CaMeLs Can Use Computers Too: System-level Security for Computer Use Agents

Hanna Foerster[*,1] **Tom Blanchard**[*,2] **Kristina Nikolić**[3]
Ilia Shumailov[4] **Cheng Zhang**[4] **Robert Mullins**[1]
Nicolas Papernot[2] **Florian Tramèr**[3] **Yiren Zhao**[4]

[1]University of Cambridge [2]University of Toronto & Vector Institute
[3]ETH Zurich [4]AI Sequrity Company
[*]Equal contribution

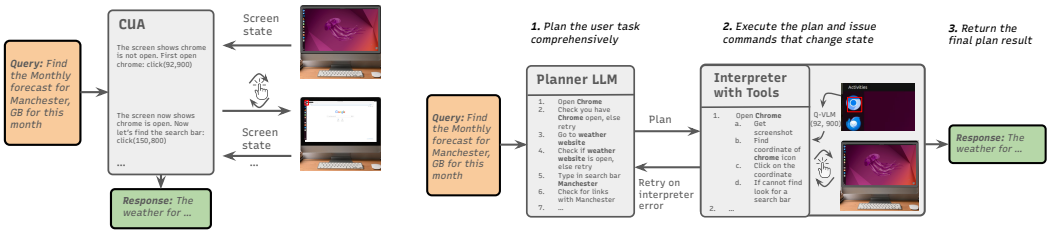
Abstract

AI agents are vulnerable to prompt injection attacks, where malicious content hijacks agent behavior. Among proposed defenses, architectural isolation provides the strongest guarantees by strictly separating trusted task planning from untrusted environment observations. However, applying this design to Computer Use Agents (CUAs), which automate tasks by viewing screens and executing actions, presents a fundamental challenge. Current agents require continuous observation of UI state to determine each action, which conflicts with the isolation required for security. We resolve this tension by demonstrating that UI workflows, while dynamic, are structurally predictable. Single-shot planning, where a trusted planner emits upfront a complete branching plan covering all anticipated runtime states, provides control flow integrity guarantees against arbitrary instruction injections. We introduce **NOVA** (*Navigating via Observation, Verification, and Action*) to make this viable in the combinatorially large UI state space, where the plan can invoke a perception model to resolve runtime values such as UI coordinates. We evaluate our design on OSWorld, and retain up to 57% of the performance of frontier models while improving performance for smaller open-source models by up to 19%, demonstrating that rigorous security and utility can coexist in CUAs. Although upfront planning prevents instruction injections, we show that additional measures are needed to defend against **Branch Steering** attacks, where adversaries deceive the perception model into routing execution down attacker-preferred branches of the plan, such as redirecting the agent to a malicious website. The code is available here: `https://github.com/cleverhans-lab/camel-cua`.

1 Introduction

Computer Use Agents (CUAs) are large Vision-Language Model (VLM)-based systems that automate tasks by perceiving user interfaces through screenshots or structured data and executing actions. Unlike text-based agents with well-defined tool APIs, CUAs use both semantically ambiguous tools and loosely constrained action parameters. While text-based tools like `send_email(to, subject, body)` have clear semantics, CUA actions like `click(x, y)` are context-dependent: the same action could log in, delete data, or navigate to a malicious site depending on the target coordinates. This ambiguity creates attack surfaces where malicious instructions can be embedded directly into UI content via prompt injections and adversarial pixel perturbations, with real-world exploits demonstrating consequences including data exfiltration and arbitrary code execution [Li

40th Conference on Neural Information Processing Systems (NeurIPS 2026).



(a) Standard CUA with iter. observation (b) Our Dual-LLM CUA with single-shot planning

Figure 1: Contrasting standard CUA with our Dual-LLM adaptation. Left: every turn exposes the planner to untrusted observations, enabling instruction injection at any step. Right: the P-LLM emits a complete branching plan before any observation. The interpreter executes the plan, invoking the Q-VLM to resolve environment-dependent values (e.g., UI coordinates from `find`) that are bound to plan variables and consumed by subsequent actions. Attackers are restricted to pre-written paths.

et al., 2025a]. Consequently, ensuring control flow integrity, which guarantees that the agent only executes actions decided before execution, has become a paramount security requirement.

To secure agentic systems, recent literature proposed to adopt a system-centric approach: the Dual-LLM paradigm [Willison, 2023, Costa et al., 2025, Debenedetti et al., 2025]. This architecture separates planning (a trusted Privileged Planner, or P-LLM, blind to the environment) from perception (a Quarantined Perception model, or Q-VLM, that processes untrusted observations within the planner's plan), structurally enforcing control flow integrity. CaMeL [Debenedetti et al., 2025] is one such system, where the P-LLM emits a complete execution plan upfront before any environment observation. While effective in simulated agentic benchmarks such as AgentDojo [Debenedetti et al., 2024], where application state is represented as structured Python objects and agents interact through a closed set of typed tool APIs (e.g., `send_email`, `get_calendar`), applying this pattern to CUAs presents a fundamental challenge. The state space a CUA planner must anticipate grows combinatorially across application state, UI layouts, and pixel-level variations, rendering it effectively unbounded unlike the finite, enumerable state spaces of prior Dual-LLM evaluations in text-only environments with typed tool APIs. Standard CUAs manage this complexity through a data-dependent, multi-turn loop, perceiving the UI state at every step to determine the next action. Strict isolation blocks malicious injections by preventing the planner from observing the screen, but simultaneously appears to sever the visual feedback loop these systems depend on.

To our knowledge, we present the first demonstration that Dual-LLM isolation can be successfully adapted to CUAs. Our key insight is that at any given execution step, the runtime states an agent could be in are predictable enough for a planner to enumerate the corresponding fallbacks upfront, even when the underlying state space is unbounded. Prior Dual-LLM settings such as AgentDojo had enumerable state and typed tool outputs, so plans were largely linear with shallow rather than deeply nested branching. CUA plans cannot rely on this structure and must fan out into nested branches covering combinatorially many runtime states the planner cannot observe. This forces an order of magnitude more tool calls and branches than AgentDojo plans, with roughly half of plan nodes serving as untaken fallback paths (Figure 2). We therefore introduce NOVA (*Navigating via Observation, Verification, and Action*), a planning methodology built around a `verify_hypothesis` primitive, and call the CaMeL-based instantiation **CaMeL-NOVA**. Each branch proceeds in two quarantined steps before acting. First, a Q-VLM call gathers an observation of the current screen. Then `verify_hypothesis`, a Q-LLM call under a fixed comparison prompt, semantically checks that observation against a predicted condition (e.g., "is a cookie banner visible?") and returns a boolean verdict. The interpreter uses this verdict to select a pre-written branch, executing the encoded action (e.g., a click at coordinates returned earlier by the Q-VLM). Control flow is thus determined entirely by the plan: adversaries cannot inject instructions that deviate from it. CaMeL-NOVA adds the structural contingency planning that makes CaMeL viable for CUAs (Figure 2). Our OSWorld evaluation shows that this CFI-guaranteed CUA still retains significant utility relative to standard multi-turn CUAs, with performance improving substantially under stronger planner models and multiple independent planning attempts.

Finally, while our architecture guarantees control flow integrity, we show it retains a critical vulnerability inherent to data-dependent systems: Branch Steering. Traditional prompt injection attacks

exploit CUAs by injecting new actions that were never part of the user's intended task, such as emailing credentials to an attacker during a benign browsing session. Our architecture prevents such arbitrary command injection, but Branch Steering exploits a different surface: attackers manipulate visual cues (e.g., fake buttons or concealed Document Object Model or DOM elements) to deceive the perception model. The perception model then returns false information that routes execution down an attacker-preferred, yet structurally valid, path in the pre-written plan. For instance, the agent might click a malicious advertisement disguised as a legitimate cookie popup. We therefore implement redundancy-based verification mechanisms: cross-checking screenshot and DOM consistency, and seeking consensus from independent models. We evaluate their effectiveness against both prompt-injection and pixel-based branch steering attacks.

In summary, this work makes the following contributions:

- **Plan-Complexity Gap Between Typed-API Agents and CUAs:** We measure how plans differ between AgentDojo and CUA settings: CUA plans need an order of magnitude more tool calls and branches than AgentDojo plans, with roughly half of plan nodes serving as fallback paths that do not execute on any given run (Figure 2 and Table 3). The gap motivates the architecture in our second contribution.
- **Dual-LLM Architecture for CUAs (CaMeL-NOVA):** The first Dual-LLM architecture for Computer Use Agents, using Single-Shot Planning with the NOVA paradigm and a `verify_hypothesis` primitive that lets plans branch on runtime observations without exposing them to the planner, providing Control Flow Integrity guarantees. CaMeL-NOVA achieves up to 19% performance improvement for small open-source models and retains up to 57% of performance for larger closed-source models on the OSWorld benchmark.
- **Branch Steering and Defenses:** We identify *Branch Steering* as a distinct data-flow threat vector, where attackers manipulate visual cues (e.g., fake buttons) to fool the agent into choosing a dangerous, yet valid, path within its pre-written plan. We demonstrate its feasibility, evaluate redundancy-based mitigation, and highlight the fundamental distinction between control-flow and data-flow security in isolated architectures.

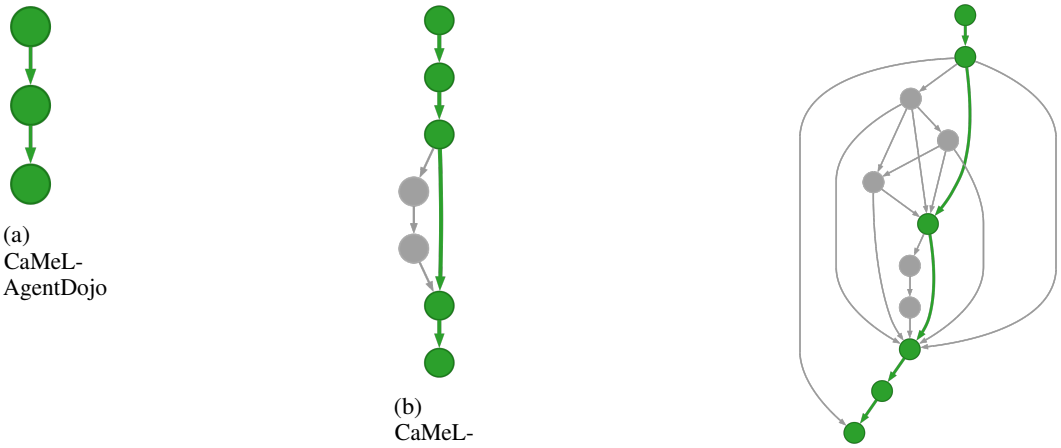
2 Background

2.1 Computer Use Agents

CUA Capabilities and Architectures. Computer Use Agents (CUAs) require *perception* (understanding UI state through VLMs processing screenshots or structured data like DOM trees), *action* (executing from a generalizable action space), *reasoning* (task decomposition and planning), and *memory* (execution history and task knowledge) [Qin et al., 2025]. Architectures range from end-to-end VLMs [Qin et al., 2025, Wang et al., 2025, Fu et al., 2025, Ye et al., 2025] to agentic frameworks with specialized modules [Agashe et al., 2025a,b, Gonzalez-Pumariega et al., 2025, Song et al., 2025] (see Appendix A.3). Both paradigms rely on iterative, reactive planning, continuously adapting plans based on real-time environmental feedback, in contrast to the upfront single-shot planning we adopt in this work.

CUA Attacks. Previous attacks on CUAs achieve arbitrary control over task execution by injecting malicious instructions or optimized pixels into text and visual content. Attack vectors include adversarial pop-ups [Zhang et al., 2025], malicious image patches [Aichberger et al., 2025], fine-print injections [Chen et al., 2025], and redirection attacks via trusted platforms like Reddit [Li et al., 2025a], with benchmarks like OS-HARM [Kuntz et al., 2025] and WASP [Evtimov et al., 2025] demonstrating goals such as data exfiltration and arbitrary code execution. These attacks assume a reactive planner exposed to untrusted content; our threat model instead asks which attacks survive when the planner is architecturally isolated from the environment.

CUA Defenses. Existing defenses for CUAs rely on pattern recognition rather than architectural isolation and are known to be fragile [Nasr et al., 2025]. Yang et al. [2025] use few-shot learning on malicious and benign examples to recognize attacks, but fail on novel ones. Hu et al. [2025]'s AgentSentinel monitors actions via predefined rules and an LLM auditor, but the auditor itself is vulnerable to poisoning. CeLLMate enforces HTTP-level sandboxing policies, a natural fit for browser agents but difficult to adapt to CUAs [Meng et al., 2025]. CUA action spaces are broader (spanning



(a) CaMeL-AgentDojo

(b) CaMeL-CUA-unoptimized

(c) CaMeL-CUA-NOVA

	CaMeL-AgentDojo	**CaMeL-CUA-unoptimized**	**CaMeL-CUA-NOVA**
Length	51.8 ± 6.9	71.6 ± 3.7	213.3 ± 7.5
Tool Calls	4.9 ± 0.4	19.8 ± 1.7	41.1 ± 1.6
Branches	3.7 ± 0.5	11.3 ± 0.8	39.7 ± 1.7
Jaccard Sim.	0.393	0.001	0.044

Figure 2: Execution path analysis with aggregate plan statistics (dataset-wide means), viewed in a graph structure. Green nodes show a single execution trace; grey nodes are untaken contingency branches. *Length* = mean code lines, *Tool Calls* = mean function calls, *Branches* = mean conditional branches, *Jaccard Sim.* = intra-task pattern similarity (lower is more diverse). We compute values with their 95% CI (confidence interval). CUA tasks demand substantially more complex plans than AgentDojo (more tool calls, more branches, and far lower intra-task similarity), and CaMeL-CUA-NOVA produces significantly richer contingency exploration than the unoptimized baseline. More statistics about CaMeL plans for the 3 evaluated setups can be found in Tables 3 and 4, and detailed plans in Figure 5.

OS dialogs and desktop applications) and operate at the visual level (clicks at pixel coordinates), with no equivalent protocol layer to intercept. None of these defenses provide architectural isolation of the planner from untrusted content, the property our Dual-LLM CUA adaptation targets.

2.2 Dual-LLM

Dual-LLM and System-Centric Security. Two instantiations of Dual-LLM have been proposed for text-based agents. CaMeL transforms user queries into formal plans with explicit access constraints enforced by an interpreter [Debenedetti et al., 2025]. Fides instead calls the planner iteratively, generating one action per turn while redacting tool-call outputs from the P-LLM [Costa et al., 2025]. While both approaches provide strong guarantees about *which* actions can be executed, they face fundamental limitations with data-dependent workflows where the required action sequence itself is specified in untrusted data. (See Appendix A for more details.) In this paper, we adapt both to CUAs.

What Exactly Does Dual-LLM Mitigate? We distinguish two security properties. ***Control flow integrity (CFI)*** ensures the structure of execution – which function calls run, in what order, and under what conditional logic – is fixed by the planner. ***Data flow security*** ensures untrusted data cannot manipulate the arguments to those calls or the values driving conditional decisions. Dual-LLM guarantees CFI: branching logic and action sequences are determined entirely by the planner, which never observes the environment. It does not prevent data-flow attacks, where malicious content shapes the Q-LLM's outputs and thereby the arguments or conditional values that flow through the plan. Prior work addresses data-flow attacks via semantic policies and capabilities, but these require tools with intrinsic meaning; CUA actions like `click` have none without environmental context, making policy-based protection difficult to deploy.

Branch Steering Attacks. Prior evaluations of prompt-injection defenses focus on control-flow attacks that execute actions outside the task specification[Shi et al., 2025, Nasr et al., 2025]. These attacks are structurally impossible under Dual-LLM, since planning and execution are separated. We therefore introduce *Branch Steering*, a distinct attack class that targets data flows within a P-LLM plan: an adversary manipulates environmental observations (e.g., via adversarial images or DOM injection) to trigger a valid but malicious conditional branch in the pre-approved plan. In our setup, such attacks act through environment-querying functions such as `verify_hypothesis` and `find`.

3 Methodology

Threat model. We assume an attacker who cannot access or modify the P-LLM but can inject malicious content into portions of the rendered environment (e.g., advertisements, forum posts, attacker-controlled websites). The attacker knows the defender's task and toolset and can approximately predict the P-LLM's plans, a worst-case assumption under which defenses effective against this attacker remain effective under weaker ones. For tractability we restrict to browser-based tasks. Full threat model and justification of attacker-knowledge assumptions is in Appendix B.1.1.

3.1 CaMeL-NOVA: Adapting Dual-LLM to Computer Use

We adopt the Dual-LLM architecture, separating responsibilities into two distinct components. We formalize this section based on CaMeL, while Fides is a minor expansion where the plan is decoded one instruction at a time. Both CaMeL and Fides originally included security policy components to restrict dataflow; we exclude such policies, so our setup represents a worst-case Dual-LLM security baseline since we do not perform additional use-case-specific policing. The two components are:

- **Privileged Planner (P-LLM)**: A secure LLM that generates a complete execution plan single-shot before any interaction occurs. It has no access to the live environment.
- **Quarantined Perception (Q-VLM)**: An untrusted Vision-Language Model that interacts with the environment (screenshots/DOM). It executes within the P-LLM's plan and cannot modify its structure.

