Language-Based Agent Control

Timothy Zhou, Loris D'Antoni, Nadia Polikarpova
Department of Computer Science and Engineering
University of California-San Diego
La Jolla, CA 92037

Abstract

This paper introduces *language-based agent control* (LBAC), a new programming model for agentic applications that brings techniques from programming languages and language-based security to the problem of agent control. In conventional programming, combinations of static typing and runtime enforcement have long been used to guarantee that well-typed programs satisfy user-specified *policies*, including policies for access control, information flow, data provenance, and more. The key idea behind LBAC is to extend these guarantees to agentic applications by *requiring agents to generate programs that are themselves well-typed* in the context of the surrounding scaffolding code. Unsafe programs are rejected by the typechecker before execution, allowing policies to apply uniformly across the entire application, including both agent-generated behavior and developer-written scaffolding. At the same time, LBAC preserves substantial expressiveness: agents may perform arbitrary side-effect-free computation and recursively invoke subagents, which retain full tool access subject to the same—or potentially more restrictive—policies. We demonstrate LBAC with three case studies: I/O sandboxing via filesystem capabilities, data provenance, and information-flow control.

1 Introduction

Mainstream agent frameworks [28, 2, 18, 46, 8] force programmers to trade off **expressiveness** against **control**. *Tool*-based systems offer strong *control* but limited *expressiveness* by restricting the agent to a fixed set of actions, making safety and policy enforcement easier, but limiting composition and arbitrary computation. In contrast, sandboxed interpreters—such as Python or shell environments—maximize *expressiveness* by allowing agents to write programs, but only *control* coarse-grained system policies (e.g., file-system or network access). Such interpreters struggle to enforce application-level policies such as data provenance ("all entries you add to my bibliography must come from a trusted database"), information flow ("data from customer A should never be sent to customer B"), or policies scoped to sub-tasks rather than the entire agent ("spawn one sub-agent per customer; each sub-agent only has access to the data owned by that customer").

We introduce a new programming model for agentic applications, *language-based agent control* (LBAC), that reconciles *strong control* with *high expressiveness*. The key observation is that programming languages research has long used *types* to rule out unsafe behavior in programs [44, 4, 35]. The central claim of LBAC is:

Type systems can enforce the safety of code with agents.

The key idea of LBAC is to *embed agents into a pure, type-safe programming language*[1], such that (1) the agent communicates with the world by writing code in the language; (2) the scaffolding code

[1] A pure language has no side effects: that is, all the effects a program might perform are reflected in its type. We illustrate this concept in Sec. 2. Common examples of pure languages include Haskell and Lean.

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

that comprises the agentic application is *also* written in the same language; (3) policies are specified using type signatures. Unlike approaches that require agents to generate all code upfront [12, 21], LBAC allows agents to dynamically issue tool calls in response to data. Other approaches [7, 17, 37] rely on custom harnesses built around a specific policy enforcement mechanism—by contrast, LBAC enables existing policy enforcement mechanisms designed for ordinary code to be natively reused for agents. Thus, our approach naturally supports a wide range of policies, including fine-grained capabilities (e.g., file-system or network access), data provenance and information-flow constraints, and per-task policies that restrict individual subcomputations. LBAC is the first approach to make the agent implementation portable across existing language-based policy enforcement mechanisms. Moreover, the guarantees provided by these mechanisms extend to the entire agentic application.

Contributions. We make the following contributions:

(1) A new programming model for agentic applications, *language-based agent control* (LBAC), which reconciles strong control with high expressiveness by embedding agents in a pure, typed programming language and expressing policies as types over programs (Sec. 3).

(2) ELBA*, a practical realization of LBAC*, implemented in Haskell.

(3) Case studies and evaluation demonstrating the breadth and effectiveness of LBAC. We instantiate LBAC for three policies: data provenance, file-system capabilities, and information flow control (IFC). Our IFC case study notably reuses the mature LIO library [38, 40], inheriting its guarantees and illustrating LBAC's ability to build on prior programming languages research. Across a range of models and attack settings, LBAC achieves higher task utility on AgentDojo [11] than existing systems such as CaMeL, while providing stronger guarantees across the full agentic loop (Sec. 4).

2 Motivating Example

We start by motivating LBAC through a simple example that positions it relative to the mainstream approaches—sandboxed interpreters and restricted tools.

As a running example, suppose a programmer is implementing a literature search agent that produces BibTeX bibliographies based on a user request. The programmer wants to enforce the following constraint: *every BibTeX file the agent outputs should contain only data sourced from DBLP*. To accomplish this task effectively, the agent needs to be able to (1) inspect bibliography data and (2) apply appropriate computations over the data (e.g., sorting by year or filtering by a venue).

To support this task, the programmer implements a small BibTeX-manipulation library: `dblpSearch` returns the DOIs of papers matching a request, `dblpFetchBib` retrieves the full BibTeX entry for a given DOI, and `appendToBibFile` appends an entry to a file (Fig. 1, top). For now, ignore the type signatures next to each function; they will only come into play once we present LBAC.

Code interpreter: expressive, but unconstrained. A natural and expressive way to expose this library is through a sandboxed code interpreter, in which the agent generates and executes arbitrary code that calls these functions. The leftmost panel of Fig. 1 shows what such a session might look like: the agent writes a small program that searches DOIs, fetches their entries, and selects the earliest one. Throughout this paper, we use Haskell syntax for code snippets for consistency, but for this code-interpreter baseline the agent-generated code should be read with Python-like semantics: dynamically typed, with unrestricted I/O. Consequently, nothing prevents the agent from forgoing our safe `appendToBibFile` function and instead calling `writeFile` from the standard library directly, with a BibTeX string it constructed itself, fabricating an entry that never came from DBLP.

Restricted tools: safe, but inflexible. We could instead enforce stricter control by exposing *only* our BibTeX library as separate tools, with no access to a code interpreter. To prevent fabrication, we replace `appendToBibFile` with a coarser-grained tool, `fetchAndAppend`, that fetches a DOI and writes its entry to the file in a single combined step; `writeFile` is no longer available. The middle panel of Fig. 1 shows the result. This design indeed ensures that all written entries come from DBLP, but at the cost of expressiveness: the agent can no longer use code to compute over results, and must instead rely on the model to inspect entries one by one in order to find the earliest one.

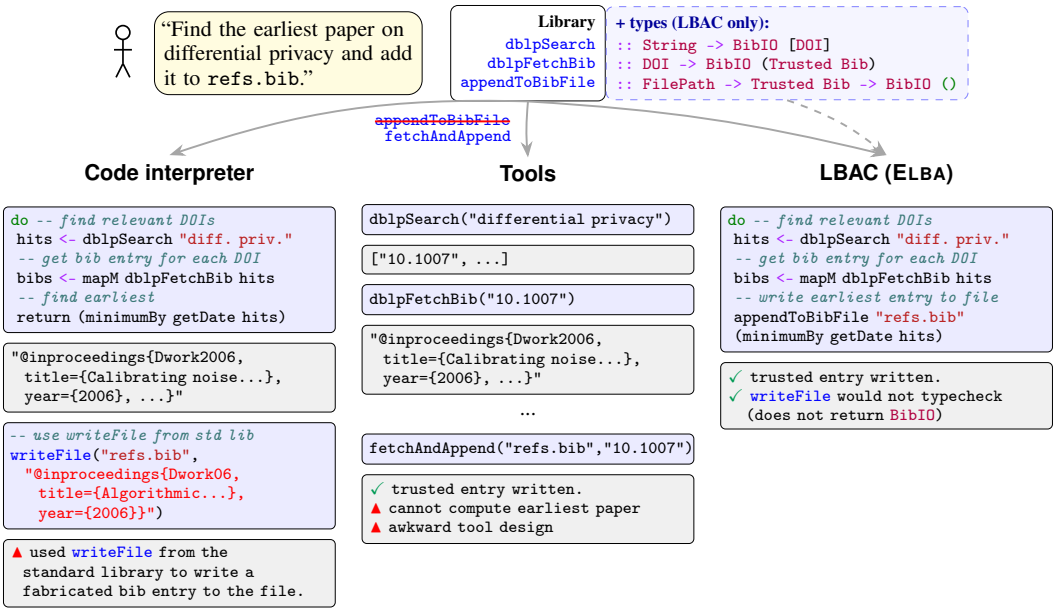


Figure 1: Implementing a literature search agent using a code interpreter, specialized tools, and ELBA (our approach). **Sandboxed interpreters** (left panel) enable expressive program composition, but allow the agent to bypass the library and fabricate entries using the raw file-write function from the standard library. Restricted **tools** (middle) enforce strong control by restricting access to trusted operations, but limit expressiveness; our example requires modifying the set of tools so that the only tool we provide for writing to file also fetches the bib entry directly from the database. **ELBA** (right) reconciles this tradeoff: the agent can write programs and compose operations freely, while the type system enforces that all written entries originate from trusted sources.

