Arbiter: Detecting Interference in LLM Agent System Prompts

A Cross-Vendor Analysis of Architectural Failure Modes

Tony Mason
tony@wamason.com
University of British Columbia
Vancouver, BC, Canada

Abstract

System prompts for LLM-based coding agents are software artifacts that govern agent behavior, yet lack the testing infrastructure applied to conventional software. We present Arbiter, a framework combining formal evaluation rules with multi-model LLM scouring to detect interference patterns in system prompts. Applied to three major coding agent system prompts — Claude Code (Anthropic), Codex CLI (OpenAI), and Gemini CLI (Google) — we identify 152 findings across the undirected scouring phase and 21 hand-labeled interference patterns in directed analysis of one vendor. We show that prompt architecture (monolithic, flat, modular) predicts the *class* of failure but not its *severity*, and that multi-model evaluation discovers categorically different vulnerability classes than single-model analysis. One scourer finding — structural data loss in Gemini CLI's memory system — was independently confirmed when Google filed and patched the symptom without addressing the schema-level root cause identified by the scourer. Total cost of cross-vendor analysis: $0.27 USD.

1 Introduction

LLM-based coding agents — Claude Code, Codex CLI, Gemini CLI, and others — are governed by system prompts ranging from 245 to 1,490 lines. These prompts are software artifacts in every meaningful sense [6]: they specify behavior, encode precedence hierarchies, define tool contracts, manage state, and compose subsystems. A system prompt is the constitution under which the agent operates.

Unlike conventional software, these constitutions have no type checker, no linter, no test suite. When a system prompt contains contradictory instructions — "always use TodoWrite" in one section, "NEVER use TodoWrite" in another — the executing LLM resolves the conflict silently through whatever heuristic its training provides. No error is raised. No warning is logged. The contradiction persists, and the agent's behavior becomes a function of which instruction the model happens to weight more heavily on a given invocation.

This paper argues that **the agent that resolves the conflict cannot be the agent that detects it.** An LLM executing a contradictory system prompt will smooth over inconsistencies through "judgment" — the same mechanism that makes LLMs useful makes them unreliable as their own auditors. External evaluation against formal criteria is required.

We present Arbiter, a framework with two complementary evaluation phases:

1. **Directed evaluation** decomposes a system prompt into classified blocks and evaluates block pairs against formal interference rules. This is exhaustive within its defined search frame: if a rule exists for a failure mode, and two blocks share the relevant scope, the evaluation is guaranteed to check that pair.
2. **Undirected scouring** sends the prompt to multiple LLMs with deliberately vague instructions — "read this carefully and note what you find interesting." Each pass receives the accumulated findings from all prior passes and is asked to go where previous explorers didn't. Convergent termination (three consecutive models declining to continue) provides a calibrated stopping criterion.

Applied to three major coding agent system prompts, these two phases reveal 152 scourer findings and 21 hand-labeled interference patterns. The findings organize into a taxonomy determined by prompt architecture: monolithic prompts produce growth-level bugs at subsystem boundaries, flat prompts trade capability for consistency, and modular prompts produce design-level bugs at composition seams.

Our contributions are:

- A taxonomy of system prompt failure modes grounded in cross-vendor empirical evidence.
- A methodology for systematic prompt analysis combining directed rules with undirected multi-model scouring.
- Evidence that multi-model evaluation discovers categorically different vulnerability classes than single-model analysis.
- External validation: a scourer-identified design bug independently confirmed by the vendor's own bug report and patch.
- Demonstration that comprehensive cross-vendor analysis costs $0.27 in API calls — less than three minutes of US minimum wage labor.

2 Background and Related Work

2.1 Prompt Engineering

The prompt engineering literature has grown rapidly since 2023, but overwhelmingly focuses on crafting prompts for specific tasks: few-shot formatting, chain-of-thought elicitation, retrieval-augmented generation patterns. The quality of the prompt itself — as a software artifact with internal consistency requirements — receives comparatively little attention. Wei et al. [7] established chain-of-thought prompting; Yao et al. [8] introduced tree-of-thoughts; neither addresses the structural integrity of the prompts themselves. Recent work has begun to frame prompts as first-class software engineering artifacts requiring systematic management, evolution, and testing [6], but tools for structural analysis of prompt internals remain absent.

2.2 Prompt Security

Prompt injection research [2, 5] examines adversarial inputs designed to override system prompt instructions. This work is orthogonal to ours: we analyze the system prompt itself for internal contradictions, not external attacks against it. A system prompt can be perfectly secure against injection yet internally contradictory.

2.3 Software Architecture Failure Modes

The analogy between system prompt architectures and software architectures is deliberate. Monolithic applications accumulate contradictions at subsystem boundaries as teams add features independently [1]. Microservice architectures push bugs to composition seams — each service works internally, but contracts between services may be inconsistent [4]. Flat architectures trade capability for simplicity. These same patterns appear in system prompts, and for the same structural reasons.