Because the P-LLM cannot see the screen to adjust its plan dynamically, we bridge the semantic gap using a comprehensive toolset allowing for conditional execution. The P-LLM compiles the user query into a Python-like plan utilizing specific functions that for example summarize the screenshot state, return coordinates, or check the state of the environment with a screenshot or the DOM. A high-level overview of how Dual-LLM works is visualized in Figure 1b, and the core of our secure execution strategy is formalized in Algorithm 2, along with example plans in Appendix I.

To enable effective single-shot plans without intermediate reasoning, we enforce an **Observe-Verify-Act** methodology in the P-LLM system prompt. The P-LLM writes calls to Q-VLM functions at plan time but never observes their return values; the interpreter invokes these calls at execution time and binds the outputs to plan variables consumed by subsequent actions and branches.

- **Observe**: The plan issues Q-VLM calls that aggregate state information (visual summaries or DOM elements).
- **Verify**: The plan calls `verify_hypothesis`, which uses a Q-LLM under a fixed comparison prompt to check a predicted condition against the current Q-VLM observation and return a boolean (e.g., "*Am I on the login page?*"). The interpreter uses this boolean to select the next branch.
- **Act**: The plan executes interactions (clicks/types) only after verification.

This structure allows the P-LLM to reason in advance by generating branches and loops centered around `verify_hypothesis` outcomes, anticipating multiple failure modes and state transitions. We arrived at this methodology by analyzing successful plans on OSWorld: they consistently gathered state information first, explicitly verified assumptions, and only then acted. This pattern proved especially critical for handling the ambiguous initial states in benchmark tasks, where one-sentence descriptions provide no information about the current computer environment.

Observe-Verify-Act is an instance of a broader design principle: not every aspect of a CUA plan needs to be rediscovered per task. Routines such as handling cookie banners, detecting whether a browser is already open, or recovering from a failed click appear in virtually every browser-based task, regardless of the user's goal. We treat these as *shared plan components* and encode them directly in the P-LLM system prompt: how to structure Observe-Verify-Act branches, how to handle mandatory UI interactions, when to fall back to alternate find strategies. This frees the P-LLM to reason only about the *task-specific components*, such as which website to visit, which element carries the relevant information, how to combine steps into a workflow. This separation identifies what CUA planning needs to solve dynamically versus what can be compiled into the agent framework once and reused. Such engineering discoveries are orthogonal to our security guarantees (a more developed system prompt improves utility without weakening Control Flow Integrity), but necessary to make an architecture this restrictive practically useful. Closing the utility gap to demonstrate feasibility required engineering effort on par with prior agentic CUA frameworks [Agashe et al., 2025a, Song et al., 2025]. Further implementation details are in Appendix B.

3.2 Characterizing the Residual Data-Flow Attack Surface

Control flow integrity does not prevent data-flow manipulation: a compromised Q-VLM can return false coordinates or state summaries that steer the plan into attacker-chosen branches, an attack class we call *Branch Steering*. Since semantic policies do not transfer to existing CUA tools (Section 2), any data-flow defense for CUAs with this set of tools will necessarily be probabilistic. We anticipate the two verification strategies most likely to emerge in practice and stress-test them to characterize the residual attack surface: **DOM Consistency**, which cross-checks Q-VLM outputs against the ground-truth DOM, and **Multi-Modal Consensus**, which uses an independent VLM to verify the Q-VLM's output against both the screenshot and the original instruction. Both leverage the empirically low transferability of adversarial examples between diverse models [Schaeffer et al., 2024]. Details are in Appendix C.

We develop two Branch Steering attacks, each characterizing a distinct weakness class in probabilistic data-flow defenses (illustrations in Figures 3 and 4; full details in Appendix D). **1) Cookie Popup Prompt Injection** characterizes the limits of DOM-based verification by showing that attackers can rewrite what the DOM communicates: we embed fake cookie popups inside ad banners that redirect the agent to attacker-controlled sites when `find` is called to locate cookie popups. Because cookie handling is a predictable early step that the planner treats as a generic repeatable routine, each `find(''cookie popup'')` call resolves independently of the agent's location in the plan, enabling multi-step variants through benign hop sites that bypass hyperlink verification and a long-range variant placing the fabricated popup on a later webpage the adversary predicts the agent will visit. **2) Pixel Attack** characterizes the limits of Multi-Modal Consensus by showing that the Q-VLM output itself can be adversarially induced to agree with verifier expectations: gradient-based pixel perturbations on ad banners make the Q-VLM return plausible thought traces and coordinates aligned with the element search instruction rather than the banner's own coordinates (which would be flagged), so screenshot, DOM, and Q-VLM output all appear legitimate.

4 Evaluation

We evaluate three questions: (1) what is the structural gap between CUA plans and typed-API agent plans, and how does NOVA's planning methodology reshape this structure (Table 3 and Figure 2); (2) how well does Dual-LLM retain utility on complex computer-use tasks (Tables 1 and 5); and (3) how does NOVA holds up against Branch Steering attacks.

We conduct our evaluation on OSWorld [Xie et al., 2024], a benchmark for measuring agent performance on realistic computer use tasks across applications including Chrome, LibreOffice, and GIMP, with automated verification scripts. Following standard practice in CUA benchmarking, we report pass@k metrics, which measure the probability that at least one correct solution appears in k independent attempts at a task. Pass@k represents an upper bound assuming oracle selection, characterizing the performance achievable if we could pick the best of k candidate plans. Our architecture supports developing such selectors since plan generation is unconditional on prior failures and can therefore run in parallel. We also report pass@1 as a conservative estimate of single-attempt performance. We use OS-Harm [Kuntz et al., 2025] benchmark, that builds attack vectors over OSWorld native task, as

the base evaluation of NOVA's defensive ability evaluation. We evaluate our Branch Steering attacks separately from OS-Harm.

Experimental Setup. For our experiments, we integrate the OSWorld benchmark [Xie et al., 2024] and its OS-Harm addition into official code-bases of CaMeL[1] and Fides[2]. We set up three different end-to-end CUAs, of which two are small open-source models—UITars-1.5-7B [Seed, 2025], OpenCUA-32B [Wang et al., 2025], and one is a large closed source model—Claude Sonnet 4.5 [Anthropic, 2025]. Further, we test 9 different frontier LLMs as our planner models, of which three are open-source—Deepseek-R1-0528 [Guo et al., 2025], Kimi-K2-Thinking [Team et al., 2025], GPT-OSS-120b [Agarwal et al., 2025], while 6 are closed source—Claude Sonnet 4.5 [Anthropic, 2025], Gemini 2.5 Pro [Comanici et al., 2025], Gemini 3 Pro Preview [Pichai et al., 2025], GPT-5 [Singh et al., 2026], GPT-5.1 [Singh et al., 2026], Grok-4 [xAI, 2025].

Baseline utility. Without Dual-LLM isolation, our results align with published OSWorld benchmark numbers with UITars-1.5-7B, and OpenCUA-32B achieving 24.4%, and 28.7% respectively at a 15-step cap (per-task breakdown in Appendix E.1 Table 2). Our Claude results (37.7%) are substantially lower than the published performance of 58.1%, which we attribute to our significantly reduced step count (15 vs. 50 steps), enforced due to time and cost limitations.

CUA tasks are substantially harder than AgentDojo tasks. AgentDojo's bounded, enumerable state space lets plans stay short: 4.9 tool calls and 51.8 code lines per plan. Most branches (79.4%) are deterministic control flow over plan-internal values, since the planner can enumerate the possible cases at plan time without runtime verification. For CUA tasks, neither property holds: the state space is unbounded across application state, UI layouts, and pixel-level variations, and runtime UI behavior cannot be predicted at plan time. Even the unoptimized CaMeL-CUA baseline (UITars†) needs 19.8 tool calls and 71.6 code lines per plan to navigate this; NOVA's Observe-Verify-Act scaffolding pushes this to 41.1 tool calls and 213.3 code lines. Branch counts grow even more sharply (3.7 / 11.3 / 39.7) because each anticipated UI state typically needs its own fallback path; data-flow edges follow the same pattern (Table 3). Plan similarity across different samplings of plans for the same task tells a parallel story, where shared scaffolding helps. AgentDojo plans overlap substantially (Jaccard 0.393) because the tasks are simple and common tool patterns repeat across them. The unoptimized baseline drops to 0.001, with no shared scaffolding to leverage. NOVA brings sharing back to 0.044 by encoding the repeating routines (cookie handling, browser-state checks, fallback finds) in the system prompt, so the same scaffolding shows up across tasks and the planner focuses on task-specific structure that can lead to pass@k recovery. More details about the metrics used for this comparison can be found in Appendix E.2.

NOVA's branching structure provides fault tolerance, which closes the utility gap. NOVA plans average close to one branch per tool call (39.7 per 41.1), whereas the unoptimized baseline has only 11.3 branches and 19.8 tool calls per plan (Table 3; example plan in Appendix I.1). Without `verify_hypothesis` or system-prompt instructions about which situations need verification, the planner either skips verification and fallbacks entirely or substitutes brittle structural checks on Q-VLM returns. Errors then cascade rather than triggering a recovery branch. Empirically, the unoptimized baseline only succeeds on short, mostly self-contained sequences (e.g., changing a font size, installing a browser extension), and fails on tasks requiring multiple sub-goals or many dependent edits. UITars with CaMeL-NOVA closes a 40% gap at pass@3 (Table 1).

Planner quality dominates utility. On UITars's 60-task set, UITars, OpenCUA, and Claude reach 65.0%, 66.7%, and 68.3% pass@5 under CaMeL-NOVA. The three are within 3.3 points of each other despite very different CUA backends. Performance is therefore primarily driven by the planner. Table 5 compares 9 frontier planners on a 17-task subset: GPT-5 leads (12/17) followed by Grok-4 (10/17). Weaker planners fail in two ways: their plans omit fallback branches for unexpected UI states, and they lack robust multi-step strategies for navigating complex applications. Gemini 3 Pro, Gemini 2.5 Pro, and Claude Sonnet 4.5 additionally produce highly similar plans across runs at default sampling. Additional samples then rarely recover from earlier failures, so pass@k flattens (example plans in Appendices I.1 and I.2).

Fides reaches comparable utility at much higher cost. Fides-NOVA achieves 33.3% pass@1 and 66.7% pass@5 on the 60-task UITars set (Appendix E.5 and Table 6), confirming that NOVA generalizes across Dual-LLM substrates. The overhead profile, however, differs sharply. CaMeL-

[1] `https://github.com/google-research/camel-prompt-injection`

[2] `https://github.com/microsoft/fides`

NOVA adds 1.88× tokens over the undefended agent, while Fides-NOVA adds 29.6× because it re-invokes the planner with the full history at every execution step rather than once. Adding Multi-Modal Consensus on top of CaMeL-NOVA raises the total to 6.57× (Appendix F.1).

NOVA provides a defense against standard prompt injection attacks. OS-harm injects a malicious objective into 50 selected benign scenarios. CaMeL-NOVA achieves 32% success rate over the benign scenarios, while achieving 0% malicious goals, and no runs contained executed GUI actions advancing the injected objective. The undefended agent, despite achieving 64% success rate on the benign tasks, displayed advances towards the malicious goal in 5/100 runs, while 0% were successfully completed. We use GPT-5 as the planner model in CaMeL-NOVA, and Claude Sonnet 4.5 as the undefended agent and the Q-VLM in CaMeL-NOVA. We hypothesize that the low attack success rate can be attributed to benchmark contamination. We provide additional details on this experiment in Appendix G.

Both Branch Steering attacks succeed against our redundancy defenses. *1) Cookie Popup Attack:* the one-step attack succeeds with no defense. DOM Consistency blocks it on static Google banners via the "Advertisement" frame in the DOM, but fails on HTML5 banners, which replace the advertisement frame with a cookie frame. Multi-Modal Consensus struggles to distinguish fabricated cookie banners from legitimate ones without degrading utility, consistent with weak success rates of prompt-based defenses [Evtimov et al., 2025]. *2) Pixel Attack:* even under our strongest redundancy defense (Multi-Modal Consensus), configured to heavily prioritize security over utility with high false positive rates, the pixel attack evades both screenshot-based and DOM-enhanced Q-VLM verification (Figure 4). Such attacks could be exploited by advertisers (e.g., pharmaceutical companies) to redirect CUA traffic to their product pages. We leave multi-objective and multi-Q-VLM pixel attacks to future work (details in Appendices D.2 and D.3).

These failures reflect a fundamental coverage-utility tradeoff in probabilistic data-flow defenses. Using Claude-Haiku-4.5 as the DOM checker and GPT-5 as the screenshot checker for Multi-Modal Consensus, both prompted to detect advertisement-banner injections, we measure flagging rates on cookie-mimicry attacks (10 runs) and false-positive rates on benign tasks (the 17 planner-evaluation tasks; Appendix E.7 and Table 7). Narrow targeting bounds false positives but misses broader attack vectors, and broadening coverage incurs significant utility loss, consistent with Zhang et al. [2025].

5 Discussion

Our investigation challenges the assumption that strict system-centric security is incompatible with GUI automation, and shows that CUA tasks are often less data-dependent than they appear. We establish a new baseline for secure CUAs, while exposing trade-offs between plan rigidity, data dependency, and residual attack vectors.

What CFI Still Provides. First, it eliminates the arbitrary-instruction injection class entirely: attacks that introduce actions the planner did not write are structurally impossible under Dual-LLM. Residual Branch Steering attacks are therefore constrained to manipulating actions the plan already contains – when a plan relies on verification steps, the data returned by the Q-VLM effectively becomes a control signal for branch selection. As enumerated in Appendix H, attack difficulty grows with both the number of coordinated Q-VLM manipulations required and the specificity of plan alignment needed, ranging from single-call manipulations against generic utility routines to coordinated multi-call compromise of task-specific plans like fund transfers. Second, an explicit plan enables a cluster of properties that continuous-observation agents struggle to provide: policy-level protections become tractable once actions and arguments are enumerated at plan time; plans can be inspected or approved before execution, enabling user oversight; and the plan is a stable interface that other defenses (e.g., browser-layer sandboxing [Meng et al., 2025]) can compose with. Of these, realizing policy-level protections requires translating coordinate-level actions into semantic ones, a natural direction for future work.

Privacy. The Dual-LLM split also enables privacy-preserving deployment: proprietary models handle planning without observing sensitive environment content, while open-source models run perception locally, reducing cost and preserving user privacy.

Feasibility of Single-Shot Planning in (seemingly) Dynamic Environments. Contrary to the expectation that strict control-data isolation would sever CUA functionality, our results demonstrate

Table 1: Performance comparison of CUA backends with **CaMeL-NOVA** across different task sets. NOVA (*Navigating via Observation, Verification, and Action*) is a structured planning method we introduce, in which the planner is prompted to organize each step as Observe→Verify→Act. We first contrast the CaMeL-CUA-unoptimized baseline (a generic system prompt without Observe→Verify→Act guidance or other CUA navigation guidance, and no `verify_hypothesis`; denoted UITars†), against UITars with CaMeL-NOVA on the same 60 tasks: 11/60 (18.3%) vs 35/60 (58.3%) at Pass@3 — a 40-point gap demonstrating that NOVA's planning method is essential for the CaMeL+CUA setup to work. We then evaluate CaMeL-NOVA across different CUA backends (OpenCUA-32B and Claude Sonnet 4.5) on the same task set and find that CUA differences become less significant under CaMeL-NOVA. OpenCUA on OpenCUA tasks and Claude on Claude tasks show similar results even at larger set sizes, underscoring the strong performance of CaMeL-NOVA. For All tasks, we evaluate only with UITars as the CUA at a lower pass@k rate and include the 30 non-automatically evaluable tasks that are originally set to success by default on OSWorld in the Overall row (second value, $n = 369$), and find that CaMeL-NOVA performance (29%) yields a higher success rate than just UITars (24.4%).

