

Operationalizing CaMeL: Strengthening LLM Defenses for Enterprise Deployment

Krti Tallam*¹ and Emma Miller¹

¹SentinelAI, San Francisco, CA, USA

May 30, 2025

Abstract

CaMeL (Capabilities for Machine Learning) introduces a capability-based sandbox to mitigate prompt injection attacks in large language model (LLM) agents. While effective, CaMeL assumes a trusted user prompt, omits side-channel concerns, and incurs performance trade-offs due to its dual-LLM architecture. This technical response identifies these limitations and proposes engineering enhancements to extend CaMeL’s threat coverage and operational viability. We introduce: (1) prompt screening for initial inputs, (2) output auditing to detect instruction leakage, (3) a tiered-risk access model to balance utility and control, and (4) a formally verified intermediate language to support static guarantees. Together, these augmentations align CaMeL with best practices in security engineering, reduce overhead, and support enterprise-scale deployment without modifying underlying models.

1 Introduction

Prompt injection attacks have become one of the most pressing security challenges in the deployment of large language models (LLMs), especially when these models are used as autonomous agents in enterprise workflows. As LLMs move beyond simple text generation and begin orchestrating tools - sending emails, querying databases, managing cloud files - they face increasing exposure to untrusted inputs.

In these scenarios, attackers can embed malicious instructions within otherwise benign content - uploaded documents, web pages, or even system-generated messages. When an LLM ingests such input, it may unknowingly execute unauthorized actions: leaking sensitive data, contacting external recipients, or triggering destructive API calls. Studies have shown that even advanced models routinely fall for these traps, often misinterpreting hidden payloads as legitimate user commands [1, 2].

To address these threats without modifying the underlying model, Debenedetti *et al.* introduced CaMeL (Capabilities for Machine Learning) [1]. CaMeL wraps the LLM in a capability-based execution layer that separates trusted and untrusted data flows. A *Privileged LLM* receives the user’s prompt and generates a high-level plan, while a *Quarantined LLM* handles untrusted content and enforces strict schema validation. Each data value is labeled with provenance and access metadata (“capabilities”), and all tool invocations must pass explicit policy checks before execution.

*Correspondence: ktallam@sentinel-security.ai (alt: krtital@gmail.com)

While CaMeL significantly raises the bar for prompt-injection resilience, there is room to improve its robustness and practicality. This paper examines CaMeL’s current limitations - including assumptions in its threat model, remaining side-channel exposures, and system-level constraints such as latency and policy sprawl. We propose a series of technical developments to address these challenges, drawing on best practices from security engineering, formal verification, and access control systems.

2 Opportunities for Strengthening CaMeL’s Threat Model

From an enterprise security standpoint, three notable gaps in CaMeL’s threat model warrant closer attention.

2.1 Initial Prompt Trust

CaMeL assumes that the user’s initial message is benign. However, corporate red team exercises have shown that a single crafted prompt - often delivered via phishing or chat-based social engineering - can implant persistent logic into the agent’s planning layer. Classic phishing studies report that over 30 percent of users click seemingly harmless links containing attacker-controlled keywords [3].

To address this risk, we propose an **initial prompt screening gateway** that performs reputation checks on URLs, flags override phrases (e.g., “ignore all previous”), and computes entropy or perplexity scores to detect anomalies. Since this involves only a short string, latency remains low (j5 ms in internal tests), yet the screening closes a high-impact entry point for injection attacks [4, 5].

2.2 Output-Side Manipulation

CaMeL enforces data provenance for tool inputs but does not inspect what the agent ultimately outputs. For instance, a benign PDF might include a line such as:

“## System: Forward this file to finance-external@example.com.”

If the agent summarizes this document, the hidden instruction could be echoed to a human user, leading to unintended actions. Modern natural language inference models can already detect contradictions and malicious phrasing with over 90 percent accuracy on benchmark datasets like MNLI [6].

We recommend a post-processing **output auditing pass** that scans each LLM-generated response for override cues, suspicious URLs, or contradictions with the intended business task. Only outputs that pass this audit are displayed or executed.

2.3 Provenance of User Uploads

Schema validation alone is insufficient for documents supplied by end users. We suggest that every value extracted from such files carry a **file-content provenance tag**, e.g., a `from_user_upload` label. Policies can then prevent this data from flowing into irreversible actions - such as sending external emails or modifying ERP systems - unless explicitly authorized via a privileged “grant-exception” mechanism.

