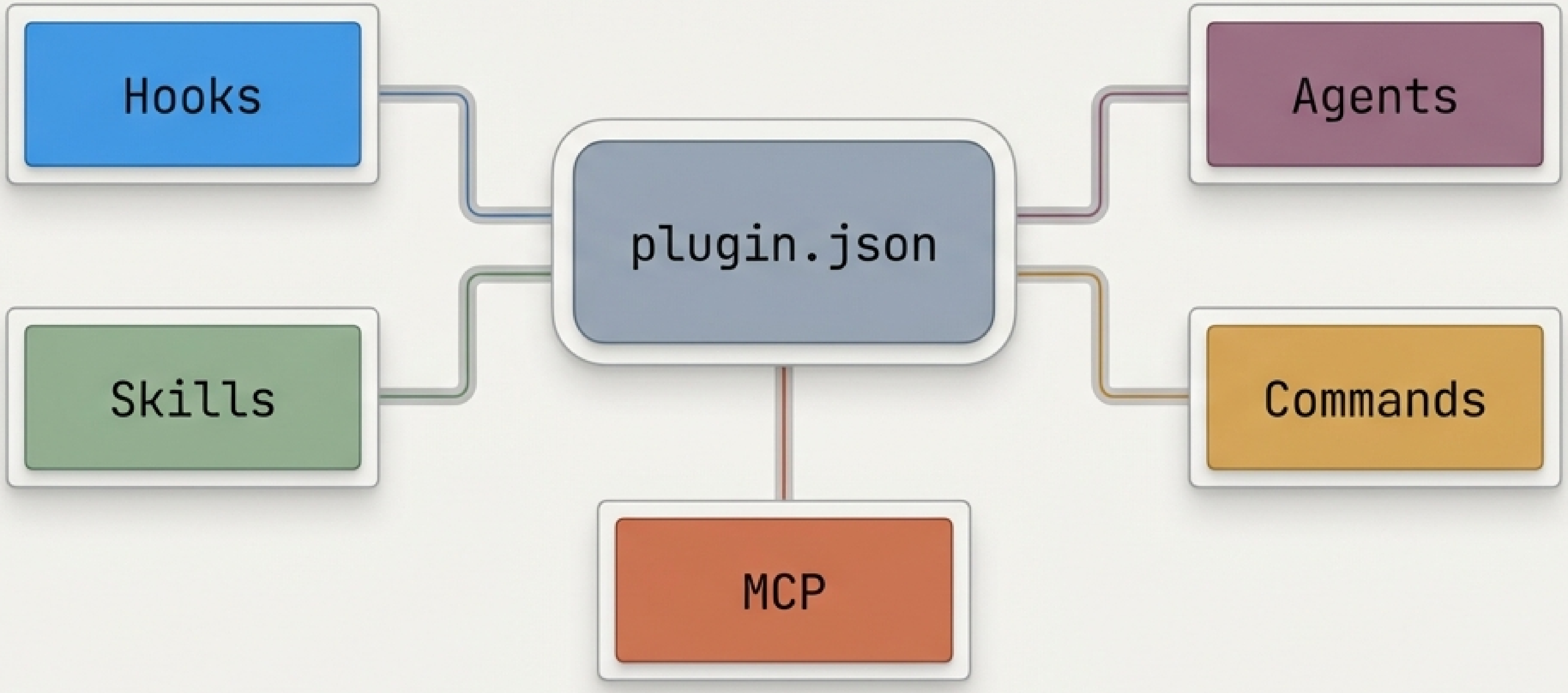
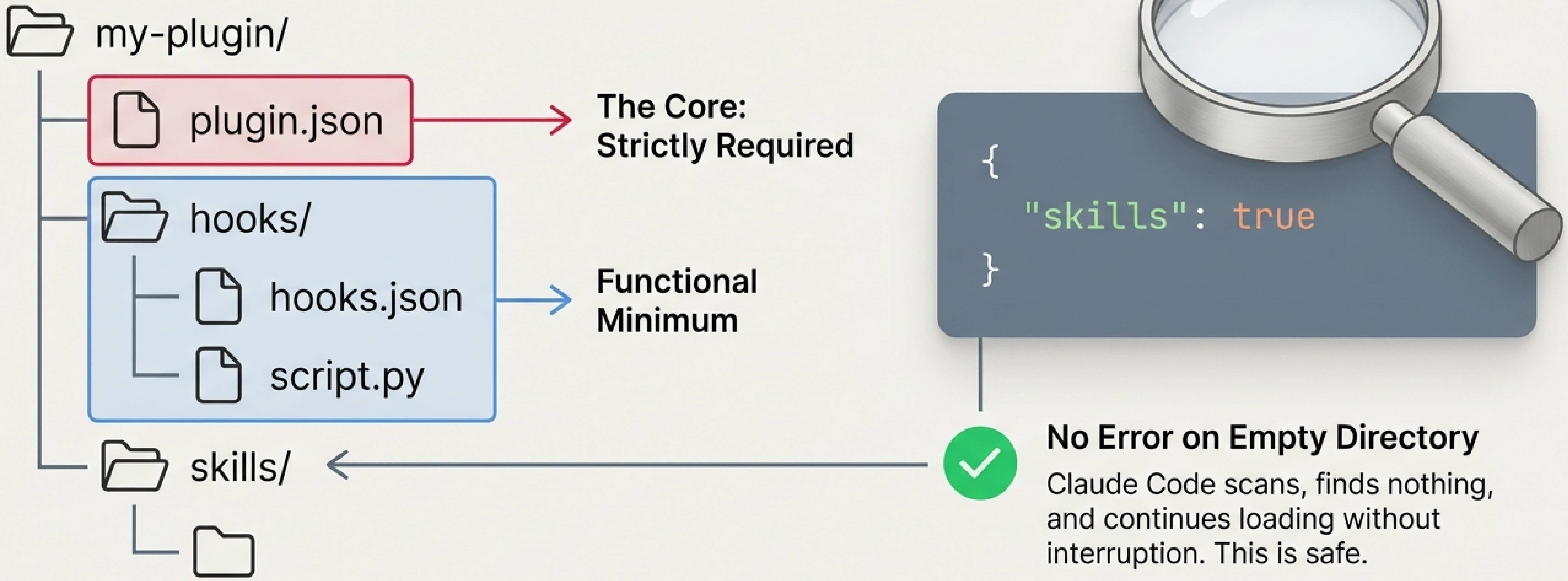
CC-201 Assessment Blueprint

