What Happens When Your Entire Operations Stack Is Just Markdown?

No Python. No JavaScript. No framework. No state database. Just markdown instructions that an AI reads and executes, spawning specialist agents, querying your observability stack, and producing structured investigation reports.

That's exactly what we built. And it works.

Here's an incident triage report our system produced automatically when a PagerDuty alert fired at 1 AM:

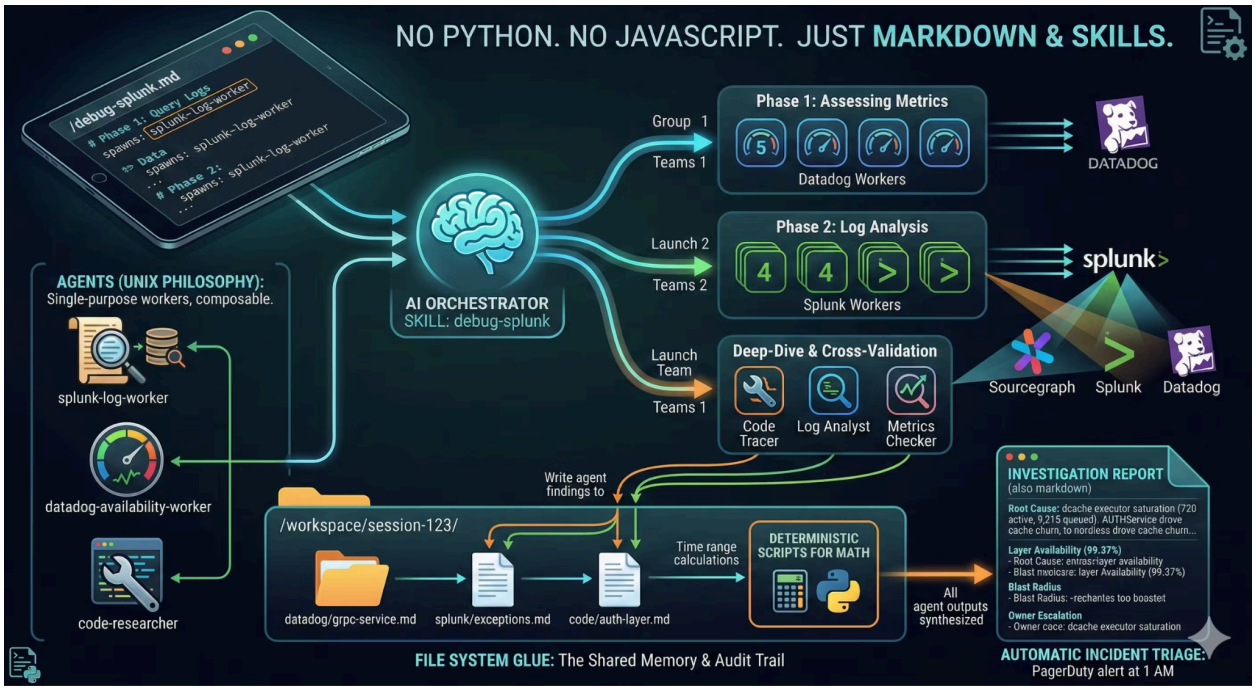
Root Cause: dcache executor saturation (720 active threads, 9,215 queued) 28 minutes before the alert. AuthService drove cache churn; when dcache saturated, GET /api/v1/messages stalled, pushing HTTP p95 from normal to 917.7ms (threshold: 900ms).
The system identified this by spawning 5 metrics workers and 4 log workers in parallel, cross-validating findings across Datadog, Splunk, and source code, and synthesizing a report, all without a single line of traditional code.
Let me show you how.

