Inside the Scaffold: A Source-Code Taxonomy of Coding Agent Architectures

Benjamin Rombaut

Abstract

LLM-based coding agents can localize bugs, generate patches, and run tests with diminishing human oversight, yet the scaffolding code that surrounds the language model (the control loop, tool definitions, state management, and context strategy) remains poorly understood. Existing surveys classify agents by abstract capabilities (tool use, planning, reflection) that cannot distinguish between architecturally distinct systems, and trajectory studies observe what agents do without examining the scaffold code that determines why. This paper presents a source-code-level architectural taxonomy derived from analysis of 13 open-source coding agent scaffolds at pinned commit hashes. Each agent is characterized across 12 dimensions organized into three layers: control architecture, tool and environment interface, and resource management. The analysis reveals that scaffold architectures resist discrete classification: control strategies range from fixed pipelines to Monte Carlo Tree Search, tool counts range from 0 to 37, and context compaction spans seven distinct strategies. Five loop primitives (ReAct, generate-test-repair, plan-execute, multi-attempt retry, tree search) function as composable building blocks that agents layer in different combinations; 11 of 13 agents compose multiple primitives rather than relying on a single control structure. Dimensions converge where external constraints dominate (tool capability categories, edit formats, execution isolation) and diverge where open design questions remain (context compaction, state management, multi-model routing). All taxonomic claims are grounded in file paths and line numbers, providing a reusable reference for researchers studying agent behavior and practitioners designing new scaffolds.

1 Introduction

Large language models have transformed software engineering practice. Tools such as Aider [Gauthier, 2024], SWE-agent [Yang et al., 2024], and OpenHands [Wang et al., 2025] can navigate unfamiliar repositories, localize bugs, generate patches, and run test suites with diminishing human oversight. The most capable of these systems resolve over half of real-world GitHub issues on the SWE-bench Verified benchmark [Jimenez et al., 2024], and seven of the agents analyzed in this study have each accumulated over 15,000 GitHub stars [Borges and Valente, 2018] (Table 1), suggesting adoption by substantial developer communities. As these agents move from research prototypes to production tooling, the scaffolding code that surrounds the language model (the control loop, tool definitions, state management, and context strategy) increasingly determines how the agent behaves, what mistakes it makes, and where it spends its token budget [Bui, 2026].

Despite this practical significance, the internal architectures of coding agent scaffolds remain poorly understood in the research literature. Existing surveys of LLM-based agents operate at the conceptual level, organizing systems by abstract capabilities (tool use, memory, planning, reflection) rather than by the implementation strategies that distinguish one production system from another [Masterman et al., 2024, Nowaczyk, 2025]. A coding agent that uses Monte Carlo Tree Search to explore candidate patches and one that uses a simple while loop with test-driven retries

both qualify as "tool-using, planning, reflective" agents under these taxonomies, yet their scaffold architectures differ in fundamental ways that affect cost, reliability, and failure modes. Empirical work on coding agents has begun to characterize their runtime behavior through trajectory analysis [Majgaonkar et al., 2026, Ceka et al., 2025], revealing that successful agents localize bugs faster, test earlier, and produce shorter action sequences than failing ones. However, these studies treat agents as black boxes: they observe what agents do, but do not examine the scaffold code that determines why. Researchers have called for architecture-aware evaluation metrics that link internal components to observable outcomes [Souza and Machado, 2026], and detailed architectural descriptions exist for individual systems [Bui, 2026], but, to the best of the current literature search, no study has systematically compared the scaffolding architectures of production coding agents at the source-code level.

This gap matters for three reasons. First, without a shared vocabulary for scaffold design, researchers studying agent behavior cannot attribute observed differences to specific architectural choices; the confound between scaffold design and model capability goes unacknowledged. Prior trajectory studies used different LLMs for different agents [Majgaonkar et al., 2026], making it impossible to isolate whether a behavioral difference stems from the scaffold or the model. Second, practitioners building new coding agents lack a systematic map of the design space. Architectural decisions (whether to use a persistent event store or a flat message list, whether to give the LLM ten specialized tools or a single shell command, whether to compact context via summarization or truncation) are currently made by reading individual codebases or blog posts, with no comparative reference (Section 2). Third, the rapid pace of development means the design space is expanding faster than it is being documented: the 13 agents analyzed in this study span release dates from June 2023 to March 2025 and represent distinct architectural strategies (fixed pipelines, sequential ReAct loops, phased workflows, depth-first tree search, and full MCTS), yet no prior work has mapped their relationships.

This paper presents a source-code-level architectural taxonomy of 13 open-source coding agent scaffolds. Each agent's implementation was analyzed at a pinned commit hash across 12 dimensions organized into three layers: control architecture (how the agent decides what to do next), tool and environment interface (how the agent interacts with code and execution environments), and resource management (how the agent manages context, state, and models) (Section 3.2). The analysis follows a qualitative case-study methodology, with every taxonomic claim grounded in file paths and line numbers from cloned repositories.

The taxonomy reveals several findings that challenge common assumptions about coding agent design. Rather than falling into discrete architectural categories, agents occupy positions along continuous spectra: control strategies range from fixed pipelines with no feedback loop (Agentless [Xia et al., 2025]) to full Monte Carlo Tree Search with reward backpropagation (Moatless Tools [Orwall, 2024]), with 7 of 13 agents using a sequential ReAct loop as their primary control structure. Loop primitives (ReAct, generate-test-repair, plan-execute, multi-attempt retry) function as composable building blocks that agents layer in different combinations, not as mutually exclusive types. Tool sets vary from zero LLM-callable tools (Aider, where the user drives all navigation) to 37 action classes (Moatless Tools), yet the underlying capability categories converge: reading, searching, editing, and executing code appear in every agent that grants the LLM autonomy. Context compaction, state management, and multi-model routing each exhibit similar spectra, with design choices at one end prioritizing simplicity and at the other prioritizing robustness or search breadth. Five cross-cutting themes, including the tradeoff between sampling and iteration, the emergence of sub-agent delegation as a first-class tool, and the architectural implications of IDE coupling, span multiple dimensions and capture patterns that do not reduce to a single axis.

The contributions of this paper are as follows:

1. A taxonomy of coding agent scaffold architectures derived from source-code analysis of 13 open-source agents, organized into three layers and 12 dimensions. This is, to the best of the current literature search, the first comparative architectural study of coding agents at the implementation level.

2. The empirical finding that scaffold architectures are better characterized as compositions of loop primitives along continuous spectra than as instances of discrete architectural types.

3. A detailed evidence base of architectural observations pinned to specific commits, providing a reusable reference for researchers studying agent behavior and practitioners designing new scaffolds.

The remainder of this paper is structured as follows. Section 2 surveys related work on agent architectures, trajectory analysis, and coding agent evaluation. Section 3 describes the agent selection criteria, analysis dimensions, and coding procedure. Section 4 presents the taxonomy across its three layers, with evidence tables and cross-cutting themes. Section 5 interprets the findings and draws out implications for agent design and evaluation. Section 6 addresses threats to validity. Section 7 concludes with a summary and directions for future work.

2 Related Work

This study sits at the intersection of several research threads. The following subsections address each in turn: conceptual taxonomies of LLM agents (Section 2.1), empirical studies of coding agent behavior (Section 2.2), architectural descriptions of individual systems (Section 2.3), and benchmark-driven evaluation (Section 2.4). Each thread contributes a different perspective on coding agents; none examines their scaffold architectures comparatively at the source-code level.

2.1 LLM Agent Architecture Surveys

A growing body of survey work organizes LLM-based agents by abstract capabilities. Masterman et al. [2024] propose a five-component reference model (reasoning, planning, tool use, memory, reflection) and classify interaction patterns across single-agent and multi-agent designs. Nowaczyk [2025] offers a capability-based taxonomy (tool-using, memory-augmented, planning, multi-agent, embodied) grounded in classical agent theory, arguing that reliability is an architectural property rather than a function of model quality alone. This claim is sympathetic to the premise of the present study; the source-code analysis presented here provides the implementation-level evidence that Nowaczyk et al.'s conceptual argument implies but does not supply, grounding the relationship between architecture and reliability in specific scaffold design choices rather than abstract capability categories. Broader surveys of LLM applications in software engineering cover coding agents as one category among many [Zhang et al., 2026]. These taxonomies provide useful conceptual vocabulary, but they operate at a level of abstraction that cannot distinguish between production coding agents. Every agent in the present corpus qualifies as "tool-using, memory-augmented, planning" under these schemes, yet their scaffold architectures differ in ways that affect cost, reliability, and failure modes. A coding agent that uses Monte Carlo Tree Search to explore candidate patches and one that uses a simple while loop with test-driven retries are indistinguishable under capability-based classification, despite fundamental differences in control flow, state management, and resource consumption. A related strand of work studies how LLMs learn to use tools at the model level (tool-calling training, function-calling fine-tuning) [Schick et al., 2023, Patil et al., 2024]; this addresses

a complementary question to ours, since the present study examines how scaffold code *exposes and orchestrates* tools, not how models learn to invoke them.

The prompting paradigms that underlie many of these conceptual categories have been formalized independently. Yao et al. [2023] introduce the *ReAct* paradigm, in which LLMs generate interleaved reasoning traces and task-specific actions in a thought-action-observation loop; seven of the 13 agents in this study use a sequential ReAct loop as their primary control structure (Table 2), and several others embed ReAct-like iteration within phases of a larger workflow. Shinn et al. [2023] formalized *Reflexion*, a verbal reinforcement learning framework where agents reflect on failed attempts and store reflections in an episodic memory buffer for subsequent retries; this pattern is foundational to the generate-test-repair loop primitive observed across the corpus (Section 4.1.1). However, both ReAct and Reflexion describe *algorithmic* paradigms, not scaffold architectures. The gap between "interleave thoughts and actions" and a production implementation with tool registration, context compaction, multi-model routing, and persistent state is precisely what this study documents.

2.2 Coding Agent Trajectory and Behavior Studies

A complementary line of work analyzes what coding agents *do* at runtime by studying their execution trajectories. Ceka et al. [2025] collect full execution traces from five agents running on SWE-bench Verified, normalize them into a unified action schema, and extract a taxonomy of "decision pathways": recurring behavioral patterns such as exploration-heavy, patch-first, and test-driven strategies. They find that bug localization is the primary bottleneck and that early test generation correlates strongly with success. Majgaonkar et al. [2026] analyze trajectory logs from OpenHands, SWE-agent, and Prometheus [Pan et al., 2026], reporting that failure trajectories are 12–82% longer than successful ones and that repository navigation dominates agent activity over patch writing. Bouzenia and Pradel [2025] study thought-action-result trajectories from three agents (RepairAgent [Bouzenia et al., 2025], AutoCodeRover, OpenHands), finding that even rare thought-action misalignment (0.5–4.8% of steps) strongly correlates with failure and that failing trajectories exhibit repetitive, non-adaptive action cycles.

These studies establish important empirical regularities, but they share two limitations that the present work addresses. First, they treat agents as black boxes: they observe *what* agents do but cannot explain *why*. A finding such as "agents with shorter trajectories succeed more often" cannot distinguish between a scaffold that enforces early termination, one that implements efficient search heuristics, and one that simply has a low iteration budget. Second, prior trajectory studies used different LLMs for different agents [Majgaonkar et al., 2026, Bouzenia and Pradel, 2025], confounding scaffold effects with model effects; for example, Majgaonkar et al. compare Claude 3.5 Sonnet-based OpenHands trajectories against DeepSeek-V3-based Prometheus trajectories, making it impossible to isolate whether behavioral differences stem from the scaffold or the model. The architectural taxonomy presented here provides the missing explanatory layer: by examining the scaffold source code that produces those trajectories, it becomes possible to attribute behavioral differences to specific design choices rather than to opaque system-level differences.

Fan et al. [2025] move from behavioral to resource-oriented analysis, measuring token consumption, cost, and computation time for five scaffolds across three LLMs. Their "token snowball effect" (linear input token growth with API calls due to naive conversation history accumulation) and "expensive failures" (failing attempts consuming up to 4× the resources of successful ones) are empirical phenomena that the present study can explain architecturally: the token snowball maps to the context compaction dimension (Section 4.3.2), and expensive failures relate to error recovery

and termination strategies. However, SWE-Effi does not examine the scaffold code that produces these cost patterns; its only architectural distinction is "agentic" versus "procedural."

2.3 Individual System Descriptions

Detailed architectural descriptions exist for several individual coding agents, and collectively they demonstrate that scaffold design choices significantly affect agent behavior. Several papers focus on how tool and environment interfaces shape agent capabilities: Yang et al. [2024] introduce the agent-computer interface (ACI) concept for SWE-agent, showing that designing custom shell commands to structure repository interaction is itself an architectural decision that affects downstream performance; Gauthier [2024] describes repository mapping via PageRank and model-specific edit format selection in Aider; and Bui [2026] describes a four-layer terminal agent architecture with dual-agent design, multi-model routing, lazy tool discovery, and adaptive context compaction, concluding that "tool reliability matters more than model capability."

Other system papers illuminate the spectrum of agent autonomy. At one end, Xia et al. [2025] argue that agentic scaffolds are unnecessary for many tasks, demonstrating that a fixed pipeline of localization, repair, and re-ranking stages can match or exceed agentic approaches on SWE-bench; this motivates the pipeline-to-agent spectrum in the present taxonomy. At the other end, Aggarwal et al. [2025] present tree-structured search with LLM-based branch selection in DARS-Agent, and Arora et al. [2024] decompose the agent into specialized sub-agents for distinct subtasks in MASAI. Between these extremes, Zhang et al. [2024b] describe AST-based context retrieval with spectrum-based fault localization in AutoCodeRover, and Wang et al. [2025] describe the event-sourced architecture and agent delegation mechanisms underlying OpenHands. Each of these papers provides architectural depth for a single system, but the design space as a whole remains unmapped: practitioners cannot compare control loop strategies, tool interface designs, or context management approaches without reading a dozen or more codebases independently. The present study extends this per-system depth across 13 agents, enabling the comparative analysis that individual descriptions cannot provide.