Mastering the core extensions, components, and practical implementation for Claude Code.



The Anatomy of a Valid Plugin

A manifest is the strictly required core. A functional plugin needs at least one executable component.



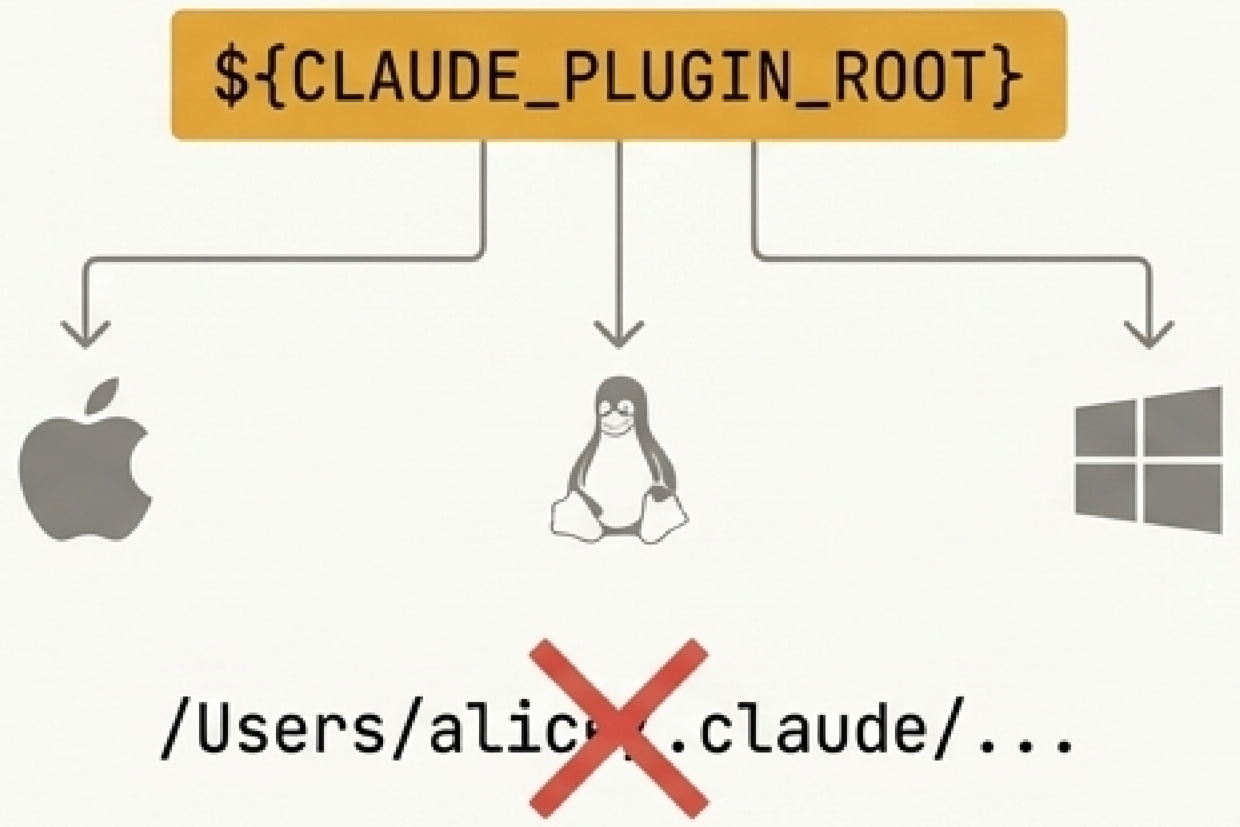
```
{
  "skills": true
}
```

No Error on Empty Directory

Claude Code scans, finds nothing, and continues loading without interruption. This is safe.

