SEMLA 2026

Lessons from Building an Enterprise Coding Agent

Open Questions and Challenges

Gustavo Pinto

br.linkedin.com/in/ghlp

mail@gustavopinto.org

gustavopinto.org

INTRODUCTION

The gap between prototype and production

LLM-based coding agents can accelerate routine development tasks. But building one that **performs well on benchmarks is fundamentally different from deploying one that developers actually use**.

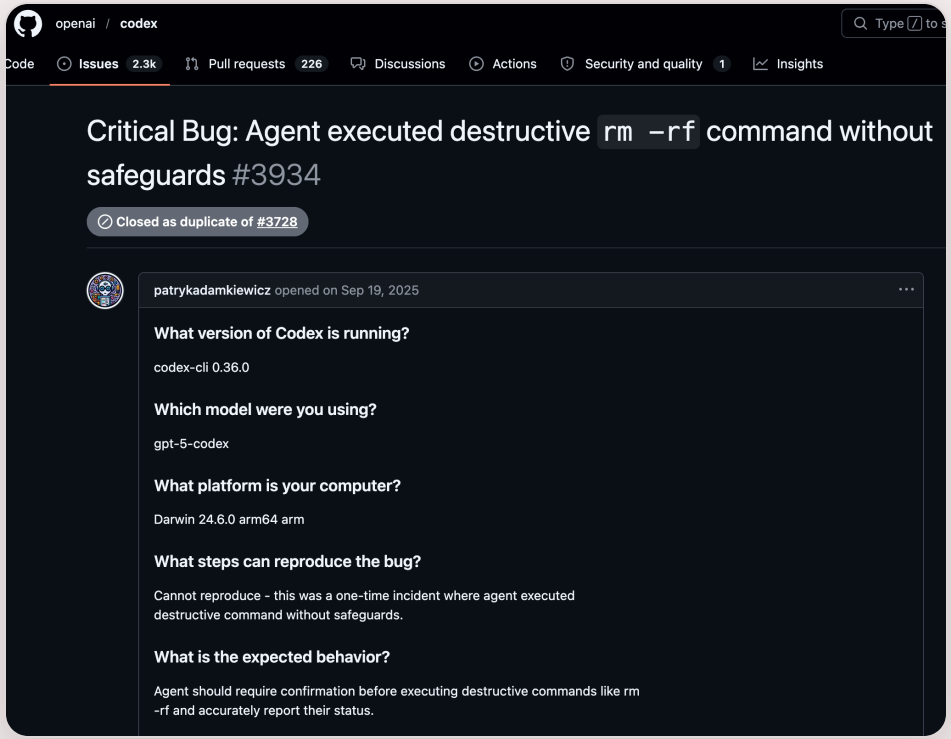
Research agents ≠ production agents.

Production demands engineering.

INTRODUCTION

When engineering challenges go unaddressed

- Agents produce **unreliable edits** that developers must manually verify and correct
- **Safety incidents** erode trust and stall adoption—e.g., an agent running `rm -rf` or unauthorized `git push --force`
- Teams invest months building prototypes that **never transition to production**



Real incident: ***openai/codex#3934***

SECTION 02

Origins of CodeGen

ORIGINS

From conversational assistant to autonomous agent

StackSpot AI

IDE conversational assistant—developers ask questions, but the tool cannot act on the environment

→

Agentic Gap

Adding agent capabilities to the enterprise codebase would require changes the platform roadmap could not absorb

→

Independent POC

Two team members build a CLI-based coding agent outside the main codebase

→

CodeGen

Production-ready agentic system used daily by real developers at Zup

ORIGINS

From conversational assistant to autonomous agent

StackSpot AI

IDE conversational assistant—developers ask questions, but the tool cannot act on the environment

→

Agentic Gap

Adding agent capabilities to the enterprise codebase would require changes the platform roadmap could not absorb

→

Independent POC

Two team members build a CLI-based coding agent outside the main codebase

→

CodeGen

Production-ready agentic system used daily by real developers at Zup

ORIGINS

From conversational assistant to autonomous agent

StackSpot AI

IDE conversational assistant—developers ask questions, but the tool cannot act on the environment

→

Agentic Gap

Adding agent capabilities to the enterprise codebase would require changes the platform roadmap could not absorb