A related but orthogonal line of work studies how developers configure coding agents. Galster et al. [2026] analyze 2,926 GitHub repositories to characterize configuration artifacts (instruction files, prompt templates, skills definitions, structured configuration) across five commercial coding tools, finding that configuration practices remain fragmented and that advanced features such as skills and sub-agent configurations are rarely used. This work is complementary to the present study: configuration artifacts control what instructions the agent receives, while scaffold architecture determines how the agent processes those instructions. A `CLAUDE.md` file that specifies "always run tests before committing" is a developer-facing configuration; the scaffold's generate-test-repair loop that implements that instruction is an architectural feature. The present study analyzes the latter.

2.4 Coding Agent Evaluation and Benchmarks

SWE-bench [Jimenez et al., 2024] has become the de facto evaluation standard for coding agents, driving rapid progress and motivating the design of many agents in the present corpus. However, its limitations are increasingly well documented: overly detailed issue descriptions inflate resolution rates [Garg et al., 2026], single-language bias limits generalizability [Jimenez et al., 2024, Xu et al., 2025], and confounded scaffold-model effects make it difficult to attribute performance to architectural choices [Fan et al., 2025]. Newer benchmarks address subsets of these concerns: SWE-bench Pro [Deng et al., 2025] introduces harder, multi-file tasks with contamination resistance, and SWE-

Compass [Xu et al., 2025] extends evaluation to eight programming languages and multiple task types. [Chen et al., 2026] examine whether agent-generated tests improve resolution rates under a minimal scaffold, finding that prompt interventions that add or remove testing change outcomes by at most 2.6 percentage points; this suggests that scaffold-level orchestration of testing (lint-test cycles, test-gated retries, MCTS reward signals), rather than model-native test-writing behavior, may be the architecturally relevant variable.

The present study deliberately does not benchmark agent performance, because benchmark scores confound scaffold architecture with model capability, prompt engineering, and incidental configuration choices (iteration limits, cost caps, default model selection); isolating the scaffold's contribution requires the kind of architectural analysis presented here, not additional benchmark runs. The taxonomy does, however, enable future controlled experiments by identifying the specific variables that would need to be held constant: for example, comparing agents with identical tool sets but different loop strategies, or identical loops but different compaction strategies, with the model held constant (Section 5.5).

[Souza and Machado, 2026] explicitly call for architecture-aware evaluation metrics that link internal agent components (planner, memory, tool router) to observable outcomes, proposing a component-to-metric mapping framework. However, their proposal is conceptual: it has not been tested on real systems and depends on architectural documentation that, prior to this study, did not exist for most coding agents. The present taxonomy provides the architectural vocabulary that such evaluation frameworks require.

3 Methodology

This study performs a source-code-level architectural analysis of open-source coding agent scaffolds, deriving taxonomic categories from observed implementation patterns rather than from documentation claims or conceptual models.

3.1 Agent Selection

Candidate agents were identified through three channels: agents evaluated on SWE-bench and reported in the leaderboard literature as of early 2026, agents with substantial adoption in developer tooling (measured by GitHub stars as a proxy for community use [Borges and Valente, 2018]), and agents cited in recent surveys of LLM-based software engineering [Zhang et al., 2026]. The search was not exhaustive within any single channel; the pool reflects the agents the researcher encountered through these sources rather than a complete enumeration of all agents meeting a fixed threshold. An initial pool of 22 candidates was narrowed using three inclusion criteria (the full candidate list and exclusion rationale for each are provided in Appendix A):

1. **Coding-specific.** The agent must be designed for software engineering tasks (code editing, bug fixing, repository navigation), not general-purpose task automation. This excluded general-purpose frameworks (Open Interpreter [Open Interpreter, Inc., 2023], Deep Agents [LangChain AI, 2025]) and multi-agent orchestration platforms (MetaGPT [Hong et al., 2024], CrewAI [CrewAI Inc., 2024]), whose unit of analysis is agent coordination rather than individual agent architecture.

2. **Open source with readable implementation.** The agent's scaffolding code (control loop, tool definitions, state management) must be available as readable source code in a public repository, pinned to a specific commit. This excluded Claude Code [Anthropic, 2025], which is

distributed as a compiled binary with no published source repository, and MASAI [Arora et al., 2024][1]. Proprietary agents whose scaffold code is not inspectable (GitHub Copilot Workspace, Cursor's AI backend, Windsurf) were also excluded on this basis.

3. **Architecturally distinct.** Near-duplicate agents were removed to avoid redundant analysis (e.g., agents sharing the same core codebase or differing only in frontend integration).

The 13 agents retained for analysis are listed in Table 1. They span two natural origin categories: *CLI agents* built for interactive developer use, and *SWE-bench agents* built primarily for automated issue resolution. One additional agent, mini-swe-agent [Yang et al., 2025], is included as a deliberate minimal baseline: it was released by the SWE-agent [Yang et al., 2024] team as a reference implementation exposing the simplest possible complete scaffold. The selection is not exhaustive but aims to cover the range of architectural strategies present in the open-source coding agent ecosystem as of early 2026.

Table 1: Agents analyzed, ordered by GitHub stars as a proxy for community adoption [Borges and Valente, 2018]. "Category" distinguishes interactive CLI tools from agents designed for automated SWE-bench evaluation. All agents were analyzed at pinned commit hashes (see Appendix B).

Agent	Category	Language	Stars	Origin
OpenCode [Anomaly, 2025]	CLI	TypeScript	135k	SST
Gemini CLI [Google, 2025]	CLI	TypeScript	100k	Google
Codex CLI [OpenAI, 2025]	CLI	Rust / TS	72k	OpenAI
OpenHands [Wang et al., 2025]	SWE-bench	Python	70k	All Hands AI
Cline [Rizwan, 2024]	CLI	TypeScript	60k	Independent
Aider [Gauthier, 2024]	CLI	Python	43k	Independent
SWE-agent [Yang et al., 2024]	SWE-bench	Python	19k	Princeton / CMU
mini-swe-agent [Yang et al., 2025]	Baseline	Python	4k	Princeton / CMU
AutoCodeRover [Zhang et al., 2024b]	SWE-bench	Python	3k	NUS / SonarSource
Agentless [Xia et al., 2025]	SWE-bench	Python	2k	UIUC
Prometheus [Pan et al., 2026]	SWE-bench	Python	1k	EuniAI
Moatless Tools [Orwall, 2024]	SWE-bench	Python	600	Independent
DARS-Agent [Aggarwal et al., 2025]	SWE-bench	Python	70	Independent

3.2 Analysis Dimensions

The analysis framework covers nine dimensions (which yield 12 taxonomy dimensions in the results, as described below). These dimensions were not fixed a priori; they emerged through iterative open coding [Strauss and Corbin, 1998] of source code during a pilot analysis of two architecturally contrasting agents: Aider (a simpler, interactive CLI scaffold) and OpenHands (a complex, event-driven, containerized scaffold). These two were chosen to maximize architectural diversity during piloting; the rationale and specific refinements are described in Section 3.4.

The initial pilot began with six candidate dimensions drawn from the conceptual agent architecture literature [Masterman et al., 2024]: control loop, tool interface, state management, context retrieval, execution isolation, and multi-model routing. During the pilot, three additional dimensions were added after the source code revealed architectural variation not captured by the initial set: tool discovery strategy (prompted by differences in how Aider and OpenHands register tools), context compaction (prompted by Aider's explicit summarization logic versus OpenHands' lack thereof), and persistent memory (prompted by Aider's conventions files). Stabilization consisted

[1]The MASAI repository contains only a README linking to the paper; no implementation code has been released.

of applying the revised nine-dimension framework to both pilot agents a second time to confirm that every dimension produced discriminating findings across both agents and that no source code observations fell outside the framework without being captured by the open-ended section. No dimensions were removed during this process. Several of the resulting dimensions align with architectural concerns identified independently in prior single-system analyses of coding agents [Bui, 2026] and conceptual agent architecture surveys [Masterman et al., 2024], providing external support for their relevance. Each dimension captures a distinct architectural decision point in the scaffold.

1. **Control loop type.** The structure of the agent's main execution loop: how it sequences LLM calls, tool dispatches, and observation handling. A sub-property, *loop driver*, records whether the loop is user-initiated or LLM-driven, a distinction added during piloting after observing that it determines several downstream architectural decisions (Section 4.1.2).

2. **Tool set and tool interface design.** The full set of tools available to the model, including how tools are defined and communicated to the LLM (e.g., JSON schema via function calling, inline system-prompt descriptions, or custom text formats). This dimension also records the edit and patch format used when the agent proposes file modifications.

3. **Tool discovery strategy.** Whether the full tool set is registered at startup (static) or whether tools are conditionally loaded, dynamically registered, or assembled at runtime based on task state (dynamic).

4. **State management strategy.** How the agent accumulates and represents history across loop iterations: the data structure (flat message list, typed event log, tree), whether state is append-only or mutable, and what is stored beyond raw messages.

5. **Context retrieval paradigm.** The mechanism by which the agent identifies relevant source code before or during task execution: LLM-directed tool calls (`grep`, `find`, AST queries), precomputed repository indexes, embedding-based semantic search, or static repository maps included in the initial prompt.

6. **Execution isolation model.** Where the agent runs shell commands and code: on the host filesystem, in a Docker container, in a sandboxed subprocess, or in a remote cloud environment. This records the trust boundary between the agent and the host system.

7. **Context compaction approach.** The strategy used when conversation history approaches the model's context limit: hard truncation, sliding window, LLM-generated summarization, selective dropping of tool results, or no mechanism present.

8. **Multi-model routing.** Whether a single model handles all agent steps or whether different models are assigned to different roles (e.g., a large model for planning, a small model for localization), and how routing decisions are made.

9. **Persistent memory.** Whether any information survives between sessions: project conventions, prior task outcomes, user preferences, or repository facts, and the storage mechanism used.

The analysis template also includes an open-ended tenth section for observations that do not fit any of the nine dimensions. This follows standard practice in qualitative coding, where emergent categories are expected alongside predefined ones [Saldaña, 2021]: forcing every finding into a

predefined category risks suppressing novel patterns. In practice, this section captured 47 cross-cutting findings (enumerated in the per-agent analysis documents) that informed the taxonomy structure presented in Section 4.

The nine analysis dimensions map to 12 taxonomy dimensions in the results. During analysis, two sub-properties proved sufficiently discriminating to warrant independent treatment: *loop driver* (a sub-property of control loop type) and *edit and patch format* (a sub-property of tool set design) each produced distinct spectra with their own evidence tables and are presented as separate dimensions in Section 4. A third dimension, *control flow implementation* (the code-level mechanism realizing the control loop: while loop, recursion, compiled graph, or exception-based signaling), emerged during analysis as an axis of variation orthogonal to loop topology and is likewise presented independently. This expansion from analysis framework to taxonomy structure is a normal outcome of qualitative coding: the framework guides data collection, but the final categories reflect what the data reveals [Saldaña, 2021].

3.3 Tool Counting Methodology

Tool counts are reported using two complementary methods to avoid conflating interface granularity with functional coverage.

Registration count tallies tools as the LLM sees them: each separately registered callable (function calling schema entry, system-prompt-defined command, or equivalent) is counted once. This measure is sensitive to how scaffold developers chose to partition functionality.

Capability category count groups tools by what they do: read, search, edit, execute, validate, and repository state. These categories were derived inductively from the pilot analysis: after cataloguing every tool across the first two agents, recurring functional roles were grouped into categories, which were then validated against the remaining eleven agents. This measure is comparable across agents regardless of granularity choices. An agent with a single bash tool and an agent with separate run_command, run_tests, and run_linter tools both cover the execute category. Both counts are reported in the results; the capability category count is used for cross-agent comparison.

3.4 Analysis Procedure

Each agent was analyzed using a structured template derived from the nine dimensions established in Section 3.2, applied consistently across all 13 agents. The template separates three levels of description, following the principle that empirical claims should be traceable to primary data sources rather than inferred from secondary documentation [Runeson and Höst, 2009]:

- **Observation**: what the code does, described in terms of data flow and control flow with specific file and line references.
- **Classification**: how the observed behavior maps to a taxonomy dimension, with explicit justification for the mapping.
- **Evidence**: a file path and line number pinned to the specific commit hash analyzed.

For each agent, the template produces a structured document covering all nine dimensions plus the open-ended section, with a full call-chain trace of the main control loop as the entry point. The template was piloted on the same two agents used for dimension development (Section 3.2): Aider and OpenHands. This pilot served a distinct purpose from the dimension stabilization; it tested whether the three-level observation format produced consistent and sufficiently detailed analysis

documents. The pilot also surfaced the need for a dual tool counting methodology (Section 3.3) after observing that raw tool counts were not comparable across agents with different interface granularity. The complete analysis documents for all 13 agents follow this template; individual sections of each analysis are referenced throughout the results where they serve as primary evidence for specific claims.

All analyses were pinned to specific git commit hashes, listed in Appendix B, because several agents under study (Cline, Aider, OpenHands, Gemini CLI) were under active development during the analysis period and unpinned analysis would yield unreproducible results. Where source code was ambiguous, the analysis records uncertainty explicitly rather than assigning a confident classification. Dimensions that genuinely do not apply to an agent (for example, persistent memory in agents with no inter-session storage) are recorded as absent, not omitted. All analyses were conducted by the author with substantial use of LLM-based coding assistants for code navigation, call-chain tracing, and initial summarization of unfamiliar codebases. All LLM-generated observations were verified against the source code before inclusion in the analysis documents; the analysis documents themselves record file paths and line numbers to enable independent verification. The single-author design is a threat to construct validity; mitigations are discussed in Section 6.

3.5 Scope and Limitations

This study is purely taxonomic. No performance benchmarking was conducted, and no claims are made about correlations between scaffold design and task success rates. The decision to exclude performance analysis reflects two limitations in the available data: first, SWE-bench results across agents are not directly comparable because different agents used different underlying models, and model capability is a larger confounder than scaffold design; second, documented solution leakage in SWE-bench issue descriptions [Garg et al., 2026] makes raw pass rates an unreliable signal regardless of these controls. The corpus is limited to open-source agents with readable source code; threats arising from this restriction are discussed in Section 6.