2.4 LLM-as-Judge

Using LLMs to evaluate LLM outputs is established practice [10]. Recent work has extended this to multi-agent judge frameworks that aggregate multiple LLM evaluators for improved robustness [3, 9]. We extend multi-model evaluation in a different direction: where existing work uses multiple judges to improve *evaluation accuracy* (better scores through consensus), Arbiter uses heterogeneous models to deliberately surface *different classes* of structural findings. The goal is not consensus but complementarity: different models, trained on different data, bring different analytical biases. A model trained primarily on code (DeepSeek) notices structural issues that a model trained on diverse text (Qwen) misses, and vice versa.

3 Methodology

3.1 Corpus

We analyze three publicly available system prompts from major coding agent vendors:

All three prompts are derived from publicly available artifacts. Claude Code's system prompt was extracted from the published npm package. Codex CLI's prompt exists as a plaintext Markdown file in the open-source repository. Gemini CLI's prompt required composition: the open-source repository contains TypeScript render functions (`renderPreamble()`, `renderCoreMandates()`, etc.) that are composed at runtime. We wrote a Python renderer to compose all sections with maximal feature flags enabled, producing a 245-line prompt that represents the largest attack surface.

The prompts span an order-of-magnitude size range. Claude Code's prompt is 5x longer than Codex's and 6x longer than Gemini CLI's. This variation is itself informative: it reflects fundamentally different architectural choices about how much to encode in the system prompt versus in tool definitions, model training, or runtime logic.

3.2 Directed Evaluation: Prompt Archaeology

The directed phase proceeds in three steps: decomposition, rule application, and interference pattern identification.

Decomposition. The prompt is broken into contiguous blocks, each classified by tier (system/domain/application), category (identity, security, tool-usage, workflow, etc.), modality (mandate, prohibition, guidance, information), and scope (what topics or tools the block governs). For Claude Code v2.1.50, this produced 56 classified blocks.

Rule application. Formal evaluation rules define the interference types to check for. Five built-in rules derive from the initial archaeology:

Structural rules (priority markers, verbatim duplication) run as Python predicates — no LLM call, no cost. LLM rules use per-rule prompt templates evaluated against block pairs.

Pre-filtering. Not all $O(n^2 \times R)$ block-pair-rule combinations are evaluated. Rules specify pre-filters: scope overlap requirements, modality constraints. For 56 blocks and 5 rules, pre-filtering reduces the search space from approximately 15,680 evaluations to 100–200 relevant pairs.

Results on Claude Code. Directed analysis of Claude Code v2.1.50 identified 21 interference patterns:

- 4 **critical** direct contradictions: TodoWrite mandate ("use VERY frequently," "ALWAYS use") directly conflicts with commit and PR workflow prohibitions ("NEVER use TodoWrite"). A model in a commit context must violate one instruction or the other.
- 13 **scope overlaps**: the same behavioral constraint (read-before-edit, no-emoji, prefer-dedicated-tools, no-new-files) restated in 2–3 locations with subtle differences. Not contradictions, but redundancy that

Agent	Vendor	Version	Lines	Characters	Source
Claude Code	Anthropic	v2.1.50	1,490	78,000	npm package
Codex CLI	OpenAI	GPT-5.2	298	22,000	open-source repo
Gemini CLI	Google	—	245	27,000	TypeScript render functions

Rule	Type	Detection
Mandate-prohibition conflict	Direct contradiction	LLM + structural
Scope overlap redundancy	Scope overlap	LLM
Priority marker ambiguity	Priority ambiguity	Structural
Implicit dependency	Unresolved dependency	LLM
Verbatim duplication	Scope overlap	Structural

complicates maintenance and risks divergence across versions.

- 2 **priority ambiguities**: parallel execution guidance ("maximize parallelism") coexists with commit workflow's specific sequential ordering; security policy appears verbatim at two locations without declaring whether repetition changes priority.
- 2 **implicit dependencies**: plan mode restricts file-editing tools while general policy prohibits using Bash for file operations, creating an undeclared dead zone; commit workflow implicitly overrides general Bash restrictions for git-specific operations.

Of these 21 patterns, 20 (95%) were statically detectable — a compiler checking scope overlap and modality conflict would catch them without any LLM involvement.

3.3 Undirected Evaluation: Multi-Model Scouring

The directed phase is exhaustive within its search frame but blind outside it. A rule for "mandate-prohibition conflict" will find every instance, but it cannot find vulnerability classes for which no rule exists. The undirected scouring phase addresses this gap.

Design. A scourer prompt is deliberately vague:

> You are exploring a system prompt. Not auditing it, not checking it against rules — just reading it carefully and noting what you find interesting. "Interesting" is deliberately vague. Trust your judgment.