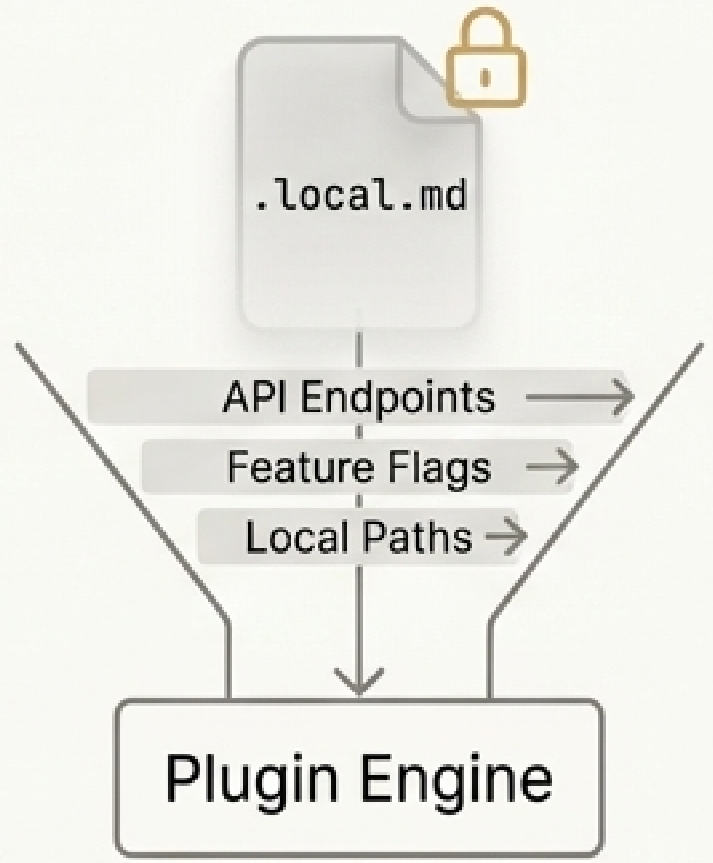
Environment Sandbox: Portability and Configuration

Portability via Dynamic Resolution



Hardcoding paths breaks portability across diverse filesystems. Always resolve absolute paths dynamically at runtime.

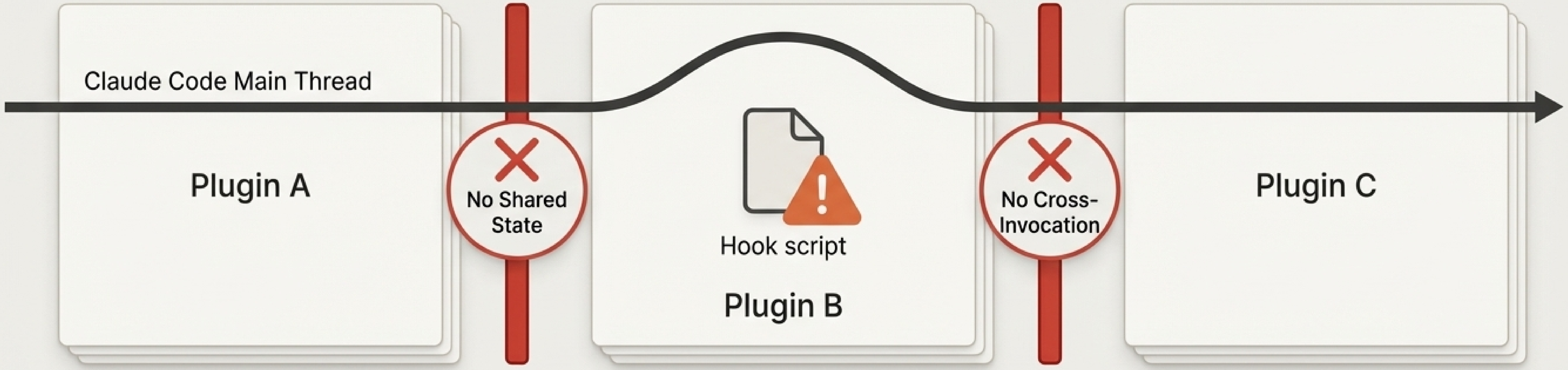
Configuration via `.local.md`



Environment-specific settings belong in a gitignored .local.md file at the plugin root, injecting runtime context without exposing repository secrets.

The Isolation Principle & Fail-Open Safety Net

Plugins operate in strict isolation. They cannot read shared variables, invoke cross-**plugin components**, or bring each other down.



The Fail-Open Default

If a hook crashes or times out, Claude Code proceeds as if the hook returned an "allow" status. Plugins must enhance the system, not become single points of failure. (Exception: Security-critical hooks require explicit mitigation).

