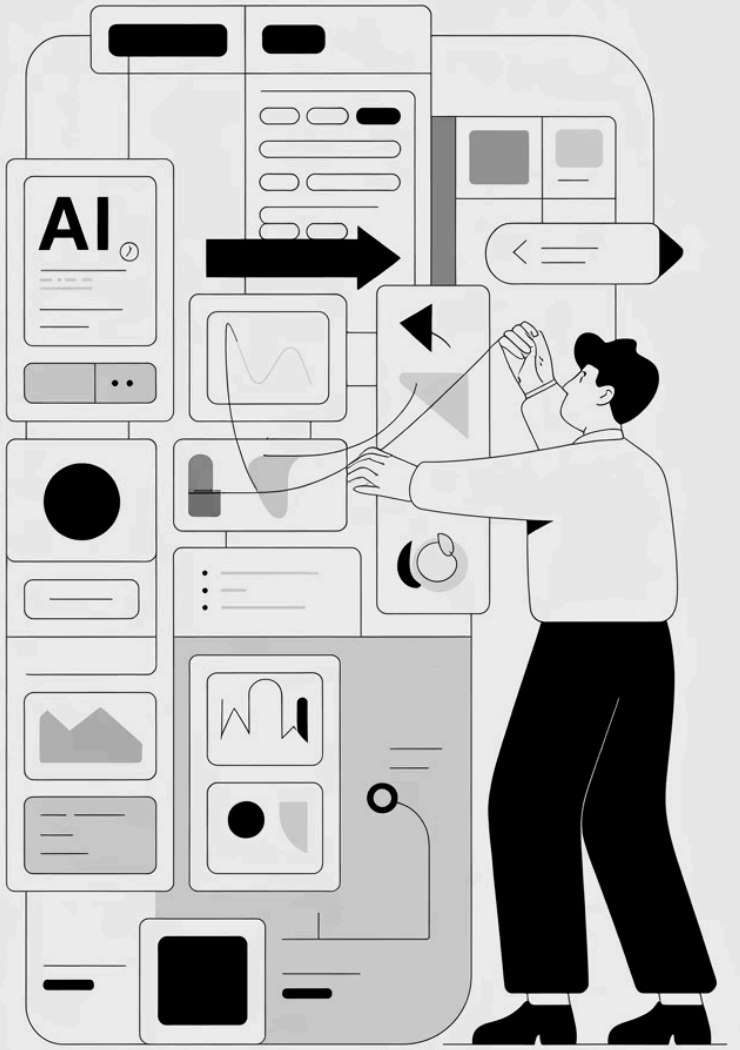


Unlock AI's True Potential.

Stop Fighting Your AI:

Engineering Prompts That Actually Work

You've likely experienced the inconsistent magic of AI prompts. Today, we move beyond basic trial-and-error to a strategic, production-ready approach. Discover how clear structure, deep context, and understanding model-specific patterns transform your AI interactions from "kind of works" to "consistently delivers." Get ready to elevate your AI game.

Find Slides @ www.martinrojas.dev

by Martin Rojas

Agenda

Prompt Architecture

The anatomy of production prompts

Core Engineering Techniques

Clarity, Chain-of-Thought, Constraints, Compression

AI-as-Coach

Using AI to improve your own prompts

Advance Patterns

Layering techniques for reliable results

Part 1: Prompt Architecture

The Anatomy of Production Prompts

Component Stack

Component	Purpose	DevOps Example
System Message	Sets behavior and role	"You are a senior SRE writing blameless postmortems"
Instruction	What to do	"Generate a postmortem for this incident"
Context	Background data	"Alert timeline, service logs, deployment history"
Examples	Pattern demonstration	"Sample: 'At 02:14 UTC, the API gateway began returning 503 errors...'"
Constraints	Output limits	"Blameless tone, 5-section format, action items must have owners"
Delimiters	Section separation	###, ---, """

Think of prompts as modular—each component serves a distinct purpose. Like a well-structured business report, clarity in structure leads to clarity in output.