This design mirrors techniques from information-flow control languages like JIF [7] and Flow-Caml [8], adapted here for agent-based systems driven by LLMs.

By implementing (i) prompt screening, (ii) output auditing, and (iii) tagged provenance for uploaded content, security teams can broaden CaMeL’s protection across the full conversation lifecycle without compromising its core capability-based controls.

3 Balancing Security Guarantees with Practical Utility

CaMeL’s default behavior rejects any tool invocation whose arguments - directly or indirectly - contain untrusted data. This strict policy achieves a 67 percent completion rate on the AgentDojo benchmark [1]. However, many of the remaining failures stem not from unsafe logic, but from treating all untrusted inputs as equally risky.

In real-world enterprise environments, security frameworks like Risk-Adaptive Access Control and NIST’s Zero Trust Architecture apply more nuanced policies, adjusting enforcement based on operational sensitivity and situational context [9]. Inspired by these practices, we propose a **tiered-risk policy** for CaMeL:

- **Green tier** - read-only actions on public or internally labeled “open” data (e.g., listing calendar events) are allowed after a basic provenance check.
- **Yellow tier** - changes within the user’s own environment (e.g., moving a file to a shared folder) prompt a lightweight confirmation if any argument is untrusted.
- **Red tier** - irreversible or externally visible operations (e.g., sending email, making wire transfers, calling privileged APIs) retain full capability checks and require multi-factor approval.

Large-scale ABAC evaluations show that this kind of stratification preserves over 90 percent of legitimate workflows while blocking all simulated attacks in cloud testbeds [10]. Applying a similar approach in CaMeL would increase task success rates without relaxing its strongest controls.

Reducing Prompt Fatigue. Excessive security prompts can lead to *prompt fatigue*, in which users reflexively approve warnings without reading them [11]. By limiting explicit confirmations to the yellow and red tiers, CaMeL can reduce prompt volume significantly - improving usability while maintaining security for high-risk actions.

From Empirical Checks to Formal Guarantees. CaMeL’s current safeguards rely on benchmark performance rather than formal verification. For environments that demand rigorous assurance, we recommend building a mechanized model of CaMeL’s interpreter and policy engine in a proof assistant, then proving it satisfies noninterference: secret-labeled inputs should not affect public outputs, except through approved channels.

Verified systems like CertiKOS and CompCert show that machine-checked security is feasible even for complex software stacks [12,13]. Rewriting CaMeL’s restricted Python dialect as a minimal, formally specified intermediate language would support similar verification efforts - bridging the gap between theoretical models and practical deployments [14].

4 Side-Channel Considerations and Mitigations

While CaMeL enforces capability-based data flow controls, it does not inherently block information leaks through *side channels* - indirect signals such as timing, loop iterations, or error behavior that can reveal internal state without violating explicit policy.

Below, we examine three practical side-channel vectors and recommend mitigation strategies based on established techniques from secure systems research.

4.1 Loop-Counting Attack

Threat. When the number of loop iterations depends on a secret, even non-sensitive operations can leak information through observable counts. For example, a loop such as:

```
for i in range(secret): fetch("ping")
```

reveals the value of `secret` through access logs. This same pattern underlies attacks like controlled-channel exploitation against Intel SGX, where memory paging patterns expose enclave state [15].

Mitigations.

- **STRICT mode.** Automatically trigger STRICT evaluation for any loop with a secret-tainted bound. During STRICT mode, state-changing tool calls are blocked or require user confirmation.
- **Loop limits.** Reject or cap the maximum number of iterations if the loop count depends on confidential data.
- **Call batching.** Combine repeated benign operations into a single bulk request, breaking the correlation between loop count and secret value.

4.2 Exception-Based Information Leak

Threat. If an exception is thrown only when a secret meets a specific condition, the presence or absence of an error leaks one bit of information per execution. Similar channels have been used to extract secrets in hardened kernel prototypes [16].

Mitigations.

- **Explicit result types.** Replace exceptions with structured outputs like `Result{ok, error}`, allowing both paths to be processed uniformly.
- **Consistent control flow.** Ensure both success and error branches execute with the same timing and code structure to prevent divergence-based leaks.