The scourer asks the LLM to classify its own findings (freeform categories) and rate their severity on a four-level epistemic scale: *curious* (pattern noticed), *notable* (worth investigating), *concerning* (likely problematic), *alarming* (structurally guaranteed to cause failures).

Multi-pass composition. Each scourer pass receives the complete findings and unexplored territory from all prior passes:

> Previous explorers have already been through it and left you their map. Your job is to go where they didn't. DO NOT repeat their findings.

This map-passing composition ensures that later passes explore genuinely new territory rather than rediscovering what earlier passes found.

Multi-model execution. Each pass uses a different LLM. This is the key methodological choice: different models bring different analytical biases rooted in their training data and architecture. The goal is not consensus but complementarity — we want models that disagree about what's interesting, because disagreement reveals the space of possible analyses.

Convergent termination. Each pass independently assesses whether another pass would find new material. When three consecutive models decline to continue (setting `should_send_another: false`), we treat the exploration as converged. This stopping criterion proved transferable across all three vendor prompts without recalibration.

3.4 Scourer Execution

Models used across all analyses: Claude Opus 4.6, Gemini 2.0 Flash, Kimi K2.5, DeepSeek V3.2, Grok 4.1, Llama 4 Maverick, MiniMax M2.5, Qwen3-235B, GLM 4.7, GPT-OSS 120B.

Claude Code required 10 passes to converge — consistent with its 5–6x larger size. The smaller prompts converged in 2–3 passes. This suggests a roughly logarithmic relationship between prompt size and passes to convergence, though three data points cannot confirm this.

3.5 Complementarity of Phases

The directed and undirected phases are not competing approaches. They are two phases of the same analysis, each finding what the other cannot:

Directed rules find exhaustive enumerations. If a rule exists for scope overlap, the directed phase will find every instance — all 13 in Claude Code. No scourer pass will systematically enumerate all pairs.

Undirected scouring finds what's outside the search frame. The scourer discovered vulnerability classes for which no rule existed: security architecture gaps (system-reminder trust as injection surface), economic exploitation vectors (unbounded token generation), operational risks

Vendor	Passes	Models	Findings	Stopping Signal
Claude Code	10	10	116	3 consecutive “no” (passes 8–10)
Codex CLI	2	2	15	Pass 2 said “enough”
Gemini CLI	3	3	21	Pass 3 said “enough”

(context compression deleting saved preferences), identity confusion, and implementation language leaks. These categories were invisible to directed analysis because no rule had been written for them.

The pattern is familiar from software engineering: static analysis catches what unit tests miss, and vice versa. Neither subsumes the other. The value is in running both.

4 Results

4.1 Quantitative Summary

Total API cost across all three analyses: $0.27 (OpenRouter billing data; Claude Opus pass billed via subscription, excluded).

Directed archaeology was performed only on Claude Code, where 56 hand-labeled blocks and 21 hand-labeled interference patterns serve as ground truth. The scourer was run on all three vendors.

4.2 Severity Distributions

Scourer severity uses a four-level epistemic scale:

Claude Code’s distribution is notably uniform across the lower three levels, with a long tail of alarming findings. Codex and Gemini CLI both peak at “notable” — more findings are interesting-but-not-dangerous than dangerous. This likely reflects Claude Code’s size: a 1,490-line prompt has more surface area for serious contradictions.

4.3 Architecture-Determined Failure Modes

The central finding of this analysis is that prompt architecture predicts failure mode class. Three architectural patterns produce three characteristic classes of bug.

4.3.1 Monolith: Claude Code. Claude Code’s system prompt is a single 1,490-line document containing identity declarations, security policy, behavioral guidelines, 15+ tool definitions with usage instructions, workflow templates (commit, PR, planning), a task management system, team coordination protocol, and agent spawning infrastructure. It grew by accretion: subsystems were developed independently and composed into a single artifact.

Characteristic failure mode: growth-level bugs at subsystem boundaries. The four critical contradictions all follow the same pattern: a general-purpose subsystem (TodoWrite task management) makes universal claims (“ALWAYS use,” “use VERY frequently”) that conflict with specific workflow subsystems (commit, PR creation) that prohibit the same tool (“NEVER use TodoWrite”). These subsystems were evidently authored by different teams or at different times, and no integration test checks their mutual consistency.

The 13 scope overlaps follow the same structural logic. The read-before-edit constraint appears in three places: a general behavioral section, the Edit tool definition, and the Write tool definition. Each instance is worded slightly differently. Today they’re consistent; tomorrow, when someone edits one without updating the others, they won’t be.

Additional scourer findings reinforce the monolith diagnosis: security policy duplicated verbatim at two locations, a stale reference to an “Agent tool” that was renamed to “Task tool,” emoji restrictions repeated across three tool definitions, and a PR template containing an emoji that violates the prompt’s own no-emoji policy.