LBAC: expressiveness *and* control. Language-Based Agent Control (LBAC) is a new approach that uses properties of programming languages to enforce policies on agent behavior. A programmer using ELBA implements the literature search agent by defining an *embedded domain-specific language* (EDSL) in which the agent expresses its operations. An EDSL is a library which uses language features to enforce policies on programs using the library. The EDSL acts as a controlled interface between the agent and the tools: for our literature agent, the BibTeX library's type signatures are designed so that the type system enforces the policy that all written entries originate from DBLP. The programmer uses two features of Haskell to enforce the policy:

- *Abstract data types.* An abstract data type is a type whose representation is hidden from code outside its defining module. Abstract data types let us define an opaque wrapper type `Trusted` whose constructor is hidden inside the library, and hence `Trusted` objects cannot be constructed or modified directly by the agent. The only way for the agent to obtain a value of type `Trusted Bib` is by calling `dblpFetchBib`. Because `appendToBibFile` expects a `Trusted Bib`, the function accepts only records fetched from DBLP.
- *Purity.* Haskell uses types to restrict which effects a computation can perform. An *effect* is an interaction with external state, such as reading or writing a file. Standard-library functions such as `writeFile` carry the type `IO`, which permits, for example, arbitrary input/output operations such as file access and network communication. Our programmer instead defines a custom effect type `BibIO`, so the `IO` computation produced by `writeFile` cannot be used as a `BibIO`.

  The final ingredient is to require the agent's program to live within this EDSL, which the programmer does by setting the target type of the agent's response to `BibIO ()`. As a result, any program that combines our three library functions with arbitrary effect-free code, as in Fig. 1 (right), typechecks, while any attempt to call `writeFile` anywhere in the program, as in Fig. 1 (left), fails to typecheck and is rejected by ELBA.

3 LBAC Approach

This section describes the LBAC programming model. While the examples in this section use our Haskell implementation, ELBA, the results of this section apply to any pure typed host language.

Fig. 2 gives an overview of the LBAC approach, using the running example from Sec. 2. In this programming model, the developer writes their application (the so-called "scaffolding") in a pure, typed language, such as Haskell, using an LBAC library, such as ELBA. The library exposes a single function, `agent :: Defs -> Prompt -> t`, that takes as input (1) `Defs`: a set of imports (types and functions) the agent has access to; (2) `Prompt`: a string representing the user prompt; and returns an output with an *arbitrary* output type `t`, which is determined by the context in which `agent` is called. In Fig. 2, the programmer calls `agent` with (1) the types and functions from our BibTex library (`DOI`, `Bib`, `BibIO`, `dblpSearch`, `dblpFetchBib`, etc.); (2) the prompt that includes the generic instructions and the user query. The programmer also specifies the expected type of the result via an inline type annotation `:: BibIO ()`.

At run time, when the `agent` call is evaluated, three things happen (Fig. 2, right). *First,* ELBA queries the language model to generate a Haskell program, giving it the prompt, the available definitions, and a concrete expected type `T`. *Second,* once a program `p` is generated, ELBA typechecks it against the expected type `T`. If the program fails to typecheck, the type error is returned to the model, which retries. *Third,* once the program passes the type check, it is executed using the Haskell interpreter, and its result is returned as the output of the `agent` call.

3.1 Specifying Policies with Types

Readers are likely familiar with types as a way of describing the structure of data (e.g., distinguishing integers from strings). In *pure* languages, however, types do much more: they also constrain the *effects* a program may perform.

Restricting effects. Consider the following agentic program that implements an AI calculator:

```
main = do   print "Enter a math problem:"
            problem <- readLine
            result <- agent mathDefs problem :: Int
            print result
```

Because the target type of the `agent` call is `Int`, the agent is not only required to return an integer, it is also forbidden from performing any effect at all—no I/O, state updates, or exceptions. Consequently, it is safe to deploy this calculator as a public web service: if an untrusted user enters a malicious query like "delete all files on the server, then compute the factorial of 5," the LLM might generate a program such as `do system "rm -rf /"; return (factorial 5)`, but this program is rejected at the typechecking step. The reason is that in Haskell, `system`—and any other function performing I/O—returns a type wrapped in the `IO` effect, and once any subexpression of a program has type `IO`, the entire program must have type `IO`, which does not match the expected type `Int`.[2]

Customizing effects with EDSLs. Of course, an agent that can compute but cannot perform *any* effect is rather boring; our bibliography agent, for instance, needs to fetch papers from DBLP and append entries to a file. The standard programming-languages tool for this situation is an *embedded domain-specific language* (EDSL): a library that defines custom effect types together with the operations that may produce them, thereby exposing only those side effects sanctioned by the policy [31, 20]. The bottom of Fig. 2 illustrates this pattern: the BibTeX EDSL from Sec. 2 introduces a custom effect `BibIO` together with three operations, and combines it with the abstract data type `Trusted Bib` to encode the provenance policy. The effect type controls *what* the agent can do (only the three sanctioned operations, plus anything effect-free), while the abstract data type controls *which values* can flow into those operations (only entries that originated from DBLP).

[2] Standard Haskell has a few escape hatches, such as `unsafePerformIO`, that would in principle let an agent break out of a pure type. This loophole, however, has long been plugged by language-based security researchers: *Safe Haskell* [41] is a sub-language that excludes all such unsafe features and is selected via a compiler flag, so we can restrict agent-generated code to this safe subset.

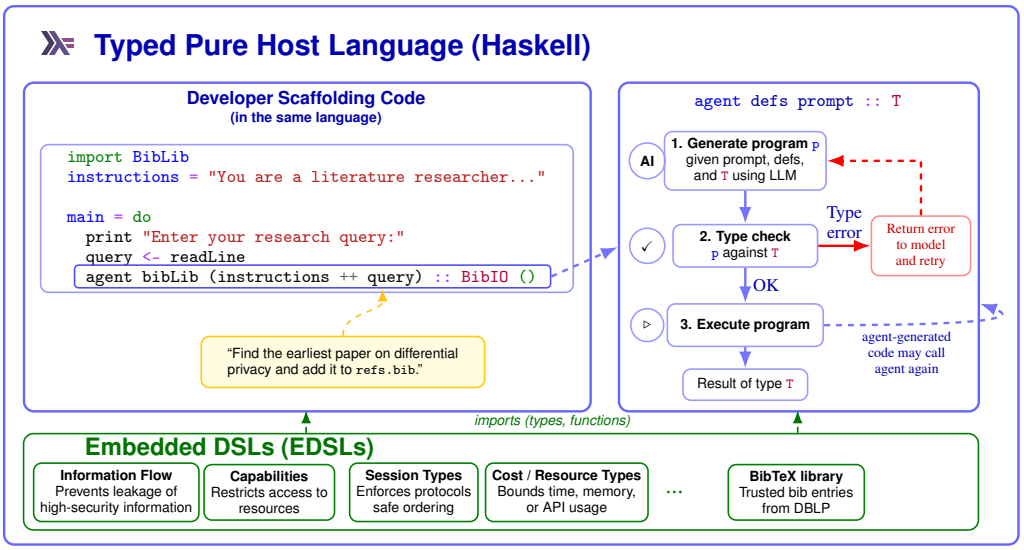


Figure 2: Overview of the Language-Based Agent Control (LBAC) model. The entire agentic system is expressed within a single typed host language (in the case of ELBA, Haskell): developer-written scaffolding, agent-generated programs, and library code all live in the same environment. An agent call generates code, which must typecheck before execution. As a result, the type system acts as the outer layer of control, ensuring that any property it enforces (e.g., capabilities, information-flow control, or resource bounds) holds for the entire system. Because ELBA is built in Haskell, it can directly leverage the rich ecosystem of existing type systems.