4 Results

The analysis of the 13 open-source coding agent scaffolds reveals a taxonomy organized into three layers: control architecture (how the agent decides what to do next), tool and environment interface (how the agent interacts with code and execution environments), and resource management (how the agent manages context, state, and models). Within each layer, architectural choices fall along continuous gradients rather than into discrete categories. In this section, each dimension is presented as a range of observed strategies, agents are placed along it with source-code evidence, and cross-cutting patterns are identified. Figure 1 provides an overview of the full taxonomy structure.

4.1 Layer 1: Control Architecture

The control layer determines how an agent orchestrates its actions. Three dimensions capture the key variation: (1) the topology of the control loop, (2) what drives it, and (3) how it is implemented in code.

4.1.1 Control Loop Strategies

Control loops range from fixed pipelines with no feedback to tree-structured search with backpropagation (Table 2), and these loop types are not mutually exclusive; agents frequently nest

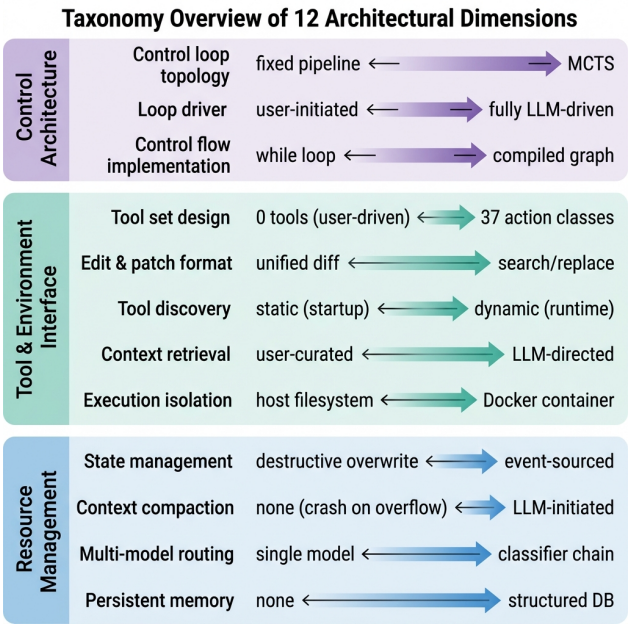

Figure 1: Taxonomy overview: 12 dimensions organized into three architectural layers. Each dimension is a continuous spectrum with observed endpoints from the corpus of 13 agents. Agents occupy positions along these spectra rather than falling into discrete categories (Section 5.1).

one loop type inside another. This composability means the table understates the actual control complexity: an agent classified as a "phased loop" may contain a full ReAct loop inside each phase. AutoCodeRover, for example, runs a multi-turn LLM interaction inside each stage of its pipeline (`agent_search.py:88--163`): a generator function yields tool selections to the caller, which executes them and sends results back, making it simultaneously a phased agent and an iterative agent depending on the level of abstraction. Moatless Tools takes composability further by decoupling the inner agent from the outer control flow entirely: its `ActionAgent` (a single-step executor: one LLM call producing actions, then executing them) can be driven either by `AgenticLoop` (which calls it repeatedly to produce ReAct behavior) or by `SearchTree` for MCTS-based exploration, with no changes to the agent code itself. This separation means that the choice between sequential and tree-search exploration is a configuration decision rather than an architectural one.

Tree search strategies. Three agents span a gradient from flat sampling to informed search, illustrating increasingly sophisticated approaches to exploring the space of possible solutions.

At the simplest end, Agentless uses independent sampling: after localizing the relevant code, it prompts the LLM to generate candidate patches independently (the default configuration generates 20; the original paper uses 4 location samples $\times$ 10 patches each for $\sim$40). Each generation sees the same context but may produce a different fix. It then selects the final patch by majority vote across these candidates. There is no tree structure and no interaction between candidates; each patch is generated in isolation.

DARS-Agent introduces tree-structured search into its main execution loop. Unlike Agentless, which separates localization and repair into distinct phases, DARS-Agent has no phase separation: search commands are available alongside edit and execute commands throughout. The agent builds a search tree where each node represents an action (such as editing a file, creating a file, or submitting a patch). At each of these branch points, the agent generates multiple alternative actions,

Table 2: Control loop strategies. Agents ordered from least to most flexible exploration strategy.

Position	Agents	Mechanism
Fixed pipeline	Agentless	10-stage pipeline of independent scripts connected by JSONL files on disk. No feedback loop between stages.
User-driven loop	Aider	Outer loop is user-initiated (each edit cycle requires user input). Inner generate-test-repair loop is autonomous for up to `max_reflections` iterations.
Sequential ReAct loop	SWE-agent, OpenHands, Codex CLI, Gemini CLI, mini-swe-agent, Cline, OpenCode	Standard thought–tool–observation cycle. LLM selects the next action; loop terminates on completion signal or budget exhaustion.
Phased loop	AutoCodeRover, Prometheus	Distinct stages with different tool access. AutoCodeRover: search-then-patch phase separation. Prometheus: LangGraph [LangChain, 2024] state machine with explicit edges.
Depth-first tree search	DARS-Agent	ReAct steps form nodes in a search tree. At branch points, the environment is reset and replayed from root; an LLM critic selects among sampled alternatives.
Full MCTS	Moatless Tools	Select–Expand–Simulate–Backpropagate with reward values (-100 to $+100$) and visit counts. Pluggable selector interface; a discriminator selects the best finished trajectory.

then uses an LLM critic to select among them: the critic is prompted with the alternatives and responds with a `<best_action_index>` tag indicating its choice. However, unlike classical tree search, DARS-Agent has no numeric reward signals and no backpropagation of results to earlier nodes; the critic makes greedy local decisions without considering how earlier choices affected later outcomes.

Moatless Tools implements full Monte Carlo Tree Search (MCTS) [Browne et al., 2012], the same algorithm used in game-playing systems like AlphaGo [Silver et al., 2016]. Each node in the search tree receives numeric reward values (ranging from -100 to $+100$), and the algorithm maintains visit counts to balance exploration of untried paths against exploitation of promising ones. After expanding a node, rewards are backpropagated up the tree to update ancestor nodes, allowing the search to learn from later outcomes and redirect effort toward more promising branches (`search_tree.py:326--345`).

The progression from flat sampling to full MCTS reveals a design tradeoff: richer search strategies require a mechanism to manage execution state across branches. Moatless Tools addresses this with shadow-mode execution, where file modifications are tracked in memory rather than written to disk; this allows the search to branch at any point without costly filesystem operations. DARS-Agent takes a different approach: at each branch point, it resets the Docker environment to a clean state and replays all actions from the root of the tree to reach the desired branch, which is correct but expensive at deeper search depths.

The remaining CLI agents (Cline, Codex CLI, Gemini CLI, OpenCode) all implement standard sequential ReAct loops without phased structure or tree search, placing them in the same middle band as the other ReAct agents in Table 2. Aider is a special case: although it appears in this band, its outer loop is user-driven rather than agentic (Section 4.1.2), with the LLM producing text-format edits rather than selecting tools. However, Aider's inner loop (`base_coder.py:932`) is autonomous: after the LLM produces edits, the scaffold runs linting and tests, and if either fails, re-prompts the LLM with the error output for up to `max_reflections` iterations. This generate-test-repair cycle is the only part of Aider where multi-turn LLM interaction occurs without user input.

4.1.2 Loop Driver

Orthogonal to loop topology is the question of what drives the loop (Table 3), and this dimension is arguably the most fundamental architectural distinction. Aider sits at one extreme: the LLM never runs grep, never opens a file it was not given, and never decides "I should look at module Y." All navigation responsibility falls on the user, with a PageRank-weighted repo map providing passive context. At the other extreme, 9 of 13 agents give the LLM full autonomy over tool selection, with Prometheus occupying a hybrid position where the LLM drives tool selection within each graph node but scaffold-controlled edges govern transitions between nodes. Between these poles, Agentless and AutoCodeRover occupy an intermediate position where the scaffold sequences phases but the LLM makes decisions within each phase.

Table 3: Loop driver strategies. Who decides what happens next?

Driver	Agents
User-driven	Aider: the LLM has 0 callable tools. The user selects files (`/add`), runs searches, and provides context. The LLM produces edits in a text format parsed by the scaffold (`base_coder.py:2296--2304`).
Scaffold-driven	Agentless, AutoCodeRover: the scaffold controls sequencing and calls the LLM at fixed points. In Agentless, each LLM call is single-turn with no conversation state. In AutoCodeRover, the scaffold manages phase transitions across up to four stages: optional reproducer generation, optional SBFL fault localization, search, and patch generation.
LLM-driven	SWE-agent, OpenHands, Codex CLI, Gemini CLI, Cline, mini-swe-agent, Moatless Tools, DARS-Agent, OpenCode: the LLM selects tools and controls exploration. Prometheus is a hybrid: within each graph node, the LLM drives tool selection via ReAct, but transitions between nodes are scaffold-controlled (conditional edges on state fields such as `state["reproduced_bug"]`).

The loop driver has implications beyond control flow. User-driven agents sidestep the bug localization bottleneck identified in prior trajectory analyses: if the user selects files, incorrect localization is a user error, not an agent failure. LLM-driven agents must solve localization as part of the task, making retrieval strategy (Section 4.2.4) a critical design choice.

4.1.3 Control Flow Implementation

The semantic loop types above are implemented through four distinct code-level mechanisms (Table 4): (1) imperative while loops, where the scaffold calls the LLM in a `while True` loop and breaks on a termination signal (8 of 13 agents); (2) recursion, where each tool-use turn triggers a recursive function call (Cline); (3) graph-as-control-flow, where a compiled state machine defines transitions and cycles (Prometheus); and (4) exception-based signaling, where special exception types carry control messages between the loop body and the outer handler (mini-swe-agent).

The graph-based approach is qualitatively different from the others: control flow is inspectable, serializable, and checkpointable by the framework. Prometheus's subgraphs define their own state types, and each subgraph is wrapped in a `SubgraphNode` class that translates between the parent graph's state and the child graph's state. For example, the parent `IssueState` holds `issue_title`, `issue_body`, and `issue_comments`; the `IssueBugSubgraphNode` wrapper passes these as keyword arguments to `IssueBugSubgraph.invoke()`. At least four levels of nesting exist: `IssueGraph` → `IssueBugSubgraph` → `IssueVerifiedBugSubgraph` → `ContextRetrievalSubgraph`.

Cline's recursive implementation is the only instance of recursion for the main agent loop in the corpus. While semantically equivalent to iteration, it means the JavaScript call stack grows with each tool-use turn. In practice, Node.js default stack limits (typically thousands of frames) are unlikely to be reached during normal sessions, but the design has architectural consequences:

Table 4: Control flow implementation variants. This table classifies agents by code-level mechanism, which is orthogonal to the semantic loop strategy in Table 2; an agent classified as "user-driven" by loop strategy (e.g., Aider) may still use an imperative while loop at the implementation level.

Implementation	Agents
Imperative while loop	SWE-agent, OpenHands, Codex CLI, Gemini CLI, mini-swe-agent, Aider, OpenCode
Fixed pipeline (no loop)	Agentless: sequential scripts; each LLM call is single-turn with no agent loop. The Anthropic path contains a bounded `for` loop (up to 10 iterations, `model.py:148--284`), but the primary architecture has no loop.
Recursion	Cline: `recursivelyMakeClineRequests` (`task/index.ts:2268`). The call stack grows linearly with conversation length.
Graph-as-control-flow	Prometheus: LangGraph compiled state machine with explicit edges (`issue_graph.py:22--134`). Cycles in the graph create loops; recursion limits (30 for `IssueBugSubgraph`, 150 for `BugReproductionSubgraph`; dynamic formulas scale to hundreds of steps for multi-candidate runs) serve as the termination guarantee.
Exception-based signaling	mini-swe-agent: `InterruptAgentFlow` hierarchy (`Submitted`, `LimitsExceeded`, `FormatError`) carries messages as payloads. The `run()` method catches these, injects messages into history, and either continues or breaks (`default.py:88--96`).

each recursive frame retains local state, making the control flow harder to serialize or checkpoint compared to iterative approaches. Seven agents (SWE-agent, OpenHands, Codex CLI, Gemini CLI, mini-swe-agent, Aider, OpenCode) use imperative while loops. Agentless has no agent loop at all (its pipeline is a fixed sequence of scripts), though its Anthropic code path contains a small iteration loop. Codex CLI's while loop is notably more complex than the others: it uses a two-level event-driven architecture with async channels, though the logical pattern remains a standard ReAct loop. Within this group, OpenCode is architecturally distinctive for layering a global publish-subscribe event bus on top of its while loop (`packages/opencode/src/bus/`). Components communicate via typed events rather than direct function calls. No other CLI agent in the corpus uses an event bus for inter-component communication; OpenHands uses event sourcing for state management (Section 4.3.1), but its event stream serves a different purpose (persistence and replay) than OpenCode's pub/sub bus (decoupled runtime communication).

4.2 Layer 2: Tool and Environment Interface

The interface layer captures how agents interact with code and execution environments. Five dimensions characterize the design space: (1) tool set design, (2) edit and patch format, (3) tool discovery strategy, (4) context retrieval paradigm, and (5) execution isolation.

4.2.1 Tool Set Design

Tool counts range from 0 (Aider) to 37 action classes (Moatless Tools), but raw counts obscure a convergence in capability categories (Table 5). Despite this range, the same four core capability categories (read, search, edit, execute) appear across all LLM-driven agents (as classified in Section 4.1.2). A fifth category, validate (dedicated test-running or linting tools), appears only in Moatless Tools; other agents subsume validation under their execute capability. The two scaffold-driven agents that use LLM-callable tools (AutoCodeRover, Agentless) intentionally restrict their tool sets to subsets of these categories, reflecting their phased architectures. Agents with fewer tools achieve coverage through composition: mini-swe-agent's single bash tool covers all four categories by delegating to shell commands. Agents with more tools decompose these categories into

finer-grained operations (Moatless Tools has separate FindClass, FindFunction, FindCodeSnippet, SemanticSearch, GrepTool, and GlobTool actions for the search category alone). OpenHands stands out for including a BrowserGym-based headless browser [Le Sellier De Chezelles et al., 2024] as a first-class tool, the only agent in the corpus with a built-in web browsing tool for navigating and interacting with web pages. It also provides a think tool for logging reasoning without side effects and a request_condensation tool that lets the LLM request context compaction (Section 4.3.2). OpenCode has a comparable tool count (18 built-in) and includes distinctive tools not found in other agents: experimental LSP integration for symbol navigation, a skill meta-tool that loads user-defined domain-specific instructions and workflows from the filesystem, and a batch tool for grouping multiple operations. Codex CLI (∼20+ tools) is notable for including meta-tools that let the LLM discover additional tools at runtime: tool_search queries registered app tools, and tool_suggest recommends tools for the current task. It also exposes a request_permissions tool that lets the LLM ask for elevated sandbox access mid-session, making the agent's own permissions a negotiable resource rather than a fixed constraint.