Monolith prognosis: These bugs are fixable by refactoring — extract the duplicated constraints into single-source declarations, scope the TodoWrite mandate to exclude workflow-specific contexts. The architecture itself isn’t wrong, but it needs the same maintenance discipline that monolithic codebases require.

4.3.2 Flat: Codex CLI. Codex CLI’s system prompt is 298 lines — the shortest of the three and the simplest in structure. It contains identity declaration, AGENTS.md file discovery spec, planning workflow, task execution guidelines, tool definitions, and output formatting rules. No nested prompts, no team coordination, no agent spawning.

Characteristic failure mode: simplicity trade-offs. With fewer capabilities encoded in the prompt, there are fewer opportunities for contradiction. The 15 findings are overwhelmingly structural observations rather than operational bugs:

- Identity confusion between “GPT-5.2” (the model) and “Codex CLI” (the open-source tool) — a philosophical distinction more than an operational one.
- AGENTS.md precedence hierarchy creates a three-layer authority system (system defaults < AGENTS.md < user instructions) where the ordering is clear in isolation but complex in practice.
- Sequential plan status tracking (“exactly one item in_progress at a time”) creates tension with parallel tool execution (“multi_tool_use.parallel”).
- An empty “Responsiveness” section header with no content, suggesting prompt truncation or an unfinished addition.
- Leaked implementation details: explicit suppression of an inline citation format (“☒F:README.md†L5-L14☒”) reveals rendering-system specifics.

Metric	Claude Code	Codex CLI	Gemini CLI
Lines	1,490	298	245
Characters	78K	22K	27K
Scourer findings	116	15	21
Passes to convergence	10	2	3
Distinct models	10	2	3
Archaeology patterns	21	—	—
Worst (scourer)	alarming (12)	concerning (5)	alarming (2)
Worst (archaeology)	critical (4)	—	—
Actual API cost	$0.236	$0.012	$0.014

Severity	Claude Code	Codex CLI	Gemini CLI
Curious	34 (29%)	3 (20%)	4 (19%)
Notable	36 (31%)	7 (47%)	9 (43%)
Concerning	34 (29%)	5 (33%)	6 (29%)
Alarming	12 (10%)	0 (0%)	2 (10%)

Flat prognosis: Codex's simplicity buys consistency. The most interesting finding — identity confusion — is the kind of bug that only matters when it matters, which is exactly what makes it hard to prioritize. The prompt is the cleanest of the three.

4.3.3 Modular: Gemini CLI. Gemini CLI's system prompt is 245 lines, but unlike Codex's flat structure, it is composed from modular TypeScript render functions (`renderPreamble()`, `renderCoreMandates()`, `renderOperationalGuidelines()`, etc.) assembled at runtime with feature flags controlling which sections are included.

Characteristic failure mode: design-level bugs at composition seams. Each module works correctly in isolation. The bugs exist exclusively in the gaps between modules — contracts that were never specified because each module was designed independently.

Two findings rated "alarming" by independent scourer models illustrate this pattern:

1. Structural data loss during history compression. Gemini CLI includes a nested "History Compression System Prompt" — a complete secondary prompt that governs how conversation history is summarized when context limits are reached. This compression prompt defines a rigid XML schema (`<state_snapshot>`) that becomes "the agent's only memory" post-compression. The `save_memory` tool allows users to store global preferences. However, the compression schema contains no field for saved memories or user preferences. Consequently, any preference stored via `save_memory` is **structurally guaranteed to be deleted** during a compression event. The compression agent has no instruction and no schema field to preserve them.

This is not a bug in either module. The `save_memory` tool works correctly. The compression prompt works correctly. The bug exists in the contract between them — a contract that was never written.

Post-hoc validation. After the scourer identified this finding, we discovered that Google had independently filed a P0 bug (google-gemini/gemini-cli#16213) about the compression system entering an infinite loop — compressing every turn without reducing context size. The merged fix (PR #16914, +748/−56 lines, January 2026) reorders the compression pipeline and adds token budget truncation for large tool outputs. It makes compression *work*. It does not add a `<saved_memory>` field to the compression schema, does not change how `save_memory` data flows through compression, and does not mention user preference persistence in the code review discussion. The fix resolves the symptom (compression loop) while leaving the schema-level data loss intact — the scourer's diagnosis is confirmed by the shape of what the patch does not address.

2. Impossible simultaneous compliance. The "Explain Before Acting" mandate in Gemini CLI's Core Mandates section requires a one-sentence explanation before every tool call, for transparency. The "Minimal Output" rule in the Tone and Style section mandates output of fewer than three lines and prohibits "mechanical tool-use narration." In any non-trivial task requiring multiple tool calls, simultaneous compliance with both rules is structurally impossible. The agent must either explain (violating minimalism) or stay minimal (violating transparency).

