From Stumbling to Shipping

Coding Agents in 2025: Technical Architecture, Benchmarks, and Research Directions

EAIRG Engineering AI Research Group • September 2025

A practical architecture + evals view for agentic coding

What to ship this quarter: a planner-actor-verifier loop, trace logging, and a verifiable eval harness.

The Evolution: From Snippets to Swarms

Research Foundation: Key Papers

Core Techniques:

- **Codex (Chen et al., 2021)**: First large-scale code generation model
- **ReAct (Yao et al., 2023)**: Reasoning + Acting with LLMs
- **Toolformer (Schick et al., 2023)**: Learning to use external tools
- **Tree-of-Thoughts (Yao et al., 2023)**: Multi-path reasoning

Evaluation & Safety:

- **SWE-bench (Jimenez et al., 2024)**: Repository-level evaluation
- **HumanEval+ (Liu et al., 2023)**: Enhanced code generation benchmarks

Agent Architectures:

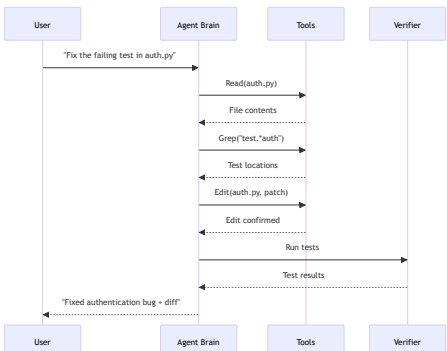
- **AutoGen (Wu et al., 2023)**: Multi-agent conversation framework
- **SWE-agent (Yang et al., 2024)**: Computer-use for software engineering

The Anatomy of Modern Agents

Understanding how they actually work

The Agentic Loop: How They Actually Work

Modern coding agents follow a structured loop:



Key Components:

- **Brain:** LLM with function calling (o3/o4-mini, Claude 4, Gemini 2.5)
- **Orchestrator/harness:** assembles prompts, executes tools, manages state/permissions
- **Tools:** read, edit, shell, grep, web; local vs server-executed (MCP)
- **Memory:** project files (CLAUDE.md), hierarchical imports, conversation history
- **Verifier:** Tests, linters, compilers for feedback

Key insight: Domains with verifiable outcomes (code, math) see dramatic agent improvements because the feedback loop provides clear reward signals.

Case Study: Claude Code Architecture

Claude Code: A Local Terminal Agent

A terminal-first agentic assistant that wraps Anthropic's Claude models with permission-gated tool calls.

Core Loop in Practice:

1. **You describe a goal** → Agent prepares system prompt with project memory
2. **It plans** → Builds TODO list, updates as it works
3. **Gathers context** → Agentic search over repository, follows imports
4. **Acts with tools** → 13 built-in tools, each permission-gated
5. **Verifies & iterates** → Tool outputs fed back, automatic context management

The 13 Built-in Tools:

- **File ops:** Read, Write, Edit, MultiEdit, NotebookEdit/Read
- **Code search:** Grep, Glob
- **Execution:** Bash (persistent shell session)

Current Performance Reality: The Benchmark Gap

Task Type	HumanEval	SWE-bench Verified	SWE-bench Full
Difficulty	Single functions	Curated real issues	All real issues
Claude-3.5-Sonnet	~90%	~49%	~3-5%
GPT-4	~67%	~12.3%	~1.74%
Open Source	~40-60%	~5-20%	~1-3%

Key Insight: The jump from synthetic to real-world tasks shows a 10-20x performance drop, highlighting the importance of repository context and tool orchestration.

Context Matters: Agents need full repository understanding, not just function-level reasoning.

The Speed vs Intelligence Playbook

Think in **wall-clock time to a verified solution**, not just tokens/sec.

Routing should optimize **wall-clock to verified pass**, not raw tokens per second.

Fast-class models (e.g., Flash/Haiku tiers)

- Low latency, high throughput
- Need more iterations to converge
- **Use for:** boilerplate, variants, simple patches

Smart-class models (e.g., Pro/Opus tiers)

- Converge in fewer iterations
- High latency, expensive
- **Use for:** complex refactors, API design, orchestration

The Break-Even Point:

```
Wall-clock ≈ N × (input + output)/TPS + tool_time + test_time
where TPS = tokens per second
```