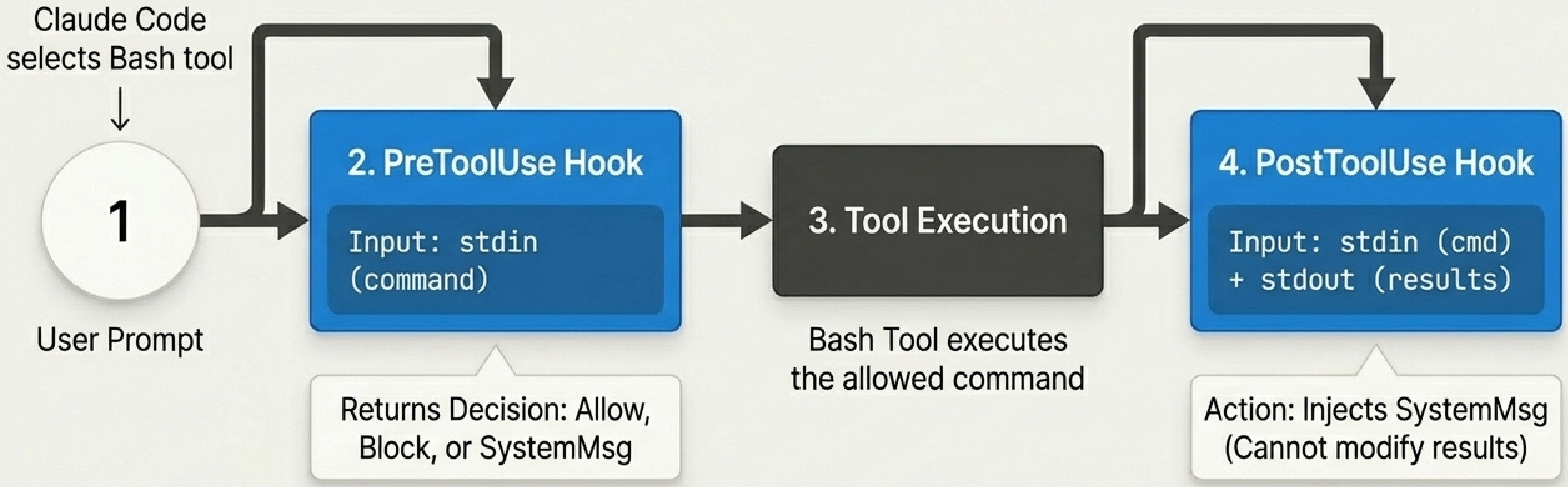
The Five Pillars of Extension

Component selection is the most consequential architectural decision in plugin development.

Component	Trigger Mechanism	Primary Function
Hooks	Automatic (Lifecycle Events)	Silent validation, logging, and guardrails bracketing tool execution.
Skills	Loaded on Demand	Injects specialized knowledge, reference files, and guided step-by-step workflows.
Agents	Dispatched via Task Tool	Executes autonomous, isolated tasks requiring dedicated context and specific reasoning models.
Commands	User Invoked (Slash Commands)	Transforms complex multi-step workflows into explicit, single-invocation user actions.
MCP Servers	External API Integration	Connects Claude Code to external databases, live services, and third-party platforms.

The Hook Execution Lifecycle

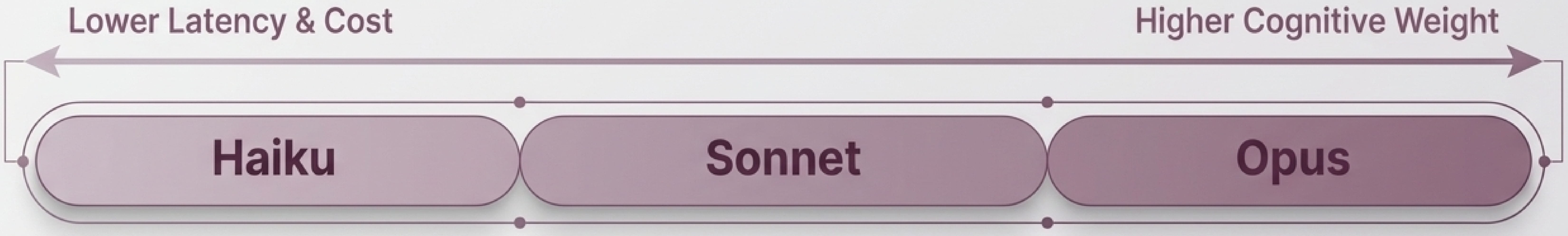
Hooks bracket tool execution. Crucially, a **PostToolUse** hook can observe output and inject context, but it **cannot modify the actual tool result** returned to the user.


Claude Code selects Bash tool
1
User Prompt
2. PreToolUse Hook
Input: stdin (command)
Returns Decision: Allow, Block, or SystemMsg
3. Tool Execution
Bash Tool executes the allowed command
4. PostToolUse Hook
Input: stdin (cmd) + stdout (results)
Action: Injects SystemMsg (Cannot modify results)