Markdown: Structuring Prompts for Precision

Markdown syntax provides a clear, standardized way to format your AI prompts, ensuring models accurately interpret structure, hierarchy, and emphasis.

`# Heading`	Defines sections, outlines hierarchy	`# Root Cause Analysis`
`**Bold Text**`	Emphasizes keywords or concepts	`**Severity:** SEV1`
`- List Item`	Presents clear, actionable points	`- Step 1: Check deployment timeline`
`` `Code` ``	Isolates specific terms or variables	Filter where `service = 'api-gateway'`
`` ```Block``` ````	Encapsulates larger blocks of text/code	```` ```Alert JSON here``` ````

Our Running Example

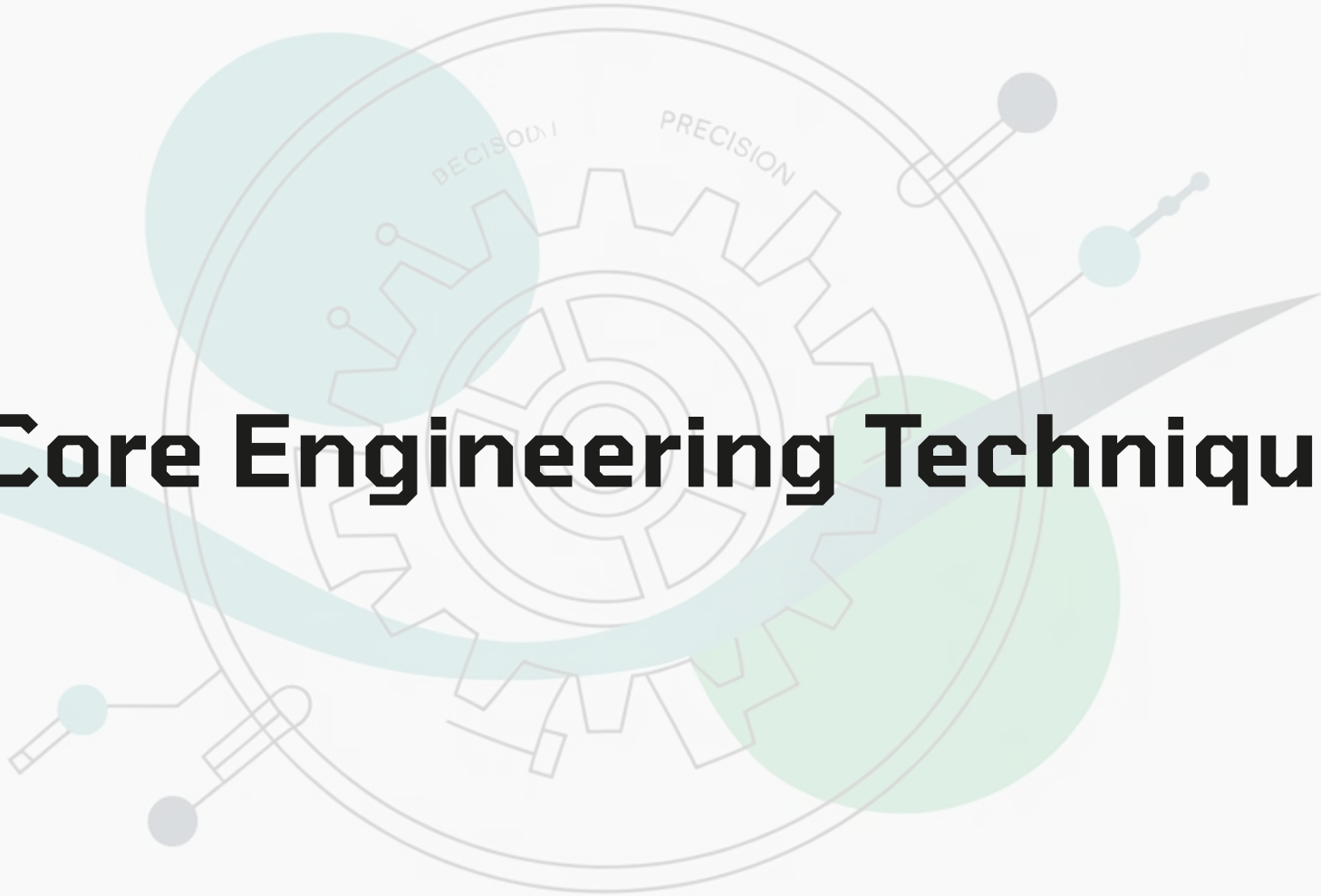
Base Prompt: Incident Postmortem Generator