→

Independent POC

Two team members build a CLI-based coding agent outside the main codebase

→

CodeGen

Production-ready agentic system used daily by real developers at Zup

ORIGINS

From conversational assistant to autonomous agent

StackSpot AI

IDE conversational assistant—developers ask questions, but the tool cannot act on the environment

→

Agentic Gap

Adding agent capabilities to the enterprise codebase would require changes the platform roadmap could not absorb

→

Independent POC

Two team members build a CLI-based coding agent outside the main codebase

→

CodeGen

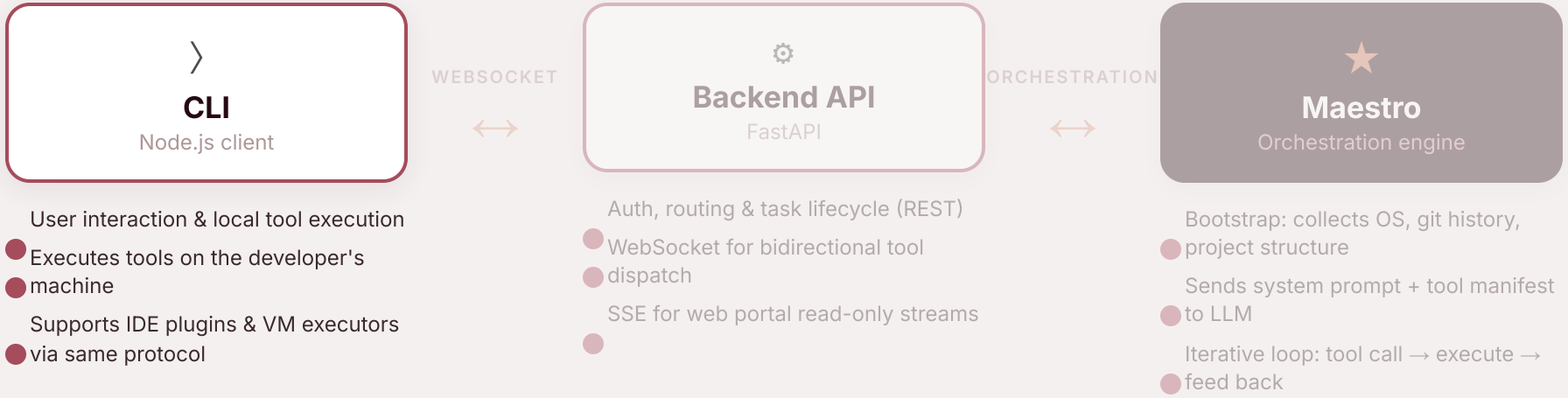
Production-ready agentic system used daily by real developers at Zup