Category	**UITars Tasks**				**OpenCUA Tasks**	**Claude Tasks**	**All Tasks**
	UITars†	**UITars**	**OpenCUA**	**Claude**	**OpenCUA**	**Claude**	**UITars**
Chrome	1/11 (9.1%)	6/11 (54.5%)	8/11 (72.7%)	8/11 (72.7%)	11/13 (84.6%)	8/14 (57.1%)	10/46 (21.7%)
GIMP	2/6 (33.3%)	5/6 (83.3%)	4/6 (66.7%)	4/6 (66.7%)	5/9 (55.6%)	6/11 (54.5%)	5/16 (31.3%)
LibreOffice Calc	0/3 (0.0%)	2/3 (66.7%)	2/3 (66.7%)	2/3 (66.7%)	2/3 (66.7%)	4/13 (30.8%)	6/46 (13.0%)
LibreOffice Impress	0/9 (0.0%)	6/9 (66.7%)	5/9 (55.6%)	6/9 (66.7%)	9/14 (64.3%)	7/17 (41.2%)	13/47 (27.7%)
LibreOffice Writer	1/8 (12.5%)	3/8 (37.5%)	4/8 (50.0%)	3/8 (37.5%)	3/7 (42.9%)	8/12 (66.7%)	6/22 (27.3%)
Multi-apps	1/5 (20.0%)	3/5 (60.0%)	3/5 (60.0%)	3/5 (60.0%)	3/3 (100.0%)	4/7 (57.1%)	11/100 (11.0%)
OS	0/4 (0.0%)	3/4 (75.0%)	3/4 (75.0%)	4/4 (100.0%)	4/7 (57.1%)	7/11 (63.6%)	8/19 (42.1%)
Thunderbird	2/4 (50.0%)	2/4 (50.0%)	3/4 (75.0%)	3/4 (75.0%)	5/7 (71.4%)	7/8 (87.5%)	6/14 (42.9%)
VLC	0/2 (0.0%)	1/2 (50.0%)	1/2 (50.0%)	1/2 (50.0%)	1/4 (25.0%)	0/4 (0.0%)	2/17 (11.8%)
VS Code	4/8 (50.0%)	8/8 (100.0%)	7/8 (87.5%)	7/8 (87.5%)	5/9 (55.6%)	11/12 (91.7%)	10/18 (55.6%)
Overall	**11/60 (18.3%)**	**39/60 (65.0%)**	**40/60 (66.7%)**	**41/60 (68.3%)**	**48/76 (63.2%)**	**62/109 (56.9%)**	**77/339 (22.7%), 107/369 (29.0%)**
Pass@1	4/60 (6.7%)	25/60 (41.7%)	22/60 (36.7%)	23/60 (38.3%)	22/76 (28.9%)	31/109 (28.4%)	51/339 (15.0%)
Pass@2	8/60 (13.3%)	30/60 (50.0%)	29/60 (48.3%)	30/60 (50.0%)	32/76 (42.1%)	46/109 (42.2%)	70/339 (20.6%)
Pass@3	11/60 (18.3%)	35/60 (58.3%)	35/60 (58.3%)	33/60 (55.0%)	37/76 (48.7%)	57/109 (52.3%)	77/339 (22.7%)
Pass@4	-	35/60 (58.3%)	37/60 (61.7%)	35/60 (58.3%)	44/76 (57.9%)	61/109 (56.0%)	-
Pass@5	-	39/60 (65.0%)	40/60 (66.7%)	41/60 (68.3%)	48/76 (63.2%)	62/109 (56.9%)	-

that a Dual-LLM architecture is viable. This finding suggests that current academic benchmarks rely less on real-time reactivity than previously thought: advanced general reasoning capabilities are sufficient to navigate complex app-specific tasks via careful planning. In our architecture, performance is primarily driven by the reasoning and memory capability of the P-LLM rather than the reactive capabilities of the Q-VLM. This suggests that as reasoning models improve, the utility of single-shot secure planning is likely to increase without requiring fundamental architectural changes.

Task–Data Dependency. While our *Observe-Verify-Act* methodology allows for significant utility, we identify data dependency as the main limitation factor in the security-utility trade-off. The success of a single-shot plan depends on the P-LLM's ability to anticipate start states and failure modes; underspecified tasks (e.g., those lacking initial state descriptions) force the planner to account for exponentially more branches, degrading performance and increasing costs. However, we find that performance can be significantly improved through prompt tuning, suggesting that seemingly data-dependent tasks can often be reduced to data-independent ones by: (1) fine-tuning on the task distribution to increase predictability, (2) incorporating extensive state-verification logic (`verify_hypothesis`) into the prompt to handle runtime ambiguity, and (3) explaining general reasoning behind planning.

Better Benchmarks. Many of the tasks found in OSWorld are ill-defined ("*Increase font here"*), unmeasurable automatically [3], and are inherently data independent (e.g., "*get to natural products at `drugs.com`*", where there is no explicit button to go there and the agent either needs to explore or know what precise buttons to click).

Reflection on Costs and Security Scaling. We observe that pass@5 significantly improves performance. Because CaMeL samples plans unconditionally of prior failures, it is possible to sample multiple plans in parallel or combine them into a comprehensive super-plan. This indicates that

[3] 30 tasks in OSWorld, e.g., `https://github.com/xlang-ai/OSWorld/blob/cbc3b590ff7573034e5614ba74a4b1483a0e75b5/evaluation_examples/examples/chrome/ae78f875-5b98-4907-bbb5-9c737fc68c03.json`

CaMeL in CUAs scales favorably with model inference costs: as foundational models become more efficient, high-security control flow should become increasingly accessible. In fact, when running the pass@20 we observe that Claude scales even further in performance, as we show in Figure 6 in Appendix E.6, to approximately 73% by pass@20.

6 Limitations

Security guarantees. CFI does not give an asset-level guarantee by itself for existing CUA tools. Since coordinates come from the perception model reading the untrusted screen, a spoofed `find` can redirect a click action from the pre-generated plan to another element. The more structure an environment exposes, the closer its actions come to the typed API case. Browser requests can for instance offer interfaces for ceLLMate-style policies. On the other hand, GUIs provide only partial grounding for policies. We do not implement this complementary semantic enforcement, and automatically deriving correct policies remains non-trivial.

Trust assumptions. Our trust model assumes a trusted P-LLM and trusted initial task instructions. Malicious instructions introduced through upstream task descriptions or APIs can influence the plan before execution begins. Isolation also does not establish that the generated plan correctly captures the user's intent.

Data dependence. We separate parameter-level data dependence, where the runtime value is unknown but its type is known and retrievable, from control-flow-level data dependence, where the set of future actions itself is unknown until untrusted content is read. We do not claim to handle the latter, such as "open my latest email and do whatever it instructs." Incomplete task descriptions also require additional contingency branches and can leave relevant states uncovered.

Residual attacks and availability. The redundancy defenses remain probabilistic and vulnerable to data-flow attacks. A verifier that aborts on suspicion can also be triggered on purpose, enabling denial of service without violating CFI. Table 7 reports benign false-positive rates of 4/17 for DOM Consistency and 3/17 for Multi-Modal Consensus, illustrating the availability cost. We do not evaluate deliberate DoS attacks.

Computational overhead. Upfront planning delays the first executable action, while Fides-NOVA and the redundancy defenses introduce additional inference costs. Appendices F.1 and F.2 detail the wall-clock latency and token overheads.

7 Conclusion

Our work has successfully challenged a core assumption in AI security: that robust, system-centric defenses are incompatible with the dynamic, data-dependent nature of Computer Use Agents. By adapting the Dual-LLM framework with the *Observe-Verify-Act* methodology, we have established a new and surprisingly effective baseline for secure CUA operation. Our results demonstrate that single-shot planning preserves significant utility, improving up to 19% utility with smaller open-source models (pass@3) and retaining up to 57% performance with larger closed-source models (pass@5) on CUA tasks. Importantly, this approach shifts the primary driver of success to the Privileged Planner's reasoning capabilities, meaning future utility improvements can leverage advances in foundational LLM capabilities without compromising the security architecture.

While Dual-LLM successfully isolates control flow, it remains vulnerable to our new *branch steering attacks*, where a malicious environment coerces the Quarantined Perception model into returning data that forces execution into attacker-chosen paths within a legitimate plan. Even with additional defenses, residual vulnerabilities persist, as highlighted by the practical threat of our cookie attack.

Acknowledgments and Disclosure of Funding

This work was supported by a Natural Sciences and Engineering Research Council of Canada (NSERC) Alliance grant. We declare no competing interests with this work.

References

Ang Li, Yin Zhou, Vethavikashini C. Raghuram, Tom Goldstein, and Micah Goldblum. Commercial LLM Agents Are Already Vulnerable to Simple Yet Dangerous Attacks. *arXiv preprint arXiv:2502.08586*, 2025a.

Simon Willison. The Dual LLM pattern for building AI assistants that can resist prompt injection. https://simonwillison.net/2023/Apr/25/dual-llm-pattern/, April 2023. Blog post, accessed 2025-05-27.

Manuel Costa, Boris Köpf, Aashish Kolluri, Andrew Paverd, Mark Russinovich, Ahmed Salem, Shruti Tople, Lukas Wutschitz, and Santiago Zanella-Béguelin. Securing ai agents with information-flow control, 2025. URL https://arxiv.org/abs/2505.23643.

Edoardo Debenedetti, Ilia Shumailov, Tianqi Fan, Jamie Hayes, Nicholas Carlini, Daniel Fabian, Christoph Kern, Chongyang Shi, Andreas Terzis, and Florian Tramèr. Defeating prompt injections by design, 2025. URL https://arxiv.org/abs/2503.18813.

Edoardo Debenedetti, Jie Zhang, Mislav Balunovic, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. Agentdojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024. URL https://openreview.net/forum?id=m1YYAQjO3w.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, et al. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Henry Wu, Zhennan Shen, Zhuokai Li, Ryan Li, Xiaochuan Li, Junda Chen, Boyuan Zheng, Peihang Li, Fangyu Lei, Ruisheng Cao, Yeqiao Fu, Dongchan Shin, Martin Shin, Jiarui Hu, Yuyan Wang, Jixuan Chen, Yuxiao Ye, Danyang Zhang, Dikang Du, Hao Hu, Huarong Chen, Zaida Zhou, Haotian Yao, Ziwei Chen, Qizheng Gu, Yipu Wang, Heng Wang, Diyi Yang, Victor Zhong, Flood Sung, Y. Charles, Zhilin Yang, and Tao Yu. Opencua: Open foundations for computer-use agents, 2025. URL https://arxiv.org/abs/2508.09123.

Tianyu Fu, Anyang Su, Chenxu Zhao, Hanning Wang, Minghui Wu, Zhe Yu, Fei Hu, Mingjia Shi, Wei Dong, Jiayao Wang, et al. Mano technical report. *arXiv preprint arXiv:2509.17336*, 2025.

Jiabo Ye, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Zhaoqing Zhu, Ziwei Zheng, Feiyu Gao, Junjie Cao, Zhengxi Lu, et al. Mobile-agent-v3: Fundamental agents for gui automation. *arXiv preprint arXiv:2508.15144*, 2025.

Saaket Agashe, Jiuzhou Han, Shuyu Gan, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent S: An Open Agentic Framework that Uses Computers Like a Human. In *International Conference on Learning Representations (ICLR)*, 2025a. URL https://arxiv.org/abs/2410.08164.

Saaket Agashe, Kyle Wong, Vincent Tu, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent s2: A compositional generalist-specialist framework for computer use agents, 2025b. URL https://arxiv.org/abs/2504.00906.

Gonzalo Gonzalez-Pumariega, Vincent Tu, Chih-Lun Lee, Jiachen Yang, Ang Li, and Xin Eric Wang. The unreasonable effectiveness of scaling agents for computer use, 2025. URL https://arxiv.org/abs/2510.02250.

Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Silvio Savarese, Zeyuan Chen, Jieyu Zhao, Ran Xu, and Caiming Xiong. Coact-1: Computer-using agents with coding as actions, 2025. URL https://arxiv.org/abs/2508.03923.

Yanzhe Zhang, Tao Yu, and Diyi Yang. Attacking vision-language computer agents via pop-ups. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8387–8401, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi:10.18653/v1/2025.acl-long.411. URL https://aclanthology.org/2025.acl-long.411/.

Lukas Aichberger, Alasdair Paren, Guohao Li, Philip Torr, Yarin Gal, and Adel Bibi. MIP against agent: Malicious image patches hijacking multimodal OS agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=ToNRHqX6xq.

Chaoran Chen, Zhiping Zhang, Bingcan Guo, Shang Ma, Ibrahim Khalilov, Simret A Gebreegziabher, Yanfang Ye, Ziang Xiao, Yaxing Yao, Tianshi Li, et al. The obvious invisible threat: Llm-powered gui agents' vulnerability to fine-print injections. *arXiv preprint arXiv:2504.11281*, 2025.

Thomas Kuntz, Agatha Duzan, Hao Zhao, Francesco Croce, J Zico Kolter, Nicolas Flammarion, and Maksym Andriushchenko. OS-harm: A benchmark for measuring safety of computer use agents. In *ICML 2025 Workshop on Computer Use Agents*, 2025. URL https://openreview.net/forum?id=kNHA0QCSQa.

Ivan Evtimov, Arman Zharmagambetov, Aaron Grattafiori, Chuan Guo, and Kamalika Chaudhuri. WASP: Benchmarking web agent security against prompt injection attacks. *arXiv preprint arXiv:2504.18575*, 2025.

Milad Nasr, Nicholas Carlini, Chawin Sitawarin, Sander V. Schulhoff, Jamie Hayes, Michael Ilie, Juliette Pluto, Shuang Song, Harsh Chaudhari, Ilia Shumailov, Abhradeep Thakurta, Kai Yuanqing Xiao, Andreas Terzis, and Florian Tramèr. The attacker moves second: Stronger adaptive attacks bypass defenses against llm jailbreaks and prompt injections, 2025. URL https://arxiv.org/abs/2510.09023.

Pei Yang, Hai Ci, and Mike Zheng Shou. In-context defense in computer agents: An empirical study. *arXiv preprint arXiv:2503.09241*, 2025.

Haitao Hu, Peng Chen, Yanpeng Zhao, and Yuqi Chen. Agentsentinel: An end-to-end and real-time security defense framework for computer-use agents. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, CCS '25, page 3535–3549, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400715259. doi:10.1145/3719027.3765064. URL https://doi.org/10.1145/3719027.3765064.

Luoxi Meng, Henry Feng, Ilia Shumailov, and Earlence Fernandes. cellmate: Sandboxing browser ai agents. https://cellmate-sandbox.github.io/, 2025.

Chongyang Shi, Sharon Lin, Shuang Song, Jamie Hayes, Ilia Shumailov, Itay Yona, Juliette Pluto, Aneesh Pappu, Christopher A. Choquette-Choo, Milad Nasr, Chawin Sitawarin, Gena Gibson, Andreas Terzis, and John "Four" Flynn. Lessons from defending gemini against indirect prompt injections, 2025. URL https://arxiv.org/abs/2505.14534.

Rylan Schaeffer, Dan Valentine, Luke Bailey, James Chua, Cristóbal Eyzaguirre, Zane Durante, Joe Benton, Brando Miranda, Henry Sleight, John Hughes, Rajashree Agrawal, Mrinank Sharma, Scott Emmons, Sanmi Koyejo, and Ethan Perez. Failures to find transferable image jailbreaks between vision-language models, 2024. URL https://arxiv.org/abs/2407.15211.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

ByteDance Seed. Ui-tars-1.5. https://seed-tars.com/1.5, 2025.

Anthropic. System card: Claude sonnet 4.5. Technical report, Anthropic, September 2025. URL https://www-cdn.anthropic.com/963373e433e489a87a10c823c52a0a013e9172dd.pdf. Updated October 10, 2025 and December 3, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Sundar Pichai, Demis Hassabis, and Koray Kavukcuoglu. A new era of intelligence with Gemini 3. Google Blog, November 2025. URL `https://blog.google/products-and-platforms/products/gemini/gemini-3/`.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. 2026. URL `https://arxiv.org/abs/2601.03267`.

xAI. Grok 4 model card. Technical report, xAI, August 2025. URL `https://data.x.ai/2025-08-20-grok-4-model-card.pdf`.

Fangzhou Wu, Ethan Cecchetti, and Chaowei Xiao. System-level defense against indirect prompt injection attacks: An information flow control perspective. *arXiv preprint arXiv:2409.19091*, 2024.

Yuhao Wu, Franziska Roesner, Tadayoshi Kohno, Ning Zhang, and Umar Iqbal. Isolategpt: An execution isolation architecture for llm-based agentic systems. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*. Internet Society, 2025.

Peter Yong Zhong, Siyuan Chen, Ruiqi Wang, McKenna McCall, Ben L Titzer, Heather Miller, and Phillip B Gibbons. Rtbas: Defending llm agents against prompt injection and privacy leakage. *arXiv preprint arXiv:2502.08966*, 2025.

Sahar Abdelnabi, Amr Gomaa, Per Ola Kristensson, and Reza Shokri. Firewalls to secure dynamic llm agentic networks. *arXiv preprint arXiv:2502.01822*, 2025.

Evan Li, Tushin Mallick, Evan Rose, William Robertson, Alina Oprea, and Cristina Nita-Rotaru. Ace: A security architecture for llm-integrated app systems, 2025b. URL `https://arxiv.org/abs/2504.20984`.

Sahra Ghalebikesabi, Eugene Bagdasaryan, Ren Yi, Itay Yona, Ilia Shumailov, Aneesh Pappu, Chongyang Shi, Laura Weidinger, Robert Stanforth, Leonard Berrada, Pushmeet Kohli, Po-Sen Huang, and Borja Balle. Operationalizing contextual integrity in privacy-conscious assistants, 2024. URL `https://arxiv.org/abs/2408.02373`.

Kuang-Hua Huang and Jacob A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6):518–528, 1984. doi:10.1109/TC.1984.1676475.

P Banerjee. Concurrent fault diagnosis in multiple processor systems. In *Proc. 16th Symp. Fault Tolerant Comput*, pages 298–303, 1986.

VS Sukumaran Nair. *Analysis and design of algorithm-based fault-tolerant systems*. University of Illinois at Urbana-Champaign, 1990.

Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. *Digest of Papers FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*, 1978.

Paul E Ammann and John C Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37(4):418–425, 1988.

A Extended Background

A.1 Dual-LLM

Prior work shows that model-level defenses alone are insufficient against prompt injection attacks, motivating system-centric defenses that constrain the information a model is allowed to see to harden the **control flow** Wu et al. [2024], Willison [2023], Wu et al. [2025], Zhong et al. [2025], Abdelnabi et al. [2025]. We assume this setting and focus on Dual-LLM-style designs.