This is our starting point—already decent but not production-ready. We'll transform this throughout the presentation.

> Write a postmortem for last night's outage.

What's missing?

- Which service? What's the severity level?
- No data sources specified (logs? alerts? deployment history?)
- No expected format — every engineer writes them differently
- Blameless tone not specified — AI may assign fault to individuals
- Two implicit questions bundled: what happened + how to prevent it



Part 2: Core Engineering Techniques

Foundational Techniques in Action

The Basic Building Blocks

Zero-shot — Direct Instruction

```
Write a postmortem for the API gateway outage.
```

✅ Simple, fast | ❌ Inconsistent quality, may fabricate timeline details

Few-shot — Pattern Learning

```
Example 1 (Bad Deploy): "API gateway v3.1.2 caused 503s
— rollback resolved in 23 min
→ ROOT CAUSE: missing env var in deploy config"

Example 2 (Resource Exhaustion): "Redis OOM after traffic
spike — scaled cluster in 41 min
→ ROOT CAUSE: cache sizing assumptions invalid for peak
load"

Example 3 (Dependency Failure): "Auth service timeouts
cascaded to checkout — circuit breaker
opened in 18 min → ROOT CAUSE: upstream token
validation latency spike"

Based on the incident data, identify the root cause category
and write a matching postmortem.
```

✅ Adapts to context | ❌ Token intensive

One-shot — Format Setting

```
Example: "On 2026-03-12, the payment service returned
503 errors for 47 minutes
affecting ~12K users. Root cause: memory leak introduced
in v2.3.1 deploy.
Resolved by rollback to v2.3.0."

Now write a postmortem for this incident: {incident_data}
```

✅ Consistent format | ❌ Limited pattern learning

Role-based — Behavioral Framing

```
You are a senior SRE at a company with a strong blameless
postmortem culture.
Write a postmortem that focuses on systemic
improvements and process gaps,
not individual fault.
```

✅ Consistent voice | ❌ May override other instructions

Technique 1: Clarity & Specificity

The Ambiguity Tax

❌ Vague Postmortem Request:

```
Write a postmortem for last night's outage.
```

Problems: Which service? What severity? What data to use? What audience? Blameless?

✅ Refined with Specificity:

```
You are a senior SRE writing a blameless postmortem for the engineering team.

Tone: Blameless — focus on systems and conditions, not people.
Example: "The deploy pipeline lacked a timeout validation step" not "Alice forgot to set the timeout."
Audience: Engineering team + VP Engineering
Format: 5 sections with bold headers. Stop after the Action Items table.

Using the provided incident data below, write a postmortem covering:
1. Timeline of key events (first alert to full resolution)
2. Root cause (what technical condition caused the failure)
3. User impact (requests affected, error rate, SLO breach details)
4. Contributing factors (conditions that amplified impact or slowed detection)
5. Action items (owner, action, due date, priority)

---
{incident_data}
---
```

Model-Specific Notes (April 2026)

GPT-5.4

Drop the "act as" framing — move to exact output specs. Be explicit about verbosity, tone, answer length, and section order. Vague requests don't get expanded; they get answered literally. For a postmortem, specify the completion criteria: "Stop after the Action Items table. Do not add a summary paragraph." Place variable input (the incident data) at the end of the prompt for better cache performance.