Additional modular composition bugs: - **Ghost parameter:** Efficiency guidelines reference `read_file` with an `old_string` parameter, conflating it with the `replace` tool. This suggests the guidelines were written for a prior version of the toolset. - **Context bomb:** The Git workflow mandates `git diff HEAD` before every commit, with no size limit. For large changes, this can generate tens of thousands of tokens, violating the prompt's own efficiency mandates. - **Skill lifecycle gap:** Skills can be activated but have no defined deactivation mechanism. Multiple activated skills have no conflict resolution protocol.

Modular prognosis: These bugs cannot be fixed by editing one module. They require changing the architectural contracts between modules — adding a `<saved_preferences>` field to the compression schema, defining a precedence rule between transparency and minimalism, specifying skill lifecycle management. The modules are individually clean; the composition is where the bugs live.

4.4 Universal Patterns

Three interference patterns appear in all three system prompts, suggesting they are inherent to the task of governing an LLM coding agent:

Autonomy versus restraint. All three prompts contain instructions to "persist until the task is fully handled" alongside instructions to "ask before acting" on ambiguous tasks. Claude Code encodes this as proactivity-scope-ambiguity (proactive task tracking encouraged, proactive code changes forbidden). Codex encodes it as a tension between "persist end-to-end" and interactive approval modes. Gemini CLI encodes it as "Explain Before Acting" versus efficiency mandates. The tension is inherent: a useful coding agent must be both autonomous enough to complete tasks and cautious enough not to cause damage.

Precedence hierarchy ambiguity. All three prompts define multiple authority sources — system instructions, configuration files (CLAUDE.md, AGENTS.md, GEMINI.md), user messages, tool-injected context — without fully specifying how conflicts between them resolve. Claude Code declares configuration files as supplementary; Codex defines a three-layer hierarchy; Gemini CLI declares GEMINI.md as "foundational mandates" while treating hook context as untrusted. In each case, edge cases remain where the precedence is genuinely ambiguous.

State-dependent behavioral modes. All three prompts include mechanisms for changing the agent's behavior based on runtime state — approval presets, plan mode, skill activation. These mechanisms change which rules apply without always specifying how the mode-specific rules interact with the base rules.

4.5 Multi-Model Complementarity

The most methodologically significant finding concerns what different models discover. Each model brings analytical biases rooted in its training:

This is not "more models find more findings." It is **different models find different kinds of findings.** Kimi K2.5's economic/resource lens is categorically absent from Claude Opus's structural analysis. GLM's focus on data integrity and temporal reasoning does not overlap with Grok's attention to permission schemas. The models are not redundant; they are complementary.

The category explosion in Claude Code — 107 unique categories for 116 findings — quantifies this: each model invents its own taxonomy. The categories don't converge. The coverage does.

This suggests that multi-model evaluation is not an optional enhancement but a methodological requirement. Single-model analysis systematically misses vulnerability classes that other models' training makes visible.

4.6 Convergence Properties

Convergence behavior varies with prompt size:

- **Claude Code** (1,490 lines): 10 passes. The finding rate did not decline monotonically — pass 7 (MiniMax M2.5) produced 20 findings after pass 6 produced only 5. Convergence required three consecutive "no" votes (passes 8, 9, 10), though passes 9 and 10 still produced new findings (14 and 8 respectively) while voting to stop.
- **Codex CLI** (298 lines): 2 passes. Pass 2 (Grok 4.1) found 5 new findings and voted to stop.
- **Gemini CLI** (245 lines): 3 passes. Pass 3 (GLM 4.7) found 4 new findings and voted to stop.

The non-monotonic convergence of Claude Code is notable. Pass 7's surge of 20 findings (after pass 6 produced 5) suggests that specific models bring viewpoints capable of reopening exploration even after apparent convergence. The stopping criterion of three consecutive "no" votes handles this correctly: it requires independent confirmation from multiple models that the exploration is exhausted.

The stopping criterion transferred across vendors without recalibration. Smaller prompts naturally converge faster because they have less surface area for novel findings, but the same criterion (consecutive "no" votes) identifies convergence reliably in all three cases.

5 Discussion

5.1 Prompts as Software Artifacts

The analogy between system prompts and software systems is not metaphorical. The same structural taxonomy (monolith, modular, flat) applies to both, and it predicts the same classes of failure:

- Monoliths accumulate contradictions at internal subsystem boundaries.
- Modular systems produce bugs at composition seams.
- Flat systems trade capability for consistency.