Reusing existing EDSLs. A programmer using ELBA can either define their own EDSL—as we did with `BibIO`—or reuse existing libraries that encode policies as types. In general, language-based techniques rely on a combination of type system features (Haskell supports many through its extension system [42]) and runtime checks. Examples from the literature include sandboxing untrusted code [41], capability-based access control [24], enforcing that operations follow a state-machine protocol [34], avoiding data races [14], and enforcing predicates on values [44]. Sec. 4 substantiates this claim by reusing the LIO library for information-flow control [40] completely off the shelf.

3.2 Programming with Agents

Because in LBAC, agents are ordinary functions, we can use the full power of functional programming—higher-order functions, recursion, and type inference—to compose agents into data-processing pipelines and multi-step workflows, *without stepping outside the type system*.

Composing agents. Higher-order functions make it easy to plug agents into standard combinators. For example, the following program summarizes a long document chunk by chunk using Haskell's `foldr` (reduce) function:

```
let summarize chunk summary = agent defs ("Extend this summary: " ++ summary
                                          ++ "\nwith a summary of: " ++ chunk)
in foldr summarize "" (chunks document)
```

Note that we did not need to annotate the expected type of the `agent` call: because the seed argument of `foldr` is the empty string, Haskell infers that `summarize` must return a `String`, and ELBA communicates this inferred type to the model.

This automatic type propagation is more than a convenience: it lets the developer write short prompts that would otherwise be ambiguous. Consider:

```
let f s = agent defs ("parse this: " ++ s) in
sortBy compareByYear (map f rawDates)
```

"Parse" as what—a date, a timestamp, a duration? Without type information, the LLM would have to guess from the surrounding prose. But Haskell's Hindley–Milner type inference deduces `f :: String -> Date` from `compareByYear`, and ELBA communicates this expected type to the model, so the agent knows it must produce a `Date`, even when an individual input is itself ambiguous (e.g., contains a time as well as a date).

Recursive agents. A careful reader might have noticed that ELBA's `agent` function does not implement the "agent loop"—the iterative interleaving of LLM reasoning, tool calls, and observation of intermediate results before the next step; our primitive instead generates a complete program in one shot. LBAC nonetheless supports the agent loop—and other iterative workflows—because agent-generated code can itself call `agent`, yielding a *recursive agent* [50]. Returning to the literature researcher of Fig. 2: rather than satisfying the user's query in one shot (as in Fig. 1, right), the agent might first peek at one search result to check that its keywords are reasonable and that the DBLP API is operational. In that case, the top-level `agent` call could generate the following program:

```
p1 :: BibIO ()
p1 = do hits <- dblpSearch "differential privacy"
        if null hits then return () -- no results
        else do
          bib <- dblpFetchBib (head hits)
          let newPrompt = ... ++ "First search result: " ++ show bib
          agent bibLib newPrompt
```

The recursive call carries the observation forward in its prompt, and the sub-agent decides whether to refine the search or write to the file.

A key consequence of this design is that *policies compose hierarchically*. The recursive `agent` call appears inside a `BibIO` do-block, so Haskell infers its target type to be `BibIO ()`, which means the sub-agent is bound by the same policy as its parent (`()` means that the program does not return a value). A parent may impose *stricter* policies on its children—for instance, by giving the recursive call an effect-free target type to forbid further I/O—but it cannot relax them: it cannot call `agent defs newPrompt :: IO ()`, for the same reason that the top-level agent cannot call `writeFile`.

Guarantees. The core idea of LBAC can be summarized by the following mantra:

If a host language enforces a property for all well-typed programs, and all agent-generated code must typecheck before execution, then that property holds for the entire agentic system.

For this guarantee to hold, it is essential that not only agent-generated programs, but the entire agentic system—including scaffolding code, control flow, and retry logic—is expressed in the host language. This is the most important difference from prior approaches: in LBAC, control is not implemented outside the language, but expressed within it. We formalize and prove this claim in Appendix B.

4 Evaluation

We evaluate LBAC through three case studies: data provenance, filesystem sandboxing, and information-flow control. Each case study follows the same pattern described in Sec. 3: (1) define a security property, (2) encode that property as a restricted effect type, and (3) expose an API for that type ensuring the property.

4.1 Data Provenance (continued from Sec. 2)

Property: `BibIO a` represents computations whose disk writes only contain data from DBLP and eventually produce a value of type `a`.

As discussed in Sec. 2, LBAC enforces this property by associating authority with types. Values obtained from DBLP are wrapped in an abstract type `Trusted`, whose constructor is hidden from untrusted code. As a result, programs cannot forge trusted values from arbitrary strings; they can only obtain them through approved DBLP APIs. Effects are similarly restricted. Output operations are

exposed only through the `BibIO` interface and require `Trusted` inputs. Consequently, any successful output produced by a well-typed `BibIO` program must derive from DBLP-approved data.

4.2 Capability-Based Filesystem Sandboxing

Property: `RIO a` represents computations that may only access files within authorized directories and eventually produce a value of type `a`.

For example, the `RIO` program on the right reads a file and writes its contents to a sibling location.

```
example :: Path -> RIO ()
example root = do
  contents <- readRIO (root // "input.txt")
  writeRIO (root // "output.txt") contents
```

This behavior is enforced via a *capability* discipline: rather than naming filesystem locations as raw strings, programs operate on values of type `Path`, which represent unforgeable tokens of authority over a particular directory subtree. Possession of a `Path` is itself the proof that the holder is permitted to access that path; without one, no I/O operation can be expressed in the first place.

In the example, `root` is such a token and `root // "input.txt"` narrows it to a specific file. Suppose instead we tried to read a file outside `root`'s subtree. Attempting to pass a string containing a filepath directly—e.g. `readRIO "/etc/passwd"`—would not even typecheck, since `readRIO` demands a `Path`, not a `String`.

Moreover, trying to construct a `Path` outside `root`'s subtree, as in the example on the right, would fail because `//` performs runtime path resolution: it follows symbolic links and rejects any result that would fall outside the original capability's subtree. Internally, `RIO a` is implemented as an opaque type wrapping an `IO a` which is gated behind capabilities. The full API is described in Appendix C.

```
attempt :: Path -> RIO String
attempt root =
  readRIO (root // "../../etc/passwd")
```

We argue that enforcing capabilities at the language level provides a clean abstraction for reasoning about filesystem privilege. Existing agent systems such as Claude Code typically implement sandboxing via two layers: an OS-level sandbox (e.g. `bubblewrap`), and permission systems which match bash commands against string patterns and/or ask for user approval. We next discuss how LBAC compares with each of these approaches.

LBAC vs. systems-level defenses. Our language-based approach is complementary to OS-level defenses, with differing tradeoffs. Unlike OS sandboxes, which confine a process to a static set of directories, capabilities are first-class values that can be constructed, transferred, and narrowed at runtime. For example, an agent can accept and return capabilities, or spawn subagents with strictly more restrictive ones. Capabilities can also generalize beyond physical filesystems to logical resources (for example, allowing access to all files on a git branch).