Claude Opus 4.7

State the full task in the first sentence — avoid building context progressively across the prompt. Show what you want rather than listing what to avoid: provide a one-sentence example of the blameless tone you expect rather than writing "don't assign blame." Only include guidance that directly shapes the output — over-engineering degrades performance on this model.

Gemini 3.1 Pro

Use this structure: Role → Goal → Constraints → Example → Output format. Be transactional, not conversational. Pin hard constraints explicitly: "3 contributing factors, ≤ 40 words each, no passive voice." Place your incident data before your question — anchor with "Based on the incident data above..." — Gemini 3.1 reasons better when the context precedes the instruction.

Technique 2: Chain-of-Thought (CoT)

Make the Model Think Like an Analyst

❌ Direct Approach:

```
Write a postmortem for the API gateway outage.
```

✅ With Chain-of-Thought:

```
Analyze this incident step-by-step before writing the
postmortem:

1. First, reconstruct the timeline: when did alerts fire, when did
engineers engage,
   when was the root cause identified, when was service restored?
2. Then, identify the root cause: what specific technical failure
triggered the incident?
3. Next, assess the impact: how many requests failed, what was
the error rate,
   which SLOs were breached?
4. Identify contributing factors: what conditions amplified the
impact or slowed
   detection and response?
5. Determine action items: what specific changes would prevent
recurrence
   or improve response time?
6. Now write the postmortem using only facts established in
steps 1–5.

Think through each step before writing.
```

Advanced with XML Tags (Claude-optimized)

Prompt you send:

```
Think through each step before writing. Use <reasoning> tags to show your work,
then write the postmortem inside <postmortem> tags.

---
{incident_data}
```

What the model returns:

```
<reasoning>
  Timeline: Alert at 02:14, PagerDuty page at 02:17, engineer engaged 02:23,
        root cause identified 02:58, fix deployed 03:31, resolved 03:47
  Root cause: nginx config in v4.2.1 set upstream timeout to 100ms (was 5000ms),
        causing cascade of 503s under normal load
  Impact: 847K requests failed, 8.2% error rate, availability 97.1% vs 99.9% SLO
  Contributing factors: no load test in staging, alert threshold at 5% not 1%
  Action items: timeout validation in deploy checklist, lower alert threshold
</reasoning>

<postmortem>
  ## API Gateway Incident Postmortem — 2026-04-18
  **SUMMARY:** nginx timeout misconfiguration in v4.2.1 caused 503 errors for 93 minutes,
  affecting 847K requests and breaching the availability SLO.
  [Full postmortem follows...]
</postmortem>
```

The tags give you two things: visible reasoning you can audit before the doc reaches stakeholders, and a clean postmortem you can extract directly.

Why CoT matters for incident analysis: Prevents the model from guessing root causes or fabricating timeline details. You can trace the logic and catch discrepancies before the document reaches stakeholders.

Technique 3: Format Constraints

Structure = Reliability

❌ Unstructured:

```
Write a postmortem about the outage.
```

✅ With Format Constraints:

```
Write the incident postmortem with EXACTLY this structure:

SUMMARY: [One sentence: what failed, for how long, user impact]

TIMELINE:
- [HH:MM UTC] [Event — what happened, what was observed]
(Use only timestamps from the provided alert data)

ROOT CAUSE: [2–3 sentences, technical, blameless]

CONTRIBUTING FACTORS:
- [Factor 1]
- [Factor 2]

ACTION ITEMS:
| Owner | Action | Due Date | Priority |
|-------|--------|----------|----------|

Return ONLY the formatted postmortem. No additional commentary.
```