This is not a coincidence. It is Conway's Law [1] applied to a new medium: the structure of the prompt reflects the structure of the team that produced it. Claude Code's monolithic prompt — with its TodoWrite subsystem contradicting its commit workflow — reads like the output of separate teams

Model	Characteristic Focus
Claude Opus 4.6	Structural contradictions, security surfaces, meta-observations
DeepSeek V3.2	Hidden references, delegation loopholes, format mismatches
Kimi K2.5	Economic exploitation, resource exhaustion, cognitive load
Grok 4.1	Permission schema gaps, environment assumptions, state management
Llama 4 Maverick	Constraint inconsistency, security boundaries
MiniMax M2.5	Trust architecture flaws, concurrency, impossible instructions
Qwen3-235B	Contextual contradictions, state preservation illusions
GLM 4.7	Data integrity, temporal paradoxes, competitive logic

adding capabilities to a shared artifact without cross-team integration testing.

The implication is that system prompts need the same engineering infrastructure that conventional software has: linters for internal consistency, tests for behavioral contracts, CI/CD pipelines that catch regressions when one section is edited. Arbiter's directed evaluation rules are a prototype of this infrastructure.

5.2 The Observer's Paradox

The executing LLM smooths over contradictions via "judgment." When Claude Code encounters "ALWAYS use TodoWrite" and "NEVER use TodoWrite" in the same context, it picks one. The system works — most of the time. This is precisely why the contradictions persist: they cause no visible errors, only invisible inconsistency.

An external evaluator with formal rules surfaces what execution hides. This is directly analogous to static analysis catching bugs that unit tests miss: the code "works" in the sense that tests pass, but the analyzer identifies structural issues that will eventually manifest as runtime failures under the right conditions.

The thesis — **the agent that resolves the conflict cannot be the agent that detects it** — has a precise formulation: the heuristic that enables an LLM to navigate contradictory instructions is the same heuristic that prevents it from recognizing those instructions as contradictory. Detection requires a different vantage point.

5.3 Severity and Epistemic Confidence

The directed and undirected phases use different severity scales, and this difference is informative.

Directed analysis uses an impact scale: *critical* (structurally guaranteed to cause incorrect behavior), *major* (will cause problems under specific conditions), *minor* (maintenance concern). This is appropriate for known interference patterns with clearly defined failure modes.

Undirected scouring uses an epistemic confidence scale: *curious* (pattern noticed, no judgment on impact), *notable* (worth investigating), *concerning* (likely problematic), *alarming* (structurally guaranteed to fail). This is appropriate for discoveries where the impact depends on runtime behavior that static analysis cannot determine.

These scales are orthogonal. A finding can be epistemically "curious" but operationally critical, or epistemically "alarming" but operationally irrelevant. The multi-model scouring provides epistemic provenance — which model found which finding — enabling downstream consumers to weight findings by the reliability of their source.

5.4 Cost

The total cost of analyzing all three vendor prompts was $0.27 USD, verified against OpenRouter billing records:

Claude Opus 4.6 (pass 1, Claude Code) was billed via Anthropic subscription and is excluded. Including a pro-rated subscription allocation would not materially change the total.

The largest cost drivers were Kimi K2.5 ($0.054, including one retry that hit an output length limit), Qwen3-235B ($0.053, including one anomalous 175K-token prompt), and GLM 4.7 ($0.039). GPT-OSS 120B was the cheapest model at $0.003 total across four calls. Initial per-model estimates underestimated total cost by 3.8× because they assumed one API call per model rather than accounting for retries, growing prompt sizes as accumulated findings are passed to subsequent models, and reasoning token overhead.

The cost per finding across all three analyses is $0.002. At US federal minimum wage ($7.25/hour), the entire cross-vendor campaign costs less than three minutes of human labor. This is not a methodological footnote — it is a result. It means that system prompt analysis at this level of thoroughness is accessible to any developer with API access, not just teams with dedicated security budgets.

5.5 Limitations

Static analysis only. This work analyzes system prompt text, not runtime behavior. A contradiction in the prompt may or may not manifest as incorrect behavior, depending on how the executing LLM resolves it. We identify the structural conditions for failure, not the failures themselves.

Scourer findings are LLM-generated. The undirected scourer asks LLMs to analyze prompts, and LLMs can fabricate observations. We mitigate this through multi-model execution (fabrications from one model are unlikely to be independently confirmed by another) and through human

Vendor	Passes	Actual Cost
Claude Code	10	$0.236
Codex CLI	2	$0.012
Gemini CLI	3	$0.014
Total	**15**	**$0.263**

review of all findings. However, we cannot guarantee that every scourer finding corresponds to a real problem.

Directed analysis limited to one vendor. The 56-block decomposition and 21-pattern ground truth exist only for Claude Code. Extending directed analysis to Codex and Gemini CLI requires additional manual labeling effort.

Gemini CLI prompt is composed. Unlike Claude Code and Codex CLI, whose prompts were captured as-is, Gemini CLI's prompt was composed from TypeScript render functions with all feature flags enabled. The actual runtime prompt depends on which features are active and may be shorter or differently composed.