SECTION 03

CodeGen Internals

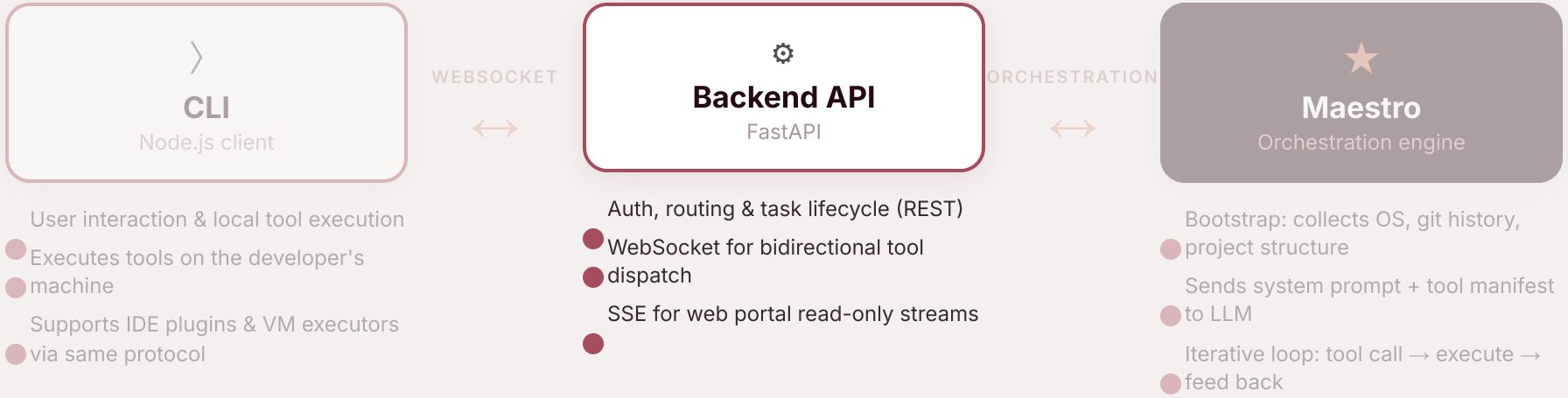
ARCHITECTURE

Three-tier architecture



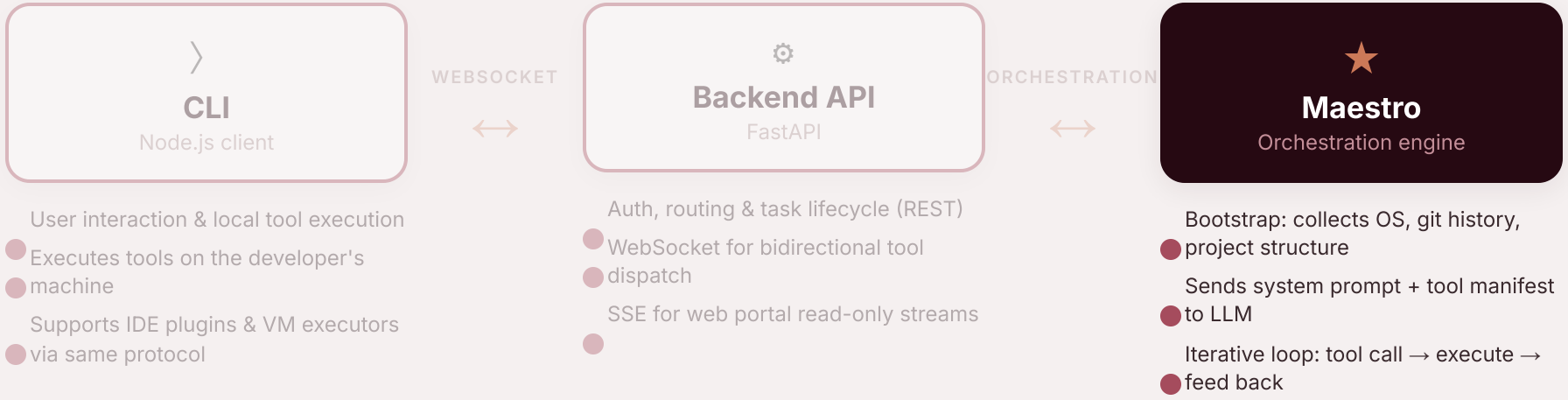
ARCHITECTURE

Three-tier architecture



ARCHITECTURE

Three-tier architecture



ReAct loop: **Reason** → **Act** (tool call) → **Observe** (result) → **Iterate** until final response

ARCHITECTURE

Tool manifest

Tool design quality is a **first-class citizen** of agent effectiveness.

`read`

Retrieves file contents. Enforces a **read-before-edit policy** to prevent stale-context errors and hallucinated edits.

`edit`

Targeted string replacement rather than full-file rewriting—mitigates LLM truncation failures on large files.

`shell`

Executes terminal commands subject to **multiple guardrail layers**: command blocking, human approval mode, full audit logging.

EXAMPLE: TOOL SPECIFICATION

```
"WriteFile": {
  "name": "WriteFile",
  "description":
    "Writes content to a specified
     file, creating it if it doesn't
     exist and overwriting if it does.",
  "parameters": {
    "type": "object",
    "properties": {
      "file_path": {
        "type": "string",
        "description":
          "The path to the file."
      },
      "content": {
        "type": "string",
        "description":
          "The content to write."
      }
    },
    "required":
      ["file_path", "content"]
  }
}
```