Table 5: Tool set size and capability coverage. Tool count reflects LLM-callable tools; user-facing commands (e.g., Aider's ∼38 slash commands) are excluded.

Agent	Tools	Read	Search	Edit	Execute	Validate	Notes
Aider	0			✓*			*Text-parsed edits, 13 formats
Agentless	0–1			✓*			*1 simulated tool in Anthropic path
AutoCodeRover	8	✓	✓				All search/read; no edit tools
OpenHands	9+	✓		✓	✓		+MCP; search via bash
Moatless Tools	15–37	✓	✓	✓	✓	✓	37 classes; ∼15 typical per session
SWE-agent	3–35	✓	✓	✓	✓		3 default; 35 across 15 bundles
DARS-Agent	∼15	✓	✓	✓	✓		Inherited from SWE-agent fork
Codex CLI	∼20+	✓	✓	✓	✓		+MCP; +sub-agent spawning
Gemini CLI	17+	✓	✓	✓	✓		+MCP; +tool discovery subprocess
Prometheus	17	✓	✓	✓	✓		1–12 bound per graph node
OpenCode	18+	✓	✓	✓	✓		+MCP; +plugins; +custom tools
Cline	27+	✓	✓	✓	✓		+MCP; largest flat built-in set
mini-swe-agent	1				✓		All capabilities via bash

Per-node tool scoping. Prometheus is the only agent that binds different tool subsets to different decision points. Its `EditNode` sees 5 tools (file operations), while its `BugReproducingWriteNode` sees only read_file, and its `BugFixVerifyNode` sees only run_command. This structural guardrailing constrains the action space at each step. AutoCodeRover achieves a similar effect through phase separation: the patch agent is not given search tools and is instead prompted to produce patches directly from the context gathered during the search phase. This is a workflow-level constraint rather than a configurable binding.

Search-only tools in AutoCodeRover's localization phase. AutoCodeRover's search agent has 8 LLM-callable tools, all of which are read-only search operations; the search LLM cannot edit files, run commands, or modify state. This is the most constrained tool set of any agent phase that uses LLM-callable tools. AutoCodeRover does generate patches, but this happens in a separate patch agent phase where the LLM is prompted to produce a patch directly in its output, without calling any tools. The philosophy is that localization and repair are distinct tasks with different tool requirements: the search phase needs structured code navigation, while the patch phase needs only the context gathered during search and the LLM's code generation capability.

Proxy agent for tool invocation in AutoCodeRover. AutoCodeRover uses a secondary LLM call (`agent_proxy.py:81--82`) to convert the search agent's natural-language tool selections into structured JSON. The search agent "thinks out loud" about which tools to call; the proxy agent extracts the structured invocation with up to 5 retries. This adds one LLM call per search round (with 15 rounds possible, that is up to 15 extra LLM calls) but avoids requiring the search agent to produce structured output directly. No other agent uses a secondary LLM call for tool call parsing; all others rely on function calling APIs, regex parsing, or XML extraction.

4.2.2 Edit and Patch Format

How agents translate LLM output into code changes reveals a convergence toward a shared interface (Table 6). The str_replace_editor tool, which takes old_str and new_str arguments for exact string replacement, appears in 5 of 13 agents (OpenHands, SWE-agent, Codex CLI's apply_patch in freeform mode, Agentless as one of three repair output formats, and Moatless Tools as StringReplace). This convergence is notable because these agents were developed independently; the shared interface reflects a common discovery that exact string matching is more reliable than line-number-based or unified-diff-based editing for LLM-generated patches [Yang et al., 2024].

Table 6: Edit/patch format variants.

Format	Agents
String replacement (function calling)	OpenHands and SWE-agent (str_replace_editor), Codex CLI (apply_patch), OpenCode (edit)
Write tool (function calling)	Gemini CLI, Cline
Text-parsed edit blocks	Aider (13 formats), DARS-Agent
Simulated tool use	Agentless (Anthropic path)
Pydantic-schema actions	Moatless Tools, Prometheus

On the parsing side, SWE-agent supports 10 different output parsers (`FunctionCallingParser`, `ThoughtActionParser`, `XMLThoughtActionParser`, and others), enabling the same tool set to work across models with different output conventions. Together with Aider's 13 model-specific edit formats, this represents the highest degree of model-agnosticism in output interfacing observed in the corpus; however, the two approaches address different layers (SWE-agent adapts tool *parsing*, while Aider adapts the *edit format* itself).

Aider is an outlier with 13 registered edit formats, each implemented as a separate coder subclass. The format is selected based on model capabilities: some models handle unified diffs better, others work better with SEARCH/REPLACE blocks. This treatment of edit format as a first-class, model-specific architectural component is unique in the corpus, though OpenCode implements a simpler version of the same idea: its tool registry selects between edit (string replacement) and apply_patch (unified diff) based on model capabilities, offering two formats rather than thirteen.

Agentless' "simulated tool use" in its Anthropic path is architecturally unique. When using Anthropic's API, the LLM calls a str_replace_editor tool, but every call receives the same hardcoded response: "File is successfully edited" regardless of input. The LLM never sees actual file state after its edits; the tool calls are extracted and applied post-hoc. This uses the tool-calling API as a structured output extraction technique rather than for actual execution.

The remaining agents use variants of the same approach. AutoCodeRover uses custom XML-like `<original>`/`<patched>` tags, while Gemini CLI and Cline combine whole-file write tools with search-and-replace tools functionally similar to str_replace_editor. mini-swe-agent is the only agent that edits files directly via shell commands rather than producing patches for the scaffold to apply.

4.2.3 Tool Discovery Strategy

Tool discovery ranges from fully static (tools fixed at initialization) to genuinely dynamic (tools added during a session) (Table 7). Nearly half of the agents (6 of 13) use fully static tool registration: all tools are defined at initialization and remain unchanged throughout the session. This is the default approach for benchmark agents (Agentless, AutoCodeRover, mini-swe-agent, DARS-Agent, Moatless Tools) and for Aider, where the LLM has no callable tools at all. SWE-agent and OpenHands occupy a middle ground: their tool sets are config-conditional (different tool bundles load depending on configuration), but once loaded, the set is fixed for a given attempt. (In SWE-agent's retry loop, different attempts can load different tool bundles if the `agent_configs` specify different configurations.)

Table 7: Tool discovery strategies.

Strategy	Agents
Static (all at init)	Aider, Agentless, AutoCodeRover, mini-swe-agent, DARS-Agent, Moatless Tools
Per-phase scoping (static per node)	Prometheus, AutoCodeRover
Config-conditional	SWE-agent, OpenHands
Per-turn dynamic rebuild	Codex CLI (`built_tools()` called every sampling request, `codex.rs:6156--6164`)
Dynamic (MCP + subprocess)	Gemini CLI (tool discovery subprocess, `tool-registry.ts:312--439`), Cline (MCP servers connected/disconnected at runtime), OpenCode (static core tools + dynamic custom tools from config dirs, plugin system, and MCP integration)

At the dynamic end, Codex CLI's per-turn tool rebuild is distinctive: while most agents construct the tool set once (or once per phase), Codex CLI calls `built_tools()` for every LLM sampling request, incorporating any MCP server changes, newly enabled connectors, or dynamic tools added during the session. The tool set is, in principle, different at every LLM call. Gemini CLI's `toolDiscoveryCommand` feature takes a different approach: an external subprocess outputs `FunctionDeclaration[]` as JSON, allowing arbitrary tool sources without modifying the agent code. Cline and OpenCode similarly support runtime tool changes through MCP server connections and plugin systems. Prometheus and AutoCodeRover fall between static and dynamic: their tool sets are fixed within each phase or graph node, but different phases expose different tool subsets (as discussed in Section 4.2.1).

4.2.4 Context Retrieval Paradigm

Context retrieval (how the agent finds relevant code) shows substantial architectural variation, with seven distinct strategy types ranging from simple keyword search to knowledge graph traversal (Table 8). The 13 agents divide into two retrieval paradigms that reflect fundamentally different assumptions about where code understanding should live.

The first paradigm, which eight agents adopt (SWE-agent, OpenHands, Codex CLI, Gemini CLI, Cline, mini-swe-agent, DARS-Agent, OpenCode), treats the LLM as a navigator: the agent provides general-purpose shell tools (grep, find, ripgrep) and relies on the LLM to formulate queries, interpret results, and decide where to look next. The scaffold provides no code understanding of its own; it simply executes the commands the LLM requests and returns the output.

The second paradigm invests in scaffold-side code understanding, building structured representations of the codebase before or during the task. Aider constructs a dependency graph from

Table 8: Context retrieval strategies. Agents may use multiple strategies.

Strategy	Agents	Mechanism
Keyword/regex search	SWE-agent, OpenHands, Codex CLI, Gemini CLI, Cline, mini-swe-agent, DARS-Agent, OpenCode	LLM invokes grep, find, ripgrep as tools.
Repo map (static analysis)	Aider	PageRank-weighted tree-sitter tag index. Builds a NetworkX dependency graph; identifiers mentioned in conversation get a 10x boost; chat files get a 50x boost (`repomap.py:365--574`).
AST-aware search	AutoCodeRover, Moatless Tools, DARS-Agent, Prometheus	search_class, search_method: structure-aware queries over pre-built AST indices. AutoCodeRover's AST tools are Python-only; Prometheus's tree-sitter-based tools cover 20 languages.
Knowledge graph traversal	Prometheus	Neo4j graph built from tree-sitter ASTs covering 20 languages. 11 tools (10 graph traversal plus `read_file`) query `FileNode`, `ASTNode`, `TextNode` entities (`graph_traversal.py:93--586`).
Embedding-based semantic search	Moatless Tools	FAISS [Johnson et al., 2021] vector store via LlamaIndex [LlamaIndex, 2022] (`code_index.py:57`). The only agent with embedding-based retrieval as an LLM-callable tool.
Hierarchical localization	Agentless	File → class/function → line-level narrowing across pipeline stages. Each level sees only what the previous level identified (`FL.py:313--681`).
Classical fault localization	AutoCodeRover	SBFL with Ochiai scoring (`analysis/sbfl.py`). Unique in the corpus.

parsed source code and uses PageRank to rank file relevance (discussed below). Agentless narrows from files to classes to lines across successive LLM calls, each seeing only what the previous stage identified. AutoCodeRover and Moatless Tools parse source code into abstract syntax trees (ASTs), enabling structure-aware queries like "find all methods named `process`" rather than text-pattern matching. Prometheus goes furthest, constructing a Neo4j [Neo4j, Inc., 2024] graph database from ASTs covering 20 languages and providing 11 tools (10 graph traversal plus `read_file`) for querying relationships between code entities.

The paradigm choice correlates with loop driver (Section 4.1.2): scaffold-driven agents tend to invest in retrieval infrastructure, while LLM-driven agents tend to trust the LLM to navigate with general-purpose tools. This correlation is not surprising: if the scaffold controls sequencing, it has the opportunity (and the need) to pre-process the repository; if the LLM controls exploration, it can issue search commands on demand.

Aider's PageRank repo map. Aider's retrieval mechanism applies Google's PageRank algorithm [Brin and Page, 1998] to source code structure. Tree-sitter [Brunsfeld et al., 2018] parses every file to extract symbol definitions and cross-file references, forming a directed dependency graph (`repomap.py:365--574`). PageRank scores rank files by structural centrality, with context-aware boosts: identifiers mentioned in conversation receive a 10x weight, files added to the chat receive 50x, and a binary search fills the token budget in rank order. No other agent in the corpus uses graph-theoretic relevance ranking. The approach trades startup cost (parsing the full repository) for context quality: rather than relying on the LLM to navigate the codebase, Aider precomputes relevance from the repository's actual dependency structure.

AutoCodeRover's SBFL. AutoCodeRover is the only agent that incorporates classical fault localization [Wong et al., 2016]. When failing tests are available, it runs spectrum-based fault localization (SBFL) with Ochiai suspiciousness scores [Abreu et al., 2007]: methods executed frequently by failing tests and rarely by passing tests rank highest (`analysis/sbfl.py`). The top 5 suspicious methods are presented to the LLM as advisory input, supplementing its search-based exploration. This bridges two otherwise disconnected communities (automated fault localization and LLM-based repair) [Zhang et al., 2024a]; no other agent in the corpus uses test-execution-based localization.

4.2.5 Execution Isolation

Execution isolation spans from no sandboxing to speculative in-memory execution (Table 9). For tree-search agents, execution isolation interacts directly with search cost. DARS-Agent's environment reset performs a full Docker reset and replays all actions from root to the branch point. At depth N, this requires re-executing N commands. DARS addresses this cost by limiting branching to specific action types (edit, create, append, submit) via `action_expansion_limit` (`default_dars.yaml:179--184`). Moatless Tools avoids replay entirely: its shadow mode tracks file modifications in a `FileContext` object without writing to disk, and each node clones its parent's file context. This enables branching at any step without environment cost.

Table 9: Execution isolation strategies.

Isolation level	Agents
None (local shell)	Gemini CLI, Aider, OpenCode
Stateless subshells	mini-swe-agent (`subprocess.run()`, `local.py:28--39`)
Platform sandboxing	Codex CLI (Bubblewrap+Landlock on Linux, Seatbelt on macOS)
Docker container	SWE-agent, OpenHands, DARS-Agent, AutoCodeRover, Prometheus
Shadow git checkpoints	Cline (isolated git repo tracks state per tool execution, `CheckpointTracker.ts`)
Shadow mode (in-memory)	Moatless Tools (`shadow_mode` flag, `agent.py:57`)
Not applicable	Agentless: LLM never executes arbitrary commands; Docker used only for test execution via SWE-bench harness

Cline's shadow git. Cline's `CheckpointTracker` creates an isolated git repository per workspace that records file system state after each tool execution. This enables diff-based rollback without touching the user's real git history, and without requiring Docker. It handles nested git repos by temporarily disabling them and validates against sensitive directories. This is a middle ground between no isolation and containerization, specific to the IDE context where Cline operates.

