

ACE: A Security Architecture for LLM-Integrated App Systems

Evan Li*, Tushin Mallick*, Evan Rose*, William Robertson, Alina Oprea, and Cristina Nita-Rotaru

Northeastern University

{li.evan1, mallick.tu, rose.ev, w.robertson, a.oprea, c.nitarotaru}@northeastern.edu

Abstract—LLM-integrated app systems extend the utility of Large Language Models (LLMs) with third-party apps that are invoked by a system LLM using interleaved planning and execution phases to answer user queries. These systems introduce new attack vectors where malicious apps can cause integrity violation of planning or execution, availability breakdown, or privacy compromise during execution.

In this work, we identify new attacks impacting the integrity of planning, as well as the integrity and availability of execution in LLM-integrated apps, and demonstrate them against IsolateGPT, a recent solution designed to mitigate attacks from malicious apps. We propose **Abstract-Concrete-Execute (ACE)**, a new secure architecture for LLM-integrated app systems that provides security guarantees for system planning and execution. Specifically, ACE decouples planning into two phases by first creating an abstract execution plan using only trusted information, and then mapping the abstract plan to a concrete plan using installed system apps. We verify that the plans generated by our system satisfy user-specified secure information flow constraints via static analysis on the structured plan output. During execution, ACE enforces data and capability barriers between apps, and ensures that the execution is conducted according to the trusted abstract plan. We show experimentally that ACE is secure against attacks from the INJECAGENT and Agent Security Bench benchmarks for indirect prompt injection, and our newly introduced attacks. We also evaluate the utility of ACE in realistic environments, using the Tool Usage suite from the LangChain benchmark. Our architecture represents a significant advancement towards hardening LLM-based systems using system security principles.

I. INTRODUCTION

Large language models (LLMs) have shown remarkable performance in language generation [1]–[6], motivating their integration with external systems. This integration is commonly realized through third-party applications (or apps) that connect an LLM with external APIs to enable seamless interactions between users and third-party services. These LLM-integrated apps can support a variety of tasks such as booking flights, reserving restaurants, and managing emails.

Several major LLM orchestration frameworks [7]–[9] have emerged to facilitate the development of apps. These frame-

works provide centralized management of prompts and dynamic, iterative generation of multi-step LLM workflows. Specifically, a central “system LLM” iterates between successive *planning* and *execution phases*. Each planning phase decides the next operations towards answering a user query based on the results of prior execution steps. Given a plan, the LLM then carries it out in a subsequent execution phase, potentially invoking apps and accessing context to do so. Planning and execution phases are interleaved, resulting in dynamic control flow that is dependent on user instructions, app descriptions, and intermediate system outputs.

To support this dynamic orchestration, the system relies on structured representations in the form of app schemas and app descriptions. An app schema formally defines the structure, expected inputs and outputs, and operational interface of an app, while an app description provides semantic metadata about the app’s capabilities, behavior, and usage context. These representations enable the system LLM to reason about available functionality, plan appropriate execution, and coordinate the invocation of apps in accordance with user intent.

Security is a major concern for LLM-integrated app systems, as they introduce new attack vectors from malicious apps installed on user devices, including indirect prompt injection [10], denial of service and privacy leakage [11]. Based on the attacker objective and the system component being attacked we classify such attacks as (1) *integrity violation of planning* – attacks that impact the integrity of the planning phase; (2) *integrity violation of execution* – attacks that impact the integrity of the execution phase; (3) *availability breakdown of execution* – attacks that interrupt the normal execution of the LLM system; and (4) *privacy compromise of execution* – attacks that cause leakage of sensitive user information from the execution environment.

Recent advances in system-level defenses for LLM-integrated app systems focus on mitigating prompt injection and related security threats posed by untrusted third-party data sources. These defenses primarily leverage isolated execution or control how data propagates within an LLM-integrated app system. Information flow control mechanisms [12] enforce separation between trusted planning and untrusted execution, while isolation architectures [13] decouple application logic through modular components to prevent shared context compromise. However, existing defenses assume a *weak adversary* that cannot manipulate the app description and schema and use an interleaved plan-execute approach that does not establish

* Equal Contribution

sufficiently comprehensive security boundaries between the system LLM and untrusted third-party apps.