NotebookLM

Agent Cognitive Matching

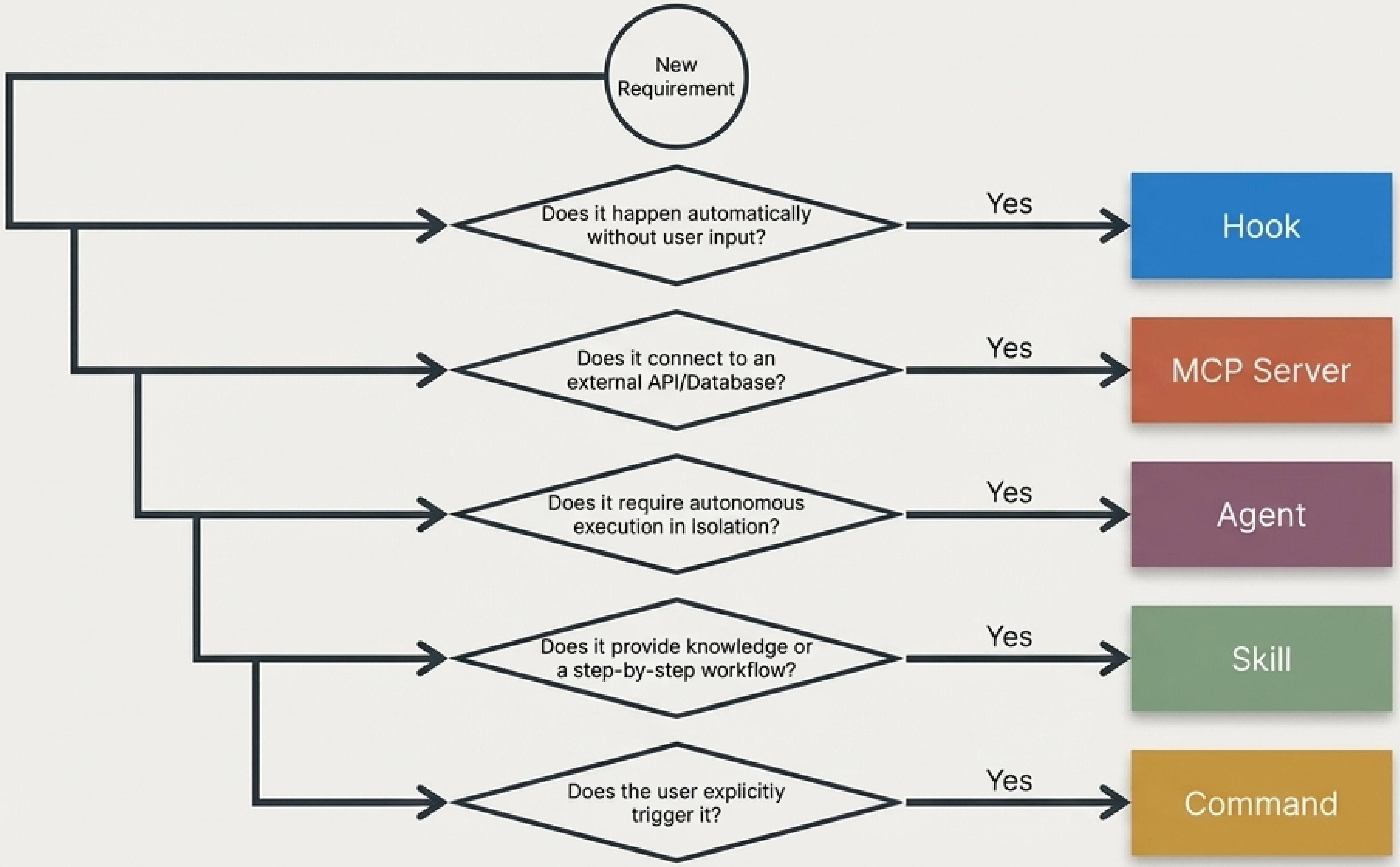
Agent behavior is governed by YAML frontmatter. Match the AI model to the task's cognitive requirements to optimize speed, cost, and quality.



Haiku	Sonnet	Opus
- Fast and cost-efficient	- Balanced reasoning & speed	- Deep, complex reasoning
- Rapid classification	- Standard logic workflows	- Architectural planning
- Structured data extraction	- General coding tasks	- Navigating dense logic
- Quick formatting		