SECTION 04

Design Decisions

DESIGN DECISIONS

14 decisions across three dimensions

Architecture & Framework

1. LangChain's chain model was inadequate for the agentic loop
2. Manual implementation gave more control than early framework adoption
3. Transitioning toward modern orchestration as frameworks converged
4. FastAPI chosen for native async support
5. Session-scoped memory already delivers substantial UX value
6. Reasoning delegated to the LLM, not hand-coded in the orchestrator
7. Strong model capabilities don't eliminate need for orchestration

Tool Design & Safety

1. Tool design quality proved more impactful than prompt-only tuning
2. Edit tool uses targeted string replacement, not full-file rewrites
3. Read-before-edit policy prevents hallucinated edits
4. Shell tool requires multiple guardrail layers
5. Policy consistency across tools is required for effective safety
6. Strict code quality enforcement as additional safety net

Human Oversight & Adoption

1. Approval mode serves as trust-calibration during onboarding
2. Separating planning from execution addresses single-pass limitations
3. Progressive deployment mirrors individual trust-building patterns
4. Most decisions involved balancing competing concerns

Read the full paper atarxiv.org/pdf/2604.09805

DESIGN DECISIONS

14 decisions across three dimensions

Architecture & Framework

1. LangChain's chain model was inadequate for the agentic loop
2. Manual implementation gave more control than early framework adoption
3. Transitioning toward modern orchestration as frameworks converged
4. FastAPI chosen for native async support
5. Session-scoped memory already delivers substantial UX value
6. Reasoning delegated to the LLM, not hand-coded in the orchestrator
7. Strong model capabilities don't eliminate need for orchestration

Tool Design & Safety

1. Tool design quality proved more impactful than prompt-only tuning
2. Edit tool uses targeted string replacement, not full-file rewrites
3. Read-before-edit policy prevents hallucinated edits
4. Shell tool requires multiple guardrail layers
5. Policy consistency across tools is required for effective safety
6. Strict code quality enforcement as additional safety net

Human Oversight & Adoption

1. Approval mode serves as trust-calibration during onboarding
2. Separating planning from execution addresses single-pass limitations
3. Progressive deployment mirrors individual trust-building patterns
4. Most decisions involved balancing competing concerns

Read the full paper atarxiv.org/pdf/2604.09805

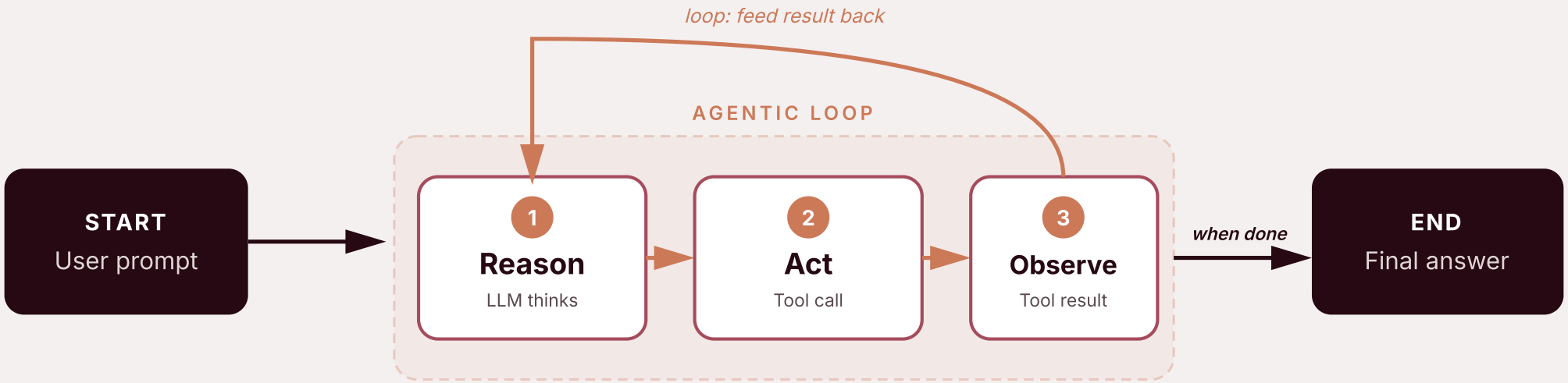
ARCHITECTURE & FRAMEWORK

LangChain's chain model was inadequate for the agentic loop

At project inception, the team experimented with **LangChain** as an orchestration framework. Its early abstractions were designed around **linear chains**—a unidirectional pipeline where each step feeds into the next.