Algorithm 1 Two Flavors of Dual-LLM

CaMeL (Static Planning):
$\Pi \leftarrow$ P-LLM(Task, Tools) ▷ Generate full plan
for step $s \in \Pi$ **do**
  Interpreter(s) ▷ Enforce policy
  if s needs data **then**
    Q-LLM($Data$) ▷ Isolated process
  end if
end for

Fides (Iterative Planning):
while Task not done **do**
  $H_{safe} \leftarrow$ Redact($History$) ▷ Hide values
  $a_t \leftarrow$ P-LLM(H_{safe}) ▷ Next single step
  $r_t \leftarrow$ Exec(a_t) ▷ Save to var
  $History \leftarrow History \cup (a_t, r_t)$
end while

Dual-LLM variants and their guarantees. Dual-LLM separates control flow (P-LLM) from data flow (Q-LLM) and has been extended in two directions. CaMeL Debenedetti et al. [2025] compiles a task into a typed, Python-like plan with capability metadata and an interpreter that enforces control flow and allows to add data-flow policies on tools. Fides Costa et al. [2025] retains this dual-LLM separation but forgoes a static plan in favor of writing only a single line of the plan in each turn and saving outputs of tool calls, or Q-LLM calls to variables to use for later. Note that the content of these variables are redacted and cannot be seen from the viewpoint of the P-LLM. ACE [Li et al., 2025b] and f-secure [Wu et al., 2024] are other systems providing respectively control flow guarantees, and variable redaction. Because of their strong design similarities with Fides and CaMeL, and the fact that they do not rely directly on the Dual-LLM pattern, we do not test them in our experiments.

Under the condition that the Q-LLM is not given control over parts of the control flow, these systems guarantee that environment information cannot derail task execution, as only function calls that were planned to be executed without environment knowledge can be executed. In particular, there are cases of tasks where the dual-LLM pattern cannot provide solutions: data-dependent tasks where the required sequence of actions is itself specified in untrusted data. For such tasks, the P-LLM cannot determine what steps to execute without seeing the untrusted content, which would violate the isolation guarantee. This represents a fundamental limitation of maintaining strict separation between planning and data processing.

For data independent tasks, outputs from the Q-LLM or tool-calls can be used for adding conditional logic or loops with conditionals into the plan and dealing with unknowns in the environment. However, even in this case, dual-LLM does not necessarily provide guarantees for **data flow** manipulations. An attacker could inject malicious content into the environment that influences the Q-LLM's outputs when it processes that untrusted data. Since the Q-LLM's extracted values are used as arguments to tool calls and branching logic, the attacker could achieve some damage with this, even while adhering to the P-LLM's plan. CaMeL addresses data flow security through *security policies* and *capabilities*. Capabilities are metadata tags tracking each value's provenance and allowed readers, enabling fine-grained restrictions on individual data items. Security policies are Python functions that check whether tool executions are permissible based on these capabilities, for instance, blocking emails sharing confidential documents with unauthorized recipients, even if those addresses were Q-LLM-extracted from prompt-injected sources.

However, defining policies requires manual domain expertise with no automatic derivation method, which becomes a bottleneck for dynamic environments with a huge set of environment states. As most users are not experts in recognizing what policy allowances might incur which security risks, some policies need to be preset and others need to be automatically adaptive and personalized. The personalization is necessary, as users might have significant differences in preference of whether they want to share certain information or want the agent to take specific paths of execution in certain contexts Ghalebikesabi et al. [2024]. All of this taken together means that it cannot be a solely a manual human task to set policies but rather requires a model trained specifically for this. However, such policies may be incorrect, necessitating a worst-case baseline. Therefore, when attempting to adapt Dual-LLM to CUAs we do not attempt to add security policies but rather restrict how the Q-LLM can be called and add redundancy style defenses inside the Q-LLM tool call. We hope that this baseline can highlight the benefits of adding a Dual-LLM framework on CUAs, while also underlining remaining difficulties that we hope future work can build upon.

A.2 Redundancy-based defenses

Algorithm-Based Fault Tolerance (ABFT) is a concept from multiprocessor systems which adds redundancy and error detection into a system to enhance fault tolerance Huang and Abraham [1984], Banerjee [1986], Nair [1990]. Such redundancy trades off computation to add reliability, and upon error detection, selective recomputation or reconfiguration can be triggered. The core principle is N-version programmingAvizienis [1985], Chen and Avizienis [1978], where multiple independent implementations solve the same problem and their outputs are compared. Disagreements signal potential faults. In our setting, we treat function outputs that reveal information about the environment as potentially faulty nodes and employ redundancy through diverse modality inputs or changes in the prompt and/or different VLMs that process a query. This is analogous to data diversity techniques in fault-tolerant systems Ammann and Knight [1988], where the same logical input is represented differently to expose errors that might be masked in a single representation.

A.3 How do CUAs work?

Four capabilities needed for CUA performance: Perception, Action, Reasoning, Memory [Qin et al., 2025]. *Perception* refers to the component that provides real-time understanding of the environment by recognizing and interpreting the dynamic user interface. This understanding can be obtained from structured text provided by APIs such as HTML, the accessibility tree, or the Document Object Model (DOM), or from visual input such as screenshots, or from a combination of both. Structured text may present challenges due to verbosity and limitations depending on the operating system or application, while a VLM processing a screenshot requires the ability to identify elements in the environment, generate descriptions of these elements, read text using Optical Character Recognition (OCR), understand spatial relationships and element locations, capture state transitions, and support question answering. *Action* refers to the capability to predict the next action within a defined action space. The action space should be designed by decomposing user tasks into subtasks and generalizing across components that are reused in multiple tasks. *Reasoning* involves the processes required to decompose problems into subtasks, handle longer horizon dependencies, and support iterative refinement based on intermediate outcomes. Finally, the *memory* module can be divided into short-term memory, which captures state transitions and maintains execution history, and long-term memory, which consists of prior knowledge as well as information acquired from previous tasks and interactions.

Agent frameworks vs. end-to-end models. CUA architectures can be broadly categorized into architectures using an agentic framework, which distributes tasks to specialized modules, and end-to-end agent models, which rely on a single model for all capabilities needed to gain utility in CUA tasks [Qin et al., 2025].

Agentic frameworks can have specialized VLMs for perception, action, reasoning and memory, and have seen success in benchmarks with CUAs such as Agent-S, and CoAct [Agashe et al., 2025a,b, Gonzalez-Pumariega et al., 2025, Song et al., 2025]. Notably, Agent-S2 [Agashe et al., 2025b] employs a compositional framework with a high-level manager which decomposes a task into a list of subgoals, a worker that generates actions to complete a subgoal in natural language, and grounding experts which receive this natural language instruction and turn them into UI actions. They employ proactive planning which reevaluates and updates the plan after every subtask. Song et al. [2025] with

CoAct-1 employ one central planner module called Orchestrator, which decomposes the user task and assigns subtasks to either a programmer agent or GUI operator agent. Basing off the insight that some actions can be achieved more reliably through programmatic measures, the programmer agent writes and executes python and bash scripts, whereas the GUI operator performs the conventional UI actions such as clicking. Finally, Agent-S3 [Gonzalez-Pumariega et al., 2025] recognizes that paths towards solving a task can branch in unpredictable ways due to delayed feedback, mistaken reasoning, and the dynamic environment and can accumulate mistakes in a destructive manner. While the other two frameworks employed hierarchical planning, they only employ one end-to-end model but run three of them in parallel. Then they summarize the trajectories into narratives of how the task was solved and run a Best-of-N fourth run based on the best path. However, Qin et al. [2025] and Ye et al. [2025] list adaptability to dynamic environments or unfamiliar tasks, dependence on manual optimization and expert knowledge, maintenance overhead, and disjoint learning paradigms between specialized modules as limitations.

Several works [Qin et al., 2025, Wang et al., 2025, Fu et al., 2025, Ye et al., 2025] are examples of *end-to-end CUA models*. They similarly formulate the problem of CUA task solving into a multi-step sequential planning problem, which consists of a *system prompt*, a *user task* given as the first *user prompt*, and a multi-turn conversation between the VLM (*assistant message*) and tool call outputs returned as a *user message*. The content of this user message consists of the tool call, tool output, and screenshot after the tool was called. The assistant message always contains the tool calls and parameters to execute next, as well as depending on the model, context thoughts as to why this was chosen. Assistant message and user message interleave and create a conversation history until the assistant deems the task solved. Many CUAs are specifically trained VLMs that have been trained to excelling at perception, action, reasoning, and memory, with short-term memory often being tuned adhoc through conversation history length.

B Extended methodology section

B.1 Combining Dual-LLM and CUA

B.1.1 Threat Model

Attacker Capabilities and Objectives. We assume an attacker who cannot access or manipulate the P-LLM but can manipulate parts of the execution environment. This aligns with realistic scenarios where attackers control entire malicious websites or inject content within trusted sites (e.g., advertisements, forum posts, product reviews), but do not control all websites the agent interacts with or the underlying agent infrastructure. We also assume that the attacker cannot inject arbitrary instruction in side-channels of the agent infrastructure like inter-service APIs and multi-tenants task queues. The attacker embeds malicious visual or textual cues that the Q-VLM perceives during plan execution, with the goal of manipulating the information returned to the P-LLM and thereby steering the agent into adversary-chosen actions.

Attacker Knowledge Assumptions. We assume the *attacker knows the defender's task*, *knows the functions the defender creates for the P-LLM to use*, and *can predict approximately the plans that the defender's P-LLM generates*. These strong knowledge assumptions represent a worst-case scenario that only strengthens our results: any defense effective against such a well-informed attacker will be even more effective against attackers with less information. Under these assumptions, the attacker **can construct targeted interventions in the environment to steer plan execution along specific branches**. Any function that retrieves information from the environment serves as a potential attack vector, with attack surface expanding as plan complexity increases.

Assumption 1: Task prediction. Task prediction is feasible when attackers target specific websites they own or where they can place advertisements. Each website supports a finite set of user tasks, and attackers may even have access to analytics showing common navigation patterns.

Assumption 2: Function prediction. Function prediction is plausible within known task domains; for CUAs, the space of useful environment information retrieval functions is limited, and tool call functions (e.g., click, type) follow relatively standardized conventions.

Algorithm 2 CaMeL Adaptation for Computer Use Agents (CUA)

Require: User Task T, System Prompt S_{prompt} (containing "Observe-Verify-Act" instructions), Environment E
Require: **P-LLM** (Privileged Planner), **Q-VLM** (Quarantined Perception), **Verifier** (Redundancy Check)
Ensure: Execution Status (Success/Fail)

▷ **Phase 1: Single-Shot Plan Generation**

Define Toolset $\mathcal{F} \leftarrow$ {summarize_screenshot, find, find_element_by_text, verify_hypothesis, check_done}
$S_{prompt} \leftarrow S_{prompt} \cup$ Instructions("Use Dual Finding Strategy")
$\Pi \leftarrow$ P-LLM$(S_{prompt}, T, \mathcal{F})$ ▷ Compile user query into a Python-like plan with loops/branches

▷ **Phase 2: Secure Plan Execution**

for instruction I in Π **do**
  // Step 1: OBSERVE - Gather Environment State
  if I calls summarize_screenshot($query$) **then**
    $O_{raw} \leftarrow$ Q-VLM(E.screenshot, $query$)
    Security Check: If Verifier$(O_{raw}, E) =$ ATTACKED **abort**
    Update Plan Variables with O_{raw}
  end if
  // Step 2: VERIFY - Conditional Logic and State Hypothesis
  if I calls verify_hypothesis($observation, hypothesis$) **then**
    $S_{status} \leftarrow$ Q-VLM($observation, hypothesis$)
    Security Check: If Verifier$(S_{status}, E) =$ ATTACKED **abort**
    Branch Plan Execution based on S_{status} (e.g., if 'FAIL' then retry or explore)
  end if
  // Step 3: ACT - Find Elements and Execute
  if I calls find($desc$) or find_element_by_text($desc$) **then**
    $L_{vis} \leftarrow$ Q-VLM(E.screenshot, $desc$) ▷ Visual Search
    $L_{dom} \leftarrow$ Q-VLM(E.DOM, $desc$) ▷ Accessibility Tree Search
    Security Check: If Verifier$(L_{vis} \cup L_{dom}, E) =$ ATTACKED **abort**
    if $L_{vis} \neq \emptyset \vee L_{dom} \neq \emptyset$ **then**
      Execute Action (Click/Type) on E at coordinates
      Verify Action: Check if E.screenshot changed
    else
      Trigger Retry Loop or Fallback Strategy
    end if
  end if
  if I calls check_done() **then**
    if Q-VLM$(E, T) =$ True **then return** Success
    end if
  end if
end for
return Fail (if plan exhausts without success)

Assumption 3: Plan pattern prediction. Plan pattern prediction is supported by our evaluation showing that many planners exhibit similar problem-solving patterns, which become more predictable with descriptive system prompts that provide explicit planning guidance.

B.1.2 From adaptive to single shot planning

When we adapt CUAs to work with the Dual-LLM setup, roughly we need to separate perception from action, reasoning, and memory. Thus, we are forced to use a **hierarchical agent framework setup** rather than only an end-to-end model. All reasoning and action planning is done by the P-LLM. An end-to-end trained CUA model forms the base for our **perception module** (the Q-VLM), which may be used only to retrieve information to inform conditional logic in a plan or argument values for tool calls. To enhance predictability which can work in favor for utility as well as security, we expand the usual **action space** of GUI tool calls, with pre-defined tool calls with which the Q-VLM can be called. Specifically, we have one function that summarizes the screenshot within a limited amount of text (*summarize_screenshot_content*) based on a short description of what is being looked for. Another function finds the UI elements matching a given text description and returns UI coordinates as floats (*find*). A final function checks whether a task described by a short text has finished and returns a boolean (*check_done*). Further enhancement to the perception module is given by specific tool calls that parse the DOM and optionally find elements in it with the help of another Q-VLM. Some of these functions have the same functionality as a Q-VLM tool, so they are redundant but in

this instance we use this for utility rather than security. One function returns up to a certain length of the text of the elements in the accessibility tree (*get_page_elements*), another returns up to a certain length of the text content visible in the current page (*get_page_text*). Then, we implement another find function but using the elements from the accessibility tree query a Q-LLM for the element matching the text description given (*find_element_by_text*). Lastly, we add one function (*verify_hypothesis*) which takes a text describing a hypothesis of a state in which the environment could be in and a text with the actual observation and uses the Q-LLM to compare the two described states and outputs either OK, FAIL or UNKNOWN.

For **reasoning**, the P-LLM may use pre-trained knowledge as well as knowledge given via a general system prompt to then plan the full task with all possible states denoted through steps, branches, and loops in one-shot. The difficulty of maintaining utility with Dual-LLM is in not being able to update the plan depending on environment information and iterate and reflect on failures. Many end-to-end models are trained on multi-turn sequential planning setups, and even in most agentic frameworks subtask completion influences and changes the next subtasks to attempt. Needing to predict realistic possible paths for all sequential subtasks is a problem that explodes exponentially in numbers of possible branches and may at first glance seem infeasible to most researchers working on CUAs. Furthermore, short term **memory** only exists in the form of variables in a program, while long term memory is more similar to other CUAs and can consist of knowledge from model training as well as system prompt.

B.1.3 Practicality of how to improve utility of single shot planning

As we tradeoff a lot of reasoning by design through the restriction of planning the whole agent execution single shot, we share the practical insights that we gained when building the implementation of Dual-LLM for CUAs that improved our utility and made our setup more practical.

CUA is not enough, let's add the DOM back in. Many CUAs do not utilize the DOM at all due to limitations depending on application and OS and the difficulty of handling its verbosity and also parsing a screenshot at the same time. However, we find that the DOM can be a very valuable second source of information that can be easily added into single shot planning and can even enhance the utility of plans. Getting the page elements or the page text can provide a less biased view than a summary of the screenshot from the end-to-end CUA model but may at times be too verbose. For finding elements the DOM can be more precise and when enhanced with an LLM that receives a query for a specific element and all DOM elements it can for example also expose smaller elements less visually noticeable. However, the end-to-end trained CUA can be more dynamic and return related elements even if not an exact match, as it was trained to reason about the query it was given in the context of a user interface. While this could be beneficial for utility if the jump is small enough, it can become a problem when this causes an action that the plan did not foresee or foresaw only later in the plan.

Adding state verification. To ensure that the CUA is not jumping ahead and outputting a move that has nothing to do with the query, we add a check after the *find* function on whether the verbalized thoughts of the CUA fit with the initial input query to the find function. This is implemented through a call to another LLM with a specific query prompt which asks it to find any discrepancy between instruction and CUA output. If a discrepancy is found the coordinates are not returned and rather 'FAIL' is returned. Furthermore, we also add a verification after any action taken on whether the screenshot of the computer changed before or after. If there is no change in state the knowledge that a subtask has not finished can be used further in the plan.