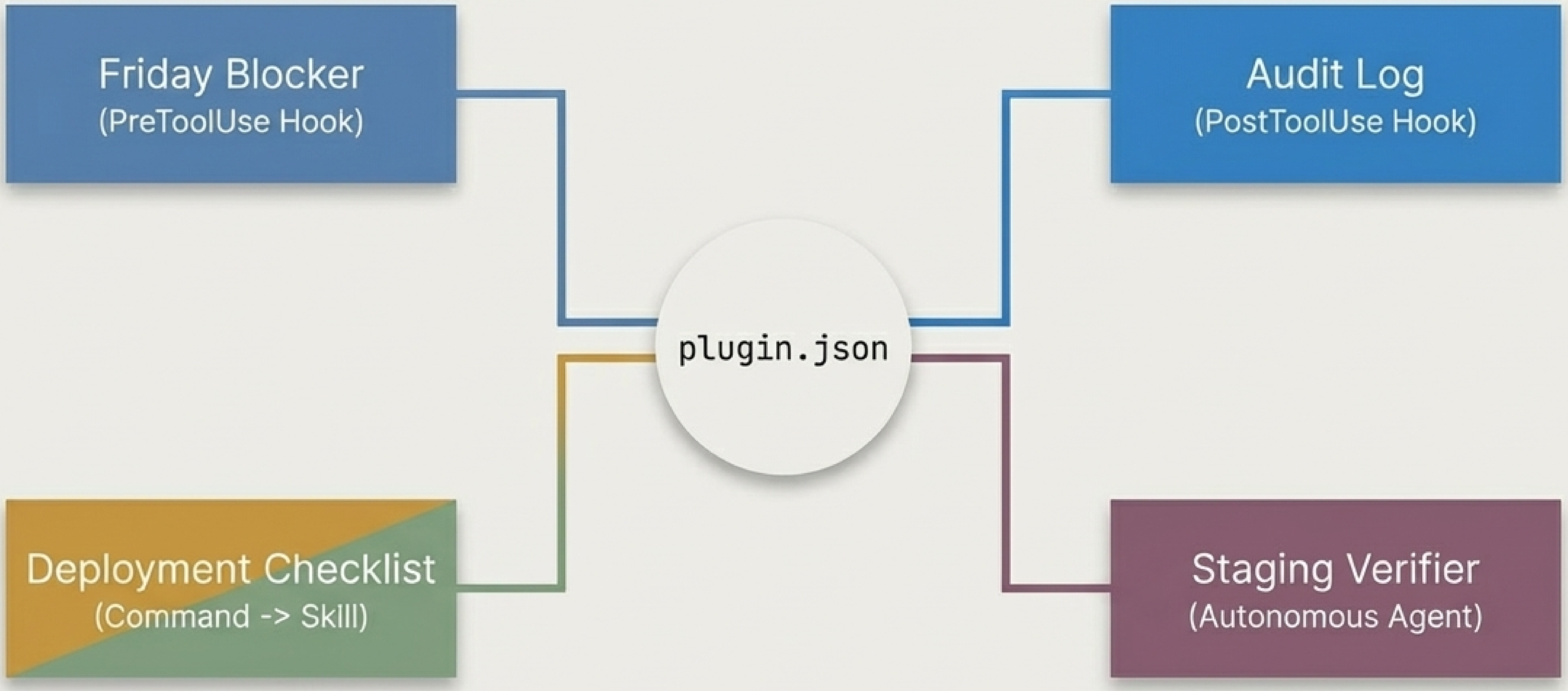
The Component Routing Blueprint

Transforming workflow problems into plugin architecture requires routing the requirement to the correct execution model.


New Requirement
Does it happen automatically without user input?
Yes
Hook
Does it connect to an external API/Database?
Yes
MCP Server
Does it require autonomous execution in Isolation?
Yes
Agent
Does it provide knowledge or a step-by-step workflow?
Yes
Skill
Does the user explicitiy trigger it?
Yes
Command

Part 2 Practical: Deploy-Guard Architecture

Scenario: Build a 50-point plugin for a deployment team that prevents weekend releases, guides the deployment process, and autonomously verifies staging logs.



Deploy-Guard: The Core Manifest

The foundation is a syntactically valid JSON manifest with all component flags enabled, matched perfectly to a convention-compliant directory structure.

plugin.json

```json
{
  "name": "deploy-guard",
  "hooks": "hooks/hooks.json",
  "skills": true,
  "agents": true,
  "commands": true
}
```

Directory Structure

```
deploy-guard/
├── plugin.json
├── hooks/
│   ├── hooks.json
│   ├── deploy-block.py
│   └── audit-log.py
├── skills/
│   └── deploy-checklist/
│       └── SKILL.md
├── agents/
│   └── staging-verifier.md
└── commands/
    └── deploy.md
```

Deploy-Guard Automation: Implementation

Fulfill automated constraints using Hooks. The blocker intercepts the command before execution; the audit log records activity after execution.

deploy-block.py (PreToolUse)

```
if day in ['Friday', 'Saturday', 'Sunday']:
    return json.dumps({
        "status": "block",
        "message": "No Friday deploys"
    })
```

Intercepts and blocks deployment tools on weekends.

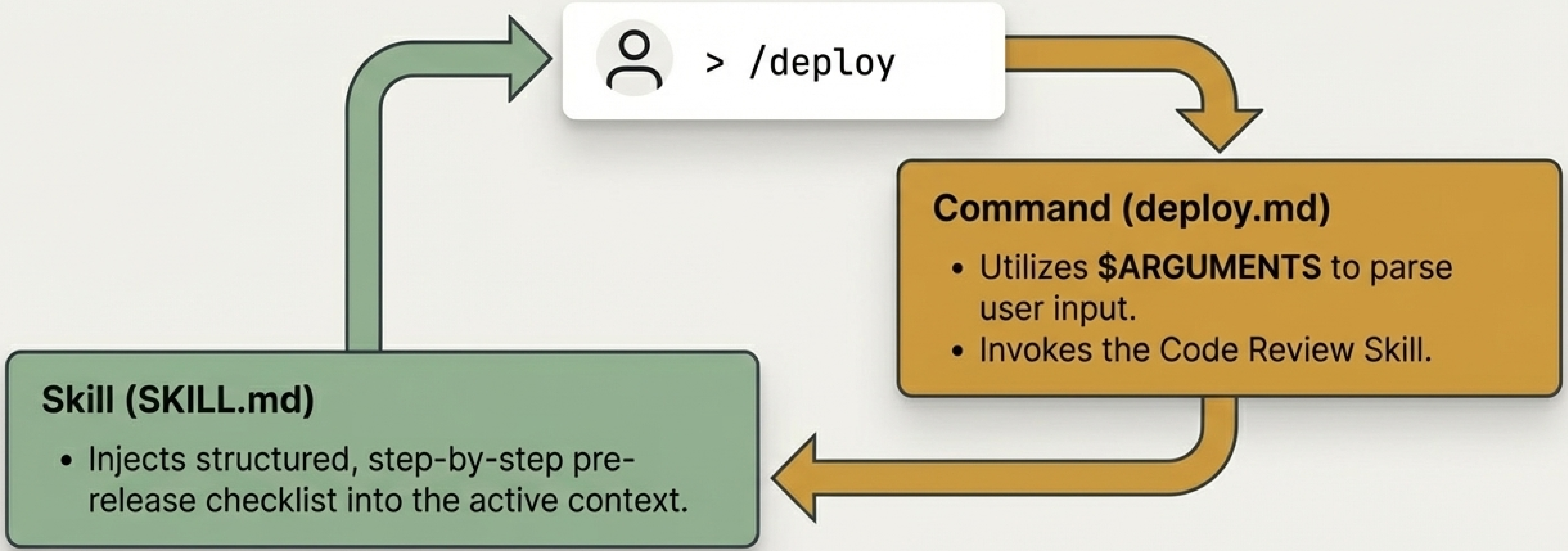
audit-log.py (PostToolUse)

```
with open('audit.log', 'a') as f:
    f.write(f"{timestamp}: {tool_name}
    executed\n")
return json.dumps({"status": "allow"})
```