Safety approaches diverge. Agents without container isolation pursue strikingly different safety strategies, suggesting the ecosystem has not converged on an isolation standard for interactive agents.

Gemini CLI uses a rule-based policy engine that assigns per-tool approval requirements. Before executing a tool call, the engine checks whether the tool requires user confirmation, is automatically allowed, or is blocked entirely. The rules are configurable, letting users adjust the safety boundary for their workflow.

Codex CLI combines two mechanisms. At the OS level, it uses platform-specific sandboxing (Bubblewrap [Flatpak Project, 2016] and Landlock [Salaün, 2017] on Linux, Seatbelt on macOS) to restrict filesystem and network access. On top of this, a Guardian safety subagent, a separate

LLM (`gpt-5.4`)[2], evaluates each tool call's risk with structured scoring on a 0–100 scale, blocking calls above a threshold of 80 (`guardian.rs`). This is the only agent in the corpus that uses an LLM to evaluate the safety of another LLM's actions.

Aider takes the simplest approach: it relies on the user being present and supervising. Since the user manually selects which files to edit and reviews all proposed changes before they are applied, the human serves as the safety boundary.

Cline provides the most granular approval system in the corpus: per-tool and per-scope approval settings, with a `CommandPermissionController` that blocks dangerous shell operators. Settings range from full autonomous mode ("YOLO") to per-command-pattern approval, giving users fine-grained control over the trust boundary. OpenCode uses a policy-based permission system similar to Gemini CLI's: tools can request runtime user approval with three options (Allow Once, Always Allow, Reject) via a permission callback in individual tool implementations, but provides no OS-level sandboxing. mini-swe-agent uses stateless subshells, providing process-level isolation without containerization.

The five Docker-based agents (SWE-agent, OpenHands, DARS-Agent, AutoCodeRover, Prometheus) rely on container boundaries as their primary safety mechanism: all commands execute inside a container, and the host filesystem is never directly exposed. OpenHands' Docker implementation is architecturally distinctive: a FastAPI action server runs *inside* the container, and the host-side agent controller communicates with it via HTTP. This creates a clean API boundary between the agent and the execution environment, unlike other Docker agents that exec commands directly into containers. Agentless occupies a unique position: because the LLM never executes arbitrary commands (it only generates text that the scaffold parses), the isolation requirements are fundamentally different from agentic approaches. Docker is used only for test execution via the SWE-bench harness, not for agent safety.

The spread from human supervision to LLM-based safety evaluation reflects an open design question: as agents gain more autonomy (Section 4.1.2), the safety mechanism must scale correspondingly, but no standard approach has emerged.

4.3 Layer 3: Resource Management

The resource management layer addresses how agents handle the constraints of finite context windows, API costs, and session boundaries. Four dimensions capture the variation: (1) state management, (2) context compaction, (3) multi-model routing, and (4) persistent memory.

4.3.1 State Management

How agents represent and maintain conversation state ranges from simple destructive overwrite to full event sourcing [Fowler, 2005] (Table 10). At one extreme, Aider's two-list design (`cur_messages` and `done_messages`) is simple but destructive: summarization replaces the contents of `done_messages`. At the other extreme, OpenHands' event-sourced architecture stores immutable events and computes views via the `View` class; condensation inserts markers rather than deleting events, preserving the full audit trail. Between these extremes, OpenCode's SQLite-backed hierarchy uses append-only messages with 12 typed part variants, the most granular state representation in the corpus after OpenHands and the only one backed by a relational database rather than in-memory structures. Agentless sits at the opposite end of the spectrum: it has no conversation

[2]Model identifiers such as `gpt-5.4`, `gpt-5.1-codex-mini`, and `gpt-5.3-codex` reflect the strings observed in the analyzed commit (Appendix B). These names may differ in later versions of Codex CLI.

state at all, since each LLM call is single-turn. State between pipeline stages is represented as immutable JSONL files on disk, one JSON object per problem instance per line. This file-based state bus makes the pipeline trivially resumable (`--skip_existing` checks instance presence in output) and parallelizable, but means there is no conversation history to manage because there is no conversation. Codex CLI stands out among the flat-list agents by adding dual persistence: an append-only JSONL rollout file for human-readable replay alongside a SQLite database for queryable state and session resumption. It is also the only flat-list agent that supports undo (`Op::Undo`) and thread rollback (`Op::ThreadRollback`). Cline maintains two parallel message lists, one for the LLM API and one for the UI, synchronized via mutex-protected concurrent access; this dual-list architecture reflects its IDE integration, where the UI needs a richer representation than the API.

Table 10: State management strategies.

Strategy	Agents
Destructive	Aider: summarization overwrites `done_messages` (`base_coder.py:1024--1034`)
Flat list, preserved	SWE-agent, Codex CLI, Gemini CLI: raw history kept; filtered views created for LLM. mini-swe-agent: raw history kept with no filtering (messages sent to the LLM as-is)
Typed event log	OpenCode: SQLite-backed message/part hierarchy with 12 part types (Drizzle ORM). Append-only messages with mutable part states (`message-v2.ts`)
Graph-scoped	Prometheus: separate message lists per graph node (`edit_messages`, `analyzer_messages`, `context_provider_messages`), reset at retry boundaries
Tree-structured (MCTS)	Moatless Tools: nodes in a tree with per-node file context snapshots, visit counts, and reward values
Tree-structured (greedy)	DARS-Agent: nodes in a tree with per-node expansion candidates and critic responses (`dars_agent.py:294--297`). No MCTS statistics; branching uses greedy LLM critic selection, and state is recovered via Docker reset and action replay rather than file context snapshots.
Event-sourced	OpenHands: immutable `EventStream`; views computed from condensation markers (`memory/view.py:13--96`)

Prometheus's graph-scoped state is structurally unique. Rather than a single growing conversation, each LLM node in the graph maintains its own message list. `ResetMessagesNode` clears specific message lists when the graph cycles back for retries, preventing unbounded accumulation without any token-counting logic.

Tree-structured state stores per-node metadata that flat lists cannot represent, but the two tree-search agents use it differently. Moatless Tools maintains MCTS statistics (visit counts and reward values) and clones file context snapshots at each branch point. DARS-Agent stores expansion candidates and critic responses but has no MCTS statistics: it uses greedy LLM critic selection and recovers branch state by resetting the Docker environment and replaying actions from the root, rather than maintaining per-node file snapshots.

4.3.2 Context Compaction

All agents face the constraint of finite context windows, and compaction strategies range from no management to LLM-initiated compaction (Table 11).

The corpus reveals two distinct philosophies. "Prevention" agents bound context growth structurally: Prometheus scopes messages per graph node and resets them at retry boundaries; AutoCodeRover caps search rounds at 15 and shows at most 3 full results per query; Moatless Tools limits trajectory depth through tree structure. "Cure" agents let context grow and compress when needed: Aider, OpenHands, Gemini CLI, and Codex CLI all trigger LLM-based summarization at token thresholds. The prevention approach avoids summarization cost and information loss but

Table 11: Context compaction strategies.

Strategy	Agents	Mechanism
None (crash on overflow)	mini-swe-agent	Unbounded growth; agent crashes on `ContextWindowExceededError`.
Rule-based truncation	SWE-agent, DARS-Agent	SWE-agent: 7 composable processors; keep first + last N observations, elide rest. Polling parameter for prompt-cache preservation. DARS-Agent: adds `Last5Observations` in its fork.
Structural isolation	Prometheus, AutoCodeRover	Prometheus: per-node message scoping + resets at retry boundaries. AutoCodeRover: round limits (15) + result truncation.
Token-based selective inclusion	Moatless Tools	Greedy recent-first selection within token budget; per-observation `summary` fallback.
LLM summarization (scaffold-triggered)	Aider, OpenHands, Gemini CLI, Codex CLI, OpenCode	Automatic at token threshold. Aider: recursive hierarchical summarization. Codex CLI: pre-turn and mid-turn modes. OpenCode: two-phase (prune old outputs, then LLM summarization).
LLM summarization + verification	Gemini CLI	Summarize, then "Probe" verification turn to check for information loss.
LLM-initiated compaction	Cline	condense tool: LLM decides when to compact. Also supports scaffold-triggered compaction at token threshold.

requires the scaffold to anticipate context growth patterns. Agentless sidesteps the compaction problem entirely: because each LLM call is single-turn with no conversation history, "compaction" reduces to fitting code into a single prompt. When prompts exceed the 128K token limit, Agentless progressively drops files from the end of the ranked list and compresses function bodies to signatures via `libcst`. This pre-computation approach eliminates the need for runtime compaction by never accumulating conversational state. Among the "cure" agents, OpenCode's two-phase strategy is the most surgical: it first prunes verbose old tool outputs, maintaining message structure but replacing outputs older than the most recent 40,000 tokens with truncation markers. Only then does it trigger LLM-based summarization via a dedicated compaction agent that can use a cheaper model. This preserves more conversational context than pure truncation while being more targeted than full summarization. Codex CLI distinguishes between pre-turn and mid-turn compaction: pre-turn compaction runs before each user turn and clears reference context so the next turn reinjects initial context cleanly; mid-turn compaction runs when the token limit is hit during tool execution and injects initial context above the last user message, matching the model's training expectations about where context appears. This awareness of compaction timing relative to the conversation structure is unique in the corpus.

SWE-agent's polling parameter. SWE-agent's `LastNObservations` processor includes a `polling` parameter that reveals a subtle interaction between compaction and API cost. Many LLM providers offer prompt caching, where consecutive calls sharing the same message prefix skip reprocessing. Without polling, every new observation changes which messages are included, invalidating the cache. With polling, truncation changes occur at a rate of only $1/$`polling` per step, keeping the prefix stable across consecutive calls. This cost optimization operates at the intersection of two otherwise independent concerns and has not been documented in prior analyses of SWE-agent.

Gemini CLI's verification probe. Gemini CLI is the only agent that validates its own context compaction. After LLM summarization, it runs a "Probe" turn where the model checks whether critical information was lost. This self-correction mechanism addresses a known failure mode of LLM summarization (lossy compression of technical details) at the cost of an additional LLM call per compaction event.

Cline's LLM-initiated compaction. Cline's condense tool gives the LLM agency over its own context management. The LLM can proactively request summarization if it judges the context to be unwieldy. No other agent in the corpus delegates the compaction decision to the LLM. OpenHands provides a `CondensationRequestAction` that is conceptually similar, but compaction can also be triggered automatically when history exceeds condenser thresholds. OpenHands' condenser architecture is the most extensible in the corpus: nine pluggable implementations composable into pipelines via a registry pattern. Because condensation operates at the view level (inserting `CondensationAction` markers rather than deleting events), the raw event stream is never modified, enabling session replay even after aggressive compaction.

4.3.3 Multi-Model Routing

Multi-model routing (using different LLMs for different subtasks) ranges from single-model simplicity to multi-strategy classifier chains (Table 12). Two agents use a single model throughout: mini-swe-agent and Agentless. OpenHands provides extensible routing infrastructure but defaults to single-model operation; its only implemented router makes narrow decisions based on image presence and token limits. The remaining 10 agents route to multiple models, but for different reasons.

Cost optimization is the primary driver. Across the agents that use multiple models, the dominant motivation is cost: routing mechanical tasks to cheaper models while reserving expensive ones for reasoning. Aider's "weak model" handles summarization and commit messages; in architect mode, a third "editor model" receives the plan and generates edits. OpenCode extends this role-based approach with per-sub-agent model overrides.

Prometheus applies the same principle at a finer granularity. Each node in its LangGraph state machine is hard-coded to use either the "advanced" or "base" model. Reasoning-heavy nodes (analysis, editing, and patch selection, the last of which makes 10 majority-vote LLM calls) use the advanced model, while mechanical nodes (knowledge graph traversal, test execution) use the base model. Because the routing is per-node rather than per-role, the cost savings scale with the proportion of mechanical steps in the graph.

Gemini CLI's classifier chain. Gemini CLI's 7-layer routing strategy is the most complex routing mechanism observed. Each layer implements a different strategy, and the first to produce a decision wins, so simple cases resolve cheaply while ambiguous ones fall through to more sophisticated classifiers. The most distinctive layer is the optional `GemmaClassifierStrategy`, which runs a lightweight Gemma model [Gemma Team et al., 2024] locally for client-side routing decisions, avoiding an API call just to select a model. No other agent performs client-side model selection.

Actor-critic in tree search. Moatless Tools' value function (`value_function/base.py`) assigns numeric rewards to search tree nodes. When configured to use a different model than the action agent, the result is an actor-critic architecture: one model generates actions while a separate model

Table 12: Multi-model routing strategies.

Strategy	Agents	Mechanism
Single model	mini-swe-agent, Agentless (per path)	One model throughout.
Router abstraction (default single)	OpenHands	`RouterLLM` base with `MultimodalRouter`: routes by image presence and token limits (`llm/router/rule_based/impl.py:16--81`). Default: single model via `noop_router`.
Role-based	Aider, OpenCode	Main/weak/editor for different subtasks. Aider: weak model for summarization and commit messages (`models.py:607--608`); editor model for architect mode (`architect_coder.py:22--25`). OpenCode: per-agent model overrides for build, plan, explore, and compaction agents (`agent/agent.ts:78--233`).
Plan/Act mode-based	Cline	Independent model per mode (`api/index.ts:76--149`). LLM switches modes via tool calls.
Per-attempt cycling	SWE-agent, AutoCodeRover	Different models for retry attempts. SWE-agent: round-robin through `agent_configs` (`agents.py:303--319`). AutoCodeRover: round-robin through `model_names` (`inference.py:98--114`).
Safety-focused (Guardian)	Codex CLI	Separate model (`gpt-5.4`) evaluates tool call risk (`guardian.rs`).
Task-based dual-model	Prometheus	Advanced model for reasoning (analysis, editing, patch selection); base model for mechanical tasks (retrieval, verification) (`llm_service.py:23--38`). Hard-coded per graph node.
Actor-critic	Moatless Tools	Value function (potentially different model) scores nodes (`value_function/base.py`).
Classifier chain	Gemini CLI	7-layer priority routing: fallback → override → approval mode → Gemma classifier → LLM classifier → numerical classifier → default (`modelRouterService.ts:39--67`). Optional local Gemma model for client-side routing.

evaluates them. No other agent uses separate models for generation and evaluation; DARS-Agent's LLM critic performs a similar role but uses the same model for both, meaning the critic shares the generator's biases.