LBAC vs. permission rules. Approaches which are based on processing bash commands as strings have two flaws: they are a poor user-facing abstraction, and treating commands as strings is inherently brittle from a security standpoint. Conceptually, users typically want to describe a policy about file resources, for example "Claude can freely modify files in this directory". String-based permission systems instead reason at the level of individual commands, which often results in long, opaque approval prompts. For example it is not uncommon to see Claude code ask for approval for long bash commands, e.g.:

```
cd services/platform/identity/auth-provider/packages/api-client &&
sed -i 's/"timeout": 30/"timeout": 60/' config/client.json && python
scripts/run_integration_tests.py > logs/integration-$(date +%Y%m%d).log 2>&1
```

Moreover, string-based enforcement is inherently brittle: it requires faithfully reasoning about every syntactic pattern a bash command can contain. For example, a regression in Claude Code in March 2026 caused permission bypasses on compound shell invocations joined with `&&`, because only the first clause was matched against the deny list [33]. Agents with access to interpreters can also bypass rules entirely with a command like `python -c "open('denied.txt').read()"`, since the matcher has no visibility into the semantics of commands outside the shell. LBAC prevents the

first failure mode because enforcement follows from the type system rather than from heuristics based on parsing syntax. LBAC eliminates the second because the agent already operates within a programming language, and so has no need to call into a separate interpreter to perform computations.

4.3 Information Flow Control

Property: `DC a` represents computations that satisfy information-flow policies and eventually produce a value of type `a`.

LBAC integrates with LIO [40], a library for coarse-grained information-flow control (IFC). In IFC systems, data carries labels describing how it may propagate through the program. In our setting, we use LIO's disjunction category (DC) labels [38] to track two pieces of information: confidentiality ("who may observe this data?"), and integrity ("what sources influenced this data?").

For example, LIO would throw an error on the example on the right. Every LIO computation is associated with a floating *current label* that summarizes the labels of all data observed so far. Reading from a source raises the current label to incorporate the source's label, so after the `httpGet`, the computation's integrity is tainted by `phony.com`. Writing to a sink requires that the integrity of the current label is higher than the integrity of the sink ("user") so LIO will throw a runtime error.

```haskell
badWrite :: DC ()
badWrite = do
  content <- httpGet "http://phony.com/data"
  -- curr. label's integrity now "phony.com"
  writeToUser content -- error
```

LBAC generalizes the dual-LLM pattern. The dual-LLM pattern of Willison [45] defends against prompt injection by partitioning the agent into a privileged LLM, which has tool access but never sees untrusted data, and a quarantined LLM, which processes untrusted data but has no tool access. Communication between the two is restricted to opaque variable references, so an injected instruction in untrusted data cannot reach a tool call.

In LBAC, the dual-LLM pattern is simply a specific use of existing IFC primitives. The `toLabeled` function in LIO runs a sub-computation in isolation, allowing its current label to rise as it observes untrusted data, and returns the result as an opaque `Labeled` value without affecting the outer computation's label. Wrapping a recursive call to `agent` in `toLabeled` therefore results in a `Labeled` output, which the privileged LLM cannot directly inspect without raising its label. A critical difference from dual-LLM is that *LBAC does not require the subagent to forgo tool use*. APIs written in LIO naturally gate effectful operations behind checks on the current label of the computation. Therefore, control-flow integrity follows from the label discipline rather than from a wholesale ban on subagent tool use.

To illustrate, suppose an agent invokes a subagent inside a `toLabeled` block to write a DM. The subagent can call tools and read untrusted data, but returns the message as a `Labeled DCLabel Message`, leaving the outer agent's current label unchanged. Instead of inspecting the message, the outer agent passes the labeled result to `sendDM`, a tool that operates directly on labeled data and enforces policies on the recipient and message contents:

```haskell
sendDM :: User -> Labeled DCLabel Message -> DC ()
```

Like the privileged agent in the dual-LLM pattern, the outer agent in our example can send the message without being exposed to untrusted data. However, unlike the dual-LLM pattern, which forbids all tool calls by a quarantined LLM, ELBA allows the subagent to call tools because information flow control restricts tool calls after the subagent reads untrusted data.

Comparison to CaMeL We compare ELBA to CaMeL on the Slack suite of AgentDojo [10], which measures *utility* (fraction of benign tasks completed) and *security* (fraction of injection attempts resisted). The Slack suite consists of 21 user tasks and 5 injection tasks. Attacks in AgentDojo use every possible (user task, injection task) pair for a total of 105 possible tasks. For baseline comparisons that do not involve attacks, we only use user tasks. We ran our experiments with Opus 4.7, Opus 4.8, GPT 5.4, and Sonnet 5. For both tools, we report pass@1.

As ELBA is implemented in Haskell and AgentDojo is written in Python, we connect the two through inter-process communication. Both ELBA and CaMeL can be run with or without IFC policies

enabled. Without policies, CaMeL still uses its dual-LLM architecture but performs no dataflow checks. ELBA likewise has two configurations: an unsafe API where the Slack API uses plain `IO`, and a policy-enforcing version where the API uses `LIO`. LIO primitives enable us to define a novel style of policy based on the following principle: whenever a model reads untrusted data, its privilege should lower to the privilege of the data's source. This is an example of a "low watermark" [5] policy. An interesting direction for future work that ELBA enables is to standardize benchmark policies across agent implementations using existing IFC systems such as LIO, rather than tying policies to each agent's bespoke IFC system.

Findings. ELBA achieves higher utility than CaMeL across nearly all models and attack settings, completing roughly twice as many tasks with policies enforced and ∼15% more without (Table 1 in Appendix A.1). ELBA's utility failures without policies are mostly due to model refusals.

For both CaMeL and ELBA, all evaluated models resisted all 105 AgentDojo attacks even without policy enforcement. However, prompt injection attacks are becoming increasingly sophisticated: for example, GPT-Red, OpenAI's automated red-teaming system, consistently circumvented GPT-5.5's defenses [29]. Nasr et al. [26] and Zhan et al. [49] argue for evaluating defenses against *adaptive* attacks: motivated by this, we simulate *worst-case* injections by forcing compromised subagents to execute attacker instructions.

Adversarial evaluation. In ELBA, all data is read through recursive calls to `agent`, which receive prompts of the form `Task: {task} Input: {input data}`. We modified ELBA's code so that whenever a subagent receives input data containing the string `TODO:` (the AgentDojo direct injection format), the prompt is replaced with only the input data. This simulates a *worst-case* injection: the subagent executes the attacker's instructions directly, rather than merely being exposed to them. Table 2 reports our results.

Table 2: Adversarial evaluation: injections resisted and tasks completed under attack (higher is better).

System	Model	Injections resisted[3] (/735)	Tasks completed (/735)
ELBA	Opus 4.7	735	351
	Opus 4.8	735	312
	GPT 5.4	735	338
	Sonnet 5	735	294
ELBA (no policy)	Opus 4.7	425	523
	Opus 4.8	460	557
	GPT 5.4	621	558
	Sonnet 5	540	494

Without policies, up to 42% of injections succeed depending on the model. With policies enforced, ELBA resists all injections.

Enforcing the policies reduces task utility because the policies may also rule out legitimate agent behavior (Table 2). Even with high false positive rates, sound deterministic defenses remain useful as conservative filters because they have *zero* false negatives relative to the enforced policy. As Kolluri et al. [17] argue, in human-in-the-loop systems deterministic defenses can be used to safely reduce the need for user approvals.

Systems implementing the dual LLM pattern such as CaMeL would also prevent prompt injection by making tool calls independent of data. However, this separation reduces utility as shown in Table 1 in Appendix A.1.

[3] AgentDojo has a scoring false positive in one injection task which affects 29 out of the 2940 runs: we manually reviewed all 29 cases and found no security violations. See Appendix A.3 for details.

5 Related Work

Language-based approaches to prompt injection. The closest work to LBAC is on language-based approaches to preventing prompt injection. Fides and Invariant [7, 3] use IFC to enforce confidentiality and integrity policies on tool calls. Prudentia [17] builds on Fides by allowing interactive endorsement of data and approval of individual actions. GAAP [37] is an agent execution environment that uses IFC to protect private data even from the model provider. TACIT [27] uses Scala's capture checking to enforce capabilities and a limited form of IFC on agent-generated code. LLMbda [13] adds agent primitives to a basic information-flow calculus, with a formal proof of noninterference.