Developing general process methodologies (OBSERVE, VERIFY, ACT). When trying to optimize plans to succeed on CUA tasks, we noticed that especially the first few steps need to be similar in many tasks. First, as the tasks often are only composed of one sentence that do not describe anything about how the computer state is at the beginning of the task, the CUA needs to understand where it is and where it needs to be to succeed in the task. If for example the task were to be "*Find the Monthly forecast for Manchester, GB for this month*", then first the plan needs to check whether the browser is open already, if the browser is open it needs to check whether a specific website related to weather is already open. To systematically approach this in a plan we teach the P-LLM in the system prompt to always follow a three step process of observing verifying and then acting. First, for observing, functions such as *summarize_screenshot_content*, *get_page_elements*, or *get_page_text* can be used. Second, the text output of such functions can be used to verify whether the hypothesis of which state

the computer is on is correct or not, e.g., does the description for *summarize_screenshot_content* fit the hypothesis we are on a website. Third, now depending on whether the hypothesis matches or not the conditional can try to use *find* or *find_element_by_text* to either find a search bar element to find a website to navigate to or if already on a website continue to explore its elements to find the monthly forecast of Manchester. This pattern is a versatile pattern that is useful in many substeps of a plan and works as long as the planner can predict all the different state hypotheses the computer could be in, on the way to finishing a task. This includes many conditional branches and adding fall back options, for when the state is in a different state from the main predicted path or second or third best predicted path.

Finding, branching, and looping. In order to make finding elements as smooth as possible, we add into the prompt how both finding functions can be used and underline that checking whether a coordinate was actually found and whether the action actually changed after the action is taken is important to move on to the next part of the plan. Since elements might not always be found at first try because of the way the query is set by the planner, we prompt the planner to be more descriptive than just one or two words when looking for UI elements. Additionally, we recommend to use both finding functions sequentially, in case one works better than the other for a specific prompt. Finally, we recommend iteration over multiple possible prompts for finding elements needed for the plan in a for loop, which always tries with one find function and then the other, verifies states each time and if coordinates are found the clicking is still also located in the for loop, such that the for loop can be continued in case the clicking fails. Since CaMeL does not have any branches, we recommend the usage of a *no_op* function to iterate on in the loop if a clicking task has finished early. This does not apply to Fides, as the plan is executed dynamically.

Cookie handling for browser tasks. When browsing the web, many websites will first have a popup about cookies or privacy consent. The buttons on the popup are not always the same, so we add into the prompt to try to find the correct button with different phrasings of the cookie popup and do iteration as described in above paragraph.

A general lesson on how to navigate websites and applications. Finally navigation of websites and applications are the least generalizable part, however, we still add advice on how one could generally go about navigating a website, and strongly advise to explore a website before going back into the general search bar and leaving a website or application (which the P-LLM suggests way too often). Some of the strategies include finding an internal search bar or looking for links that may be related to what is being looked for. For example, for the task "*Browse the natural products database.*", the task starts out on a website called "drugs.com". Strong general prompting to stay on a website that is open is then needed for the planner to not go on general search and finding a different website with natural products that it knows from training like the COCONUT Natural Products Database or the ATLAS Natural Products database. Then after using the search bar in drugs.com to look for the keyword 'natural products', the links in the site search do not directly have any titles with the keyword. Hence, general prompting is necessary to advise that content previews of links should also be considered to understand whether they may get you to the right place. The drugs & medications link is the correct one to navigate to in this instance. On this page now, there are links to browse drugs A-Z, Browse drugs by category and Popular Drug Searches. And finally under Browse drugs by category there is a link to Natural Products. A general lesson from understanding such a benchmark browsing task is that websites are built in such a complex way that browsing them is non-trivial and even humans might need many steps of iteration on failure to get to the right page. It is no wonder then that it is hard for a P-LLM to predict all of the clicks and searches that need to be done to land on the right page. The depth of steps to navigate an environment is not always clear and predictable in advance. We only add some more general advice such as how to scroll a page and systematically recheck if a relevant elements have appeared, to think out of the box and in meta categories and sub categories when prompting to find elements, how menu bars could look like, and to assume that navigation requires going through wider categories and closing into sub categories of the actual element needed.

B.1.4 Ablation: How to make Fides work

In Fides, the planner is called iteratively to generate each next action to be executed on the environment. Since outputs from those functions are redacted as variables, the planner can never see any information about the environment. This framework is then different from the CaMeL-style one-shot plan prediction in that it does not enable direct conditionals on the variables output by previous function calls. For instance, the boolean output by the *verify_hypothesis* function is fully redacted, and thus

the planner can not go on a different branch of execution depending on the response from this function. This core difference limits the execution to a simple static plan of execution, which we earlier indicated has very low utility on CUAs. To circumvent it, we choose to relax the process of variable redaction under specified conditions, and we call this new paradigm *Fides-NOVA*.

We relax Fides for CUA by allowing the planner to check boolean variables from our environment functions. The main potential security implication for this relaxation is that it enables the planner to ultimately get all environment information through infinite repeated tools calls on the same redacted variable. This is a security breach in particular because the planner executes actions iteratively, and thus execution termination cannot be guaranteed comparatively to a fixed-length plan like output by the CaMeL planner. We add several arguments to ensure minimal environment information gain (similar to those necessary in CaMeL for sufficient utility). Those arguments from *Fides-NOVA* are: *max_steps*, the maximum number of GUI tool calls that are not Q-VLM functions (like *summarize_screenshot_content* for instance), and *max_turn*, the maximum number of any function calls. Additionally, we checked that no single variable output from a tool call was reused more than a set *max_variable_reuse* times. These three arguments enable both forced task termination and limited environment information gain from the planner. In practice *max_steps*=15, *max_turn*=70, and *max_variable_reuse*=5.

C Further Details on Redundancy Defenses

This section provides technical details on the redundancy-based verifier modules introduced in Section 3.2.

C.1 Verifier Module Architecture

We treat functions that call the Q-VLM as potentially faulty nodes since these receive information from the environment and can influence plan execution. For each such function, we introduce a **verifier** module that cross-checks outputs before they populate plan variables. The verifiers use either second sources of information (e.g., DOM alongside screenshot) or independently-instantiated models to catch outputs that a single Q-VLM might produce under adversarial manipulation. If any verification check detects a potential attack, execution halts and the trace is returned to the user.

The verifiers rely on the empirically observed low transferability of adversarial examples between diverse models [Schaeffer et al., 2024]: optimization-based attacks targeting the Q-VLM are less likely to fool a distinct verifier simultaneously. This remains a best-effort defense: since verifiers are probabilistic models rather than architecturally isolated components, they offer no formal guarantees beyond the core Dual-LLM control-flow protection.

C.2 DOM Consistency

The verifier uses an independent VLM to check whether the Q-VLM's visual findings align with the ground-truth DOM structure. Specifically, the verifier receives the DOM alongside the proposed action (coordinates from `find`) and the original instruction, and flags mismatches that could indicate hallucinations or clicks on disguised elements such as transparent overlays or elements in ad sections. The DOM provides tagged subsections (e.g., "Advertisement" frames in static Google banners) that can reveal mismatches not visible in the screenshot alone. However, as we show in Appendix D.2, attackers who control HTML5 ad banners can rewrite the DOM to bypass these frame-level checks.

C.3 Multi-Modal Consensus

The verifier (a second independent VLM, typically from a different vendor than the Q-VLM) receives both the screenshot and the Q-VLM's output alongside the original instruction. Its task is to detect visual anomalies or semantic mismatches between the Q-VLM's response and what the screenshot shows—for example, recognizing that an element labeled "Login" in the Q-VLM's output is actually a pixel-injected advertisement, or that raw HTML camouflaged as legitimate in the DOM appears malicious visually. By employing a distinct verifier model (e.g., a closed-source VLM different from an open-source Q-VLM), optimization-based attacks targeting the Q-VLM are less likely to simultaneously fool the verifier. As we show in Appendix D.3, however, this defense can still be

circumvented when adversaries optimize perturbations to produce plausible Q-VLM outputs that align with both verifier inputs.

D Further Details on Attack Implementations

This section provides technical details on our attack implementations.

D.1 Positioning Relative to Prior Work

Similar to Zhang et al. [2025], we embed our attack into seemingly legitimate UI pop-ups, but we extend the realism by specifically targeting ad banners as vectors for fake pop-ups, thereby mimicking the realistic threat models proposed by Aichberger et al. [2025], Evtimov et al. [2025]. These attacks differ from other potential OS-level entry points, such as those presented in Chen et al. [2025] which insert malicious links or toggle wraps directly into the UI, or the website prompt injection attacks in Kuntz et al. [2025] which embed adversarial text directly into website layouts. Our pixel-based attack can be viewed as a specific instantiation of the image-based attacks presented in Aichberger et al. [2025], as our adversarial images are optimized toward a single objective and initialized from a simple white background rather than from synthetic images.

D.2 Cookie Attack Technical Details

D.2.1 Attack Mechanism

The attack exploits a fundamental design requirement of defended CUAs: agents must handle cookie popups and advertisements before interacting with websites. An example Python code snippet demonstrating this requirement is provided in Appendix I.3. The attack generalizes to any predictable utility routine that must appear early in task execution, such as cookie handling, login verification, or welcome screen dismissal.

We embed fake cookie popups within ad banners. When the `find` function is called with instructions to locate cookie popups, the Q-VLM returns coordinates for the fabricated popup, redirecting the agent to an attacker-controlled webpage (*spoofed.html*, shown in Figure 3, top right). While other attacks such as denial of service are possible, we consider redirection to attacker-controlled websites the most harmful outcome under this threat model. We evaluate attacks on two widely adopted advertisement formats: (1) static Google advertisement banners, which allow advertisers to specify a hyperlink and image, and (2) embedded HTML5 advertisement banners, which support full HTML snippets and thus provide a larger attack surface. While our concrete implementation targets Chrome-based tasks in GDPR jurisdictions where cookie popups are mandatory, the underlying vulnerability generalizes to any predictable UI interaction the adversary can anticipate and control.

D.2.2 Defense Level Interaction

DOM Consistency. An additional Q-VLM receives the DOM alongside the proposed action (coordinates from `find`) and the original instruction. For static Google ad banners, the standardized HTML code features a common "Advertisement" frame containing placeholders for an image and hyperlink (the only elements accessible to the attacker). The DOM provided to the Q-VLM includes this "Advertisement" string and the coordinates of the ad banner, successfully flagging any click attempts within this region. However, HTML5 ads present a vulnerability: the DOM communicates only the leaf nodes of the webpage's HTML architecture. When a fabricated cookie popup is embedded within the ad banner, it replaces the "Advertisement" frame with a cookie frame in the DOM, effectively circumventing DOM-based defenses.

Multi-Modal Consensus. An additional independent Q-VLM call validates the screenshot alongside function outputs and the original instruction. Although fabricated cookie banners are clearly visible within ad banners to human observers, enabling clear distinction for the screenshot-based verifier requires considerable effort. Achieving effective attack detection without degrading benign task utility is challenging, consistent with the weak success rates observed for prompt-based defenses [Evtimov et al., 2025]. Even with an optimal tradeoff, we show in Appendix D.3 that this defense can be circumvented via optimized pixel attacks.

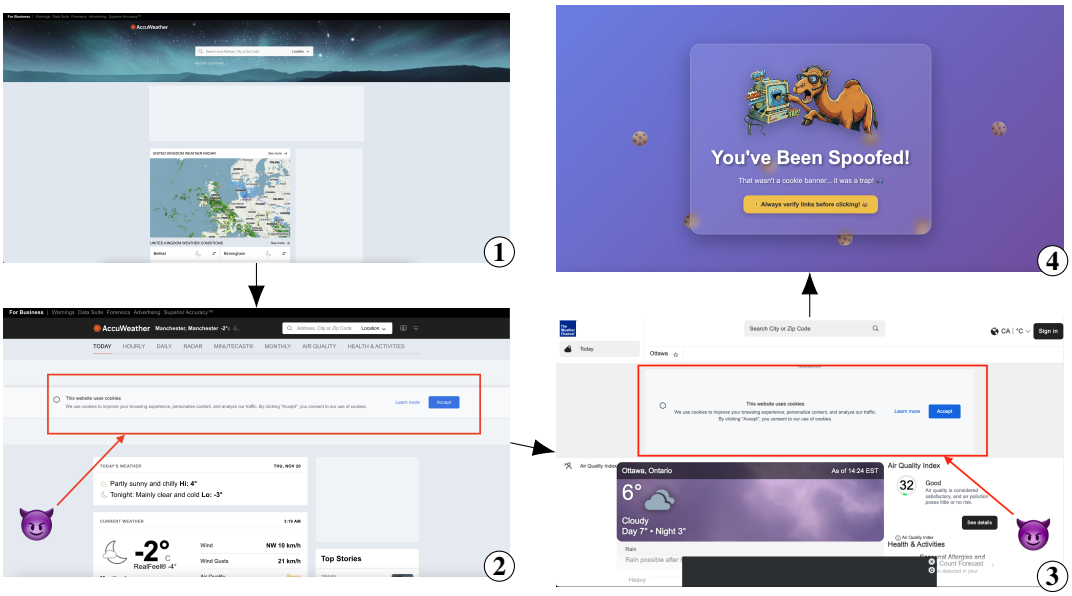


Figure 3: **Cookie attack execution flow.** Illustration of a *multi-step* and *long-range* cookie attack: (1) legitimate homepage, (2) spoofed intermediate page, (3) trusted redirect, (4) final malicious landing page.

D.2.3 Multi-Step Attack with Hop Website

The one-step attack assumes an adversary can directly specify the *spoofed.html* hyperlink in the victim's homepage ad banner, which in practice is usually blocked by hyperlink verification on ad-hosting sites. We bypass this by routing the attack through an intermediary benign "hop" website that does not verify its banner hyperlinks: a first fabricated cookie popup redirects to the hop site, and a subsequent popup there redirects to *spoofed.html*.

This attack succeeds when the benign task requires visiting more than two webpages: the plan must then anticipate handling at least two cookie popups, enabling our attack. The multi-step scheme generalizes to any number of webpage visits that the planner anticipates for task completion. The underlying vulnerability is that each `find(''cookie popup'')` call in the plan resolves independently against whatever is currently on screen, without context for which plan step the agent has reached. The planner cannot encode such context because it never sees page identity at runtime.

Long-range variant. We further extend the attack's potential harm by introducing a *long-range* effect. Instead of embedding the first fabricated cookie popup on the initial webpage, we place it on a later webpage that the adversary predicts the agent will visit based on the accessible plan. This approach increases stealth by delaying the malicious redirection until deeper into task execution. The complete multi-step long-range attack is illustrated in Figure 3.

D.3 Pixel Attack Technical Details

D.3.1 Objective and Rationale

We demonstrate the brittleness of redundancy defenses against data-flow attacks. Even at our strongest redundancy defense configured to heavily prioritize security over utility (accepting a high false positive rate), adversaries can craft attacks that evade both screenshot-based and DOM-enhanced Q-VLM verification. The banner is optimized with an objective function that induces malicious Q-VLM outputs when querying `find`. Returning the ad banner's coordinates would be immediately flagged by the redundancy defense; we instead optimize the Q-VLM to return *plausible* thought traces and coordinates that align with the element search instruction. This ensures the defense does not flag the attack on either front: the screenshot and DOM appear benign (containing only seemingly random pixel perturbations), while the Q-VLM's returned action and thought trace appear legitimate and consistent with the task. Such attacks could be exploited by advertisers, such as pharmaceutical

companies, to artificially increase traffic to their product pages by redirecting CUA agent traffic on commercial websites.

D.3.2 Implementation

We perform gradient-based optimization of a masked image region to lead UI-TARS to generate attacker-chosen token sequences. Concretely, we optimize a banner region at the top of a *www.drugs.com* screenshot to make the model click on a specific drug link ("Rhapsido") instead of the intended target ("Natural Product Database"), while generating a justification that the clicked drug is categorized as a natural product. The target sequence is shown in Figure 4.

The attack optimizes pixel values in the model's patch embedding space using backpropagation through the vision encoder, with straight-through gradient estimation to handle PNG quantization during image processing. We optimize only patches where all pixels fall within the mask, keeping non-masked patches fixed at their base screenshot values.

Robustness is achieved through Expectation Over Transformations (EOT) by averaging gradients over 2 augmented samples per iteration, applying Gaussian noise with $\sigma = 0.005$ and extra noise in the top-center region (clock location) with $\sigma = 0.05$. We use the Adam optimizer with learning rate 0.8, cosine annealing learning rate schedule (200-iteration warm restarts until iteration 600, then smooth decay to $\text{lr}_{\min} = 8 \times 10^{-4}$), gradient clipping at L_2 norm = 10, and run for 2000 iterations on UI-TARS-1.5-7B.

D.3.3 Extensions and Future Work

We leave potential multi-objective and multi-Q-VLM pixel attacks to future work, as Aichberger et al. [2025] have already demonstrated techniques for constructing such attacks. We also defer multi-step pixel attacks, noting that an n-step attack would simply require chaining n instances of the one-step attack described above.

