed-methods case study. *arXiv preprint arXiv:2509.20353*, 2025.
- Yifan Sui, Han Zhao, Rui Ma, Zhiyuan He, Hao Wang, Jianxun Li, and Yuqing Yang. Act while thinking: Accelerating llm agents via pattern-aware speculative tool execution. *arXiv preprint arXiv:2603.18897*, 2026.
- Weiwei Sun, Xuhui Zhou, Weihua Du, Xingyao Wang, Sean Welleck, Graham Neubig, Maarten Sap, and Yiming Yang. Training proactive and personalized llm agents. *arXiv preprint arXiv:2511.02208*, 2025.
- The Linux Foundation. Linux foundation announces the formation of the agentic AI foundation (AAIF), anchored by new project contributions including model context protocol (MCP), goose and AGENTS.md. Linux Foundation Press Release, 2025. <https://www.linuxfoundation.org/press/linux-foundation-announces-the-formation-of-the-agentic-ai-foundation>.
- Janelle Teng Wade, Lance Co Ting Keh, Talia Goldberg, David Cowan, Grace Ma, Bhavik Nagda, Brandon Nydick, and Bar Weiner. AI infrastructure roadmap: Five frontiers for 2026. Bessemer Venture Partners, <https://www.bvp.com/atlas/ai-infrastructure-roadmap-five-frontiers-for-2026>, 2026.
- Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.
- Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024a.
- Shawn Wang. Agent engineering. Latent Space, <https://www.latent.space/p/agent>, 2025.
- Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024b.
- Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024c.

- Lilian Weng. LLM-powered autonomous agents. <https://lilianweng.github.io/posts/2023-06-23-agent/>, 2023.
- Michael Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, 2009.
- Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First conference on language modeling*, 2024.
- Qing Xiao, Xinlan Emily Hu, Mark E Whiting, Arvind Karunakaran, Hong Shen, and Hancheng Cao. Ai hasn’t fixed teamwork, but it shifted collaborative culture: A longitudinal study in a project-based software development organization (2023-2025). *arXiv preprint arXiv:2509.10956*, 2025.
- Bin Xu. Ai agent systems: Architectures, applications, and evaluation. *arXiv preprint arXiv:2601.01743*, 2026.
- Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *arXiv preprint arXiv:2502.12110*, 2025.
- John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.
- Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ -bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.
- Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*, 2025a.
- Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model-based agents. *ACM Transactions on Information Systems*, 43(6):1–47, 2025b.
- Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023.
- Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Gptswarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*, 2024.

Appendix

A Package Structure

This appendix shows what each part of the TypeScript package does at runtime.

A.1 Directory-to-Responsibility Map

The package (Figure 9) is organized around a `src/` directory. Table 7 lists the key files that form the main subsystems.

Table 7 Key files by approximate size and runtime responsibility.

| File | Size | Responsibility |
|-----------------------------|-------|------------------------------------|
| <code>main.tsx</code> | 804KB | Entry point, mode dispatch, setup |
| <code>query.ts</code> | 68KB | Core agent loop, 5 context shapers |
| <code>QueryEngine.ts</code> | 47KB | SDK/headless conversation wrapper |
| <code>Tool.ts</code> | 30KB | Tool interface, types, utilities |
| <code>history.ts</code> | 14KB | Global prompt history |
| <code>mcp/client.ts</code> | Large | MCP client (8+ transport variants) |
| <code>compact.ts</code> | Large | Compaction engine |
| <code>AgentTool.tsx</code> | Large | Agent tool, subagent dispatch |
| <code>runAgent.ts</code> | Large | 21-parameter agent lifecycle |

The `tools/` directory contains approximately 42 subdirectories implementing tools, with the corresponding schema, description, permission requirements, and execution logic. The `commands/` directory contains approximately 86 slash command subdirectories.

Key service directories include `services/tools/` (StreamingToolExecutor, toolOrchestration, toolExecution), `services/compact/` (compaction engine), and `services/mcp/` (MCP client and configuration). The permission infrastructure spans `utils/permissions/` (rule evaluation, classifier), `hooks/useCanUseTool.tsx` (permission handler), `types/permissions.ts` (mode definitions), and `types/hooks.ts` (event schemas).

A structural quirk: `query.ts` (file) and `query/` (directory) coexist. The file contains the main query loop; the directory houses helper modules for loop configuration and context assembly.

A.2 Conditional Tool Availability

The `getAllBaseTools()` function (`tools.ts`) constructs different tool sets depending on mode, build, environment, and feature flags (Table 8). The model may see as few as 3 tools in simple mode (Bash, Read, Edit) or 40+ tools in a full internal build with all features enabled.

A.3 Cross-File Dependencies

The import graph includes the following dependency structure. `QueryEngine.ts` delegates to `query.ts` for turn execution. `query.ts` imports from `services/tools/` (StreamingToolExecutor, runTools) and `services/compact/` (autoCompact, buildPostCompactMessages). `QueryEngine.ts` imports from `memdir/` for memory and prompt assembly. The code explicitly avoids circular imports: `types/permissions.ts` was extracted to break import cycles, and `setCachedClaudeMdContent()` in `context.ts` avoids a cycle through the `permissions/filesystem` path.

Table 8 Conditional tool availability categories.

| Category | Examples |
|-----------------|--|
| Always included | AgentTool, BashTool, FileReadTool, FileEditTool, FileWriteTool, SkillTool, WebFetchTool, WebSearchTool |
| Environment | GlobTool/GrepTool (unless embedded), ConfigTool (ant-only), PowerShellTool (Windows) |
| Feature flag | TaskCreate/Get/Update/List (todoV2), EnterWorktreeTool (worktree), TeamTools (swarms), ToolSearchTool |
| Null-checked | SuggestBackgroundPRTTool, WebBrowserTool, RemoteTriggerTool, MonitorTool, SleepTool |

B Evidence Base and Methodology

This appendix describes the evidence sources, the analytic procedure, and the epistemological constraints of this study.