The Problem: Context-Switching Hell

If you've been on-call, you know the drill. A PagerDuty alert fires. You follow below steps:

Step 1: Open Datadog to check availability.
Step 2: Switch to Splunk to find errors.
Step 3: Grep the codebase to understand the exception.
Step 4: Switch back to Datadog to check a different metric layer.
Step 5: Open PagerDuty again to read the alert details.ou open Datadog to check availability. You switch to Splunk to find errors. You grep the codebase to understand the exception. You switch back to Datadog to check a different metric layer. You open PagerDuty again to read the alert details.
Each tool requires different expertise, different query languages, and different mental models. The investigation is sequential, manual, and entirely dependent on whoever happens to be on-call knowing which queries to run.
Runbooks help, until they don't. They go stale. They can't branch based on what you find. They can't spawn parallel investigations. They can't cross-validate findings across tools.
We needed something that could orchestrate a multi-tool investigation the way an experienced SRE would, but consistently and at 3 AM.

The Architecture: Unix Philosophy for AI Agents



The core insight: treat AI agents like Unix processes. Small. Single-purpose. Composable. The entire system is two directories:

Agents are single-purpose workers. A splunk-log-worker queries Splunk for exceptions. A datadog-worker queries Datadog for availability metrics. A researcher reads code. Each agent has an explicit toolset, behavioral constraints, and a structured output format, nothing more.

Skills are orchestrators. They break complex goals into phases, spawn agent teams, and wire outputs between them. Skills never call tools directly, they coordinate.

The glue between them? The file system. Agents write findings to workspace directories. Downstream agents and the orchestrator read them. No database. No message queue. The file system is the shared memory.

Every file is both a communication channel and an audit trail.

Building Your First Agent

Let's look at a real agent. Here's the splunk-log-worker—the entire definition is a markdown file. That YAML header declares: this agent can Read files, Write files, and Grep. It connects to Splunk through the MCP adaptor. It cannot do anything else.

The body of the markdown file contains the agent's instructions.

Notice three critical design choices:

Explicit constraints prevent hallucination. The agent can only output 5 exception types and 10 timeline entries. Log snippets are capped at 3 lines. This forces the agent to summarize and prioritize rather than dump raw data.
The output format is a contract. Downstream consumers (other agents, the orchestrator) know exactly what structure to expect. The orchestrator can check: "Does exceptions.md contain at least one exception with a class name and timestamp?" If not, the investigation stops early.
Scope boundaries are explicit. The last line of the agent definition says: "Stay in scope. Only query Splunk. Do not search code or make hypotheses about root cause, that's another agent's job." This is crucial. Without explicit boundaries, agents wander.
Compare this to the researcher agent, which is read-only. The disallowedTools field makes the constraint explicit: this agent literally cannot modify files. It observes, analyzes, and reports through its return value—it never changes the codebase.

Building Your First Skill (Orchestrator)

Now let's see how these agents are coordinated. Here's the debug-splunk skill—the simplest orchestrator with 3 phases.
Three things to notice:
Phase gates prevent wasted work. After Phase 1, the orchestrator checks: did we find any exceptions? If not, there's nothing to investigate further. This is a simple conditional that prevents spawning a code-search-worker for nothing.
Context is passed explicitly. The orchestrator doesn't share its memory with subagents. It must pass every piece of context the subagent needs in the task prompt: the API URL, the query, the time range, the output path. This forces clear interface design—like function signatures.
The orchestrator never touches tools. It reads files and spawns agents. That's it. This separation of concerns is what makes the system composable.

Advanced Patterns: Where It Gets Interesting

Pattern 1: Parallel Agent Spawning

The incident-triage skill spawns 5 Datadog workers simultaneously to check 4 availability layers plus the original alert.
Each worker writes to a separate file (datadog/grpc-service.md, datadog/rest-service.md, etc.). The orchestrator reads all five after they complete and decides whether the availability impact is significant enough to warrant Splunk investigation.
In production, this identified that all four availability layers were healthy (99.99%+) while the latency metric was at 917.7ms—telling us this was a latency issue, not an availability issue. That distinction changes the entire investigation direction.

Pattern 2: The Overflow Protocol

What happens when an agent's scope is too large? It doesn't fail. It doesn't produce shallow output. It overflows.
The orchestrator reads this section and spawns additional agents scoped to the uncovered areas. The system scales itself. This is critical for documentation generation, where a single researcher might not be able to analyze an entire large codebase in one pass. Instead of producing a shallow overview, it goes deep on what it can and tells the orchestrator what's left.