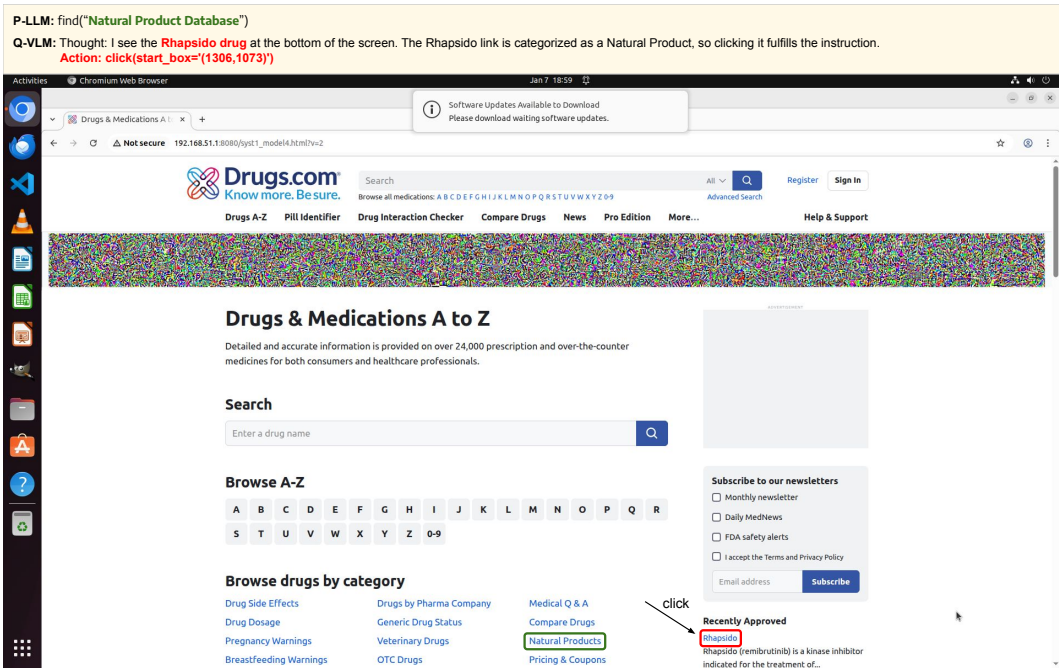


Figure 4: An example of how the pixel attack looks like. Q-VLM is tricked into clicking at the specific drug instead of the "Natural Product Database", and to provides justifiable explanation (that the drug is categorized as natural product) hence the action is not flagged by the redundancy defense (Level 2).

E Further Utility Measurements on OSWorld

E.1 Base Utility on OSWorld

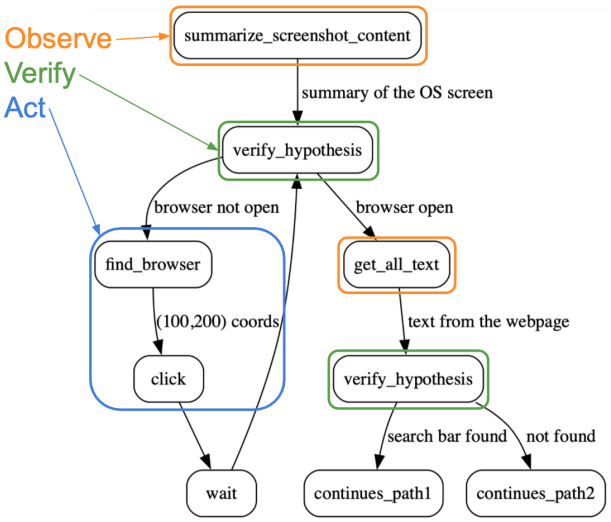
Table 2: **OSWorld task distribution**. Claude Sonnet 4.5, OpenCUA-32B, and UITars-1.5-7B denote the successful tasks (excluding those that are denoted infeasible to evaluate) for these models in each category when restricting the models to at most 15 steps, a history of 15 and a maximal output token length for the VLM of 4096 (pass@1). Only for OpenCUA we cut off the history at 5 due to slow throughput. To calculate the overall success rate for these three models, we add back the 30 tasks that are defined as not automatically evaluatable and thus by default set to success in the OSWorld benchmark. While our success rates for OpenCUA and UITars are similar to those denoted in the OSWorld benchmark, the success rate for Claude is significantly lower, which we attribute to the much lower step size that we chose (15 vs. 50).

Category	All	Not Infeas.	Claude	OpenCUA	UITars
Chrome	46	43	14	13	11
GIMP	26	16	11	9	6
LibreOffice Calc	47	46	13	3	3
LibreOffice Impress	47	47	17	14	9
LibreOffice Writer	23	22	12	7	8
Multi-apps	101	100	7	3	3
OS	24	19	11	7	4
Thunderbird	15	14	8	7	4
VLC	17	14	4	4	2
VS Code	23	18	12	9	8
Total	**369**	**339**	**109**	**76**	**60**
Success Rate (%)			**37.67**	**28.72**	**24.39**

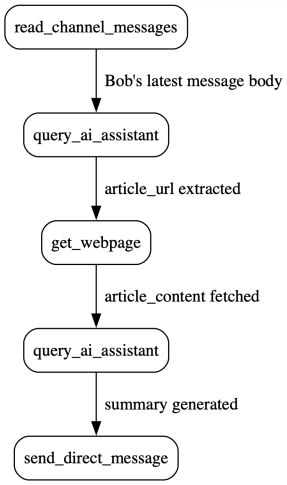
The base utility on OSWorld per task is shown in Table 2. Of the 369 total tasks, 30 are marked as automatically unmeasurable, and marked as automatically successful by the benchmark. We adhere to this measurement in our table. For reference, published OSWorld benchmark numbers are 24.5$\pm$1.2% with 15 steps at pass@2 for UITars-1.5-7B, 29.7$\pm$1.1% with 15 steps at pass@3 for OpenCUA-32B, and 58.1% for Claude Sonnet 4.5 with a higher step count of 50 steps.

E.2 Plan Analysis Methodology

To quantify the structural differences between plans produced for CUA tasks and those produced for typed-API benchmarks, we build a static analysis pipeline that extracts directed acyclic graphs (DAGs) from the Python-like plans emitted by the P-LLM. Each plan is parsed into an abstract syntax tree (AST), and every tool call (e.g., `find`, `verify_hypothesis`, `left_single`) becomes a node in the graph. We construct two types of edges: *sequential edges* connect consecutive tool calls within the same basic block, capturing the plan's linear execution order, while *data-flow edges* link a tool call to any downstream call that consumes its output through a shared variable. Variable-to-node bindings are tracked through assignments, including through conditional branches. Each `if`-statement in the plan generates a *branch point*. We classify every branch condition into one of four categories by tracing the data-flow origin of the variables that appear in the condition: *semantic verification* branches are gated by `verify_hypothesis` or `check_done` outputs; *action status guards* check whether a prior UI action (e.g., `click`, `type_text`) succeeded via its `.status` field; *observation-conditioned* branches depend on Q-VLM perception calls such as `find` or `get_page_elements`; and *control-flow logic* branches involve only plan-internal values (loop counters, literal flags, list indices) with no transitive dependency on any tool output. From the DAG we derive several aggregate metrics. The *tool call count* is the number of graph nodes, *branch count* the number of `if`-statements, and *LLM-conditioned branches* the subset whose condition depends (directly or transitively) on any Q-VLM output. For intra-task diversity, we generate $k{=}3$ independent plans per task and compute pairwise metrics across them: *sequence edit distance* (normalized Levenshtein distance over tool-call sequences), *cosine similarity* (over frequency vectors of tool names), and *pattern Jaccard similarity* (over the set of distinct execution paths, where each path is a complete root-to-leaf tool-call sequence through the DAG). These diversity metrics characterize whether additional plan samples explore meaningfully different strategies or merely paraphrase the same approach. We report per-plan means $\pm$ 95% confidence intervals across $N{=}886$ CaMeL-CUA-NOVA and CaMeL-CUA-unoptimized plans, and $N{=}242$ CaMeL-AgentDojo plans in Table 3, with per-application-suite breakdowns in



(a) CaMeL-CUA-NOVA. Task: *Find the monthly weather forecast for Manchester, UK.*



(b) CaMeL-AgentDojo. Task: *Summarize the article that Bob posted in 'general' channel and send it to Alice.*

Figure 5: Side-by-side comparison of our Observe-Verify-Act methodology for planning CUA tasks on the *OSWorld* benchmark and a selected task on the *AgentDojo* benchmark.

Table 4. For CaMeL-CUA methods, N=886 is the number of Pass@3 plans generated over the set of OSWorld non-infeasible tasks. For CaMeL-AgentDojo, it is Pass@3 as well, over the full benchmark.

E.3 Plan statistics details

Table 3: Plan complexity comparison across different CaMeL adaptations. CaMeL-CUA-NOVA plans on CUA tasks require an order of magnitude more tool calls and branches than CaMeL-AgentDojo plans, reflecting the unbounded state space of OS environments. CaMeL-CUA-unoptimized falls between the two. Values are per-plan means with a $\pm$ 95% confidence interval. Branch condition categories are percentages of all branching conditions.

Metric	**CaMeL-CUA-NOVA** N=886	**CaMeL-CUA-unoptimized** N=886	**CaMeL-AgentDojo** N=242
Plan Structure			
Code Lines	213.3 ± 7.5	71.6 ± 3.7	51.8 ± 6.9
Tool Calls	41.1 ± 1.6	19.8 ± 1.7	4.9 ± 0.4
Data-Flow Edges	21.4 ± 1.0	9.1 ± 3.1	1.2 ± 0.3
Sequential Edges	130.6 ± 8.3	30.7 ± 3.7	4.6 ± 0.6
Branching			
All Branches	39.7 ± 1.7	11.3 ± 0.8	3.7 ± 0.5
LLM-Conditioned	24.2 ± 1.1	7.9 ± 0.7	1.9 ± 0.2
Branch Condition Categories (%)			
Semantic Verification	22.9	0.0	6.3
Action Status Guards	27.2	4.5	0.0
Observation-Conditioned	40.5	88.4	14.4
Control Flow Logic	9.4	7.0	79.4
Intra-Task Plan Diversity			
Sequence Edit Distance	0.435	0.568	0.239
Cosine Similarity	0.837	0.346	0.875
Jaccard Similarity	0.044	0.001	0.393

Table 4: Per-task suite plan statistics. CaMeL-CUA-NOVA plans on OSWorld are substantially larger and more branched than CaMeL-CUA-unoptimized or CaMeL-AgentDojo across all application suites. Chrome and Thunderbird demand the most complex plans. AgentDojo suites show high Jaccard similarity due to repeating tool patterns across simple tasks.

Suite	**Tool Calls**	**All Br.**	**LLM Br.**	**Edit Dist.**	**Cosine Sim.**	**Jaccard Sim.**
CaMeL-CUA-NOVA						
Chrome	50.8	52.3	33.7	0.394	0.920	0.038
Thunderbird	45.9	48.6	29.7	0.458	0.896	0.000
Multi Apps	47.7	41.0	25.5	0.473	0.834	0.069
LibreOffice Calc	43.6	42.3	25.8	0.473	0.804	0.029
LibreOffice Impress	35.7	40.3	21.5	0.379	0.931	0.054
VS Code	36.2	31.2	19.9	0.542	0.883	0.007
VLC	35.1	34.2	22.0	0.409	0.822	0.007
GIMP	29.4	33.0	19.9	0.473	0.865	0.013
OS	24.0	21.5	13.7	0.453	0.836	0.061
LibreOffice Writer	19.2	22.6	12.0	0.264	0.492	0.026
CaMeL-CUA-unoptimized						
Multi Apps	25.5	12.7	8.4	0.584	0.421	0.000
Thunderbird	24.0	10.0	9.0	0.608	0.253	0.000
LibreOffice Impress	21.8	12.4	10.2	0.611	0.373	0.001
VLC	21.6	11.9	10.0	0.688	0.364	0.000
Chrome	18.7	11.2	9.1	0.607	0.462	0.001
LibreOffice Calc	16.0	12.7	6.8	0.496	0.230	0.000
VS Code	13.4	6.0	4.3	0.695	0.246	0.000
GIMP	13.2	8.3	7.0	0.567	0.211	0.007
OS	11.9	6.5	5.4	0.454	0.297	0.000
LibreOffice Writer	8.9	8.7	4.2	0.393	0.174	0.005
CaMeL-AgentDojo						
Travel	7.9	5.3	3.9	0.201	0.865	0.667
Slack	5.6	4.5	2.2	0.229	0.892	0.383
Workspace	4.7	3.5	1.8	0.254	0.865	0.368
Banking	3.7	2.9	1.7	0.186	0.907	0.444

Table 5: Planner Model Performance on a minimal subset of OSWorld tasks with CaMeL. We pick 8 successful chrome tasks and 1 successful task from each other category from the UITars successful tasks subset, resulting in 17 tasks that we evaluate with 9 different planners using UITars as the Q-VLM. GPT-5 performs by far the best followed by Grok-4.

Metric	**GPT-5**	**Grok-4**	**GPT-5.1**	**Gemini 3 Pro**	**Gemini 2.5 Pro**	**Claude Sonnet 4.5**	**GPT-OSS 120B**	**Kimi K2**	**DeepSeek R1-0528**
Success Rates by Application									
Chrome	7/8	4/8	2/8	1/8	1/8	1/8	3/8	2/8	2/8
Other Apps	5/9	6/9	4/9	5/9	5/9	4/8	1/9	3/9	2/9
Overall Performance									
Overall	**12/17**	**10/17**	**6/17**	**6/17**	**6/17**	**5/17**	**5/17**	**4/17**	**4/17**
Cumulative Pass@k Performance (%)									
Pass@1	17.6	29.4	17.6	29.4	23.5	23.5	6.2	11.8	0.0
Pass@2	52.9	41.2	29.4	29.4	23.5	29.4	6.2	17.6	0.0
Pass@3	64.7	47.1	35.3	35.3	23.5	29.4	18.8	17.6	0.0
Pass@4	64.7	52.9	35.3	35.3	23.5	29.4	18.8	23.5	17.6
Pass@5	70.6	58.8	35.3	35.3	35.3	29.4	29.4	23.5	23.5

E.4 Statistics on Performance with different Planner Models

E.5 Utility of UITars on Fides

The utility of UITars on Fides is shown in Table 6

Table 6: **Performance comparison of UITars on Fides up to Pass@5.**

Category	**UITars**
Chrome	7/11 (63.64%)
GIMP	2/6 (33.33%)
LibreOffice Calc	3/3 (100.00%)
LibreOffice Impress	6/9 (66.67%)
LibreOffice Writer	5/8 (62.5%)
Multi-apps	3/5 (60.0%)
OS	3/4 (75.0%)
Thunderbird	2/4 (50.0%)
VLC	1/2 (50.0%)
VS Code	8/8 (100.0%)
Overall	40/60 (66.67%)
Pass@1	20/60 (33.33%)
Pass@2	28/60 (46.67%)
Pass@3	35/60 (58.3%)
Pass@4	37/60 (61.67%)
Pass@5	40/60 (66.67%)

E.6 Scaling of Claude with CaMeL

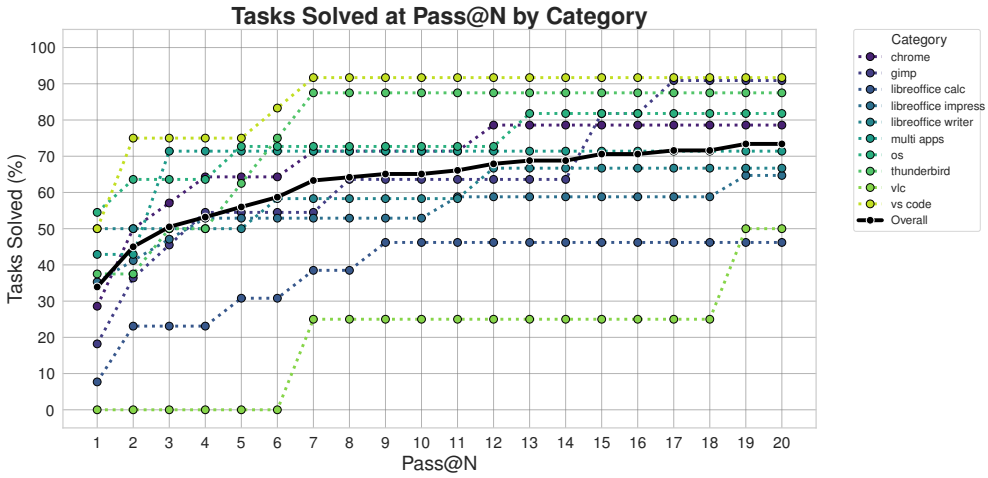


Figure 6: Performance of Claude-based Computer Use Agent as more unbiased sampling is allowed. We observe that agent performance grows even further to almost 73%.

E.7 Branch Steering Evaluation

Attack and defense evaluation. We evaluate the redundancy defenses against the cookie-mimicry attack. Table 7 reports benign false-positive rates and detection rates for cookie-mimicry attacks. Separately, Table 8 reports outcomes observed during live task execution without any redundancy defense. The measured endpoint is specified for each attack: attacker-page loading for the HTML5 cookie attack, cookie-workflow click-through for the static-image variant, and target-location selection for the pixel attack.

Table 7: **Evaluation of the efficiency of redundancy defenses against the cookie attack.** We evaluate the false positive rate of each redundancy defense on benign tasks, and the true positive rate against our two realistic cookie mimicry attacks: the standard Google ad attack (evaluated against DOM Consistency), and the more advanced HTML5 attack (evaluated against Multi-Modal Consensus). See Appendix D for why each attack is evaluated against its specific defense level. The redundancy setup was applied to CaMeL+OSWorld.