B.1 Evidence Base and Evidence Tiers

Claims in this paper are grounded at three evidence tiers:

- **TIER A (product-documented):** Claims drawn from official Anthropic documentation and engineering publications. These establish product intent but may not reflect internal implementation.
- **TIER B (code-verified):** Claims citing specific files and functions in the extracted TypeScript codebase (v2.1.88, obtained from a publicly available npm package extraction). This is the strongest evidence tier.
- **TIER C (reconstructed):** Claims derived from community analysis, OpenClaw structural comparison, or inference from code patterns. These are stated with hedging language.

The source corpus comprises approximately 1,884 files totaling roughly 512K lines of TypeScript. OpenClaw is used for calibration rather than ground truth.

B.2 Design-Space Analytic Procedure

Design questions were identified by examining each subsystem for recurring choice points where alternative designs exist in other production agents. Claude Code’s answers to each question were traced through specific source files and function implementations (TIER B evidence). The five-value framework (human decision authority, safety, security, and privacy, reliable execution, capability amplification, and contextual adaptability) was identified from official documentation and creator statements (TIER A), then traced through thirteen design principles to architectural decisions. Long-term capability preservation is treated separately as an evaluative lens rather than a design value, because it is not prominently reflected as a design driver in the architecture or in Anthropic’s stated values (Section 2.4). Token economics serves as a cross-cutting constraint that bounds all five values simultaneously, revealing how individual subsystem choices interact under shared resource pressure.

B.3 Limitations

- **Static snapshot.** Analysis reflects one version (v2.1.88). Feature flags (*e.g.*, `TRANSCRIPT_CLASSIFIER`, `CONTEXT_COLLAPSE`) create build-time variability; different build targets may produce functionally different applications.
- **Reverse-engineering epistemology.** Source code reveals implemented structure, control flow, dependencies, and feature gates. It cannot confirm design intent, enabled production flags, runtime prevalence, or unshipped behavior.
- **Single-system analysis.** Findings describe Claude Code’s design space, not the entire design space of coding agents. Generalizations are bounded.
- **OpenClaw snapshot.** The OpenClaw analysis reflects a specific development state and may not represent its current capabilities.

| Source Structure (v2.1.88) | Runtime Responsibility |
|--|--|
| Entry & Startup | |
| main.tsx | Application entry point, mode dispatch, signal handlers |
| replLauncher.tsx | Interactive REPL composition (components + screens) |
| entrypoints/ | SDK & headless startup (coreTypes.ts, coreSchemas.ts) |
| cli/ | CLI argument handlers (agents, auth, mcp, plugins) |
| UI Layer | |
| components/, screens/ | Terminal UI building blocks (ink framework), screen composition |
| outputStyles/ | System-prompt output style logic |
| Core Loop | |
| query.ts | Agentic query loop (queryLoop AsyncGenerator), 5 shapers |
| query/ | Loop config helpers (context assembly is in context.ts) |
| QueryEngine.ts | Headless/SDK conversation wrapper (delegates to query.ts) |
| context.ts | Context assembly (getSystemContext, getUserContext) |
| Tools & Commands | |
| Tool.ts | Tool interface and types (execution lives in services/tools/) |
| tools/ | 42 concrete tool implementations (Bash, Read, Edit, Agent, ...) |
| services/tools/ | Tool execution and orchestration (not registration) |
| commands/ | 86 slash command implementations |
| Safety & Permissions | |
| utils/permissions/ | Deny-first rule evaluation, yoloClassifier (auto-mode) |
| types/permissions.ts | 7 permission mode definitions (5 external + auto + bubble) |
| hooks/useCanUseTool.tsx | Permission handler (coordinator, swarm, classifier, interactive) |
| components/permissions/ | Permission dialog UI (PermissionDialog.tsx, per-tool prompts) |
| Extensibility | |
| plugins/ | Plugin loader, manifest validation, component registration |
| skills/ | Skill loader, SKILL.md frontmatter parsing, bundled skills |
| utils/hooks.ts | Hook registry, lifecycle dispatch across 27 event types |
| types/hooks.ts + schemas/hooks.ts | Hook schemas (Zod) + types (cmd/prompt/http/agent) |
| Context & Memory | |
| services/compact/ | 5-layer compaction (budget, snip, micro, collapse, auto) |
| memdir/ | Auto memory loading, entry cap enforcement |
| utils/claude.md.ts | CLAUDE.md 4-level hierarchy, @include processing |
| state/ | Runtime application state |
| Persistence | |
| history.ts | Global prompt history (history.jsonl, reverse-order reader) |
| utils/sessionStorage.ts | Per-session JSONL transcripts, sidechains, file-history |
| Services & Integration | |
| services/ | MCP client (8+ transports), API adapters, LSP, analytics |
| remote/ | Remote execution backend support |
| coordinator/ | Multi-agent coordination mode, worker management |
| Additional Infrastructure | |
| bootstrap/, bridge/, constants/, server/ | App init, WebSocket communication, API configuration |
| ink/, keybindings/, vim/, buddy/, ... | Terminal rendering, input handling, optional features |

Figure 9 Extracted package structure mapped to runtime responsibilities. Left column: TypeScript source directories and key files. Right column: inferred runtime roles. This appendix represents reconstructed analysis (Tier C evidence), not official Anthropic documentation.