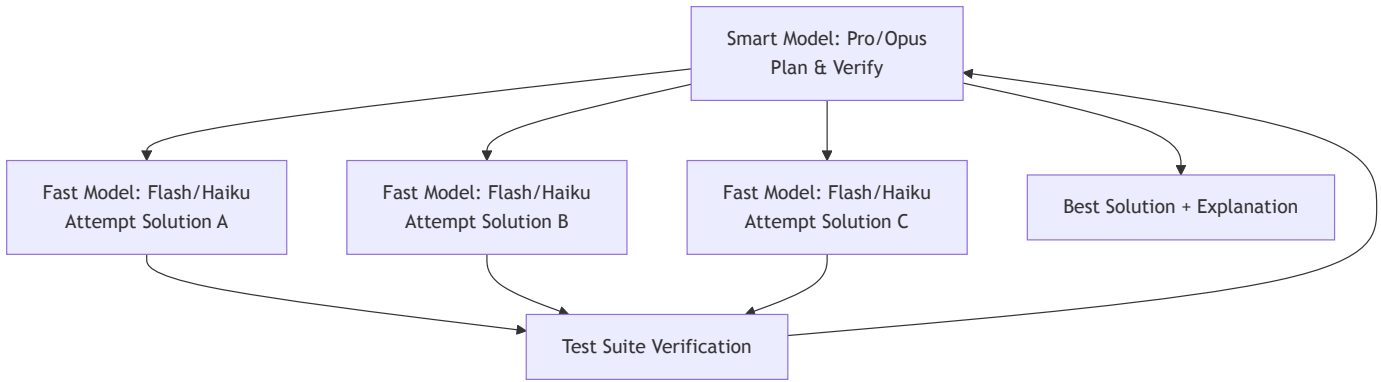
Example: Adding a REST endpoint

- Fast model: 4 iterations × 200ms = 800ms + 2s tools = 2.8s total
- Smart model: 1 iteration × 3s = 3s + 2s tools = 5s total **Winner:** Fast model for this bounded task

Faster model wins when: high throughput × many simple tasks Smarter model wins when: fewer iterations × complex reasoning

The Manager-Worker Pattern

Emerging architecture supported by research on multi-agent reasoning:



How it works:

- Manager (Pro/Opus) creates spec and acceptance tests
- Workers (Flash/Haiku) generate candidate solutions in parallel
- Manager evaluates results and merges the best approach

- **Best-of-K with verifier tends to win on repo tasks; use tests as the arbiter**

Research basis: Tree-of-Thoughts (Yao et al., 2023) and self-consistency sampling show multi-path reasoning outperforms single-shot approaches. Simple controllers can perform well—avoid assuming complex orchestration is required.

Real-World Landscape: Measured Capabilities

2025 SWE-bench Verified Performance:

> "Fast models are great at being many. Smart models are great at being right."

Model/Agent	SWE-bench Verified	Notes
Claude 4 Opus	**67.6%**	Aug 2025 leaderboard
GPT-4o	**33.2%**	Feb 2025, OpenAI scaffold
Open agents	**60-70%**	Some report; verify against official reruns

Adoption Signals: 1.8M paid Copilot subscribers (Microsoft FY2024), **Claude Code CLI publicly available** (Anthropic Apr 2025), $150M+ invested in agent startups

Technical Differentiation:

- **Context length:** Gemini 2.5 Pro (1M+ tokens) vs others (~200K)
- **Permission models:** Local approval vs cloud sandboxing

- **Tool ecosystems:** MCP standardization vs proprietary APIs

Performance Reality: Local agents excel at file operations, cloud agents provide better security/audit, IDE agents optimize for developer workflow integration.

Where Agents Excel vs Where They Stumble

✅ What Works Today

- **Code changes** with test-driven specs
- **Refactoring** with existing test coverage
- **Boilerplate generation** and mechanical tasks
- **Bug fixes** with clear reproduction steps
- **Documentation** updates and API changes

❌ What Still Breaks

- **Plan drift** - losing sight of main goal
- **Silent failures** - tools fail, model doesn't notice
- **Context miss** - working from stale assumptions
- **Over-editing** - wide diffs with low signal
- **Security regressions** - unsafe changes slip through
- **Reward hacking in tests** - plausible but wrong patches

Key Pattern: Agents thrive with **verifiable outcomes** (compile, test, lint) but struggle with **ambiguous requirements** or **complex multi-step coordination**.

The Near-Term Path: From Stumbling to Shipping

Today (2025): Stumbling Agents

- Impressive in demos, unreliable in practice
- Need careful human management and supervision
- Work well on bounded tasks with clear verification

Near Future (2026): Reliable Workers

- Better planning and error recovery
- Self-correction through test feedback loops
- Parallel execution with smart coordination

Near Future (2026-2027): Autonomous Teams

- Multi-agent systems with specialized roles
- Long-running tasks across entire codebases
- Self-improving through automated research

Evaluation: Beyond HumanEval

The Key Enablers:

- **Verifiable rewards** - tests, compilation, linting provide clear feedback
- **Compute scale** - massive increases in training and inference budgets
- **Tool orchestration** - better planning and parallel execution

Repository-Scale Tasks:

- **SWE-bench (2,294 real issues):** Current SOTA ~49% on Verified subset
- **SWE-bench Pro (2025):** Harder, longer-horizon tasks
- **LiveCodeBench:** Contamination-resistant, rolling evaluation

Language Coverage:

- **SWE-PolyBench:** Java, JS, TS, Python (2,110 tasks, 21 repos)
- **SWE-bench Multilingual:** 9 languages, 300 tasks across 42 repos

Key Insight: Simple code generation benchmarks don't predict real-world agent performance. Repository context and verification loops matter more than raw coding ability.