Structured JSON Output (For System Integration)

```
Return ONLY valid JSON matching this schema:

{
  "incident_id": "string (e.g. INC-2026-0418)",
  "severity": "SEV1|SEV2|SEV3",
  "service": "string",
  "start_time": "ISO8601",
  "end_time": "ISO8601",
  "duration_minutes": integer,
  "summary": "max 100 chars",
  "timeline": [
    {
      "timestamp": "ISO8601",
      "event": "string",
      "actor": "system|engineer|automated"
    }
  ],
  "root_cause": "string",
  "contributing_factors": ["string"],
  "user_impact": {
    "requests_failed": integer,
    "error_rate_pct": decimal,
    "slo_breached": boolean
  },
  "action_items": [
    {
      "owner": "string",
      "action": "string",
      "due_date": "YYYY-MM-DD",
      "priority": "P1|P2|P3"
    }
  ]
}

NO additional text. Only JSON.
```

Success Rate: 92% valid JSON (vs 45% with natural language requests)

Technique 4: Prompt Compression

Every Token Counts

❌ Verbose (147 tokens):

You are a highly experienced senior site reliability engineer with extensive experience
in incident management and blameless postmortem culture. Your task today is to carefully
analyze the incident data that has been provided to you, which contains all of the alert
timeline information, service dependency details, and deployment history from the past 24
hours, and then create a comprehensive blameless
postmortem document that would be appropriate for sharing with the engineering team and engineering leadership. Please make
sure to include a detailed timeline of events, a technical explanation of the root cause
that does not blame any individual, and a clear set of action items with owners.

✅ Compressed (42 tokens):

Senior SRE. Write a blameless postmortem from incident data.
Cover: timeline, root cause, user impact, contributing factors, action items.
Format: 5-section engineering doc.
Every action item needs an owner.

Ultra-Compressed (26 tokens):

Task: Blameless incident postmortem
Input: Alert timeline, logs, deploy history
Output: Timeline, root cause, impact, action items with owners

Savings: 71–82% fewer tokens | Same output quality | Lower costs

> **Challenge:**
>
> Take your longest prompt, cut 40% of tokens. Drop filler words: 'please,' 'could you,' 'make sure.' Use headers and lists instead of sentences.

Building Layers

Progressive Enhancement

Layer 0: Naked Prompt

Write a postmortem for last night's outage.

❌ Vague, inconsistent, may hallucinate timeline details

Layer 1: +Role

You are a senior SRE.

Write a blameless postmortem for the API gateway outage that caused
503 errors from 02:14–03:47 UTC.

✓ Consistent technical tone

Layer 2: +Specificity

You are a senior SRE writing for the engineering team.

Analyze the following incident:
- Service: API Gateway
- Error: 503s for 93 minutes
- Requests affected: ~847K
- SLO impact: Availability dropped to 97.1% (target: 99.9%)

Use the provided alert timeline and deployment history.

✓ Clear parameters

Layer 3: +Chain-of-Thought

[Previous layers...]

Before writing, work through these steps:
1. Reconstruct the event timeline from alert data
2. Identify the technical root cause
3. Assess user impact and SLO breach
4. Find contributing factors
5. Generate action items with owners and due dates

Show your reasoning, then write the postmortem.

✓ Traceable logic

Layer 4: +Format Constraints

[Previous layers...]

OUTPUT FORMAT:
SUMMARY: [1 sentence: what failed, duration, impact]
TIMELINE: [HH:MM UTC — Event]
ROOT CAUSE: [technical, blameless, 2–3 sentences]
CONTRIBUTING FACTORS: [bullet list]
ACTION ITEMS: [table: Owner | Action | Due Date | Priority]

✓ Predictable structure

Layer 5: +Validation (Production-Ready)

SYSTEM: You ONLY use facts from the provided incident data. Never estimate
timestamps or fabricate error counts. If data is missing, state "Data not available."

[All previous layers...]