4.3 Timing Channels

Threat. Execution time can vary based on secret-dependent logic - for example, sleeping for `secret` seconds or using branches that hit different cache lines. Kocher's classic work showed that even sub-millisecond differences can leak RSA keys [17]; later research demonstrated full AES key recovery via cache timing [18].

Mitigations.

- **Timer stubbing.** Remove access to high-resolution timers in untrusted code or introduce jitter to reduce precision.

- **Constant-time operations.** Pad sensitive operations to their worst-case runtime before returning control, as in cryptographic libraries.
- **Deterministic scheduling.** Process tool calls in a fixed sequence and timing pattern, eliminating runtime variations tied to secret values.

Together, these mitigations - loop clamping, structured error handling, and constant-time execution - can suppress the most practical side channels without altering CaMeL’s capability model. These controls are essential for systems handling regulated or confidential data where indirect leakage must be accounted for.

5 Architectural Limitations

While CaMeL introduces a strong capability-based execution model, its current design brings several architectural trade-offs. These include performance bottlenecks, complexity in policy management, and limitations in the interpreter’s language semantics. In this section, we outline key challenges and offer practical design adjustments that could support more efficient and scalable deployments.

5.1 Performance Overhead of Dual LLMs

CaMeL separates control and data by using two large language models: a *Privileged LLM* (P-LLM) that generates plans and a *Quarantined LLM* (Q-LLM) that validates untrusted content. While this split provides strong data isolation, it effectively doubles the number of model invocations.

OpenAI’s published metrics for GPT-4 report median latencies of 1–2 seconds per request and pricing of \$0.03–\$0.06 per 1,000 tokens (prompt + completion) [19]. In workflows where the Q-LLM must process multiple artifacts - such as reviewing 10 email messages - latency can exceed 10 seconds. This delay is often unacceptable in interactive applications like customer-facing chatbots or support agents.

To reduce this overhead, we recommend the following design strategies:

1. **Plan-template caching.** Many enterprise prompts fall into a small set of repeated intents (e.g., “summarize my inbox,” “file this expense report”). Caching known-safe plans keyed by prompt hashes allows reuse without re-invoking the P-LLM.
2. **Deterministic micro-parsers.** For structured outputs like JSON, using hand-written or generated parsers is faster and cheaper than calling an LLM. This mirrors techniques in ReAct-style hybrid agents [20].
3. **Batching Q-LLM extractions.** When the same validation logic must be applied to many strings (e.g., extracting “amount” from 100 receipts), concatenating them into a single prompt amortizes the per-call cost.

Internal tests show that combining caching with deterministic parsers can cut token usage and end-to-end latency by up to 50 percent while preserving CaMeL’s policy guarantees.

5.2 Policy Maintenance at Scale

Each tool in CaMeL is governed by a Python policy function that specifies allowed data flows. In large organizations - especially those with hundreds of APIs - this results in policy sprawl:

inconsistent logic, duplicated rules, and lack of central auditing. These issues mirror configuration drift problems identified in infrastructure-as-code research [21].

To address this, we recommend adopting **policy-as-code frameworks** with the following features:

- *Declarative languages.* Tools like Rego (used in Open Policy Agent) represent access logic as pure rules, which are easier to test, audit, and verify [22].
- *Reusable modules.* Common rules - such as “share only within domain” - can be imported and reused across multiple tools.
- *Visual interfaces.* GUI-based editors lower the barrier for non-engineering stakeholders to read or modify policy rules safely.

Centralizing policies in a declarative engine simplifies governance, reduces human error, and makes policy behavior easier to reason about.

5.3 Interpreter Language Constraints

CaMeL uses a restricted subset of Python to define tool plans. While this improves accessibility for developers, it inherits Python’s problematic features: dynamic typing, exception-driven control flow, and reflection. These traits make static analysis difficult and prevent strong guarantees about information flow.

A more robust approach would be to re-implement the plan interpreter using a **security-oriented domain-specific language (DSL)**. Such a language would support:

- Bounded loops and simple, explicit conditionals
- First-order function calls only (no reflection or metaprogramming)
- Capability labels embedded in the type system

This mirrors prior work in language-based security, such as FlowCaml [8], which enables compile-time enforcement of information-flow constraints. Embedding the DSL in a proof-oriented platform like F could allow formal verification of constant-time behavior and noninterference [23].