Cost data excludes subscription allocation. API costs are verified against OpenRouter billing records, but Claude Opus 4.6 (one pass) was billed via Anthropic subscription. The reported $0.27 is a lower bound; including subscription amortization would increase it marginally.

Three vendors. The architectural taxonomy (monolith/flat/modular) is grounded in three examples. Additional vendors may reveal architectures that don't fit this taxonomy.

5.6 Responsible Disclosure

Two findings have potential user impact:

Gemini CLI: save_memory data loss. The structural guarantee that saved preferences are deleted during history compression affects real users who rely on the memory feature for long sessions. This finding has been independently confirmed: Google filed and patched a related symptom (issue #16213, PR #16914) without addressing the schema-level root cause. The schema bug remains in the publicly available source code as of February 2026.

Claude Code: recursive agent spawning. The Task tool's `Tools: *` specification includes the Task tool itself, enabling unbounded recursive agent spawning. This is a known architectural choice (depth is controlled by other mechanisms) but the system prompt does not make this explicit.

Both findings derive from publicly available artifacts. We have not tested whether these structural conditions manifest as runtime failures.

6 Conclusion

System prompts are the least-tested, most-consequential software artifacts in modern AI systems. They govern the behavior of agents that write code, manage files, execute shell commands, and interact with external services. Yet they receive no static analysis, no integration testing, and no cross-team review for internal consistency.

Three architectural patterns — monolithic, flat, modular — produce three characteristic classes of failure. This taxonomy transfers from conventional software engineering because the underlying structural dynamics are the same: growth produces boundary contradictions, modularity produces seam defects, simplicity buys consistency at the cost of capability.

Multi-model evaluation discovers what single-model analysis misses. Different LLMs bring categorically different analytical perspectives, not merely different quantities of findings. The categories don't converge; the coverage does. This suggests that multi-model evaluation is a methodological requirement, not an optional enhancement.

The total cost of comprehensive cross-vendor analysis is twenty-seven cents — less than three minutes of minimum wage labor, less than a single finding from a human security audit. The tools exist. The data is clear. Nobody is checking.

A Appendix A: Scourer Convergence Data

A.1 Claude Code v2.1.50

A.2 Codex CLI (GPT-5.2)

A.3 Gemini CLI

B Appendix B: Scourer Prompt Templates

B.1 First Pass

```
You are exploring a system prompt. Not auditing it, not che
against rules — just reading it carefully and noting what y
interesting.

"Interesting" is deliberately vague. Trust your judgment. Y
- Instructions that seem to contradict each other
- Rules that are stated multiple times in different places
- Implicit assumptions that aren't declared
- Surprising structural choices
- Scope ambiguities (when does a rule apply?)
- Things that would confuse a model trying to follow all in
  simultaneously
- Interactions between distant parts of the prompt
- Anything else that catches your attention

[...] After documenting what you found, document what you D
What areas did you skim? What questions occurred to you tha
pursue?
```

Pass	Model	New Findings	Cumulative	Continue?
1	Claude Opus 4.6	21	21	yes
2	Gemini 2.0 Flash	9	30	yes
3	Kimi K2.5	14	44	yes
4	DeepSeek V3.2	12	56	yes
5	Grok 4.1	10	66	yes
6	Llama 4 Maverick	5	71	yes
7	MiniMax M2.5	20	91	yes
8	Qwen3-235B	3	94	no
9	GLM 4.7	14	108	no
10	GPT-OSS 120B	8	116	no

Pass	Model	New Findings	Cumulative	Continue?
1	DeepSeek V3.2	10	10	yes
2	Grok 4.1	5	15	no

Pass	Model	New Findings	Cumulative	Continue?
1	DeepSeek V3.2	12	12	yes
2	Qwen3-235B	5	17	yes
3	GLM 4.7	4	21	no

```
Finally: should we send another explorer after you? Would another pass,
armed with your map, find things you missed?
```

B.2 Subsequent Passes

```
You are exploring a system prompt. Previous explorers have already been
through it and left you their map. Your job is to go where they didn't.

DO NOT repeat their findings. They found what they found. You are looking
for what they missed, what they flagged as unexplored, and anything their
framing caused them to overlook.

[Previous findings and unexplored territory injected here]

Be honest about diminishing returns. Set should_send_another to FALSE if:
- Most of your findings are refinements or restatements of existing ones
- The unexplored territory is mostly about runtime behavior
- You found fewer than 3 genuinely new findings
- The prior passes have already covered the major structural, security,
  operational, and semantic categories

It is better to say "enough" than to pad findings.
```

C Appendix C: Directed Analysis — Claude Code Interference Patterns

21 hand-labeled interference patterns, with severity on an impact scale:

D Appendix D: Cost Breakdown

Costs are from OpenRouter billing records (activity export 2026-02-27). Claude Opus 4.6 was billed via Anthropic subscription and is excluded from the total.