Other routing strategies. Two agents use per-attempt model cycling: SWE-agent and AutoCodeRover both rotate through different model configurations on retry attempts, betting that a model that failed on one attempt may succeed if a different model tries the same task from scratch. Cline routes by mode rather than by role: its plan and act modes each use an independently configured model and the LLM switches between modes via tool calls. Codex CLI uses the most models of any agent in the corpus: the primary model for code generation, a Guardian model (`gpt-5.4`) for safety evaluation of tool calls, and two dedicated models for its memory extraction pipeline (`gpt-5.1-codex-mini` for per-rollout extraction, `gpt-5.3-codex` for consolidation; Section 4.3.4). The Guardian is the only instance in the corpus of multi-model routing for safety rather than cost or capability reasons.

4.3.4 Persistent Memory

Persistent memory (knowledge that survives across sessions) varies from nonexistent to multi-tier extraction pipelines (Table 13). A clear division exists between agents designed for benchmark evaluation and those designed for interactive use. All five agents with no persistent memory (SWE-agent, OpenHands, AutoCodeRover, mini-swe-agent, DARS-Agent) treat each task as independent, with no cross-task learning. For benchmark-only agents (SWE-agent, AutoCodeRover, mini-swe-agent, DARS-Agent), this is expected; for OpenHands, which is also widely used for interactive development (53k stars, the second most in the corpus), the absence of persistent memory is notable, though its microagent system loads static project instructions that partially fill this role. Agentless also lacks cross-task learning but supports pipeline resumability via cached outputs and embedding indices, placing it between these two groups. The five CLI agents with persistent memory (Aider, Cline, Gemini CLI, Codex CLI, OpenCode) target interactive development where remembering project conventions and past decisions has direct value. Prometheus is an outlier: despite being a SWE-bench agent (Table 1), it implements multi-tier persistence (Neo4j, PostgreSQL, Athena) more characteristic of an interactive tool, suggesting architectural ambitions beyond benchmark evaluation. Among these, what "memory" means varies significantly: Cline and Gemini CLI persist learned rules, Codex CLI extracts and consolidates memories from past sessions, while OpenCode persists full session state in SQLite, including all messages, tool outputs, token usage, and costs enabling interrupted sessions to be resumed with complete context.

Static project instructions. Several agents in the "None" category do load static, user-written project instructions that persist across sessions. OpenHands reads `.openhands_instructions` and `.cursorrules` from the workspace via its microagent system, similar to Aider's `.aider.conf.yml`. These are not classified as persistent memory because the agent never writes or updates them; they are static configuration rather than learned knowledge.

LLM as memory author. Cline and Gemini CLI share a pattern where the LLM writes its own persistent instructions. Cline's new_rule tool creates `.clinerules` files; Gemini CLI's save_memory tool appends to `GEMINI.md`, which additionally supports `@path/to/file` references for composing instructions across multiple files. These memories persist in the project repository and are loaded into future sessions' system prompts. Codex CLI takes a different approach: a background pipeline extracts and consolidates memories without the LLM explicitly deciding what to remember.

Table 13: Persistent memory strategies.

Strategy	Agents	Mechanism
None	SWE-agent, OpenHands, AutoCodeRover, mini-swe-agent, DARS-Agent	No cross-session persistence. Each run starts fresh. Trajectory files are output artifacts, not consumed by future runs.
Pipeline resumability	Agentless	No cross-task learning, but JSONL outputs enable pipeline resumability (`--skip_existing`), embedding indices persist via `--persist_dir`, and precomputed repo structures are cached via `PROJECT_FILE_LOC`. These are consumed by future runs of the same pipeline, not cross-task knowledge.
Config file loading	Aider (`.aider.conf.yml`)	Static, user-written configuration. Tags cache (`repomap.py:43, 217--265`) persists AST analysis across sessions as a performance optimization.
LLM-writable rules/memory	Cline (`.clinerules/`), Gemini CLI (`GEMINI.md`)	The LLM actively writes persistent instructions. Cline: new_rule tool (`tools.ts:31`). Gemini CLI: save_memory tool appends under "Gemini Added Memories" in `GEMINI.md` (`memoryTool.ts`).
Full session persistence	OpenCode	SQLite-backed session history with resume capability. All messages, tool outputs, token usage, and costs persist (`session/session.sql.ts:14--76`).
Background extraction pipeline	Codex CLI	Two-phase: Phase 1 extracts memories from recent rollouts (parallel, `gpt-5.1-codex-mini`); Phase 2 consolidates via sub-agent (`gpt-5.3-codex`). Usage-ranked, stale memories pruned (`memories/README.md`).
Multi-tier persistence	Prometheus	Athena (semantic memory service, HTTP API) + Neo4j (knowledge graph, 20 language ASTs) + PostgreSQL (LangGraph checkpoints). Memory-first retrieval with KG fallback (`context_retrieval_subgraph.py:159--163`).

Cross-tool compatibility. Cline reads not only its own `.clinerules/` but also `.cursorrules`, `.windsurfrules`, and `AGENTS.md`. This pragmatic interoperability reflects an emerging ecosystem where developers use multiple AI coding tools with shared project-level instructions [Galster et al., 2026].

4.4 Cross-Cutting Themes

Several findings span multiple taxonomy dimensions and do not reduce to a single axis. These cross-cutting themes emerged from the open-ended section of the analysis template (Section 3.2) and represent architectural patterns or tradeoffs that manifest differently across dimensions rather than constituting dimensions themselves. They are presented separately because forcing them into the dimensional framework would obscure their multi-dimensional nature.

4.4.1 Sampling vs. Iteration

When an LLM-generated patch fails, there are two fundamentally different ways to try again: generate another independent attempt (sampling), or refine the failed attempt using feedback from the failure (iteration). This distinction cuts across the control loop taxonomy because it describes how agents handle the *population* of solution attempts, not the structure of any single attempt.

Agentless is the purest example of sampling: it generates multiple patches independently and selects by majority voting (Section 4.1.1). Each patch is generated from the same context; no patch benefits from knowing that another failed.

Six of the nine LLM-driven agents (OpenHands, Codex CLI, Gemini CLI, Cline, mini-swe-agent, OpenCode) take the opposite approach: pure iteration. A single attempt is refined through a feedback loop where each step sees the results of previous steps. If a test fails, the agent sees

the error message and adjusts. This depth-first strategy bets that feedback is more valuable than independence, and that a single guided trajectory is more likely to converge on a correct solution than multiple unguided ones.

Majority voting also appears in Prometheus, but at a different level. Rather than generating patches independently, Prometheus calls the advanced model 10 times on the same prompt to *select among* already-generated candidate patches with early stopping when the vote lead exceeds remaining votes. Here, voting operates at the decision layer (choosing which patch to submit) rather than the generation layer (creating patches).

SWE-agent's `RetryAgent` occupies a middle ground between sampling and iteration. It generates multiple complete attempts, each a full depth-first trajectory with its own feedback loop, and selects the best via a reviewer model. Each attempt can use a different model configuration. The result combines iteration within each attempt (feedback-driven refinement) with sampling across attempts (independent trajectories evaluated post-hoc).

4.4.2 Sub-Agent Delegation

Five agents support sub-agent spawning, where the primary agent creates a secondary agent instance to handle a subtask. This matters because it enables parallelism (working on multiple files simultaneously) and specialization (delegating to agents with different tool permissions or system prompts). However, these five agents implement delegation through fundamentally different mechanisms, revealing different assumptions about who should control the delegation decision.

Codex CLI gives the LLM full control over delegation by exposing it as a suite of tools: spawn_agent, send_input, resume_agent, wait, and close_agent. The LLM decides when to spawn a sub-agent, what task to assign it, and when to collect results. Depth is limited by `agent_max_depth`, and collaboration tools are disabled at maximum depth to prevent unbounded recursion.

OpenCode takes a role-based approach: its task tool spawns sub-agents with different specializations (build, plan, explore, general), each with scaffold-enforced tool permissions. The plan agent disables file-editing tools (`edit`, `write`) for most paths but retains bash access; the explore agent enables read-oriented tools plus bash while denying write-oriented tools by default. The LLM chooses which specialist to invoke, but the available specializations and their capabilities are defined by the scaffold. This is a structural constraint, distinct from Cline's plan/act mode switching where the LLM itself decides when to transition between modes via tool calls.

OpenHands integrates delegation into its event-sourced architecture. A parent `AgentController` creates a child controller via `AgentDelegateAction` and forwards events to the delegate. Because delegation flows through the same event stream as all other actions, it is automatically captured in the agent's history and subject to the same condensation and replay mechanisms.

Cline offers simpler tool-based delegation via new_task and use_subagents. Gemini CLI supports sub-agents through its `LocalAgentExecutor`, which runs a separate ReAct loop with its own turn limits and deadline timer; uniquely, it includes a recovery phase that gives the sub-agent one final turn to produce output when the deadline expires. Prometheus's graph structure delegates implicitly through subgraph nesting rather than explicit spawning.

4.4.3 Online vs. Offline Selection

Tree-search agents face two distinct selection problems: which branch to explore next *during* the search (online guidance), and which completed solution to return *after* the search finishes (offline selection). These two problems can be solved by the same mechanism or by different ones, with different tradeoffs.

DARS-Agent separates the two cleanly. During the search, an LLM critic selects among candidate actions at each branch point; this is online guidance that shapes which parts of the search tree get explored. After the search completes, a separately trained reviewer evaluates the finished patches; this is offline selection that chooses the final output. The separation means each component can be optimized independently: the online critic needs to be fast (it runs at every branch point), while the offline reviewer can be more thorough (it runs once at the end).

Moatless Tools also separates the two components. The value function assigns numeric rewards during online MCTS simulation; a separate discriminator evaluates and selects the best completed trajectory offline. Unlike DARS-Agent's independently trained reviewer, both components can be configured to use the same model, allowing the discriminator to apply evaluation criteria consistent with the search guidance.

The contrast extends to how each agent extracts its final answer. DARS-Agent's leftmost-path extraction always takes `children[0]`, relying entirely on the online critic to have directed the search toward good solutions. Moatless Tools' discriminator actively re-evaluates all completed trajectories, potentially selecting one that was not the most-visited during search.

4.4.4 Ecosystem Maturity

The relationship between DARS-Agent and SWE-agent illustrates the current state of ecosystem maturity. When DARS-Agent needed to add tree-search capabilities on top of SWE-agent's ReAct loop, it copied and modified the entire codebase rather than extending it: `agents.py` is a 700-line copy of the original SWE-agent `Agent` class, and `dars_agent.py` is a parallel 1382-line reimplementation with tree-search logic added. This fork-based reuse means bug fixes and improvements in either project do not propagate to the other.

mini-swe-agent takes the opposite approach, reusing SWE-agent's execution environments (including its SWE-ReX Docker and Modal backends) while defining its own abstractions on top. It specifies agent, model, and environment as Python Protocols, a form of structural typing [Levkivskyi et al., 2017] where any object with the right methods automatically conforms to the interface. This makes it possible to swap in alternative implementations without modifying mini-swe-agent's code.

That fork-based and dependency-based reuse coexist for the same upstream project suggests the ecosystem has not yet stabilized around clean extension points. Moatless Tools' dual-flow architecture (Section 4.1.1) offers a glimpse of what such stabilization could look like: a clean separation between per-step logic and orchestration strategy that makes switching between sequential and tree-search modes a configuration choice.

4.4.5 IDE as Architecture

Cline is the only agent in the corpus that runs as an IDE extension rather than a standalone CLI or server [Rizwan, 2024]. This architectural choice gives it access to a category of context that CLI agents cannot obtain: the IDE's own understanding of the project.

For example, the `@problems` mention pulls from VS Code's diagnostic API, giving the agent access to the same type errors, linting warnings, and build failures that a developer sees in the editor's "Problems" panel. The `@terminal` mention accesses integrated terminal output, and commands execute in visible terminal panels via the VS Code terminal API, so the user can watch commands run in real time. The `FileContextTracker` monitors which files the model has seen, read, or modified across turns, detecting external modifications to prevent stale context during diff editing.

The tradeoff is platform lock-in. These capabilities depend on VS Code's extension API, and Cline's core features (diagnostics integration, file watching, checkpoint git integration) cannot run outside VS Code. The codebase shows signs of ongoing decoupling (a standalone terminal manager, a CLI mode), but the richest context sources remain VS Code-dependent.

5 Discussion

The results presented in Section 4 describe 12 dimensions across three architectural layers, each exhibiting a range of strategies observed in 13 open-source coding agents. This section interprets those findings along three lines corresponding to the contributions stated in Section 1: what the spectral, compositional character of the design space means for how researchers classify agents (Sections 5.1–5.2), what the taxonomy enables for researchers studying agent behavior and practitioners building new scaffolds (Sections 5.3–5.4), and what the evidence base reveals about ecosystem maturity and evaluation methodology (Sections 5.5–5.6).

5.1 Spectra, Not Categories

The most persistent finding across all 12 dimensions is that scaffold architectures resist discrete classification. Prior taxonomies of LLM agents organize systems into categories defined by abstract capabilities: tool-using, memory-augmented, planning, reflective [Masterman et al., 2024, Nowaczyk, 2025]. Every agent in the present corpus qualifies for every one of these categories, yet their implementations differ in ways that these labels cannot express. The source-code analysis reveals that the variation is better captured as continuous spectra: control loops range from fixed pipelines to full MCTS (Section 4.1.1), tool counts range from 0 to 37 (Section 4.2.1), context compaction ranges from none to LLM-initiated compression (Section 4.3.2), and state management ranges from destructive overwrite to event sourcing (Section 4.3.1). Within each spectrum, agents occupy distinct positions that reflect genuine architectural tradeoffs rather than arbitrary implementation choices.

