From Stumbling to Shipping

Coding Agents in 2025 and the Near-Term Future

EAIRG Engineering AI Research Group

The Anatomy of Modern Agents

Understanding how they actually work

The Agentic Loop: How They Actually Work

Modern coding agents follow a structured loop:

The Speed vs Intelligence Playbook

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

Fast Models (Gemini Flash, Claude Haiku)

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

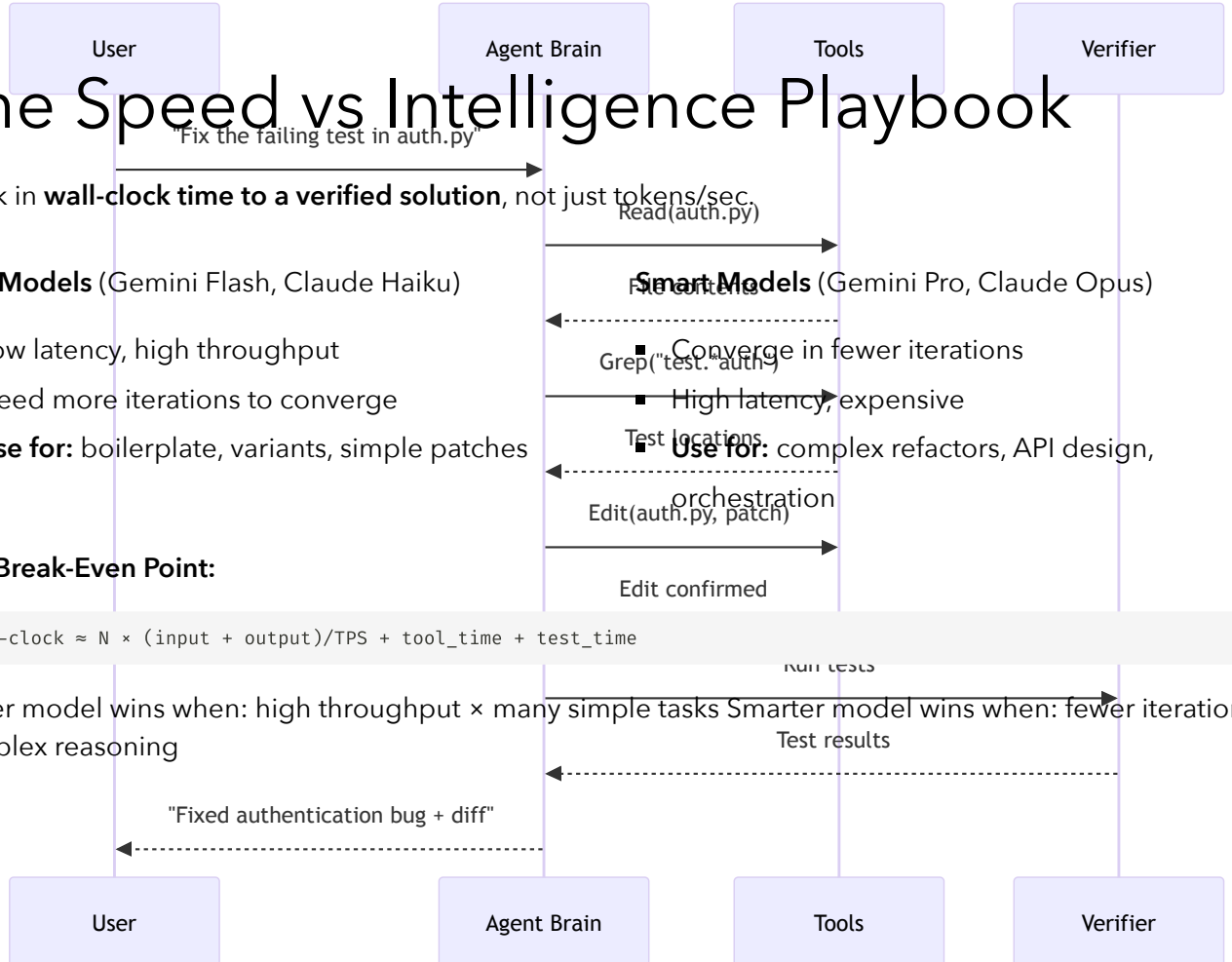
Smart Models (Gemini Pro, Claude Opus)

- 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
```

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

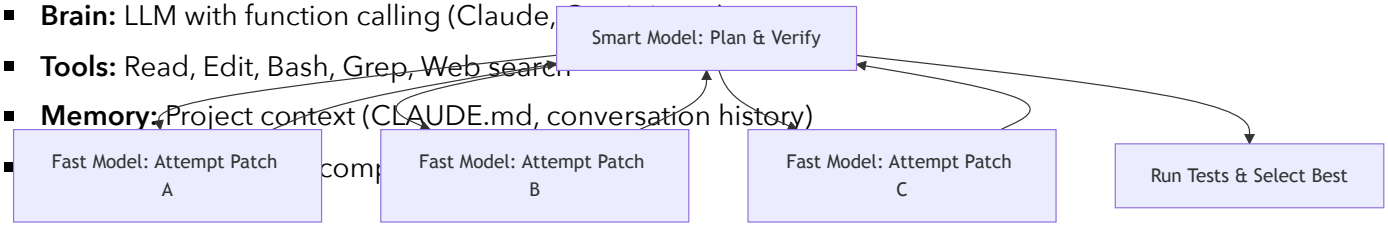


The Manager-Worker Pattern

The emerging architecture for parallel agents:

Key Components:

- **Brain:** LLM with function calling (Claude,
- **Tools:** Read, Edit, Bash, Grep, Web search
- **Memory:** Project context (CLAUDE.md, conversation history)
- comp



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
- Tests provide the final verification step

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

Real-World Landscape: Who’s Building What

Major Players:

- **Gemini CLI:** Local terminal, open source, MCP support, 1M+ context
- **Claude Code:** Local terminal, strong permissions, CLAUDE.md memory
- **GitHub Copilot:** IDE + Cloud, deep VS Code integration, PR workflows
- **Cursor:** AI-first editor, agent mode, rules for context
- **Replit Agent:** Cloud workspace, full app building, autonomous deployment

The Split:

- **Local agents:** Permission-gated, your machine, your rules
- **Cloud agents:** Sandboxed, auditable, but less filesystem access

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

❌ 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

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

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.

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.**

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.

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.

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).

Thank You

Q&A

References:

- OpenAI: Learning to Reason with LLMs
- SWE-bench Verified Paper
- METR: Recent Frontier Models Are Reward Hacking
- Amazon Q Developer Security Bulletin (CVE-2024-6991)