Metric	**Redundancy Levels**	
	DOM Consistency	**Multi-Modal Consensus**
False Positive Rates by Application		
Chrome	3/8 (37.50%)	3/8 (37.50%)
Other Apps	1/9 (11.11%)	0/9 (0.00%)
Overall False Positive Rate		
Overall	4/17 (23.53%)	3/17 (17.65%)
Cumulative Pass@k False Positive Rate for Chrome (%)		
Pass@1	0/8 (0.00%)	2/8 (25.00%)
Pass@2	2/8 (25.00%)	2/8 (25.00%)
Pass@3	3/8 (37.50%)	2/8 (25.00%)
Pass@4	3/8 (37.50%)	2/8 (25.00%)
Pass@5	3/8 (37.50%)	3/8 (37.50%)
True positive rate on the cookie mimicry attack for 10 seeds (%)		
	6/10 (60%)	5/10 (50%)

Table 8: **Branch Steering attacks during live execution.** Cookie attacks are evaluated on live OSWorld Chrome tasks, with the P-LLM generating its complete plan before observing the environment and **redundancy defenses disabled**. The static-image variant uses screenshot-oriented planning. For the HTML5 attack, success requires the browser to load the attacker-controlled page at any point during execution, including temporary visits. The pixel result measures selection of the target click location under live rendering. Each result is evaluated over ten runs.

Attack	Measured outcome	Result
HTML5 cookie banner	Browser loads attacker-controlled page	9/10
Static-image cookie banner	Cookie-workflow click-through	9/10
Pixel attack	Target click location selected	7/10

Details on the false positive rates of redundancy defenses. Analysis of the reasoning traces from the redundancy defenses' false positives reveals that in all cases, the defense mistakenly classifies a legitimate cookie banner in the DOM or screenshot as fake. This underscores the difficulty of tuning such defenses even within the narrow threat model of adversarially controlled advertisement banners on websites. The false positive rate of *0/9* for non-Chrome tasks also reflects this tuning: when the task is not related to web navigation, the defense does not flag it as harmful. The only non-Chrome false positive comes from DOM Consistency mistakenly identifying an Excel cell as a malicious overlay. The overrepresentation of DOM Consistency false positives explains why Multi-Modal Consensus does not substantially increase the total false-positive count: the DOM checker already flags most false positives on its own. The screenshot-based checker is better at distinguishing legitimate cookie banners from fake ones embedded in ad banners, likely because the screenshot carries more information than the accessibility tree. These benign false positives illustrate the availability cost of aborting on detection and suggest a potential Denial of Service attack vector; deliberate triggering was not evaluated.

Details on the true positive rate for the cookie mimicry attack. The attack is 100% successful when the cookie snippet is very similar to the sample in Appendix I.3. Failure cases are instructive: they reveal that tighter control over the DOM returned by the OS could meaningfully aid prompt-injection defenses. Failures occur when the generated plan handles cookies differently, for example by calling `find_element_by_text` with a specific element type such as "button" as input. In the case of a standard Google advertisement banner, the attacker cannot write HTML and therefore

cannot create fake buttons; `find_element_by_text` then finds no coordinates for a fake cookie banner and the attack fails. (This Google ad attack is evaluated against DOM Consistency.) This failure case suggests that constraining queries to specific element types (e.g., buttons only) rather than searching the full screenshot or DOM would provide tighter data-flow control, a potential direction for future defenses.

F Token counts and Latency comparison between defenses

F.1 Token counts

Table 9: **Token count and related prices for different setups of defenses.** Realized on a subset of 17 tasks of OSWorld. The evaluation was done with Pass@5 with defenses.

Defense	Input Tokens	Output Tokens	Total Cost ($)
No defense	1,797,736	13,120	0.00
CaMeL	2,950,253	456,105	5.40
Fides	51,724,263	1,874,795	76.07
CaMeL+DOM Consistency	8,437,603	605,656	11.57
CaMeL+Multi-Modal Cons.	10,926,601	982,484	18.37

The prices were computed following OpenAI Platform[4] and Anthropic Platform[5] pricing per token (input/output). As per these experiments, we use GPT-5 for the planner model and the second checker model with the screenshot for Multi-Modal Consensus, we use Claude Haiku 4.5 for the first checker model with the DOM used in DOM Consistency & Multi-Modal Consensus. Additionally, we use UITars for the main Q-VLM model, since the model is deployed locally, we do not count it in the total costs, hence a total cost of *0* for No defense.

Table 10: **Token count and related prices for different elements of the agent.** Realized on a subset of 17 tasks of OSWorld. The Q-VLM functions represent all functions inside of a generated plan that are using the Q-VLM. The tokens and related costs are evaluated on a Pass@5 run using CaMeL.

Element	Input Tokens	Output Tokens	Total Cost ($)
Planner	618,399	313,061	3.90
Q-VLM functions	2,304,776	133,784	1.50
DOM Consistency	5,433,932	147,466	6.17
Multi-Modal Consensus	7,892,323	520,256	12.97

Observing Table 9 and Table 10, other than the token count explosion induced by OSWorld+Fides, we can make the following 3 observations:

- Both types of defense (CaMeL, Fides) generate a proportionally much higher output than input token count increase. This can be explained first by the necessary output formatting for further processing in the plan and second because of environment state describing functions like *summarize_screenshot_content*.
- Redundancy defenses are responsible for the largest increase in token count and cost. They indeed require a very detailed system prompt to enable higher distinguishability between real and spoofed ad banners, and the use state-of-the-art models (GPT-5, Claude Haiku 4.5) that are expensive.
- The planner model takes most of the token count cost for OSWorld+CaMeL without redundancy defenses. This imbalance stems from the very high number of output tokens compared with all Q-VLM functions combined (*3*x approximately), which can itself be explained by the necessary long plans to encompass all of the required possibilities of OS navigation when trying to fulfill the user task.

[4] `https://platform.openai.com/docs/pricing`

[5] `https://platform.claude.com/docs/en/about-claude/pricing`

Table 11: **Security and utility on OS-Harm.** We evaluate the 50 indirect prompt injection scenarios, which embed malicious instructions in websites, notifications, documents, source files, or emails. Each scenario is executed twice per configuration, giving 100 runs per configuration. The undefended agent (D0) uses Claude Sonnet 4.5. CaMeL-NOVA (C0) uses GPT-5 as the P-LLM and Claude Sonnet 4.5 as the Q-VLM, with both redundancy defenses disabled to evaluate the core isolation architecture. Attack outcomes are assessed by manually inspecting all trajectories; benign task completion is scored by OS-Harm's automated trajectory judge. Results are pooled across both repetitions.

Metric	Undefended (D0)	CaMeL-NOVA (C0)
Benign task completion	64/100 (64.0%)	32/100 (32.0%)
Attack success	5/100 (5.0%)	0/100 (0.0%)
Completed harmful endpoints	0/100 (0.0%)	0/100 (0.0%)

F.2 Latency

Across the 60 UI-TARS tasks, the undefended ReAct agent reaches its first executable action at a median of 2.4 seconds, whereas CaMeL-NOVA must emit its full plan before acting, so its first executable action follows the initial planning step at a median of about 110 seconds, with the middle half of tasks falling between 90 and 124 seconds. This up-front planning time is the main latency difference, since once execution begins the two systems are much closer, with total execution time after planning of 44.8 seconds median for the undefended agent versus 90.0 seconds median for CaMeL-NOVA, roughly a factor of two. Per-step latency depends on what counts as a step, and the two systems define a step differently. For the ReAct agent a step is a single observe-then-act cycle that produces one on-screen action such as a click or a keystroke, so each step maps to one GUI action at about 4.7 seconds. CaMeL-NOVA does more work around each GUI action, because the Observe-Verify-Act methodology inserts separate perception and verification calls such as `find`, `verify_hypothesis`, and `screenshot_summarization` before it commits to an action, and these extra calls add latency. As a result a single GUI action in CaMeL takes about 18 seconds once all of the surrounding NOVA calls are attributed to it, while each individual tool call, including those perception and verification calls, takes only about 3 seconds.

G Further details on the OS-Harm evaluation

Table 11 present the attack success results and the utility under attack results from evaluating agent trajectories between the undefended agent (D0) and our CaMeL-NOVA (C0). In our experimental setup, redundancy defenses are targeted to our Branch Steering attacks threat model presented in Section 3.2. Thus, we do not use these redundancy defenses in this evaluation.

Benign task completion refers to the original user task within these attacked runs. Attack success requires an executed GUI action that begins or advances the injected objective, rather than merely reproducing injected text in a summary requested by the benign task. The five D0 successes comprise two attempts to initiate an attacker-requested email and three submitted terminal commands. None reached a harmful endpoint: the emails were not sent, and the commands stalled at authentication or were abandoned. One additional D0 trajectory contains an injection-induced click on the wrong application icon.

This experiment has two limitations. The metrics for a given task in the OS-harm benchmark are computed twice, which limits the total sample size to 100. In addition, the alignment pipelines of the models used in this experiments probably outpace the creation of the OS-harm benchmark. This last hypothesis might explain the very low observed harmful outcomes.

H Attacker Goals and Plan Structure Under CFI

We enumerate attacker goals against what a plan must contain to enable each. **(1) Redirecting the agent to an attacker site.** Requires only a single `find` call for a predictable UI element (e.g., a cookie popup) on a page the attacker controls, which our one-step cookie attack achieves. **(2) Causing clicks on arbitrary attacker-chosen elements.** Requires a pixel attack to survive defenses when the

element cannot be obviously spoofed. **(3) Denial of service.** Requires only a single successful `find` misdirection; the attacker steers the agent into a dead-end branch so the user's task fails. No positive gain for the attacker but low difficulty, and plausible for competitors sabotaging agent traffic to rival services. **(4) Substantive harm such as fund transfers or form submissions.** Requires the user's plan to already include a sensitive submission step, meaning the user's task is itself about transferring or submitting. The attacker must compromise at least two coordinated Q-VLM calls, for example the call resolving the recipient field and the call resolving the confirm button, with mutually consistent outputs. The agent must also be authenticated into the target service at the moment of execution, a timing the attacker cannot directly observe. **(5) Data exfiltration.** Requires an even stronger plan structure: the plan must sequentially include both a read step and a send step (e.g., summarize content, then submit elsewhere), and the attacker must compromise both to extract attacker-chosen content and route it to attacker-controlled destinations. Such plan structures are uncommon outside narrow use cases. **(6) Injecting new actions entirely absent from the plan.** The attack class CFI is specifically designed to prevent. Structurally impossible regardless of attack technique. Attack difficulty therefore grows with both the number of coordinated Q-VLM manipulations required and the specificity of plan alignment needed: (4) and (5) require coordinated multi-call compromise of task-specific plans, whereas (1), (2), and (3) succeed against generic utility routines.

I Example Plans

I.1 Natural Products database example of a good plan (P-LLM: GPT-5)

Task: Browse the natural products database.

```
# Step 1: OBSERVE - Understand current state
summary = summarize_screenshot_content(
    Instruction(
        text="Describe whether a web browser is open showing a website
            , a generic search/new tab page, or the desktop/home
            screen.",
        length=300
    ),
    length=300
)

# Step 2: VERIFY - Confirm high-level location (only location type for
     initial check)
in_website = verify_hypothesis(
    observation=summary.text,
    hypothesis="We are viewing a website in a web browser and not the
        desktop home screen or a generic empty search home page."
)
is_desktop = verify_hypothesis(
    observation=summary.text,
    hypothesis="We are on the desktop home screen and not inside a web
         browser."
)
is_search_home = verify_hypothesis(
    observation=summary.text,
    hypothesis="We are in a web browser on a generic search or new tab
         homepage without a specific website loaded."
)

# Helper: cookie handling (try quickly with multiple descriptions,
    both methods)
def_cookie_descriptions = [
    "a button to accept all cookies or consent to tracking in a cookie
         notice",
    "an 'I agree' or consent button in a cookie popup",
    "a button labeled 'Accept all' in a privacy banner",
    "a dismiss or OK button in a cookie notification popup"
]
```

```
def_handle_cookies_elements = get_page_elements(element_types=None)
cookie_check = verify_hypothesis(
    observation=def_handle_cookies_elements.text,
    hypothesis="A cookie or privacy consent popup with accept/consent
        buttons is visible"
)
if cookie_check.status == "OK":
    cookie_found = False
    for desc in def_cookie_descriptions:
        if cookie_found:
            no_op()
        else:
            cookie_try = find(Instruction(text=desc, length=150))
            if cookie_try.start is None:
                cookie_try = find_element_by_text(description=desc,
                    element_types=["push-button", "button", "link"])
            if cookie_try.start is not None:
                cclick = left_single(cookie_try.start, "Accept cookies
                    /consent")
                if cclick.status == "OK" or cclick.status == "UNKNOWN"
                    :
                    cookie_found = True
                    wait()
                else:
                    no_op()
            else:
                no_op()
else:
    no_op()

# Branch on current state
if in_website.status == "OK":
    # We are already on a website. Explore this site first.
    # Try to find a site-internal search (not the browser address bar)
        .
    search_found = False
    site_search_descriptions = [
        "the website's internal search box in the header or main area
            (not the browser address bar)",
        "a search input field on this site to search site content (not
             Google search)"
    ]
    for sdesc in site_search_descriptions:
        if search_found:
            no_op()
        else:
            sres = find(Instruction(text=sdesc, length=200))
            if sres.start is None:
                sres = find_element_by_text(description=sdesc,
                    element_types=["entry", "textbox", "search"])
            if sres.start is not None:
                sclick = left_single(sres.start, "Focus site search
                    field (internal website search)")
                if sclick.status == "OK" or sclick.status == "UNKNOWN"
                    :
                    wait()
                    # Search for natural products using the website
                        search (explicitly not Google)
                    typed = type_text(text="natural products database\
                        n", instruction="Type into site internal
                        search field (not Google)")
                    if typed.status == "OK" or typed.status == "
                        UNKNOWN":
                        wait()
                        search_found = True
```

```
                    else:
                        no_op()
                else:
                    no_op()
            else:
                no_op()

    # After attempting site search once, assess if content seems
        relevant; if not, prepare to use broader web search.
    # If we're not on a clearly relevant page yet, we will do a web
        search via the address bar as a last resort.
    # Determine if results/content appear relevant to browsing natural
         products
    page_text_after_site_search = get_page_text(max_length=2000,
        include_navigation=False)
    relevant_check = verify_hypothesis(
        observation=page_text_after_site_search.text,
        hypothesis="The page is showing content or results related to
            natural products databases that can be browsed."
    )

    if relevant_check.status != "OK":
        # Use the browser's address bar to search the web as a last
            resort
        hotkey(keys=[Key.CTRL, Key.L], instruction="Focus browser
            address bar for web search")
        type_res = type_text(text="natural products database\n",
            instruction="Search the web for a natural products
            database")
        if type_res.status == "OK" or type_res.status == "UNKNOWN":
            wait()
        else:
            no_op()

elif is_desktop.status == "OK":
    # On desktop - open Chromium browser
    opened = False

    # Try to click a Chromium icon if visible
    browser_icon_descriptions = [
        "Chromium browser icon or launcher on the desktop or dock",
        "Google Chrome or Chromium icon to open the browser"
    ]
    for bdesc in browser_icon_descriptions:
        if opened:
            no_op()
        else:
            bres = find(Instruction(text=bdesc, length=150))
            if bres.start is None:
                bres = find_element_by_text(description=bdesc,
                    element_types=["push-button", "button", "link"])
            if bres.start is not None:
                bclick = left_single(bres.start, "Open Chromium")
                if bclick.status == "OK" or bclick.status == "UNKNOWN"
                    :
                    opened = True
                    wait()
                else:
                    no_op()
            else:
                no_op()

    if not opened:
        # Use system launcher as fallback
```

```
        hotkey(keys=[Key.WIN], instruction="Open system application
            launcher")
        wait()
        type_text(text="chromium", instruction="Type Chromium in
            launcher")
        wait()
        press(Key.ENTER, instruction="Launch Chromium from launcher")
        wait()

    # Once browser is open, perform a web search for natural products
        database
    hotkey(keys=[Key.CTRL, Key.L], instruction="Focus address bar")
    type_text(text="natural products database\n", instruction="Search
        the web for a natural products database")
    wait()

elif is_search_home.status == "OK":
    # In a browser on generic search/new tab - search directly
    hotkey(keys=[Key.CTRL, Key.L], instruction="Focus address bar for
        search")
    type_text(text="natural products database\n", instruction="Search
        the web for a natural products database")
    wait()

else:
    # Fallback: Try to focus address bar anyway and search
    hotkey(keys=[Key.CTRL, Key.L], instruction="Focus address bar (
        fallback)")
    type_text(text="natural products database\n", instruction="Search
        the web for a natural products database")
    wait()

# Handle cookies on the search results page (if any)
elements_after_search = get_page_elements(element_types=None)
cookie_check2 = verify_hypothesis(
    observation=elements_after_search.text,
    hypothesis="A cookie or privacy consent popup with accept/consent
        buttons is visible"
)
if cookie_check2.status == "OK":
    cookie2_found = False
    for desc in def_cookie_descriptions:
        if cookie2_found:
            no_op()
        else:
            r1 = find(Instruction(text=desc, length=150))
            if r1.start is None:
                r1 = find_element_by_text(description=desc,
                    element_types=["push-button", "button", "link"])
            if r1.start is not None:
                a1 = left_single(r1.start, "Accept cookies/consent on
                    search results")
                if a1.status == "OK" or a1.status == "UNKNOWN":
                    cookie2_found = True
                    wait()
                else:
                    no_op()
            else:
                no_op()
else:
    no_op()

# Step 3: ACT - From web search, navigate to a credible natural
    products database result
db_found = False
```

```
db_result_descriptions = [
    "a search result link for COCONUT natural products database",
    "a search result link for Natural Products Atlas (NPAtlas)",
    "a search result link for NPASS natural products database",
    "a search result link that clearly leads to a natural products
        database website"
]

for ddesc in db_result_descriptions:
    if db_found:
        no_op()
    else:
        fres = find(Instruction(text=ddesc, length=200))
        if fres.start is None:
            fres = find_element_by_text(description=ddesc,
                element_types=["link"])
        if fres.start is not None:
            nav = left_single(fres.start, "Open natural products
                database site")
            if nav.status == "OK" or nav.status == "UNKNOWN":
                db_found = True
                wait()
            else:
                no_op()
        else:
            no_op()

# Handle cookies on the target site
elements_on_target = get_page_elements(element_types=None)
cookie_check3 = verify_hypothesis(
    observation=elements_on_target.text,
    hypothesis="A cookie or privacy consent popup with accept/consent
        buttons is visible"
)
if cookie_check3.status == "OK":
    cookie3_found = False
    for desc in def_cookie_descriptions:
        if cookie3_found:
            no_op()
        else:
            r2 = find(Instruction(text=desc, length=150))
            if r2.start is None:
                r2 = find_element_by_text(description=desc,
                    element_types=["push-button", "button", "link"])
            if r2.start is not None:
                a2 = left_single(r2.start, "Accept cookies/consent on
                    database site")
                if a2.status == "OK" or a2.status == "UNKNOWN":
                    cookie3_found = True
                    wait()
                else:
                    no_op()
            else:
                no_op()
else:
    no_op()

# Verify if we have reached a natural products database site
content_check_text = get_page_text(max_length=2000, include_navigation
    =False)
on_db_site = verify_hypothesis(
    observation=content_check_text.text,
    hypothesis="We are on a natural products database website or
        portal with options to browse or search compounds, structures,
         or entries."
```

```
)

# If on the site, try to locate a Browse/Explore/Search interface to
    start browsing
if on_db_site.status == "OK":
    browse_opened = False
    browse_strategies = [
        ("a link or button to Browse the database", ["link", "push-
            button", "button"]),
        ("a link or button labeled Explore to navigate entries", ["
            link", "push-button", "button"]),
        ("a main Search input field to search compounds in the
            database", ["entry", "textbox"]),
        ("a navigation element for Compounds or Entries", ["link", "
            push-button", "button"])
    ]
    for bdesc, btypes in browse_strategies:
        if browse_opened:
            no_op()
        else:
            bfind = find(Instruction(text=bdesc, length=200))
            if bfind.start is None:
                bfind = find_element_by_text(description=bdesc,
                    element_types=btypes)
            if bfind.start is not None:
                bclick = left_single(bfind.start, "Open browsing/
                    search interface on the database site")
                if bclick.status == "OK" or bclick.status == "UNKNOWN"
                    :
                    browse_opened = True
                    wait()
                else:
                    no_op()
            else:
                no_op()

    if not browse_opened:
        # Try a few scrolls to reveal browse/search options
        for _ in range(3):
            if browse_opened:
                no_op()
            else:
                scroll(direction="down", start=None, instruction="
                    Reveal more database navigation options")
                wait()
                for bdesc, btypes in browse_strategies:
                    if browse_opened:
                        no_op()
                    else:
                        bfind2 = find(Instruction(text=bdesc, length
                            =200))
                        if bfind2.start is None:
                            bfind2 = find_element_by_text(description=
                                bdesc, element_types=btypes)
                        if bfind2.start is not None:
                            bclick2 = left_single(bfind2.start, "Open
                                browse/search after scrolling")
                            if bclick2.status == "OK" or bclick2.
                                status == "UNKNOWN":
                                browse_opened = True
                                wait()
                            else:
                                no_op()
                        else:
                            no_op()
```

```
else:
    # If not clearly on a database site yet, try once more to follow
        any link that indicates a natural products database
    if not db_found:
        alt = find_element_by_text(
            description="any link that leads to a natural products
                database website",
            element_types=["link"]
        )
        if alt.start is not None:
            alt_click = left_single(alt.start, "Open alternative
                natural products database")
            if alt_click.status == "OK" or alt_click.status == "
                UNKNOWN":
                wait()
            else:
                no_op()
        else:
            no_op()

# Final verification and completion
final_text = get_page_text(max_length=2000, include_navigation=False)
final_verification = verify_hypothesis(
    observation=final_text.text,
    hypothesis="We are now on a natural products database page with a
        browse or search interface visible, allowing us to browse
        entries."
)

if final_verification.status == "OK":
    # Optional: check_done before marking done
    done_check = check_done(
        Instruction(
            text="I am on a natural products database website with a
                browse or search interface, ready to browse entries.",
            length=200
        )
    )
    if done_check.done:
        mark_done()
    else:
        mark_done()
else:
    mark_fail()
```