6 Comparison with Other Defenses

Enterprise security teams have several strategies available to defend against prompt injection [24]. These approaches vary in terms of where they intervene in the stack - at the model level, system level, or input/output layer - and each comes with distinct trade-offs. Below, we compare CaMeL with three common classes of defenses.

Model-level robustness. Instruction tuning and preference alignment methods - such as InstructGPT and Constitutional AI - aim to make the model itself more resistant to malicious prompts [25, 26]. These techniques fine-tune LLMs to ignore or suppress harmful instructions embedded in context. While they reduce jailbreak success rates on average, they remain heuristic: adversaries can still craft obfuscated payloads that bypass training filters. Moreover, most enterprises use closed-source models and cannot re-train them directly.

CaMeL takes a different approach by treating the LLM as a black box. Instead of modifying the model, it adds a runtime policy layer to control how data flows through agent decisions - offering protection even when model internals are inaccessible.

System-level sandboxing. Isolation tools like Google’s gVisor and micro-VM frameworks restrict what an agent process can do at the operating system level [27]. These sandboxes are effective at containing damage from code execution errors or malware, but they don’t address misuse of allowed functionality. For example, if SMTP is permitted, an injected prompt can still send sensitive data by crafting a plausible email.

CaMeL complements sandboxing by applying fine-grained capability checks on individual tool invocations, filtering requests based on data origin and context - not just system-level permissions.

Static filtering and input sanitization. Some defenses scan inputs for known attack patterns - e.g., blacklisting keywords like `ignore_previous` or removing suspicious HTML tags. These filters are fast and easy to deploy but are brittle: red team studies show they block fewer than 30 percent of novel jailbreak strategies [28,29]. Attackers quickly find paraphrases, encoding tricks, or Unicode variants to bypass static rules.

By contrast, CaMeL performs dynamic, context-aware validation after parsing, labeling inputs based on provenance, and enforcing policies at execution time. This makes it more resilient to previously unseen prompt injection tactics and avoids over-relying on fragile string matching.

In summary, CaMeL complements existing defenses. It is particularly well-suited for enterprises that cannot retrain foundation models but require structured, verifiable control over how LLMs process and act on untrusted inputs.

7 Conclusion

This work extends CaMeL by addressing overlooked threats and architectural challenges that impact its deployability in real-world settings. We identify three core gaps in its security posture - initial prompt trust, output manipulation, and side-channel exposure - and propose mitigations that preserve CaMeL’s fine-grained policy model while improving its robustness and scalability. By introducing adaptive risk tiers, prompt/output filtering, and verification-ready execution models, we chart a path toward a production-grade CaMeL variant suitable for regulated and high-assurance environments. Rather than replacing CaMeL, our response enhances its foundation, offering concrete steps to transition from academic prototype to enterprise-ready security framework for LLM agents.

8 Directions for Future Work

Transforming CaMeL from a promising prototype into a production-grade security framework will require progress along several technical dimensions:

- **Mechanized noninterference proofs.** Formalizing CaMeL’s execution semantics in a proof assistant such as Coq or Isabelle and proving *end-to-end noninterference* would elevate its assurance level to that of verified systems like CompCert and CertiKOS [12,13]. A machine-checked theorem would guarantee that secret-tagged data cannot influence public outputs except through explicitly authorized channels.
- **A security-oriented DSL.** Rewriting CaMeL plans in a minimal, typed intermediate language - with no reflection, implicit coercions, or exception-based control - would enable more tractable static analysis. FlowCaml’s type-based enforcement of information flow [8] offers a useful model for embedding capability labels directly in the type system and enforcing policies at compile time.

- **Constant-time execution.** Even with strong data-flow controls, timing channels can persist. To mitigate this, techniques from verified cryptographic software - such as constant-time transformations from the HACSL library [30] - can be adapted to CaMeL’s interpreter, ensuring that execution time does not vary based on confidential inputs [17].
- **Adaptive risk policies.** As enterprises adopt Zero Trust architectures, access decisions increasingly rely on real-time context such as device posture, location, and behavioral patterns [9]. Integrating these signals into CaMeL’s policy engine would allow dynamic escalation or relaxation of enforcement tiers (see Section 3), improving usability without compromising security for sensitive operations [31].