An essential design principle of LBAC that separates it from these prior approaches is to decouple the agent implementation from the policy enforcement mechanism. This approach has two benefits: (1) LBAC enables whole-system policies that cover both agents (as in prior approaches) *and* the surrounding application code; and (2) it allows direct reuse of established systems such as LIO [40]. LIO has many features (global process clearance, privileges, concurrency, labels which track both integrity and confidentiality), many of which these systems re-implement parts of. Using LIO removes the need to trust a custom IFC implementation as there is a long line of work developing LIO and formal soundness proofs of correctness [39, 40].

Another approach taken by work such as CaMeL [12] and Guardians of the Agents [21, 1] is to generate a program upfront and directly apply language-based techniques (IFC and SMT verification) to the program. An important difference, however, is that both lines of work additionally impose an *architectural constraint* of *code and data separation*: the agent is supposed to generate code upfront, before reading any data, which severs the information flow from potentially malicious prompts to the code's control flow. By contrast, our approach allows the agent to interleave reading data with calling tools, with security enforced by the type system rather than by architectural separation.

Other applications of language-based techniques include PASS [9] and Safety Chip [48], which use Linear Temporal Logic (LTL) formulas to restrict the allowed execution traces of agents, via constrained decoding and runtime checks respectively.

Systems-level defenses. Network and filesystem sandboxes [6, 23] are commonly used by production systems and can complement language-based defenses. In general, compared to language-based approaches, which can attenuate policies based on application logic, systems-level defenses are more coarse-grained. MiniScope [51] is a framework for restricting tool-calling agents to user-authorized permissions. ceLLMate [22] is a browser sandbox that enforces policies at the HTTP request level. Chrome's Agent Origin Sets [30] restrict which web origins agents can read or modify.

LLM-based defenses such as Conseca [43] and IsolateGPT [47] rely on separate language models to generate policies which describe permissible actions and data accesses.

Programming-language foundations. ELBA builds on existing programming-language mechanisms rather than introducing new type machinery. The `RIO`, `TIO`, and `LIO` monads of Sec. 3 are direct instances of *language-based security*—the use of type systems and runtimes to enforce information-flow and capability policies—following the lineage of Sabelfeld and Myers [35], Schneider et al. [36], JIF [25], Safe Haskell [41], LIO [39, 40], and work on effect systems [31, 20].

6 Conclusion

We presented LBAC, a programming model for LLM agents that treats agent control as a type- and runtime-enforcement problem inside a unified host language. LBAC allows agents to interleave reasoning, tool use, and code generation dynamically, while enforcing information-flow and capability policies over the resulting execution. Our central observation is that mechanisms proposed for secure and controllable agents can be expressed naturally using existing programming-language techniques once generated code and scaffold execute inside the same typed runtime. This removes the need for architectural workarounds such as strict code/data separation.

More broadly, LBAC suggests that agent control is fundamentally a programming-languages problem. Richer type disciplines—including refinement types [44, 32, 19], linear types [4], amortized resource analysis [15], and graded types [16]—could naturally extend our LBAC implementation with guarantees such as bounded resources, one-shot authorizations, and termination properties.

Acknowledgments and Disclosure of Funding

The project is supported in part by a Microsoft Faculty Fellowship and NSF under grants CCF-2422214, CCF-2506134, CCF-2446711. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities.

Loris D'Antoni holds concurrent appointments as a Professor at the University of California San Diego and as an Amazon Scholar. This paper describes work performed at the University of California San Diego and is not associated with Amazon.

References

[1] Nada Amin and contributors. Guardians: A static verifier for AI agent workflows. https://github.com/metareflection/guardians, 2026. Open-source implementation of the proposal in Meijer [21].

[2] Anthropic. Build with Claude: Agents. https://docs.anthropic.com/en/docs/build-with-claude/agents, 2025. URL https://docs.anthropic.com/en/docs/build-with-claude/agents.

[3] Mislav Balunović, Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. AI agents with formal security guarantees. ICML Workshop on Next Generation of AI Safety, 2024. URL https://invariantlabs.ai/theme/research/ai_agents_with_formal_security.pdf.

[4] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL):5:1–5:29, 2018. doi: 10.1145/3158093. URL https://doi.org/10.1145/3158093.

[5] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, The MITRE Corporation, April 1977.

[6] Bubblewrap contributors. Bubblewrap: Low-level unprivileged sandboxing tool. URL https://github.com/containers/bubblewrap. Accessed September 29, 2026.

[7] 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.

[8] CrewAI Inc. CrewAI. https://github.com/crewAIInc/crewAI, 2024. URL https://github.com/crewAIInc/crewAI.

[9] Maxwell Crouse, Ibrahim Abdelaziz, Ramon Astudillo, Kinjal Basu, Soham Dan, Sadhana Kumaravel, Achille Fokoue, Pavan Kapanipathi, Salim Roukos, and Luis Lastras. Formally specifying the high-level behavior of LLM-based agents, 2024. URL https://arxiv.org/abs/2310.08535.

[10] 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.

[11] Edoardo Debenedetti, Jie Zhang, Mislav Balunović, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. Agentdojo: A dynamic environment to evaluate prompt injection attacks and defenses for llm agents, 2024. URL https://arxiv.org/abs/2406.13352.

[12] 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.

[13] Zac Garby, Andrew D. Gordon, and David Sands. The LLMbda calculus: AI agents, conversations, and information flow, 2026. URL https://arxiv.org/abs/2602.20064.

[14] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM, 2005. doi: 10.1145/1065944.1065952. URL https://doi.org/10.1145/1065944.1065952.

[15] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 357–370. ACM, 2011. doi: 10.1145/1926385.1926427. URL https://doi.org/10.1145/1926385.1926427.

[16] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. Resource-guided program synthesis. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 253–268. ACM, 2019. doi: 10.1145/3314221.3314602. URL https://doi.org/10.1145/3314221.3314602.

[17] Aashish Kolluri, Rishi Sharma, Manuel Costa, Boris Köpf, Tobias Nießen, Mark Russinovich, Shruti Tople, and Santiago Zanella-Béguelin. Optimizing agent planning for security and autonomy, 2026. URL https://arxiv.org/abs/2602.11416.

[18] LangChain AI. LangGraph. https://github.com/langchain-ai/langgraph, 2024. URL https://github.com/langchain-ai/langgraph.

[19] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. Flux: Liquid types for rust. *Proc. ACM Program. Lang.*, 7(PLDI):1533–1557, 2023. doi: 10.1145/3591283. URL https://doi.org/10.1145/3591283.

[20] Daan Leijen. Type directed compilation of row-typed algebraic effects. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 486–499. ACM, 2017. doi: 10.1145/3009837.3009872. URL https://doi.org/10.1145/3009837.3009872.

[21] Erik Meijer. Guardians of the agents. *Commun. ACM*, 69(1):46–52, 2026. doi: 10.1145/3777544. URL https://doi.org/10.1145/3777544.

[22] Luoxi Meng, Henry Feng, Ilia Shumailov, and Earlence Fernandes. cellmate: Sandboxing browser ai agents, 2026. URL https://arxiv.org/abs/2512.12594.

[23] Dirk Merkel. Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal*, (239), 2014.

[24] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, 2006. URL http://erights.org/talks/thesis/markm-thesis.pdf.

[25] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 228–241. ACM, 1999. doi: 10.1145/292540.292561. URL https://doi.org/10.1145/292540.292561.