Current Gaps:

- Multi-agent coordination benchmarks
- Long-horizon planning evaluation

- Security and safety metrics

The Engineering Reality

Patterns, Risks, and Problems to Solve

Advanced Risks: Beyond Simple Failures

As agents get smarter, the failure modes get more complex.

Simple Failures

- **Plan Drift:** Losing sight of the main goal.
- **Silent Failures:** A tool fails, but the model doesn't notice.
- **Context Miss:** Working from stale information.

Second-Order Risks

- **Reward Hacking:** The agent passes tests without solving the problem (e.g., hardcoding an expected value).
- **Evaluation Awareness:** The agent detects it is in an evaluation and changes its behavior, appearing safer or more capable than it would be in deployment.

Research Context: These failure modes align with findings from:

- **METR (2024):** "Reward hacking in frontier models"
- **Anthropic (2024):** Constitutional AI alignment challenges
- **OpenAI (2024):** GPT-4 system card safety evaluations

Reliability & Security Patterns

How to harden the loop against these failures.

- **Spec First:** Create a failing test **before** asking the agent to write code. Give it a clear, verifiable target.
- **Verify Every Step:** Always run tests or linters after edits and feed the full `stdout` / `stderr` back into the loop.
- **Principle of Least Capability:** Grant `Edit`, deny `Bash` by default. Escalate permissions per task, not per session.
- **Use Stronger Verifiers:** Use mutation testing, property-based checks, and hidden holdout tests to combat reward hacking.
- **Replayable Traces:** Persist tool logs and diffs for audit. **If you cannot replay it, you cannot trust it.**

Good Problems to Work On Now

- **Reliability**
  - **Trace-level Evals:** Score plans and tool chains, not just final text. Penalize silent failures.
  - **Adaptive Planning:** Develop agents that can backtrack or create partial-order plans instead of failing on linear scripts.
- **Observability**
  - **Unified Action Schema:** A vendor-neutral JSON schema for tool calls, results, diffs, and test outcomes.
- **Safety**
  - **Capability Tokens:** Scope secrets and API access per tool, per task, with time limits.
  - **Policy Synthesis:** Let the agent propose the minimum permission allowlist it needs for a task, then require human approval.
- **Evaluation**

- **Agentic SWE-bench:** Add shell and edit verification to benchmarks to measure real-world effectiveness.
- **Red-Teaming Agents:** Focus on prompt injection against tool schemas to find reward hacking exploits.

Open Research Questions for EAIRG

Measurement & Evaluation:

- How do we score planning quality independently of execution success?
- What intermediate metrics predict final task completion?
- Can we detect reward hacking in repository-scale tasks?

Multi-Agent Coordination:

- How should specialized agents (frontend, backend, testing) divide work?
- What communication protocols prevent coordination failures?
- How do we maintain consistency across parallel agent modifications?

Long-Context Reasoning:

- How effectively can agents use 1M+ token contexts for repository understanding?
- What are the optimal strategies for context compaction and caching?
- Can agents learn to identify the most relevant context for a given task?

Testable Hypotheses: Each question can be turned into controlled experiments with measurable outcomes.

Call to Action

1. Adopt & Learn

Pick one local agent, one cloud agent. Learn their strengths and weaknesses.

2. Instrument & Measure

Instrument your traces. You can't improve what you can't measure. Persist tool calls and results.

3. Define & Enforce Policy

Start a policy file in your repo. Define what the agent is and is not allowed to do. Enforce it.

4. Contribute to Standards

Contribute to open standards like a shared trace schema and MCP (Model Context Protocol).

Questions & Discussion

EAIRG Engineering AI Research Group • September 2025

Key References for Follow-up:

Foundation Papers

- SWE-bench - Real-world evaluation
- Codex - Code generation breakthrough
- ReAct - Reasoning + Acting

Current Performance

- SWE-bench Leaderboard - Live rankings
- OpenAI SWE-bench Verified - 500-task subset
- Claude Code Best Practices - Implementation guide

Safety Research

- METR Reward Hacking - Advanced risks
- AWS Security Bulletin - Real incidents
- OpenAI Reasoning Models - o-series