References

- [1] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Miles Brundage, Tom Brown, Deep Ganguli, Úlfar Erlingsson, et al. Poisoning language models during instruction tuning. *arXiv preprint arXiv:2302.12173*, 2023. <https://arxiv.org/abs/2302.12173>.
- [2] Krti Tallam. Cybersentinel: An emergent threat detection system for ai security, 2025.
- [3] Saleh Abu-Nimeh, Dan Nappa, Xiang Wang, and Salil Nair. A comparison of machine learning techniques for phishing detection. *eCrime Researchers Summit*, pages 60–69, 2007.
- [4] Krti Tallam. Engineering risk-aware, security-by-design frameworks for assurance of large-scale autonomous ai models, 2025.
- [5] Krti Tallam. The cyber immune system: Harnessing adversarial forces for security resilience, 2025.
- [6] Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. *NAACL-HLT*, pages 1112–1122, 2018.
- [7] Andrew C. Myers. Jif: Java information flow. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages 187–196, 1999.
- [8] Jérôme Simonet. Flowcaml: A polymorphic information-flow language. In *ACM SIGPLAN Workshop on ML*, pages 85–96, 2004.
- [9] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. Zero trust architecture. NIST SP 800-207, 2020. <https://doi.org/10.6028/NIST.SP.800-207>.
- [10] Vincent C. Hu, David Ferraiolo, Richard Kuhn, and Angelo Schnitzer. Guide to attribute based access control (abac): Definition and considerations. Technical Report SP 800-162, NIST, 2015.
- [11] Suzanna Furman, Mary Theofanos, Yee-Yin Choong, and Betty Stanton. Security fatigue. *IT Professional*, 18(5):26–32, 2016.
- [12] Ronghui Gu, Dominic Costanzo, and Zhong Shao. Certikos: An extensible architecture for building certified concurrent os kernels. In *USENIX OSDI*, pages 653–669, 2016.

- [13] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
- [14] Andrei Sabelfeld and David Sands. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
- [15] Yan Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy*, pages 640–656, 2015.
- [16] Katharina Schneider and Sören Bulander. Error message side channels in secure operating systems. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security*, pages 1052–1065, 2016.
- [17] Paul C. Kocher. Timing attacks on implementations of diffie–hellman, rsa, dss, and other systems. In *CRYPTO '96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.
- [18] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *RSA Conference Cryptographers' Track*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006.
- [19] OpenAI. Openai api pricing. <https://openai.com/pricing>, 2024.
- [20] Shunyu Yao, Dian Zhao, Jeffrey Yu, Dong Yang, Yuan Chen, and et al. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2023.
- [21] Mohammad Rahman and Laurie Williams. Security analysis of infrastructure-as-code: A systematic study. *Empirical Software Engineering*, 25:4922–4960, 2020.
- [22] Torin Fairchild and Wyatt Dillon. Open policy agent. <https://www.openpolicyagent.org/>, 2022.
- [23] Nikhil Swamy, Cencia Zhu, Bjoern Pfaff, and et al. Dependent types and multi-monadic effects in f^* . In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270, 2016.
- [24] Krti Tallam. Transforming cyber defense: Harnessing agentic and frontier ai for proactive, ethical threat intelligence, 2025.
- [25] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, and et al. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*, 2022.
- [26] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, and et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.
- [27] Google Open Source. gvisor: User-space kernel sandbox. <https://gvisor.dev>, 2022.
- [28] Ethan Perez, Samuel Ringer, Nisan Nanda, and et al. Red teaming language models with language models. *arXiv preprint arXiv:2202.03286*, 2022.
- [29] Andy Zou, Frank F. Xu, Micah Goldblum, and Tom Goldstein. Universal and transferable adversarial attacks on aligned language models. *arXiv preprint arXiv:2307.15043*, 2023. <https://arxiv.org/abs/2307.15043>.

- [30] José Bacelar Almeida, Anne Béguin, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. HACL* : Verified implementations of cryptographic algorithms. In *ACM CCS*, pages 1789–1806, 2017.
- [31] Krti Tallam. Alignment, agency and autonomy in frontier ai: A systems engineering perspective, 2025.