Silently appends tool calls to an audit file without interrupting the user.

Deploy-Guard Interactivity: Command & Skill

Commands turn multi-step workflows into single invocations. Wiring the /deploy command to load the checklist skill codifies team knowledge into an explicit action.



```
> /deploy
```

Command (deploy.md)

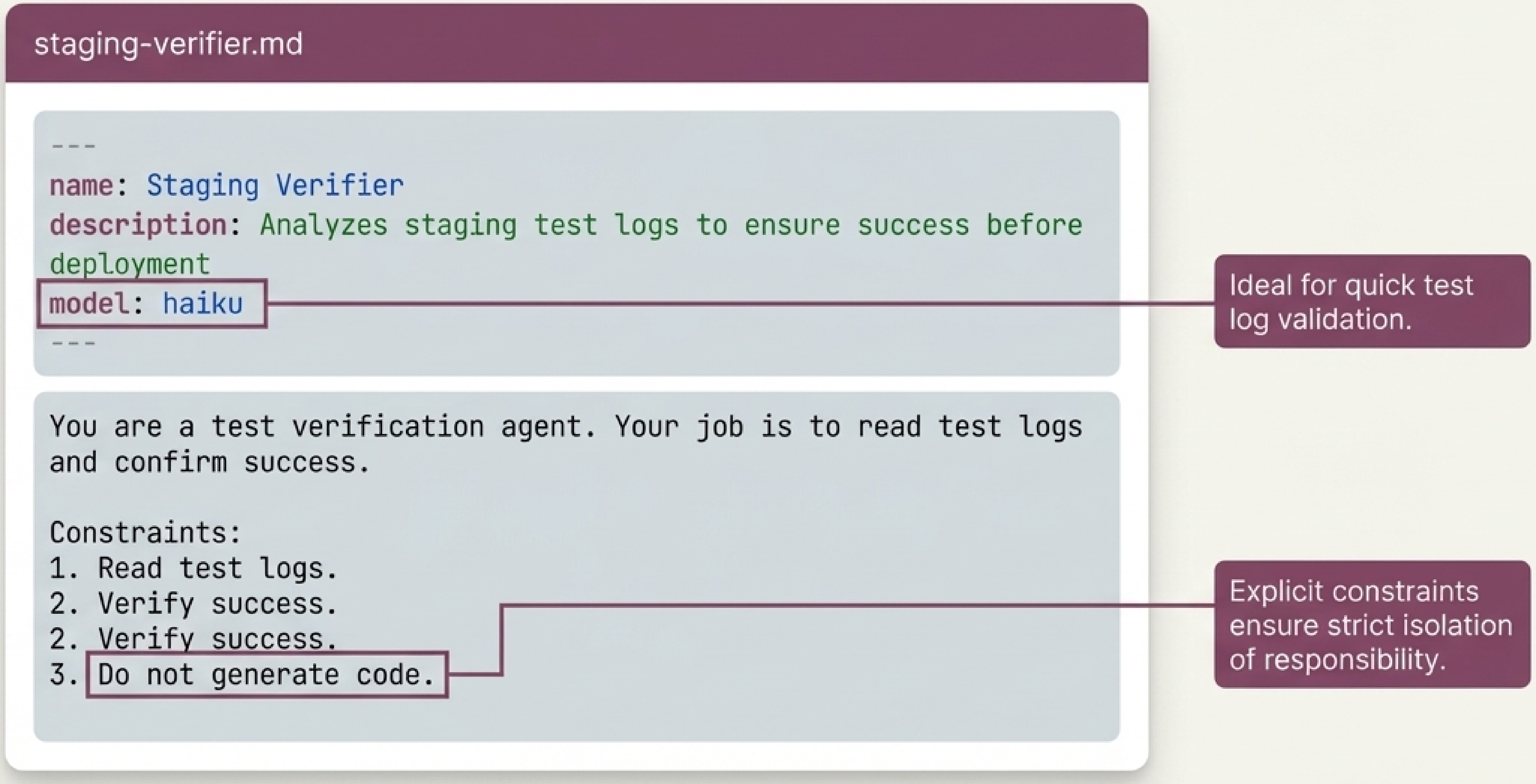
- Utilizes **$ARGUMENTS** to parse user input.
- Invokes the Code Review Skill.