This spectral character has a structural explanation: the loop primitives identified in Section 4.1.1 (ReAct, generate-test-repair, plan-execute, multi-attempt retry, tree search) function as composable building blocks that agents freely layer and nest. Because these primitives compose freely, the space of possible architectures is combinatorial rather than categorical. Assigning a single label ("ReAct agent," "pipeline agent") to a system that layers multiple primitives obscures the design decisions that actually differentiate it.

For researchers, this finding suggests that evaluating scaffold dimensions independently may be more informative than classifying whole agents. A study comparing "ReAct agents" against "pipeline agents" conflates loop topology, loop driver, tool set design, and context management into a single binary. Decomposing along the dimensions identified here would allow more precise attribution of behavioral differences to specific architectural choices. For practitioners, the composability of loop primitives suggests that design decisions may be more orthogonal than they appear: Moatless Tools demonstrates that tree search can be layered over an existing per-step agent without rewriting the agent logic (Section 4.1.1), though whether all dimension combinations are equally feasible remains an open empirical question.

5.2 Convergence and Divergence

Across the 12 dimensions, some show strong convergence among agents while others show wide divergence, and the pattern is informative. The converging dimensions tend to reflect constraints

that are external to the scaffold designer: tool capability categories converge on reading, searching, editing, and executing code (Section 4.2.1) because these are the operations that software engineering tasks require, regardless of architectural philosophy. Edit format is trending toward string replacement (Section 4.2.2) because LLMs produce more reliable edits with exact string matching than with line-number-based or unified-diff formats whether through independent discovery or imitation of successful designs. Execution isolation converges on Docker containers for benchmark agents (Section 4.2.5) because autonomous code execution without sandboxing is unacceptable for unattended evaluation. These convergences reflect solved problems or hard constraints: the design space has been explored, and practitioners have settled on solutions that work.

The diverging dimensions tell a different story. Context compaction (Section 4.3.2) exhibits seven distinct strategies across 13 agents, from no management at all to LLM-initiated compression with verification probes. State management (Section 4.3.1) ranges from destructive overwrite to event sourcing, with tree-structured, graph-scoped, and database-backed variants between them. Multi-model routing (Section 4.3.3) spans single-model simplicity to seven-layer classifier chains. These dimensions diverge because they address open design questions where no dominant solution has emerged. Context compaction, for instance, requires balancing information preservation against token cost, with the optimal tradeoff depending on task length, model capability, and cost tolerance in ways that no single strategy resolves. State management involves similar tradeoffs between simplicity, auditability, and support for branching exploration. The divergence on these dimensions is not noise; it reflects genuine uncertainty about the best approach.

This pattern has practical implications. The converging dimensions are candidates for standardization: a shared tool protocol covering the four capability categories (as the Model Context Protocol begins to attempt) would reduce duplicated effort without constraining architectural innovation. The diverging dimensions, conversely, are where research investment is most needed. The diversity of context compaction strategies, in particular, suggests that no existing approach fully solves the "token snowball" problem identified by Fan et al. [2025], where growing context from tool outputs degrades both performance and cost. The range of observed strategies (prevention through structural bounding, cure through summarization, and hybrid approaches) represents an active design frontier.

5.3 The Scaffold-Model Interface

The taxonomy reveals that scaffold design mediates model capability in ways that prior work has not systematically examined. The same underlying language model behaves differently depending on how many tools it is presented (0 in Aider versus 35 in SWE-agent, Section 4.2.1), how context is managed (full unfiltered history in mini-swe-agent versus event-sourced views with condensation in OpenHands, Section 4.3.2), what loop structure surrounds it (single-pass pipeline in Agentless versus feedback-driven iteration in SWE-agent, Section 4.1.1), and who drives the loop (user in Aider, scaffold in AutoCodeRover, LLM in OpenHands, Section 4.1.2). Each of these scaffold-level decisions shapes what the model sees, what actions it can take, and how its errors propagate or get corrected.

This observation has direct consequences for empirical studies of coding agents. Trajectory analyses that compare agents using different models [Majgaonkar et al., 2026] cannot isolate whether an observed behavioral difference stems from the scaffold or the model. The present taxonomy provides the vocabulary to describe exactly which scaffold dimensions differ between two agents, making it possible to design controlled comparisons. For example, a study could compare agents with identical tool sets but different loop strategies (Section 4.1.1), or identical loops but different compaction strategies (Section 4.3.2), holding the model constant. Souza and Machado [2026] called

for architecture-aware evaluation metrics that link internal components to observable outcomes; the 12 dimensions identified here provide the architectural variables that such metrics would need to control for.

The scaffold-model interface also explains why the loop driver dimension (Section 4.1.2) is arguably the most fundamental architectural distinction. As Section 4.1.2 documents, user-driven agents sidestep the localization bottleneck entirely, while LLM-driven agents must solve it, making retrieval strategy a critical co-design choice. The same model that performs well with user-curated context may struggle when forced to navigate a repository autonomously. This interaction illustrates why scaffold dimensions cannot be evaluated in isolation; they form an interdependent design space.

5.4 Implications for Agent Design

Several practical design lessons emerge from the taxonomy, though they should be understood as patterns observed across 13 agents rather than prescriptive recommendations.

Loop composition as a design strategy. Eleven of the 13 agents layer multiple loop primitives rather than relying on a single control structure (Section 4.1.1). Pure single-loop agents are rare: Agentless (pipeline only) and mini-swe-agent (ReAct only) are the closest examples, and both are deliberately minimalist. This pattern suggests that the ReAct loop, while foundational, is typically insufficient on its own; layering a retry, test-repair, or planning primitive on top addresses failure modes that a single feedback loop cannot handle [Shinn et al., 2023].

Tool count and the capability-confusion tradeoff. Tool counts range from 0 to 37, yet the underlying capability categories converge on four (read, search, edit, execute). This convergence at the capability level despite divergence at the tool level suggests that the four categories define a minimum viable toolset for autonomous coding agents. Beyond this baseline, the tradeoff is between expressiveness and LLM confusion: more specialized tools reduce the LLM's per-tool reasoning burden but increase the action space it must navigate [Yang et al., 2024]. Prometheus's per-node tool scoping (Section 4.2.1) and AutoCodeRover's phase separation offer two strategies for managing this tradeoff, constraining the tools visible at each decision point rather than presenting the full set at every step.

Context compaction as an architectural requirement. Every agent that gives the LLM sustained autonomy must address context growth. Section 4.3.2 documents two philosophies: prevention (bounding context structurally) and cure (compressing on demand). Prevention avoids information loss but requires anticipating growth patterns; cure is more flexible but risks lossy compression. The only agent with no compaction strategy (mini-swe-agent) crashes when the context window is exceeded, confirming that compaction is not optional for agents operating beyond trivial task lengths.

Sub-agent delegation as an emerging capability. Five of the 13 agents support explicit sub-agent spawning through five distinct mechanisms (Section 4.4.2); a sixth, Prometheus, achieves implicit delegation through subgraph nesting rather than explicit spawning. The diversity and the absence of a dominant pattern suggest that delegation is an active design frontier. The variation in who controls the delegation decision mirrors the loop driver spectrum (Section 4.1.2), suggesting that the autonomy-versus-control tradeoff recurs at every level of agent architecture.

5.5 Implications for Agent Evaluation

As noted in Section 3.5, SWE-bench comparisons between agents confound scaffold design, model choice, and configuration in a single metric. The taxonomy makes this confounding concrete. As Section 4.3.3 documents, agents use different models and different numbers of models; Codex CLI routes to four distinct models while mini-swe-agent uses one. Per-attempt model cycling in SWE-agent and AutoCodeRover (Section 4.3.3) means that a single benchmark run may involve multiple models, further complicating attribution. Controlling for these differences requires the kind of architectural decomposition that the present taxonomy provides but that existing evaluations do not perform.

The sampling-versus-iteration distinction (Section 4.4.1) poses a particularly acute evaluation challenge. Agentless's independent sampling strategy and SWE-agent's iterative retry strategy represent fundamentally different approaches to the same problem, but benchmark scores conflate single-attempt quality with multi-attempt strategy. An agent that produces mediocre individual patches but samples 40 of them and selects the best may outperform an agent that produces strong individual patches through iterative refinement. Separating these two capabilities in evaluation would require reporting both single-attempt and multi-attempt metrics, a distinction the taxonomy makes legible but current benchmarks do not enforce.

Architecture-aware evaluation need not require entirely new benchmarks. The 12 dimensions identified here suggest concrete controls that could be applied within existing evaluation frameworks. For instance, fixing the tool set (Section 4.2.1) while varying the control loop (Section 4.1.1) would isolate the effect of loop strategy on task success. Fixing the model while varying the scaffold would isolate scaffold effects from model effects. The taxonomy provides the variables; the evaluation methodology is a matter of experimental design.

5.6 Ecosystem Maturity and Standardization

The state of reuse and modularity across the corpus reflects an ecosystem that is innovating rapidly but has not yet stabilized around shared abstractions. As Section 4.4.4 documents, fork-based and dependency-based reuse coexist for the same upstream project, indicating that clean extension points have not yet emerged.

The convergence on tool capability categories without convergence on tool interfaces (Section 4.2.1) suggests a specific standardization opportunity. All LLM-driven agents need tools for reading, searching, editing, and executing code, but each agent defines its own tool schemas, parameter names, and output formats. A shared tool interface protocol could reduce the integration cost of new tools without constraining how scaffolds orchestrate them. The Model Context Protocol (MCP) [Anthropic, 2024], already supported by five agents in the corpus (OpenHands, Codex CLI, Gemini CLI, Cline, OpenCode), represents an early attempt at this kind of standardization, though it operates at the transport layer rather than defining semantic contracts for specific tool categories.

Moatless Tools' dual-flow architecture (Section 4.1.1) offers a more focused model of what modular scaffold design could look like. Its clean separation between the per-step executor (`ActionAgent`) and the orchestration strategy (`AgenticLoop` or `SearchTree`) means that adding a new exploration strategy requires implementing a new orchestrator, not modifying the agent logic. This separation of concerns is rare in the corpus; most agents tightly couple their per-step logic with their orchestration strategy, making it difficult to experiment with alternative control structures without substantial refactoring. As the ecosystem matures, this kind of modular separation may prove more valuable than tool protocol standardization, because it addresses the architectural level where the most design variation exists (Section 5.2).

Cline's IDE integration (Section 4.4.5) illustrates a different facet of ecosystem evolution: the tradeoff between platform coupling and capability richness. The IDE-native context that Cline accesses (diagnostics, file-change tracking, terminal integration) is unavailable to CLI agents, but comes at the cost of VS Code platform lock-in. For the ecosystem as a whole, this tension may resolve through protocol-level abstraction (providing IDE-quality context via standardized APIs) rather than platform convergence, but no such abstraction exists today.

6 Threats to Validity

This section organizes threats to validity following the standard framework for empirical software engineering studies [Runeson and Höst, 2009]: construct validity (whether the study measures what it claims to measure), internal validity (whether the findings follow from the data), external validity (whether the findings generalize beyond this study), and reliability (whether the study can be reproduced).

6.1 Construct Validity

The primary construct validity threat is single-author bias. All 13 agent analyses were conducted by a single author (with LLM-assisted code navigation, as described in Section 3.4), meaning that dimension classifications, evidence selection, and cross-agent comparisons reflect one person's interpretation of the source code. Two mitigations partially address this threat. First, every taxonomic claim in Section 4 is grounded in specific file paths and line numbers pinned to commit hashes (Appendix B), making each claim independently verifiable against the source code. A post-hoc verification pass checked 296 extracted claims against the cloned repositories, confirming 267, correcting 19 (primarily line number offsets from code evolution between analysis and verification dates), and accepting 10 as minor simplifications (e.g., describing a multi-step process in fewer steps than the code implements, or attributing a behavior to a single function when it spans two). The verification pass was performed by the same researcher who conducted the original analysis; while self-verification is weaker than independent review, the commit-pinned evidence trail makes independent verification straightforward. Second, the analysis template separates observation (what the code does), classification (how it maps to a dimension), and evidence (file path and line number), following case study reporting guidelines [Runeson and Höst, 2009]. This separation makes it possible for a reader to evaluate the classification independently of the observation. Nonetheless, the study would benefit from independent replication by a second analyst, particularly for dimensions where judgment calls are required (for example, whether Prometheus's graph-scoped state management constitutes a form of structural compaction or a separate architectural pattern).

A second construct validity threat concerns the dimension framework itself. Although the nine analysis dimensions were derived iteratively through open coding [Strauss and Corbin, 1998] during a pilot analysis of two architecturally contrasting agents (Section 3.2), the pilot agents (Aider and OpenHands) may not have surfaced all relevant dimensions. The open-ended tenth section of the analysis template was designed to capture observations outside the predefined dimensions, and it produced 47 cross-cutting findings that informed the final taxonomy (Section 4.4). However, dimensions that neither pilot agent exhibits and that no subsequent agent made salient could still be missing. For example, the taxonomy does not include a dimension for prompt engineering strategy (how system prompts are constructed and varied). While prompt templates are visible in source code, the dimension was excluded for scope: analyzing prompt structure, length, few-shot examples, and persona instructions across 13 agents would constitute a study in its own right, and the architectural impact of prompt differences (as opposed to scaffold differences) cannot be assessed

without runtime experimentation. This exclusion is deliberate but means that an important aspect of scaffold design is not captured.

6.2 Internal Validity

Internal validity concerns whether the observed patterns genuinely reflect architectural relationships rather than confounds. Two threats are relevant.

First, the taxonomy describes source code at pinned commits, but several agents were under active development during the analysis period (Section 3.4). Architectural features may have been added, removed, or significantly refactored between the analyzed commit and the current version. Pinning to specific commits ensures reproducibility of the reported findings but means the taxonomy is a snapshot, not a live description. The commit hashes are listed in Appendix B so that readers can assess how much each agent has evolved since analysis.