LangChain's chain model was inadequate for the agentic loop

The agentic coding assistant required a **cyclical interaction pattern**: the model requests a tool, the client executes it, and the result is fed back into the model repeatedly until the task is complete.



This fundamental mismatch forced the team to work around the framework rather than with it.

ARCHITECTURE & FRAMEWORK

Manual implementation gave more control than early framework adoption

We adopted but later abandoned LangChain.

The team implemented the agentic loop directly—the Maestro component. This gave explicit control over stop criteria, tool dispatch, WebSocket communication, and error propagation.

For novel or poorly understood interaction patterns, **manual implementation accelerates learning and provides clearer ownership of execution semantics**—advantages that outweigh framework convenience during early project phases.

ARCHITECTURE & FRAMEWORK

Transitioning toward modern orchestration as frameworks converged

As the agentic pattern became widespread, **frameworks evolved to support it natively**. LangChain introduced LangGraph, offering first-class support for cyclical tool-calling loops.

When evaluated, the team found that the design closely resembled what had already been built by hand. This convergence validated the original choices and made transition cost low.

Building manually first and adopting frameworks later—once they mature to match actual requirements—avoids both premature abstraction and long-term maintenance burden.

ARCHITECTURE & FRAMEWORK

Creating agentic apps in 2026

adopting → ~~abandoning~~

→ adopting again

DESIGN DECISIONS

14 decisions across three dimensions

Architecture & Framework

1. LangChain's chain model was inadequate for the agentic loop
2. Manual implementation gave more control than early framework adoption
3. Transitioning toward modern orchestration as frameworks converged
4. FastAPI chosen for native async support
5. Session-scoped memory already delivers substantial UX value
6. Reasoning delegated to the LLM, not hand-coded in the orchestrator
7. Strong model capabilities don't eliminate need for orchestration

Tool Design & Safety

1. Tool design quality proved more impactful than prompt-only tuning
2. Edit tool uses targeted string replacement, not full-file rewrites
3. Read-before-edit policy prevents hallucinated edits
4. Shell tool requires multiple guardrail layers
5. Policy consistency across tools is required for effective safety
6. Strict code quality enforcement as additional safety net

Human Oversight & Adoption

1. Approval mode serves as trust-calibration during onboarding
2. Separating planning from execution addresses single-pass limitations
3. Progressive deployment mirrors individual trust-building patterns
4. Most decisions involved balancing competing concerns