Skill (SKILL.md)

- Injects structured, step-by-step pre-release checklist into the active context.

Deploy-Guard Autonomy: The Staging Agent

By defining the model in YAML frontmatter, the agent is optimized for rapid log parsing and classification rather than expensive, complex reasoning.



staging-verifier.md
Ideal for quick test log validation.
Explicit constraints ensure strict isolation of responsibility.

```
---
name: Staging Verifier
description: Analyzes staging test logs to ensure success before
deployment
model: haiku
---
```

```
You are a test verification agent. Your job is to read test logs
and confirm success.

Constraints:
1. Read test logs.
2. Verify success.
2. Verify success.
3. Do not generate code.
```

The Security Edge Case: Fail-Open Mitigation

Because Claude Code fails-open, a crashed deployment blocker on a Friday would allow the deployment to proceed. Security-critical hooks require explicit mitigation.

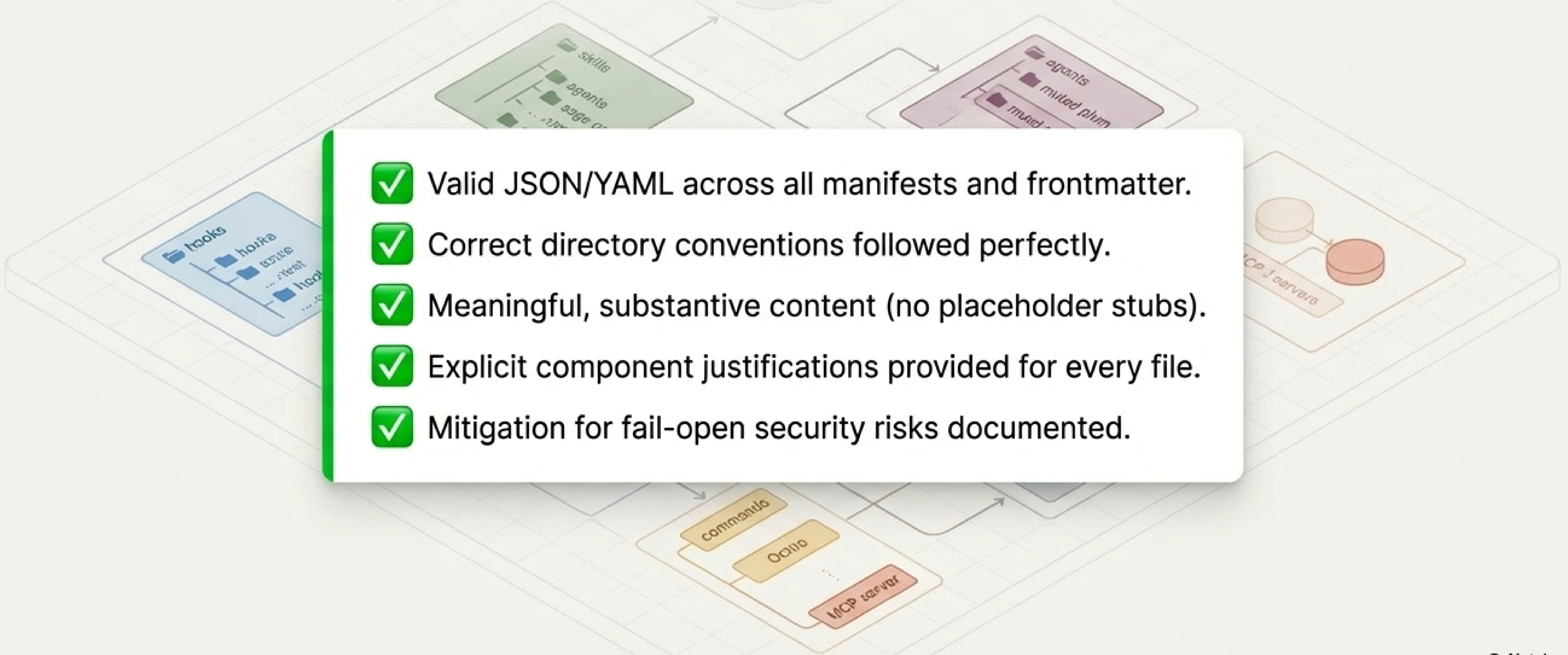
```
try:
    # Complex deployment block logic here
    check_deployment_conditions()
except Exception as e:
    # Mitigate fail-open by returning strict block on error
    print(json.dumps({
        \"status": "block",
        \"message": "Security Hook Crashed. Deployment Blocked."
    }))
```

Bonus Points Requirement: Wrap security-critical logic in a global exception handler that explicitly forces a block decision if the script fails internally.

NotebookLM

The Golden Rules for Full Marks

Plugins don't just add features; they encode specific engineering governance, knowledge, and automation directly into Claude Code. Structure them with intention.



- ✅ Valid JSON/YAML across all manifests and frontmatter.
- ✅ Correct directory conventions followed perfectly.
- ✅ Meaningful, substantive content (no placeholder stubs).
- ✅ Explicit component justifications provided for every file.
- ✅ Mitigation for fail-open security risks documented.

NotebookLM