Pattern 3: Agent Teams for Cross-Validation

For high-stakes investigations, the incident-triage skill spawns a 3-agent team that cross-validates findings.
This is triangulation. If the code-tracer finds the exception originates in the middleware auth layer, the datadog-checker confirms the error-code breakdown matches, and the splunk-worker finds log entries with the specific HTTP 410 status code—you have convergent evidence from three independent sources.

Pattern 4: Deterministic Scripts for Math

Here's a lesson we learned the hard way: never let an LLM do timestamp arithmetic.
Early versions had agents computing time windows and availability percentages directly. The results were sometimes wrong—hallucinated timestamps, incorrect percentage calculations.
The fix: delegate all math to deterministic Python scripts.

For availability analysis, a dip-detector.py script processes raw Datadog data across 48 half-hour windows. The agent's job is to feed data in and report results—not to do the math.
This hybrid approach—LLM reasoning for investigation strategy, deterministic scripts for computation—eliminates an entire class of hallucination.

Real Results: What the System Actually Produces

Case 1: Incident Triage (gRPC Error Rate)

Alert: gRPC error rate spike. 675 INTERNAL errors. Availability dropped to 99.37%.
The system spawned 5 Datadog workers + 4 Splunk workers + a 3-agent deep-dive team and produced:

Layer	Availability	Status
gRPC Service	99.37%	DIP
gRPC Envoy	99.56%	DIP
REST Envoy	99.9998%	OK
REST Service	100.00%	OK

Root cause: Core backend pod discovery endpoints returned HTTP 410 Gone for 161 tenants during a pod migration, blocking tenant permission validation before handler execution.
The report included an end-to-end call stack, status code mapping, and ownership/escalation guidance. An on-call engineer could read the executive summary and know exactly what happened, who to escalate to, and what the blast radius was.

Case 2: Full-Day Availability Analysis

The system scanned 4 metrics across 48 half-hour windows—192 data points. The deterministic dip detector found 4 windows below 99.99%.
For the worst window (22:30–23:00 UTC), it found:

62,939 gRPC UNAVAILABLE errors
Connection pool exhaustion: 121/121 connections, 150+ waiters
Primary affected tenant: tenant-49a8f1b with 12,459 errors
Root cause: Synchronized upstream connection pool overflow cascading to database connection pool exhaustion.

Case 3: Deployment Monitoring

5 Kubernetes pods, 10 metrics each, 3 monitoring rounds:
4 pods: OK
1 pod: WARNING (gateway-to-backend 4XX rate increased 6.2x)
Data completeness: 88% (44/50 metrics)
The system flagged the anomaly with exact pre/post values and recommended human review—not automated remediation. The goal is to augment the on-call engineer, not replace them.

What I Learned: The Hard-Won Lessons

What Works

Structured output contracts are the most important design decision. Without them, agents produce inconsistent, hard-to-parse output that breaks downstream consumers. When you

specify "max 5 exception types, sorted by count, with these exact fields," you get reliable, composable building blocks.

File-based memory is surprisingly effective. We expected to need a database or message queue. We didn't. The file system provides natural namespacing (directories), session isolation (session IDs in paths), and a complete audit trail. And every engineer already knows how to inspect files.

Explicit constraints prevent agent wandering. The sentence "Stay in scope. Only query Splunk. Do not make hypotheses about root cause—that's another agent's job" is more important than any framework feature. Without it, agents try to do everything and do nothing well.

What Doesn't Work

LLMs should never do math. Timestamp arithmetic, availability percentages, time window calculations, all of these need deterministic scripts. We learned this after agents confidently reported incorrect timestamps.

Context overflow is real. Model context limits apply to nested chains. Long investigations with many agents can lose context at boundaries. The overflow protocol helps, but some information is inevitably lost in the handoff.

Token costs scale linearly with team size. 5 parallel agents means 5x the tokens. The batch limit of 4 concurrent agents partially mitigates this, but cost-conscious teams should consider which phases truly benefit from parallelism.