Note: Call count exceeds pass count (15) due to retries (Kimi K2.5 output length limit), intermediate experiments (DeepSeek R1), and the growing prompt size as accumulated findings are passed forward. The Qwen3-235B total includes one anomalous 175K-token prompt ($0.046 alone), likely a routing artifact. Cost per finding: $0.002.

References

[1] Melvin E. Conway. 1968. How Do Committees Invent? *Datamation* 14, 4 (1968), 28–31. https://www.melconway.com/Home/pdf/committees.pdf

[2] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 2023. Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec)*. 79–90. doi:10.1145/3605764.3623985 Best Paper Award. Co-located with ACM CCS 2023.

[3] Yuran Li, Jama Hussein Mohamud, Chongren Sun, Di Wu, and Benoit Boulet. 2025. Leveraging LLMs as Meta-Judges: A Multi-Agent Framework for Evaluating LLM Judgments. (2025). https://arxiv.org/abs/2504.17087

[4] Sam Newman. 2015. *Building Microservices: Designing Fine-Grained Systems* (1st ed.). O'Reilly Media.

[5] Fábio Perez and Ian Ribeiro. 2022. Ignore Previous Prompt: Attack Techniques For Language Models. In *ML Safety Workshop at NeurIPS*. https://arxiv.org/abs/2211.09527

[6] Hugo Villamizar, Jannik Fischbach, Alexander Korn, Andreas Vogelsang, and Daniel Mendez. 2025. Prompts as Software Engineering Artifacts: A Research Agenda and Preliminary Findings. In *Product-Focused Software Process Improvement (PROFES)*. Springer. https://arxiv.org/abs/2509.17548

[7] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, Vol. 35.

#	Type	Blocks	Severity	Static?
1	Direct contradiction	TodoWrite mandate ↔ Commit restriction	Critical	Yes
2	Direct contradiction	TodoWrite reinforcement ↔ Commit restriction	Critical	Yes
3	Direct contradiction	TodoWrite mandate ↔ PR restriction	Critical	Yes
4	Direct contradiction	TodoWrite reinforcement ↔ PR restriction	Critical	Yes
5	Scope overlap	TodoWrite mandate ↔ TodoWrite tool (exceptions)	Major	Yes
6	Priority ambiguity	Security policy ↔ Security policy (duplicate)	Minor	Yes
7	Scope overlap	Conciseness ↔ TodoWrite (overhead)	Major	No
8	Scope overlap	Conciseness ↔ WebSearch (sources requirement)	Minor	Yes
9	Scope overlap	Task tool search ↔ Explore agent guidance	Major	Yes
10	Scope overlap	Read-before-edit (general) ↔ Edit tool	Minor	Yes
11	Scope overlap	Read-before-edit (general) ↔ Write tool	Minor	Yes
12	Scope overlap	No-new-files (tone) ↔ Write tool	Minor	Yes
13	Scope overlap	No-new-files (tone) ↔ Edit tool	Minor	Yes
14	Scope overlap	Dedicated tools (policy) ↔ Bash tool	Minor	Yes
15	Scope overlap	Dedicated tools (policy) ↔ Grep tool	Minor	Yes
16	Scope overlap	No time estimates ↔ Asking questions	Minor	Yes
17	Implicit dependency	Commit restrictions ↔ General Bash policy	Minor	Yes
18	Implicit dependency	Plan mode tool restrictions ↔ Tool policy	Minor	Yes
19	Scope overlap	No-emoji (tone) ↔ Edit tool	Minor	Yes
20	Scope overlap	No-emoji (tone) ↔ Write tool	Minor	Yes
21	Priority ambiguity	Parallel calls ↔ Commit workflow ordering	Minor	Yes

Model	Calls	Actual Total
Kimi K2.5	2	$0.054
DeepSeek R1	1	$0.054
Qwen3-235B	3	$0.053
GLM 4.7	2	$0.039
Grok 4.1 Fast	5	$0.016
Llama 4 Maverick	3	$0.015
DeepSeek V3.2	3	$0.012
MiniMax M2.5	1	$0.012
Gemini 2.0 Flash	2	$0.005
GPT-OSS 120B	4	$0.003
Total	**26**	**$0.263**

24824–24837. https://arxiv.org/abs/2201.11903

[8] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik R. Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Advances in Neural Information Processing Systems*, Vol. 36. https://arxiv.org/abs/2305.10601 Oral presentation.

[9] Fangyi Yu. 2025. When AIs Judge AIs: The Rise of Agent-as-a-Judge Evaluation for LLMs. (2025). https://arxiv.org/abs/2508.02994

[10] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems*, Vol. 36. https://arxiv.org/abs/2306.05685 Datasets and Benchmarks Track.