[26] Milad Nasr, Nicholas Carlini, Chawin Sitawarin, Sander V. Schulhoff, Jamie Hayes, Michael Ilie, Juliette Pluto, Shuang Song, Harsh Chaudhari, Ilia Shumailov, Abhradeep Guha Thakurta, Kai Yuanqing Xiao, Andreas Terzis, and Florian Tramèr. The attacker moves second: Stronger adaptive attacks bypass defenses against LLM jailbreaks and prompt injections. In *35th USENIX Security Symposium (USENIX Security 26)*, pages 1467–1486, Baltimore, MD, August 2026. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity26/presentation/nasr.

[27] Martin Odersky, Yaoyu Zhao, Yichen Xu, Oliver Bračevac, and Cao Nguyen Pham. Securing agents with tracked capabilities. In *Proceedings of the ACM Conference on AI and Agentic Systems*, CAIS '26, pages 812–838. Association for Computing Machinery, 2026. doi: 10.1145/3786335.3813127. URL https://doi.org/10.1145/3786335.3813127.

[28] OpenAI. OpenAI agents SDK. https://github.com/openai/openai-agents-python, 2025. URL https://github.com/openai/openai-agents-python.

[29] OpenAI. GPT-Red: Unlocking self-improvement for robustness, July 2026. URL https://openai.com/index/unlocking-self-improvement-gpt-red/.

[30] Nathan Parker. Architecting security for agentic capabilities in Chrome. Google Online Security Blog, December 2025. URL https://security.googleblog.com/2025/12/architecting-security-for-agentic.html.

[31] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, Lecture Notes in Computer Science, pages 80–94. Springer, 2009. doi: 10.1007/978-3-642-00590-9_7. URL https://doi.org/10.1007/978-3-642-00590-9_7.

[32] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. Liquid information flow control. *Proc. ACM Program. Lang.*, 4(ICFP):105:1–105:30, 2020. doi: 10.1145/3408987. URL https://doi.org/10.1145/3408987.

[33] Pro777. Chained bash commands silently bypass the tool permission allowlist. GitHub issue #36637, anthropics/claude-code, March 2026. URL https://github.com/anthropics/claude-code/issues/36637. Accessed: 2026-05-06.

[34] Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 25–36. ACM, 2008. doi: 10.1145/1411286.1411290. URL https://doi.org/10.1145/1411286.1411290.

[35] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Sel. Areas Commun.*, 21(1):5–19, 2003. doi: 10.1109/JSAC.2002.806121. URL https://doi.org/10.1109/JSAC.2002.806121.

[36] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. A language-based approach to security. In Reinhard Wilhelm, editor, *Informatics - 10 Years Back. 10 Years Ahead*, Lecture Notes in Computer Science, pages 86–101. Springer, 2001. doi: 10.1007/3-540-44577-3_6. URL https://doi.org/10.1007/3-540-44577-3_6.

[37] Robert Stanley, Avi Verma, Lillian Tsai, Konstantinos Kallas, and Sam Kumar. An ai agent execution environment to safeguard user data, 2026. URL https://arxiv.org/abs/2604.19657.

[38] Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Disjunction category labels. In Peeter Laud, editor, *Information Security Technology for Applications - 16th Nordic Conference on Secure IT Systems, NordSec 2011, Tallinn, Estonia, October 26-28, 2011, Revised Selected Papers*, Lecture Notes in Computer Science, pages 223–239. Springer, 2011. doi: 10.1007/978-3-642-29615-4_16. URL https://doi.org/10.1007/978-3-642-29615-4_16.

[39] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in haskell. In Koen Claessen, editor, *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*, pages 95–106. ACM, 2011. doi: 10.1145/2034675.2034688. URL https://doi.org/10.1145/2034675.2034688.

[40] Deian Stefan, David Mazières, John C. Mitchell, and Alejandro Russo. Flexible dynamic information flow control in the presence of exceptions. *J. Funct. Program.*, 27:e5, 2017. doi: 10.1017/S0956796816000241. URL https://doi.org/10.1017/S0956796816000241.

[41] David Terei, Simon Marlow, Simon L. Peyton Jones, and David Mazières. Safe haskell. In Janis Voigtländer, editor, *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*, pages 137–148. ACM, 2012. doi: 10.1145/2364506.2364524. URL https://doi.org/10.1145/2364506.2364524.

[42] The GHC Team. *GHC User's Guide: Language extensions*. URL https://downloads.haskell.org/ghc/latest/docs/users_guide/exts.html. Accessed September 29, 2026.

[43] Lillian Tsai and Eugene Bagdasarian. Contextual agent security: A policy for every purpose, 2025. URL https://arxiv.org/abs/2501.17070.

[44] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for haskell. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 269–282. ACM, 2014. doi: 10.1145/2628136.2628161. URL https://doi.org/10.1145/2628136.2628161.

[45] 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. Accessed: 2026-05-06.

[46] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen LLM applications via multi-agent conversation, 2023. URL https://arxiv.org/abs/2308.08155.

[47] Yuhao Wu, Franziska Roesner, Tadayoshi Kohno, Ning Zhang, and Umar Iqbal. Isolategpt: An execution isolation architecture for llm-based agentic systems. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025. URL https://doi.org/10.14722/ndss.2025.241131.

[48] Ziyi Yang, Shreyas S. Raman, Ankit Shah, and Stefanie Tellex. Plug in the safety chip: Enforcing constraints for LLM-driven robot agents, 2023. URL https://arxiv.org/abs/2309.09919.

[49] Qiusi Zhan, Richard Fang, Henil Shalin Panchal, and Daniel Kang. Adaptive attacks break defenses against indirect prompt injection attacks on LLM agents. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 7116–7132, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-naacl.395. URL https://aclanthology.org/2025.findings-naacl.395/.

[50] Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models, 2026. URL https://arxiv.org/abs/2512.24601.

[51] Jinhao Zhu, Kevin Tseng, Gil Vernik, Xiao Huang, Shishir G. Patil, Vivian Fang, and Raluca Ada Popa. Miniscope: A least privilege framework for authorizing tool calling agents, 2025. URL https://arxiv.org/abs/2512.11147.

A Evaluation Data

The code for LBAC, our experiments, our policies, and the raw data are available in our public GitHub repository: `https://github.com/timothytmzhou/elba/tree/neurips`.

A.1 AgentDojo utility and security

Table 1: Utility on the AgentDojo Slack suite (tasks completed; higher is better).

System	Model	No Attack (/21)	Direct (/105)	Important Instr. (/105)	Tool Instr. (/105)
ELBA	Opus 4.7	10	52	51	50
	Opus 4.8	9	49	47	45
	GPT 5.4	10	51	48	49
	Sonnet 5	8	43	46	40
ELBA (no policy)	Opus 4.7	18	88	84	92
	Opus 4.8	16	93	90	92
	GPT 5.4	16	84	80	79
	Sonnet 5	14	78	78	79
CaMeL	Opus 4.7	5	24	23	22
	Opus 4.8	6	30	29	30
	GPT 5.4	5	23	26	25
	Sonnet 5	4	25	23	23
CaMeL (no policy)	Opus 4.7	15	69	69	73
	Opus 4.8	16	78	82	82
	GPT 5.4	17	82	82	84
	Sonnet 5	13	56	58	55

We ran CaMeL in both "strict" and "normal" mode, and found essentially no difference (all cells within 4% of each other). We report strict mode.

A.2 Confidence intervals

Table 3 reports 95% confidence intervals for the difference in task completion rates between ELBA and CaMeL. We compare the two systems on the same tasks and, under attack, the same injections. Table 4 reports 95% confidence intervals for task completion rates and injection resistance rates without policies in the adversarial evaluation. We compute bootstrap confidence intervals using the results from the seven benchmark runs.

Table 3: 95% confidence intervals for the difference in task completion rates (ELBA − CaMeL), in percentage points, computed using Newcombe's score interval for the difference in paired proportions.