VALIDATION: Before writing, confirm:
- All timestamps match source alert data exactly
- Root cause is technical, not person-focused
- Every action item has a named owner and due date

✓ Production-ready

Part 3:

AI-as-Coach

The AI-as-Coach Method

Instead of guessing what makes a good prompt, let AI help you improve your prompts. This creates a feedback loop that rapidly improves your results.

Quick-Start Template

Act as a Senior Prompt Engineer. Your goal is to help me refine and optimize a prompt for maximum performance. When I provide a prompt, please follow these steps:

1. Critique: Review the prompt for structure, coherence, and clarity. Identify any ambiguous phrasing or "fluff" that might confuse an LLM.
2. Intent & Specificity: Evaluate if the prompt provides enough context and clear constraints. Identify what might be missing to achieve a high-quality result.
3. Clarifying Questions: Before providing the final version, ask me 2-3 targeted questions to bridge any gaps in context, tone, or intended output format.
4. The Optimization: Once I answer, provide the "v2.0" version of the prompt using best practices (e.g., Role-assigning, Delimiters, and Chain-of-Thought prompting).

Do you understand? If so, please ask me for the prompt you'd like to optimize.

[your current prompt]

Why This Works:

Sequential Logic

Instead of just guessing what you want, it forces the AI to stop and think (and ask you questions) before it writes the final version.

Structural Integrity

By asking for "Delimiters" and "Role-assigning," you're prompting the AI to use professional engineering techniques in the output.

A quick tip for the road

If you find the AI is still being too "wordy" in its critique, you can add a line to the end: *"Keep the critique concise and focused on actionable improvements."*

The Lean Master Prompt

We can condense multiple optimization goals into a "Lean Master Prompt" that retains the powerful recursive logic of reviewing, questioning, and refining. This version trims unnecessary verbose phrasing without sacrificing effectiveness.

> Act as a Senior Prompt Engineer. Review my next prompt for ****clarity, structure, and intent****. Identify any ambiguities or missing context, then ****ask 2-3 targeted questions**** to bridge those gaps. After I respond, provide an optimized "v2.0" using professional prompting standards (Roles, Delimiters, and Constraints). Ready?

Why this keeps the logic:

Bundled Review

"Clarity, structure, and intent" efficiently covers comprehensive critique, replacing multiple distinct review steps.

The "Stop" Command

It still forces the AI to ask questions *before* generating the final version, preventing premature "best guess" outputs.

High Standards

Referencing "professional prompting standards" succinctly instructs the AI to apply advanced techniques like role-assigning and Chain-of-Thought.

Part 4: The AI Mindset

Key Takeaways:

Structure beats sophistication — Clear formatting produces better results than clever wording

Test techniques against YOUR use cases — What works for one query may not work for another

Measure what matters — Accuracy, consistency, and speed

Build prompt templates — Reuse what works for common finance queries

Use AI to improve AI — The AI-as-Coach method creates a rapid feedback loop for prompt refinement

Immediate Action:

"Pick one advanced technique from today"

"Apply it to your most problematic prompt"

"Share results back to the community"

Part 5:

Advanced Patterns

Tree of Thought (ToT)

Explore multiple analytical approaches *simultaneously*, then evaluate which reveals the most insight.

We're investigating a P1 incident. Before writing the postmortem,
we need to understand WHY it happened.

Explore this problem through THREE different analytical lenses:

LENS A — Code/Deploy:
- Was there a recent deployment, config change, or feature flag activation?
- What changed in the system in the 24h before the incident?
- Is the root cause traceable to a specific code or configuration change?
- Was the change tested adequately before reaching production?

LENS B — Infrastructure/Capacity:
- Was there unusual load or traffic patterns before the incident?
- Did any resource (CPU, memory, connections, disk) hit limits?
- Were there upstream or downstream dependency failures?
- Is this a resource sizing or infrastructure configuration issue?