Second, some dimensions may not be as independent as the taxonomy implies. Section 5.3 notes that loop driver and retrieval strategy are correlated: scaffold-driven agents tend to invest in retrieval infrastructure while LLM-driven agents rely on general-purpose tools. Similar correlations may exist between other dimensions (for example, between tool discovery strategy and tool count, or between state management and context compaction). The taxonomy presents dimensions as independent axes, but in practice they form an interdependent design space where choices on one dimension constrain options on others. The cross-cutting themes in Section 4.4 capture some of these interdependencies, but a full analysis of dimension interactions is beyond the scope of this study.

6.3 External Validity

Three threats limit the generalizability of the findings.

First, the corpus is restricted to open-source agents with readable source code (Section 3.1). Proprietary coding agents (GitHub Copilot Workspace, Cursor's AI backend, Windsurf) and agents with compiled or obfuscated source code (Claude Code) are excluded because their scaffolding is not publicly inspectable. This introduces a survivorship bias: open-source agents may systematically differ from proprietary agents in design choices driven by business constraints, proprietary model access, or different optimization targets (user experience versus benchmark scores). The taxonomy should therefore be understood as describing the open-source design space, not the full design space of coding agents.

Second, the corpus of 13 agents, while covering a range of architectural strategies, is not exhaustive. Agents released after the analysis period or agents that did not meet the inclusion criteria may exhibit architectural patterns not represented in the taxonomy. The study aims for analytical generalizability [Yin, 2018] (the dimensions and spectra should be useful for characterizing new agents) rather than statistical generalizability (the distribution of agents across dimension positions is not claimed to be representative of any population). As the ecosystem evolves, both new dimensions and new positions on existing dimensions are likely to emerge.

Third, the corpus is dominated by agents targeting Python-language repositories, primarily because SWE-bench (the dominant evaluation benchmark) uses Python projects exclusively. Agents designed for multi-language or non-Python ecosystems may face different architectural constraints (for example, different AST parsing requirements, different build and test toolchains, or different dependency resolution patterns) that could produce architectural variation not observed here [Jimenez et al., 2024, Xu et al., 2025]. Prometheus's 20-language tree-sitter support and SWE-agent's language-agnostic shell-based approach suggest that some agents already address this

limitation, but the analysis does not systematically evaluate how architectural choices vary across target languages.

6.4 Reliability

The primary reliability threat is reproducibility. The fully specified template (Section 3.4) and pinned commit hashes (Appendix B) enable a second analyst to arrive at substantially similar observations, though classification judgments (where to place an agent on a continuous spectrum) may differ. The complete analysis documents enable independent scrutiny.

A secondary concern is that static source-code analysis misses runtime behavior. Some architectural features (MCP tool discovery in practice, whether configurable features like Moatless Tools' pluggable selector are used in typical deployments) may only be visible at runtime. The taxonomy describes architectural capability, not observed runtime behavior.

The corpus is also heavily concentrated in Python-implemented agents (10 of 13); the three TypeScript agents (Cline, Gemini CLI, OpenCode) may use language idioms (event-driven architectures, module systems) that a Python-oriented analysis framework is less attuned to. The analysis template was designed to be language-agnostic, but subtle biases in what the analyst notices cannot be ruled out.

Finally, the CLI/SWE-bench category distinction used throughout the paper (Table 1) is a simplification. Some agents straddle categories: OpenHands is categorized as SWE-bench but is also widely used as an interactive development tool, and several SWE-bench agents can be run interactively. The distinction is used descriptively (to contextualize design choices) rather than analytically (as a dimension of the taxonomy), but readers should not interpret it as a rigid boundary.

7 Conclusion

This paper presented a source-code-level architectural taxonomy of 13 open-source coding agent scaffolds, organized into three layers (control architecture, tool and environment interface, resource management) and 12 dimensions. Every taxonomic claim is grounded in file paths and line numbers from cloned repositories at pinned commits, providing an evidence base that can be independently verified and extended as the ecosystem evolves.

Three findings emerge from the analysis. First, scaffold architectures are better characterized as positions along continuous spectra than as instances of discrete types. Control strategies range from fixed pipelines to full Monte Carlo Tree Search; tool counts range from 0 to 37; context compaction spans seven distinct strategies; state management ranges from destructive overwrite to event sourcing. Prior capability-based taxonomies that classify agents as "tool-using" or "planning" cannot distinguish between systems that differ fundamentally on these dimensions. Second, the loop primitives that underlie control architectures (ReAct, generate-test-repair, plan-execute, multi-attempt retry, tree search) function as composable building blocks: 11 of the 13 agents layer multiple primitives rather than relying on a single control structure. This compositionality means the design space is combinatorial rather than categorical, and assigning a single architectural label to a system obscures the design decisions that differentiate it. Third, the dimensions themselves exhibit a pattern of convergence on externally constrained choices (tool capability categories, edit formats, execution isolation) and divergence on open design questions (context compaction, state management, multi-model routing), suggesting where the design space has stabilized and where research investment is most needed.

Several directions for future work follow from the taxonomy. The most direct extension is controlled experimentation: the 12 dimensions identify specific architectural variables that can be isolated while holding others constant. For example, comparing agents with identical tool sets but different loop strategies, or identical loops but different compaction strategies, with the model held constant, would enable causal attribution of performance differences to scaffold design rather than to the model or configuration confounds that current benchmark comparisons cannot disentangle (Section 5.5). The taxonomy provides the variables; designing the experiments is the next step.

A second direction is longitudinal analysis. The taxonomy describes a snapshot of 13 agents at pinned commits. Repeating the analysis at later commits would reveal how scaffold architectures evolve: whether converging dimensions continue to converge, whether diverging dimensions stabilize, and whether new dimensions emerge as the ecosystem matures. The commit-pinned methodology makes such longitudinal comparison straightforward.

Third, extending the corpus to proprietary agents (when architectural details become available through documentation or reverse engineering) and to agents targeting languages beyond Python would test the generalizability of the observed spectra (Section 6). The dimension framework is designed to be language- and platform-agnostic, but whether the specific positions observed here generalize to different ecosystems remains an open question.

Finally, the taxonomy enables the architecture-aware evaluation metrics that prior work has called for [Souza and Machado, 2026], but could not implement without architectural documentation. Linking specific dimension positions (loop strategy, compaction approach, tool set design) to observable outcomes (task success, token cost, trajectory length) would move the field from system-level leaderboards toward component-level understanding of what makes coding agents effective.

References

Rui Abreu, Peter Zoeteweij, and Arjan J. C. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART)*, pages 89–98. IEEE, 2007.

Vaibhav Aggarwal, Ojasv Kamal, Abhinav Japesh, Zhijing Jin, and Bernhard Schölkopf. DARS: Dynamic action re-sampling to enhance coding agent performance by adaptive tree traversal. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025.

Anomaly. OpenCode. `https://github.com/anomalyco/opencode`, 2025.

Anthropic. Model Context Protocol. `https://modelcontextprotocol.io`, 2024. Open specification for connecting AI assistants to external tools and data sources.

Anthropic. Claude Code. `https://github.com/anthropics/claude-code`, 2025. The open-source repository contains plugins, examples, and documentation, but the core agent source is distributed as compiled TypeScript bundles in the npm package.

Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. MASAI: Modular architecture for software-engineering AI agents. *arXiv preprint arXiv:2406.11638*, 2024.

Hudson Borges and Marco Tulio Valente. What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129, 2018.

Islem Bouzenia and Michael Pradel. Understanding software engineering agents: A study of thought-action-result trajectories. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2025. arXiv preprint arXiv:2506.18824.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. RepairAgent: An autonomous, LLM-based agent for program repair. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 2188–2200, 2025.

Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

Cameron B. Browne, Edward Powley, Daniel Whitehouse, et al. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

Max Brunsfeld et al. Tree-sitter: An incremental parsing system for programming tools. `https://tree-sitter.github.io/tree-sitter/`, 2018.

Nghi D. Q. Bui. Building AI coding agents for the terminal: Scaffolding, harness, context engineering, and lessons learned. *arXiv preprint arXiv:2603.05344*, 2026.

Ira Ceka, Saurabh Pujar, Shyam Ramji, Luca Buratti, Gail Kaiser, and Baishakhi Ray. Understanding software engineering agents through the lens of traceability: An empirical study. *arXiv preprint arXiv:2506.08311*, 2025.

Zhi Chen, Zhensu Sun, Yuling Shi, Chao Peng, Xiaodong Gu, David Lo, and Lingxiao Jiang. Rethinking the value of agent-generated tests for LLM-based software engineering agents. *arXiv preprint arXiv:2602.07900*, 2026.

CrewAI Inc. CrewAI. `https://github.com/crewAIInc/crewAI`, 2024.

Xiang Deng, Jeff Da, Edwin Pan, et al. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Zhiyu Fan, Kirill Vasilevski, Dayi Lin, Boyuan Chen, Yihao Chen, Zhiqing Zhong, Jie M. Zhang, Pinjia He, and Ahmed E. Hassan. SWE-Effi: Re-evaluating software AI agent system effectiveness under resource constraints. *arXiv preprint arXiv:2509.09853*, 2025.

Flatpak Project. Bubblewrap: Unprivileged sandboxing tool. `https://github.com/containers/bubblewrap`, 2016.

Martin Fowler. Event sourcing, 2005. URL `https://martinfowler.com/eaaDev/EventSourcing.html`.

Matthias Galster, Seyedmoein Mohsenimofidi, Jai Lal Lulla, Muhammad Auwal Abubakar, Christoph Treude, and Sebastian Baltes. Configuring agentic AI coding tools: An exploratory study. *arXiv preprint arXiv:2602.14690*, 2026.

Spandan Garg, Benjamin Steenhoek, and Yufan Huang. Saving SWE-Bench: A benchmark mutation approach for realistic agent evaluation. In *Proceedings of the 5th IEEE/ACM International Conference on AI Engineering – Software Engineering for AI (CAIN)*, 2026. arXiv preprint arXiv:2510.08996.

Paul Gauthier. Aider: AI pair programming in your terminal. https://aider.chat, 2024.

Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, et al. Gemma: Open models based on Gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024.

Google. Gemini CLI. https://github.com/google-gemini/gemini-cli, 2025.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, et al. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations (ICLR)*, 2024.

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? In *International Conference on Learning Representations (ICLR)*, 2024.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2021.

LangChain. LangGraph. https://github.com/langchain-ai/langgraph, 2024.

LangChain AI. Deep Agents. https://github.com/langchain-ai/deepagents, 2025.

Thibault Le Sellier De Chezelles, Maxime Gasse, Alexandre Drouin, Massimo Caccia, Léo Boisvert, Megh Thakkar, Tom Marty, Rim Assouel, Sahar Omidi Shayegan, Lawrence Keunho Jang, et al. The BrowserGym ecosystem for web agent research. *arXiv preprint arXiv:2412.05467*, 2024.

Ivan Levkivskyi, Jukka Lehtosalo, and Łukasz Langa. PEP 544 – protocols: Structural subtyping (static duck typing). https://peps.python.org/pep-0544/, 2017.

LlamaIndex. LlamaIndex. https://github.com/run-llama/llama_index, 2022.

Oorja Majgaonkar, Zhiwei Fei, Xiang Li, Federica Sarro, and He Ye. Understanding code agent behaviour: An empirical study of success and failure trajectories. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2026. arXiv preprint arXiv:2511.00197.

Tula Masterman, Sandi Besen, Mason Sawtell, and Alex Chao. The landscape of emerging AI agent architectures for reasoning, planning, and tool calling: A survey. *arXiv preprint arXiv:2404.11584*, 2024.

Neo4j, Inc. Neo4j graph database. https://neo4j.com, 2024.

Sławomir Nowaczyk. Architectures for building agentic AI. *arXiv preprint arXiv:2512.09458*, 2025.

Open Interpreter, Inc. Open Interpreter. https://github.com/OpenInterpreter/open-interpreter, 2023.

OpenAI. Codex CLI. https://github.com/openai/codex, 2025.

Albert Orwall. Moatless Tools. https://github.com/aorwall/moatless-tools, 2024.

Yue Pan, Zimin Chen, Siyu Lu, Zhaoyang Chu, Xiang Li, Han Li, Yang Feng, Claire Le Goues, Federica Sarro, Martin Monperrus, and He Ye. Prometheus: Towards long-horizon codebase navigation for repository-level problem solving. *arXiv preprint arXiv:2507.19942*, 2026.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Saoud Rizwan. Cline. `https://github.com/cline/cline`, 2024.

Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

Mickaël Salaün. Landlock LSM: Toward unprivileged sandboxing. In *Linux Security Summit*, 2017. `https://landlock.io`.

Johnny Saldaña. *The Coding Manual for Qualitative Researchers*. SAGE Publications, 4th edition, 2021.

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. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Débora Souza and Patrícia Machado. Toward architecture-aware evaluation metrics for LLM agents. In *Proceedings of the 5th IEEE/ACM International Conference on AI Engineering – Software Engineering for AI (CAIN)*, 2026. arXiv preprint arXiv:2601.19583.

Anselm Strauss and Juliet Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, 2nd edition, 1998.

Xingyao Wang, Boxuan Li, Yufan Song, et al. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv preprint arXiv:2407.16741.

W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2:801–824, 2025. FSE 2025. ACM SIGSOFT Distinguished Paper Award.

Jingxuan Xu, Ken Deng, Weihao Li, et al. SWE-Compass: Towards unified evaluation of agentic coding abilities for large language models. *arXiv preprint arXiv:2511.05459*, 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. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

John Yang, Carlos E. Jimenez, et al. mini-swe-agent. https://github.com/SWE-agent/mini-swe-agent, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Robert K. Yin. *Case Study Research and Applications: Design and Methods*. SAGE Publications, 6th edition, 2018.

Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. A survey of learning-based automated program repair. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–69, 2024a.

Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. A survey on large language models for software engineering. *Science China Information Sciences*, 69:141102, 2026. doi: 10.1007/s11432-025-4670-0.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024b.

A Candidate Agent Corpus

Table 14 lists the full pool of 22 candidate agents considered for this study, along with the inclusion criterion each excluded agent failed. The three inclusion criteria are defined in Section 3.1.