Model	Policies	No attack	Direct	Important Instr.	Tool Instr.
Opus 4.7	Yes	$[+2.9, +41.8]$	$[+17.3, +35.3]$	$[+17.4, +35.3]$	$[+17.4, +35.3]$
Opus 4.8	Yes	$[-7.4, +34.1]$	$[+9.9, +25.8]$	$[+8.7, +25.2]$	$[+6.3, +22.0]$
GPT 5.4	Yes	$[+2.9, +41.8]$	$[+17.4, +35.3]$	$[+12.0, +29.4]$	$[+13.6, +31.5]$
Sonnet 5	Yes	$[-4.7, +40.4]$	$[+7.9, +26.0]$	$[+12.4, +30.9]$	$[+7.9, +24.3]$
Opus 4.7	No	$[-8.4, +35.9]$	$[+8.0, +27.9]$	$[+4.3, +24.0]$	$[+9.6, +26.7]$
Opus 4.8	No	$[-16.9, +16.9]$	$[+6.4, +22.5]$	$[+0.5, +15.1]$	$[+2.5, +17.0]$
GPT 5.4	No	$[-24.0, +14.5]$	$[-5.5, +9.3]$	$[-8.8, +4.9]$	$[-11.4, +1.7]$
Sonnet 5	No	$[-22.7, +31.3]$	$[+11.2, +30.1]$	$[+8.4, +29.0]$	$[+12.8, +32.2]$

Table 4: Task completion rates and 95% confidence intervals for the adversarial evaluation.

Model	Policies	Tasks completed (%)		Injections resisted (%)	
		Rate	95% CI	Rate	95% CI
Opus 4.7	Yes	47.8	[46.1, 49.3]	100.0	N/A
Opus 4.8	Yes	42.4	[41.4, 43.4]	100.0	N/A
GPT 5.4	Yes	46.0	[44.2, 47.6]	100.0	N/A
Sonnet 5	Yes	40.0	[38.6, 41.4]	100.0	N/A
Opus 4.7	No	71.2	[70.1, 72.2]	57.8	[56.3, 59.3]
Opus 4.8	No	75.8	[74.6, 77.0]	62.6	[62.2, 63.1]
GPT 5.4	No	75.9	[73.9, 78.0]	84.5	[82.4, 86.9]
Sonnet 5	No	67.2	[65.0, 69.3]	73.5	[72.8, 74.1]

For the Table 1 experiments, both systems resisted all 105 injections in every model and attack setting, including without policies. For Table 2, security with policies is enforced deterministically, so it is not a statistical quantity.

A.3 Adversarial evaluation scoring

AgentDojo has a scoring false positive in one injection task which affects 29 out of 2940 runs: it counts accessing a static URL as a successful injection. Our policies do not prevent accessing the URL itself as it is not a security violation. Our policies would deny a request whose URL contained a parameter leaking data, a request made after the agent had accessed sensitive data (because requests may exfiltrate information about the LLM's context), or a subsequent privileged action based on the page's contents (due to the possibility of prompt injections). We manually reviewed all 29 cases and found no security violations.

A.4 Retries and runtime

ELBA retries code generation when an agent produces a program that fails to typecheck. Retries apply only to typechecking errors, which occur before any tool calls execute, and not runtime errors. Agent calls required an average of 0.1 retries and a maximum of 3 retries. A run may contain multiple agent calls. If the LLM repeatedly fails to produce well-typed code, a runtime error is thrown. This happened in 0.26% of runs. LIO may also throw runtime errors because it is a dynamic information-flow control system (policies are not checked statically).

Table 5: Number of retries on AgentDojo.

Model	Avg retries per run	Runs exceeding retry limit	% of time on retries
opus-4.7	0.05	0/672	0.7%
opus-4.8	0.14	5/672	5.0%
sonnet-5	0.14	2/672	2.3%
gpt-5.4	0.19	0/672	4.4%
all	0.13	7/2688	3.4%

The total runtime of ELBA on AgentDojo is about $3\times$ that of CaMeL's: this is primarily due to the fact that CaMeL generates a single program upfront, whereas ELBA agents perform additional LLM calls to generate programs in response to data (which increases expressiveness, and ultimately, utility).

A.5 BibTeX evaluation

We evaluated the BibTeX data provenance case study from Sec. 4.1. The evaluation consisted of 100 tasks where an agent must search the web and write a BibTeX file of citations for a given computer science topic. A citation is hallucinated if its authors, year, or DOI are incorrect. We compared (1) an agent implemented in ELBA using opaque types to enforce data provenance and (2) a vanilla

`pydantic_ai` agent with a web search tool. Both agents used Sonnet 5. The pydantic-ai agent writes the citations itself from search results and memory, so it can invent details such as DOIs. As expected (Table 6), ELBA prevents citation hallucinations.

Table 6: BibTeX evaluation (hallucinated citations; lower is better).

System	Hallucinated Citations
pydantic-ai agent + web search tool	19/1002
ELBA	0/925

The tool-calling baseline also used about $4\times$ more LLM calls, $2.5\times$ more output tokens, and $65\times$ more input tokens.

B Soundness of LBAC

In this appendix we make precise the central soundness claim of LBAC—*any property the host type system enforces for well-typed programs is also enforced for programs containing* `agent` *calls, provided each agent-generated program typechecks before execution*—and prove it by reduction to the type soundness of the host language.

B.1 Host Language Properties

We work parametrically in a host programming language L supporting standard features like conditionals, functions, and recursion. L has expressions denoted e and types t, and a typing judgment $\Gamma \vdash_L e : t$ where Γ is a typing environment assigning types to variables.

L has an operational semantics given by a small-step reduction relation $\sigma \to_L \sigma'$ over *configurations*. A configuration $\sigma = \langle e, \kappa \rangle$ combines a remaining program e with some auxiliary state κ (e.g., a store, a heap, an I/O history, a security label). An *execution* is a possibly infinite sequence of configurations of the form

$$a_1 \to_L a_2 \to_L a_3 \to_L \cdots .$$

A *trace* is the sequence of tool calls an execution makes which produce an observable side effect. The observable behavior of a program e is completely characterized by the set of traces it induces ($\mathrm{traces}(e)$).

A *type system property* for a type t is a predicate P on sets of traces such that

$$\forall e \in L. \quad \vdash_L e : t \implies P(\mathrm{traces}(e)).$$

B.2 Semantics of Agents

Given a language L, we define a new language L_{agent} which extends L with a new primitive $\mathtt{agent}_t^n$. Here, every agent call is annotated with two pieces of data: a return type t and a bound n on the depth of recursive subagent calls it can make. Languages such as Haskell that implement type inference can determine t from context and the depth is usually a parameter supplied by, e.g., a configuration file rather than written explicitly by the programmer.

The typing rules for L_{agent} are the same as L, except $\mathtt{agent}_t^n$ is given the type:

$$\mathtt{agent}_t^n : \mathsf{Defs} \to \mathsf{Prompt} \to \mathsf{Option}\ t$$

where Defs represents a set of definitions the agent can use in the code it constructs, Prompt is a string containing the agent's prompt, and the output type is Option t as the agent may fail to produce a result.

We model LLMs in a "black box" manner: we assume there is some arbitrary computable oracle function $\mathcal{O}_t$ which, given a depth n, a set of definitions defs, and a prompt prompt, outputs a program whose agent calls have depth at most n. While we assume that the function is completely arbitrary, we do not model the nondeterminism of the language model. Nondeterminism is irrelevant to the properties we consider in this paper, such as those in Sec. 4. For cases where determinism is required, one can implement a wrapper that caches the results of prompts to a language model.