LENS C — Process/Detection:
- Were alerts configured to catch this class of failure?
- How long did it take to detect vs. actual start of incident?
- Were runbooks available and up to date?
- Were escalation paths followed correctly?

After completing all three analyses:
1. Compare the explanatory power of each lens
2. Identify which approach reveals ACTIONABLE root causes
3. Recommend which lens should drive the postmortem narrative and remediation

Present findings as:
- Executive summary (which lens won and why)
- Key insight from each lens
- Recommended action items based on the strongest analysis

When to use: When you're not sure whether an incident was caused by a bad deploy, infrastructure failure, or process gap — let the data reveal it.

Why this works: "503 errors" might be a symptom. The root cause could be a config change, an upstream dependency, or a monitoring gap that let the incident run unchecked. ToT surfaces which story the data actually supports.

Self-Consistency

Majority Vote for Accuracy

Document the root cause of the API gateway incident for the engineering postmortem.
This will appear in our quarterly reliability review, so accuracy is critical.

Perform THREE independent analyses:

METHOD 1 — Timeline Analysis:
- Reconstruct events chronologically from alert data
- Identify the first anomaly before errors began
- Trace from first anomaly to confirmed root cause
- Document user impact at each stage

METHOD 2 — Change Analysis (What Changed?):
- Review all deployments in 24h before incident
- Review config changes, feature flag activations, infrastructure changes
- Identify which change correlates with the onset of errors
- Validate by checking error onset timestamp vs. change timestamp

METHOD 3 — Dependency Analysis:
- Map all upstream and downstream dependencies of the affected service
- Check which dependencies showed degradation at incident onset
- Identify the cascade pattern (did it originate upstream or in the service itself?)
- Cross-reference with historical incidents for pattern matching

VALIDATION:
- Compare all three root cause conclusions
- If all match: Report with HIGH CONFIDENCE
- If 2 of 3 match: Flag the outlier method, investigate why
- If all differ: STOP — escalate for data review before publishing postmortem

Output format:
Method	Root Cause Identified	Confidence	Notes
Timeline			
Change			
Dependency			

FINAL ROOT CAUSE: [statement] (Confidence: HIGH/MEDIUM/LOW)

When to use: High-stakes incidents going into quarterly reliability reviews, SLO breach reports, or board-level summaries.

Why this works: Catches errors that single-path reasoning misses. If all three methods agree on root cause, ship the postmortem. If they disagree, you've found a data quality issue or reasoning error *before* it reaches stakeholders.

ReAct Pattern

Iterative Reason → Act → Observe loops for agentic investigations.

When to use: Agentic systems where the model has access to real tools — alert queries, metrics APIs, deployment logs. Each observation shapes the next action, turning the model into an active investigator rather than a passive summarizer.

```
You are an SRE agent investigating a P1 incident. You have access to these tools:
- query_alerts(time_range)
- query_metrics(service, metric, time_range)
- query_deployments(time_range)
- query_dependencies(service)

Elevated error rates detected in the API gateway starting 02:14 UTC.
Investigate the root cause.

For each step use this exact format:
THINK: [What do I need to know? What's my current hypothesis?]
ACT: [Tool call with parameters]
OBSERVE: [Key findings from the tool response]

Continue iterating until you can state the root cause with confidence, then output:

ROOT CAUSE: [statement]
SUPPORTING EVIDENCE: [one finding per cycle that led here]
RECOMMENDED ACTION ITEMS: [owner | action | priority]
```

Why this works: The model can't hallucinate a timeline it hasn't queried. Each ACT is grounded in a real tool call, and each THINK is grounded in a real OBSERVE. The investigation is auditable — every conclusion traces back to an actual data point.

Note: This pattern requires a model with function calling or tool use enabled (e.g., via an MCP server, LangChain agent, or your platform's agentic framework). The prompt structure is the same regardless of the underlying tooling.