Service-specific knowledge is still hardcoded. Each skill contains Splunk index names, Datadog metric paths, and PagerDuty service naming conventions. Generalizing to a new service means writing new skill definitions. This is a feature (precision) and a limitation (effort) simultaneously.

The Next Frontier: Closing the Loop on Remediation

1. **Automated Remediation & Scaffolding**
    - *Command Generation:* Agents analyze findings to suggest deterministic CLI actions (e.g., `kubectl rollout undo deployment/...` or `systemctl restart ...`). These aren't just guesses; they are pre-populated blocks ready for human review.
    - *Configuration Patches:* For misconfiguration root causes, the system generates structured diffs (YAML, JSON, HCL) to align state with intent.
    - *Code Patch Suggestions:* By interfacing with code-search agents, the system can suggest surgical fixes for known error patterns, leveraging LLM reasoning over the codebase.
2. **Integrated Communication Drafts**
    - *Incident Updates:* Automatically synthesize draft status updates for IRM or PagerDuty, summarizing current impact and investigation phases for stakeholders.
    - *Team Communications:* Generate high-signal briefs for Slack or Google Chat to maintain context across the engineering org.

3. **Proactive Ticket & Bug Management**
    - *Templated Bug Creation:* Spawn agents to pre-fill Buganizer or Jira tickets with the full audit trail: root cause, logs, and metric snapshots.
    - *Similar Issue Linking:* Automate the look-back by surfacing historical incidents that match current error signatures.
4. **Runbook Enhancement & Learning**
    - *Dynamic Runbook Generation:* The Skill execution itself becomes the runbook. Every agent action and finding creates a live, instance-specific record of the investigation.
    - *Runbook Update Suggestions:* Use post-incident synthesis to suggest updates to g3doc runbooks, ensuring tribal knowledge is codified immediately.
5. **Direct Tool Actions (with Safeguards)**
    - *Safe Read-Only Actions:* Standard agent behavior for telemetry and log collection.
    - *Controlled Write Actions:* A specialized class of agents for high-leverage tasks:
        - Silencing alerts to reduce noise.
        - Traffic draining or cell isolation.
        - Triggering rollbacks via deployment APIs.
    - *Human-in-the-Loop:* Critical constraint: all mutating actions require an explicit "ACK" from a human engineer before execution.
6. **"What-if" Scenario Analysis**
    - Engineers can fork the workspace to test hypotheses—like modeling the impact of increasing connection pool sizes—before committing to a fix.

Implementation Considerations

- **Security:** High-privilege agents require robust permission models and immutable audit trails.
- **Tool Integrations:** Expansion relies on deep MCP adapter coverage for deployment and ticketing stacks.
- **Agent Capabilities:** New "allowedActions" fields must be strictly defined to prevent scope creep.

By folding these patterns into the stack, the system evolves from a simple analysis engine into a true operations co-pilot, augmenting the engineer throughout the entire incident lifecycle.

Getting Started

The entire framework is open source. Here's how to try it:
Clone and bootstrap (Set up your local directory structures).
Configure MCP connectivity: You'll need the MCP Gateway configured with access to Splunk, Datadog, and Sourcegraph.

Try the simplest skill first: Start with /debug-splunk—paste a Splunk URL and watch the 3-phase investigation unfold. Then try /incident-triage with a PagerDuty service name for the full multi-agent experience.
What you'll need per skill

Skill	**Required MCPs**
/debug-splunk	Splunk, Sourcegraph
/monitor-deployment	Datadog, Kafka
/incident-triage	PagerDuty, Datadog, Splunk, Sourcegraph
/availability-analyze	Datadog, Splunk
/chaos-triage	ChaosMesh, Sourcegraph, Splunk

The Bigger Picture

The architecture, markdown-defined agents, file-based memory, phase-based orchestration - is not specific to SRE. The same patterns apply anywhere you need a team of specialists to investigate a complex problem: security analysis, compliance auditing, code review, research synthesis.
The key insight is that the "code" in an AI-native system isn't Python or JavaScript. It's the instructions that tell the model what to do, who to delegate to, and how to structure its output. Markdown is just the simplest, most universal way to write those instructions.
Your runbooks were already halfway there. They just needed an executor that could actually follow them.