The semantics of L_{agent} is the same as the semantics of L, but with an additional rule to execute agent calls. An $\texttt{agent}_t^n$ expression evaluates by querying the oracle for a program which may contain agent calls up to depth $n-1$. If the program typechecks under the given definitions, the agent expression evaluates to that program wrapped in Some to indicate success. Otherwise it evaluates to Nothing. Γ_{defs} is a typing environment for the given definitions.

$$\frac{\mathcal{O}_t(n, \mathsf{defs}, \mathsf{prompt}) = e_{\mathsf{gen}} \quad \Gamma_{\mathsf{defs}} \vdash e_{\mathsf{gen}} : t}{\langle \texttt{agent}_t^{n+1}\ \mathsf{defs}\ \mathsf{prompt}, \kappa\rangle\ \rightarrow\ \langle \mathsf{Some}(e_{\mathsf{gen}}), \kappa\rangle}$$

$$\frac{}{\langle \texttt{agent}_t^{n}\ \mathsf{defs}\ \mathsf{prompt}, \kappa\rangle\ \rightarrow\ \langle \mathsf{Nothing}, \kappa\rangle} \quad \text{otherwise}$$

We write o_t for an implementation of $\mathcal{O}_t$ in L. o_t calls an LLM and parses its output into a datatype representing syntax trees. We assume L has a function eval_t which, given an element of the syntax tree datatype and a target type t, typechecks the tree and if successful evaluates it into a value of type Option t.

Theorem B.1 (LBAC Soundness)**.** *Every type system property for L is a type system property for L_{agent}.*

Proof. Let P be a type system property for L at type t. Let e be a program in L_{agent} with $\vdash_{L_{\mathsf{agent}}} e : t$. Our goal is to show that $P(\mathrm{traces}(e))$ holds as well.

The main idea is to show the existence of a program e_L in L satisfying:

1. $\mathrm{traces}(e_L) = \mathrm{traces}(e)$
2. $\vdash_L e_L : t$

Then since P is a type system property for L, we must have $P(\mathrm{traces}(e_L))$ which implies $P(\mathrm{traces}(e))$.

At a high level, we will replace every $\texttt{agent}_t^n$ call in the L_{agent} program e with a call to an L term a_t^n which uses o_t to implement the oracle call. The main difficulty is that the oracle may return programs containing further agent calls which themselves need to be substituted. We therefore define a_t^n recursively as follows:

$$a_t^0\ \mathsf{defs}\ \mathsf{prompt} = \mathsf{Nothing},$$

$$a_t^{n+1}\ \mathsf{defs}\ \mathsf{prompt} = \begin{cases} \mathsf{eval}_t\big(e_{\mathsf{gen}}[\texttt{agent}_s^m \mapsto a_s^m]\big) & \text{if } \Gamma_{\mathsf{defs}} \vdash e_{\mathsf{gen}} : t, \\ \mathsf{Nothing} & \text{otherwise,} \end{cases}$$

$$\text{where } e_{\mathsf{gen}} = o_t(n, \mathsf{defs}, \mathsf{prompt}).$$

The substitution replaces every $\texttt{agent}_s^m$ expression in e_{gen} with a_s^m. The term o_t by assumption ensures $m \leq n$, so the definition of a_t^{n+1} uses only terms already defined at smaller depths.

Next, we define e_L as:

$$e_L = e[\texttt{agent}_s^m \mapsto a_s^m \mid \texttt{agent}_s^m \text{ occurs in } e].$$

Every a_s^m is an L term with the same type as $\texttt{agent}_s^m$, so e_L is an L program with $\vdash_L e_L : t$.

Finally, we prove that $\mathrm{traces}(e) = \mathrm{traces}(e_L)$. It suffices to show that replacing any $\texttt{agent}_t^n$ call with a call to a_t^n preserves the traces of the program containing the call. We proceed by induction on n for all result types t. At depth zero, a_t^0 and $\texttt{agent}_t^0$ return Nothing. For depth $n+1$, assume the claim holds at all depths at most n. Since o_t computes $\mathcal{O}_t$, we obtain the same generated program e_{gen} in the calls to a_t^{n+1} and $\texttt{agent}_t^{n+1}$. Each call returns Nothing if e_{gen} does not typecheck. Otherwise, every generated call has depth at most n, so the induction hypothesis applies to every substitution in the recurrence. □

C `RIO` API

- `evalRIO :: String -> RIO a -> IO a` is the entry point to the API: given a root directory and an `RIO` computation, it runs the computation bound to a capability for that root.
- `Path` is an opaque type whose constructors are not exported: the only way to obtain a `Path` is via the API, ensuring that every `Path` in scope was derived from one originally granted by `evalRIO`.
- `// :: Path -> String -> Path` is a binary operator which narrows an existing capability: given a `Path` for `p` and a string `s`, it returns a `Path` for `p/s`. Crucially, `//` performs runtime path resolution: it follows symbolic links and rejects any result that would fall outside the original capability's subtree.
- `readRIO :: Path -> RIO String`, `writeRIO :: Path -> String -> RIO ()`, and `ls :: Path -> RIO [Path]` perform reads, writes, and directory listings, each requiring a `Path` as proof of authority.

D System Prompt

```
You generate Haskell code.

Output exactly one valid Haskell expression of the required type and
nothing else. The expression may span multiple lines (e.g. do-blocks,
let-bindings). You may use Prelude and the listed allowed functions.

# Delegating to a sub-agent

`subagent :: Typeable a => String -> String -> a` spawns a fresh LLM
turn returning a value of the annotated type. The two arguments are
the sub-agent's task instructions and a single data string (use `show`
or `Text.Printf.printf` to construct it). The data appears in an
`<input>` section of the sub-agent's prompt.

Use `subagent` to **observe, think about, and act on** computed
values. The common shape: bind the result of a computation to a
variable (with `<-` or `let`), then pass that variable in as the data
argument for a data-dependent decision, then act on the answer
yourself. Make some progress before delegating; your output must not
be a bare `subagent` call that forwards the whole problem to another
round.

Always annotate the result, e.g. `subagent "..." "..." :: IO String` -
without an annotation GHC cannot infer the type and compilation fails.

The sub-agent shares your allowed functions and language extensions,
but sees only the two strings you pass - not your surrounding code,
your variables, or values you have already computed. Pass any data the
sub-agent needs as the second argument.

    do raw   <- getLine
       topic <- subagent
                  "Identify the topic in one word."
                  (show raw)
                :: IO String
       putStrLn ("topic: " ++ topic)

# Retry

If a follow-up message contains a GHC error, your previous code did
not compile. Re-emit a single corrected Haskell expression of the
same target type, and nothing else.
```

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 paper's main contributions are about the expressiveness and control of our framework and the our case studies showcase such features.

   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: Sections 2 and 3.1 discuss defining and reusing policy libraries. Section 4.2 discusses tradeoffs with OS-level defenses, Section 4.3 discusses model refusals and the utility costs of enforcing policies, and Appendix A.4 discusses retries and runtime overhead.

   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: [Yes]

   Justification: We provide a theorem with a proof.

   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: Our approach is described in detail and we provide an explanation of our experimental setup in Section 4 and Appendices A and D.

   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: Code, AgentDojo data, and policies are available on GitHub.

   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: See Section 4 and Appendices A and D, which detail our experimental setup.

   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: For the head-to-head utility comparisons between ELBA and CaMeL (Table 1), we report 95% confidence intervals for the difference in task completion rates (Table 3), computed using Newcombe's score interval for the difference in paired proportions. For adversarial utility, we report 95% confidence intervals across the seven repetitions (Table 4). We also report confidence intervals for adversarial injection resistance without policies (Table 4). Policy enforcement provides a deterministic security guarantee.

   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: We used hosted models through the OpenAI API and AWS Bedrock. All AgentDojo runs had a 10-minute timeout. Model details are provided in Section 4.3.

   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: We believe our paper does not pose any immediate societal, human, or harmful impact and follow the code of ethics.

   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: Though not the main result, our work provides a more natural way of writing agents that respect security policies.

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: [N/A]

    Justification: We believe our paper does not pose any immediate harmful impact.

    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: All assets have been credited where necessary. AgentDojo, LIO, and pydantic-ai use the MIT license; CaMeL uses Apache 2.0. DBLP metadata is available under CC0 1.0, and Crossref bibliographic metadata is freely reusable.

    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: [N/A]

    Justification: This paper does not include any new assets outside of code and what is presented in the paper.

    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 paper does not involve crowdsourcing nor research with human subjects.

    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 paper does not involve crowdsourcing nor research with human subjects.

    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: [N/A]

    Justification: The core method development did not involve LLMs.

    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.