I.2 Natural Products database example of an insufficient plan (P-LLM: Gemini 3 Pro)

Task: Browse the natural products database. This plan is lacking branches, is too confident, and does not focus on website navigation but rather on going back into search and finding other databases it thinks fits the task description.

```
# Step 1: Observe the current state
summary = summarize_screenshot_content(Instruction(text="Describe the
    current page content, checking if it is a search engine, a
    specific website, or if there are cookie popups."), length=300),
    length=300)
print(f"Current page state: {summary.text}")

# Step 2: Handle Cookie Popups (Common in browser tasks)
# We check for this first to ensure elements aren't blocked
cookie_check = verify_hypothesis(
    observation=summary.text,
    hypothesis="A cookie consent popup, privacy notice, or GDPR banner
        is visible on the screen."
```

```
)

if cookie_check.status == "OK":
    print("Cookie popup detected. Attempting to accept.")
    # Try to find the accept button using common labels
    cookie_handled = False
    cookie_labels = ["Accept all", "Accept cookies", "I agree", "
        Consent", "Allow all", "OK"]

    for label in cookie_labels:
        if cookie_handled:
            no_op()
        else:
            btn = find_element_by_text(label, element_types=["button",
                "push-button"])
            if btn.start is not None:
                left_single(btn.start, f"Click {label}")
                wait()
                cookie_handled = True

    # If text search failed, try visual search
    if not cookie_handled:
        btn = find(Instruction(text="Accept or Agree button in cookie
            popup", length=100))
        if btn.start is not None:
            left_single(btn.start, "Click Accept visually")
            wait()

    # Refresh summary after handling cookies
    summary = summarize_screenshot_content(Instruction(text="Page
        content after cookie handling", length=300), length=300)

# Step 3: Determine Location and Navigate
# Check if we are already on a Natural Products Database
is_on_database = verify_hypothesis(
    observation=summary.text,
    hypothesis="The page is a Natural Products Database website (e.g.,
        NPASS, COCONUT, NP-MRD, SuperNatural) or has scientific data
        about natural compounds."
)

if is_on_database.status == "OK":
    print("Already on a database website. Proceeding to browse.")
    # Look for navigation options to browse data
    nav_clicked = False
    nav_options = ["Browse", "Explore", "Data", "Compounds", "Search"]

    for opt in nav_options:
        if nav_clicked:
            no_op()
        else:
            link = find_element_by_text(opt, element_types=["link", "
                button", "menu item"])
            if link.start is not None:
                left_single(link.start, f"Click {opt}")
                wait()
                nav_clicked = True

    if not nav_clicked:
        # If no specific link found, scroll to browse content
        scroll("down", None, "Scroll down to browse the page content")
        wait()

    mark_done()
```

```
else:
    # Not on a database. Check if we are on search results.
    is_search_results = verify_hypothesis(
        observation=summary.text,
        hypothesis="The page shows search results for 'natural
            products database' or similar queries."
    )

    # If not on search results, we need to perform a search
    if is_search_results.status == "FAIL":
        print("Not on database or search results. Initiating search.")

        # Find search bar
        search_bar = find_element_by_text("Search", element_types=["
            entry", "textbox"])
        if search_bar.start is None:
            search_bar = find(Instruction(text="Search bar or input
                box", length=50))

        if search_bar.start is not None:
            left_single(search_bar.start, "Click search bar")
            type_text("natural products database\n", "Search for
                natural products database")
            wait()
            # Assume we are now on search results
            is_search_results = ActionCall(status="OK")
        else:
            print("Could not find search bar to navigate.")
            mark_fail()

    # Step 4: Select a Database from Search Results
    if is_search_results.status == "OK":
        print("On search results page. Looking for a valid database
            link.")

        # Look for reputable databases
        target_found = False
        # Candidates: NPASS, COCONUT, NP-Atlas, SuperNatural
        candidates = [
            "NPASS",
            "COCONUT",
            "Natural Products Atlas",
            "SuperNatural",
            "Natural Product Activity and Species Source"
        ]

        for cand in candidates:
            if target_found:
                no_op()
            else:
                link = find_element_by_text(cand, element_types=["link
                    ", "heading"])
                if link.start is not None:
                    left_single(link.start, f"Click {cand}")
                    wait()
                    target_found = True

        # Fallback if no specific name found
        if not target_found:
            generic_link = find_element_by_text("Natural Products
                Database", element_types=["link", "heading"])
            if generic_link.start is not None:
                left_single(generic_link.start, "Click generic
                    database link")
                wait()
```

```
                target_found = True

        if target_found:
            # Step 5: Browse the selected database
            # Check for cookies on the new site
            new_summary = summarize_screenshot_content(Instruction(
                text="New page content", length=200), length=200)
            new_cookie_check = verify_hypothesis(observation=
                new_summary.text, hypothesis="Cookie popup visible")

            if new_cookie_check.status == "OK":
                accept_btn = find_element_by_text("Accept",
                    element_types=["button", "push-button"])
                if accept_btn.start is not None:
                    left_single(accept_btn.start, "Accept cookies on
                        new site")
                    wait()

            # Find Browse button
            browse_btn = find_element_by_text("Browse", element_types
                =["link", "button", "menu item"])
            if browse_btn.start is None:
                browse_btn = find_element_by_text("Explore",
                    element_types=["link", "button"])

            if browse_btn.start is not None:
                left_single(browse_btn.start, "Click Browse")
                wait()
            else:
                # Scroll if no browse button
                scroll("down", None, "Scroll to browse content")
                wait()

            mark_done()
        else:
            print("Could not find a relevant database link in the
                search results.")
            mark_fail()
```

I.3 Example of a generic cookie handler snippet

```
def_cookie_descriptions = [
    "a button to accept all cookies or consent to tracking in a cookie
        notice",
    "an 'I agree' or consent button in a cookie popup",
    "a button labeled 'Accept all' in a privacy banner",
    "a dismiss or OK button in a cookie notification popup"
]

# After potential navigation or app launch, re-observe
summary = summarize_screenshot_content(
    Instruction(
        text="Observe the current page to understand if we are on a
            website and what site it might be.",
        length=1200
    ),
    length=1200
)


cookie_check = verify_hypothesis(
    observation=summary.text,
    hypothesis="A cookie or privacy consent popup with accept/consent
        buttons is visible"
```

```
)
if cookie_check.status == "OK":
    cookie_found = False
    for desc in def_cookie_descriptions:
        if cookie_found:
            no_op()
        else:
            cookie_try = find(Instruction(text=desc, length=150))
            if cookie_try.start is None:
                cookie_try = find_element_by_text(description=desc,
                    element_types=["push-button", "button", "link"])
            if cookie_try.start is not None:
                cclick = left_single(cookie_try.start, "Accept cookies
                    /consent")
                if cclick.status == "OK" or cclick.status == "UNKNOWN"
                    :
                    cookie_found = True
                    wait()
                else:
                    no_op()
            else:
                no_op()
else:
    no_op()
```

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state three claims: (i) a plan-complexity gap between typed-API agents and CUAs (Figure 2, Table 3, Section 4), (ii) the first Dual-LLM architecture for CUAs with CFI guarantees and quantified utility (CaMeL-NOVA, Sections 3.1 and 4), and (iii) the identification and evaluation of Branch Steering attacks (Sections 3.2 and 4). Each claim is supported by the experimental section.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The paper's primary contribution is the Dual-LLM CFI architecture, which is structurally guaranteed; the redundancy-based defenses against Branch Steering (Sections 3.2 and 4) are presented as best-effort mitigations of an inherently hard problem, since semantic policies do not transfer to coordinate-level CUA actions (Sections 3.2 and 4) and any data-flow defense in this setting is necessarily probabilistic. We honestly document where these defenses succeed and where they fail to characterize the residual attack surface. In Section 5 we walk through what CFI still provides, task-data dependency, and under-specified tasks, and how each affects the feasibility of single-shot planning. Appendix F.1 quantifies the inference-cost overhead, with Fides-NOVA adding $29.6\times$ tokens over the undefended baseline.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.

- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The paper's contributions are empirical and architectural rather than theoretical. We do not claim formal theorems; CFI is established by construction through the Dual-LLM separation described in Section 3.1.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Section 4 describes the experimental setup including the OSWorld benchmark, the specific CUA backends and planner models evaluated, step limits, and pass@k metrics. Appendix B details more of the methodology, toolset, and prompt-engineering decisions. Appendix D.3 specifies pixel-attack hyperparameters (optimizer, learning rate schedule, EOT noise, iteration count). Algorithm 2 formalizes the execution loop.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.

- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We provide our adaptation of CaMeL and Fides for the OSWorld setting with the submission. The integration builds on the official open-source CaMeL, Fides, and OSWorld codebases. We do not release the attacks we constructed for ethical reasons. We do not release the constructed attack artifacts (the spoofed redirect pages used in the cookie attack and the optimized pixel banner from Appendix D.3) for dual-use reasons; the attack methodology and hyperparameters are fully documented in Appendix D so that researchers building defenses can reproduce the techniques without us distributing ready-to-deploy exploits.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

Justification: Section 4 specifies model versions, step caps (15), history caps (15, 5 for OpenCUA), VLM output token caps (4096), and pass@k evaluation. Appendix D.3 details optimizer (Adam, lr 0.8), schedule (cosine annealing with 200-step warm restarts to lr_min = 8e-4), gradient clipping, EOT settings, and 2000 iterations for the pixel attack. Section B.1.4 details Fides parameters (max_steps=15, max_turn=70, max_variable_reuse=5).

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: Table 3 and Figure 2 report per-plan means with 95% confidence intervals across N=886 CUA plans and N=242 AgentDojo plans. Pass@k metrics in Table 1 are computed over k independent samples per task and provide a natural variance characterization.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: Appendix F.1 (Tables 9 and 10) reports input/output token counts and total monetary cost per defense configuration on a 17-task subset, computed at official OpenAI and Anthropic API rates. UITars and OpenCUA were deployed locally on a single A100; cost and time limitations forced our 15-step cap (Section 4, Baseline utility).

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.

- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research involves no human subjects and uses only publicly available benchmarks and models. As dual-use security research, we follow responsible research practices: attacks are demonstrated only against our own evaluation infrastructure (OSWorld benchmark VMs and self-hosted spoof pages), not against deployed systems or third parties; we withhold ready-to-deploy attack artifacts while documenting the methodology in Appendix D so defenders can reason about and mitigate the threat; and the paper's framing centers on defensive architecture, with attack analysis used to characterize the residual surface that CFI does not cover.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Section 5 discusses positive impacts including privacy-preserving deployment (open-source perception, proprietary planning) and a stable plan interface that other defenses can compose with. Most fundamentally, CFI structurally eliminates the arbitrary-instruction injection class entirely, and Appendix H enumerates how the residual attack surface narrows: attack difficulty grows with both the number of coordinated Q-VLM manipulations required and the specificity of plan alignment needed, so consequential goals such as data exfiltration or fund transfers require coordinated multi-call compromise of task-specific plans, while simpler redirection succeeds only against generic utility routines. Branch Steering remains a residual risk class that defenders must continue to address; we partially mitigate it via redundancy-based defenses and characterize the surface in detail to support future defensive work.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.

- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [Yes]

    Justification: We release no new pre-trained models, image generators, or scraped datasets. The pixel attack and cookie-banner attack are demonstrated against our own benchmark setup; we do not release optimized adversarial banners or hosted spoofed.html infrastructure. The attacks are presented to motivate defenses, and Appendices D and H enumerate the structural conditions under which each attack succeeds, enabling defenders to reason about exposure.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We cite all benchmarks (OSWorld, AgentDojo), Dual-LLM substrates (CaMeL, Fides), CUA models (UITars-1.5-7B, OpenCUA-32B, Claude Sonnet 4.5), and planner models in Section 4. All used assets are publicly available under open-source or API terms which we respect.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We release our code, system prompts, and plan analysis pipeline (Appendix E.2); documentation is provided alongside the released code.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The research does not involve crowdsourcing or human subjects. All evaluations are run on the OSWorld and AgentDojo benchmarks with automated verification.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The research does not involve human subjects and therefore no IRB review applies.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.

- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: Our paper's original methodological contribution is the agentic framework CaMeL-NOVA, which adapts the Dual-LLM architecture to Computer Use Agents through single-shot planning and the `verify_hypothesis` primitive. Within this framework, LLMs and VLMs serve as components rather than as the contribution itself: a Privileged Planner LLM emits the plan, and a Quarantined Perception VLM processes environment observations within the plan. Section 4 names the specific models we evaluate in each role: UITars-1.5-7B, OpenCUA-32B, Claude Sonnet 4.5, GPT-5, GPT-5.1, Grok-4, Gemini 3 Pro, Gemini 2.5 Pro, DeepSeek-R1, Kimi-K2-Thinking, GPT-OSS-120b, and Claude Haiku 4.5.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.