Read the full paper atarxiv.org/pdf/2604.09805

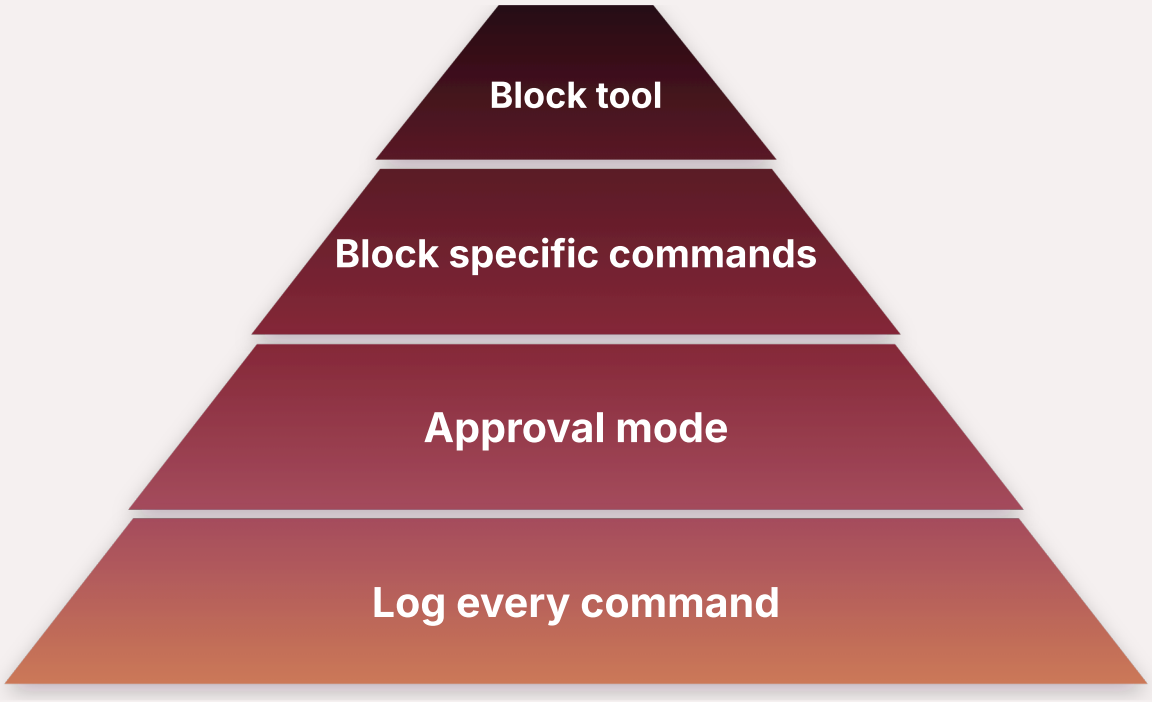
TOOL DESIGN & SAFETY

Tool design > prompt design

- **Semantic quality of descriptions** determines how well the LLM understands when and why to use a tool
- **Parameter schemas** determine whether the model can invoke the tool correctly
- **Read-before-edit policy** prevents hallucinated edits on files the model hasn't recently inspected
- **Error signaling** determines whether the model can recover from failed invocations

Tool specification is a **first-class engineering concern**—not an afterthought.

Shell safety & cross-tool policy · guardrails for `shell`



⚠ PROBLEM **Blocking direct file deletion is useless** if `shell` remains unrestricted—a shell command achieves the same effect.

TOOL DESIGN & SAFETY

The 3 layers of tool safety

specifying → restricting → auditing

DESIGN DECISIONS

14 decisions across three dimensions

Architecture & Framework

1. LangChain's chain model was inadequate for the agentic loop
2. Manual implementation gave more control than early framework adoption
3. Transitioning toward modern orchestration as frameworks converged
4. FastAPI chosen for native async support
5. Session-scoped memory already delivers substantial UX value
6. Reasoning delegated to the LLM, not hand-coded in the orchestrator
7. Strong model capabilities don't eliminate need for orchestration

Tool Design & Safety

1. Tool design quality proved more impactful than prompt-only tuning
2. Edit tool uses targeted string replacement, not full-file rewrites
3. Read-before-edit policy prevents hallucinated edits
4. Shell tool requires multiple guardrail layers
5. Policy consistency across tools is required for effective safety
6. Strict code quality enforcement as additional safety net

Human Oversight & Adoption

1. Approval mode serves as trust-calibration during onboarding
2. Separating planning from execution addresses single-pass limitations
3. Progressive deployment mirrors individual trust-building patterns
4. Most decisions involved balancing competing concerns

Read the full paper atarxiv.org/pdf/2604.09805

HUMAN OVERSIGHT

Trust is earned, not mandated

APPROVAL MODE

Every file edit and shell command requires **explicit human confirmation**.

→

AUTONOMOUS MODE `--yolo`

Agent operates with **minimal interruption**.

Developers **begin in approval mode**, then organically migrate to autonomous mode as confidence grows.

HUMAN OVERSIGHT

Balancing competing concerns

Throughout CodeGen's development, the team repeatedly faced decisions where improving one dimension came at the cost of another:

Safety via approval modes	vs.	added latency & friction
Session memory for UX	vs.	infrastructure complexity
Reasoning delegated to LLM	vs.	less deterministic control

SECTION 06

Open Questions

OPEN QUESTIONS

For researchers and tool builders

1. Is there a **methodology** for designing tools that LLMs invoke correctly?
2. Where should **reasoning** live—in the model or in the orchestrator?
3. How do we enforce **safety** when tools have overlapping capabilities?
4. How do agents earn **autonomy** beyond a binary on/off switch?
5. What should agents **remember**—and what should they forget?
6. How do we **QA code** that agents wrote?

Thanks, G.

ZUP INNOVATION

Lessons from Building an Enterprise Coding Agent

Open Questions and Challenges

Gustavo Pinto

mail@gustavopinto.org

br.linkedin.com/in/ghlp