Motivated by limitations of existing defenses, we identify and demonstrate several concrete attacks that subvert the integrity of the system planning phase as well as the integrity and availability of the system execution phase of IsolateGPT [13]. Our attacks include *Execution Flow Disruption* and *Execution Manager Hijack* created through malicious app outputs, and *Planner Manipulation* created through malicious app descriptions. To address these new attacks, we design Abstract-Concrete-Execute (ACE), a new secure architecture for LLM-integrated apps that provides comprehensive security by design. ACE is based upon the key insight that ahead-of-time planning based only on the trusted user query—as opposed to dynamic plan generation—enables principled security reasoning and static enforcement of strong security policies on plan execution. An overview of our architecture is given in Figure 1b, in contrast to existing systems using interleaved planning and execution shown in Figure 1a.

ACE separates query processing into three distinct phases: *abstract plan generation*, *concrete plan instantiation*, and *isolated plan execution*. The first phase creates an abstract execution plan using only trusted query information, thus creating a security boundary that preserves plan integrity despite the presence of untrusted apps. This approach enables reasoning about the control and information flow properties of system execution traces under an immutable rule-based plan compared to a dynamic, data-dependent plan. The separation of planning and execution phases guarantees integrity of execution, including preventing indirect prompt injection attacks arising from malicious app outputs.

The second phase instantiates the plan using registered apps, leveraging isolation to prevent malicious apps from corrupting the integrity of the abstract plan. With a complete execution plan in hand, ACE then verifies that the plan satisfies static security policies including quantification of risk and permissible information flows between the system LLM, context, and apps. By verifying concrete plan implementations against our lattice-based policy, we automatically reject implementations that violate defined information flow constraints.

The final phase executes the verified plan, leveraging system isolation primitives and controlled interfaces between components to enforce the previously-verified security policies and overall integrity of execution with respect to the concrete plan. To summarize, our contributions are:

- We demonstrate three new attacks that subvert the integrity of the system planning phase as well as the integrity and availability of the system execution phase of IsolateGPT [13].
- We propose ACE, a new secure architecture for LLM-integrated app systems providing comprehensive security by design. ACE uses the key insight that planning based on only trusted components enables principled security reasoning and static enforcement of strong security policies on plan execution. Our abstract planning mechanism stands in stark contrast to the majority of existing LLM-

based systems, which follow an interleaved plan-execute procedure to decide execution and produce a response.

- We verify that the plans generated by our system satisfy user-specified secure information flow constraints via static analysis on the structured plan output. We demonstrate that our information flow verification system successfully blocks the accidental or malicious leakage of privileged information to unqualified recipients.
- We conduct experiments to empirically demonstrate ACE’s security benefits. We show that ACE successfully prevents all attacks from INJECAGENT [14] and ASB [15], standard benchmarks for evaluating indirect prompt injection attacks. We also show that ACE prevents our newly introduced attacks. In addition, we demonstrate that ACE achieves high utility (above 80%) on the Tool Usage suite from the LangChain benchmark.

A complete version of this work, containing supplementary appendices and extended results, is available on arXiv [16]. Our code is publicly available at <https://github.com/escottrose01/ace-llm>.

II. BACKGROUND AND PROBLEM STATEMENT

We provide an overview of LLM-integrated apps and details about existing defenses against malicious apps. We then describe our problem statement and goals.

A. Overview of LLM-Integrated App Systems

LLM-integrated app systems are structured around modular, composable components—primarily apps—that expand the LLM’s functionality to perform real-world tasks. We give an example of a typical LLM-integrated app system in Figure 1a. At the core of this architecture is a system LLM that interprets user queries, formulates execution strategies, and invokes the appropriate apps to fulfill task objectives. The system LLM operates over a dynamic prompt context including the user’s input, prior dialogue, app descriptions, and any intermediate results. This context functions as transient memory, allowing the model to reason over evolving task states, maintain coherence across steps, and ensure consistency in output.

Within this framework, an app is defined by three elements: a natural language description, a schema, and a function. The description specifies the app’s purpose and operational constraint, and serves as semantic metadata for app selection and planning. The schema defines the structure of the app’s inputs and outputs. The function, typically a script or service, implements the app logic—receiving structured inputs and returning either structured or natural language outputs.

Task handling involves two conceptual phases: planning and execution. Embedded within the system LLM is a planning mechanism responsible for decomposing high-level user intent into a structured execution strategy. The planner selects relevant apps, determines their invocation order, and supplies required inputs. The resulting plan serves as a blueprint for execution, supporting both single-step and multi-step workflows.

App execution is managed by an underlying execution environment, which enforces process isolation, resource limits, and

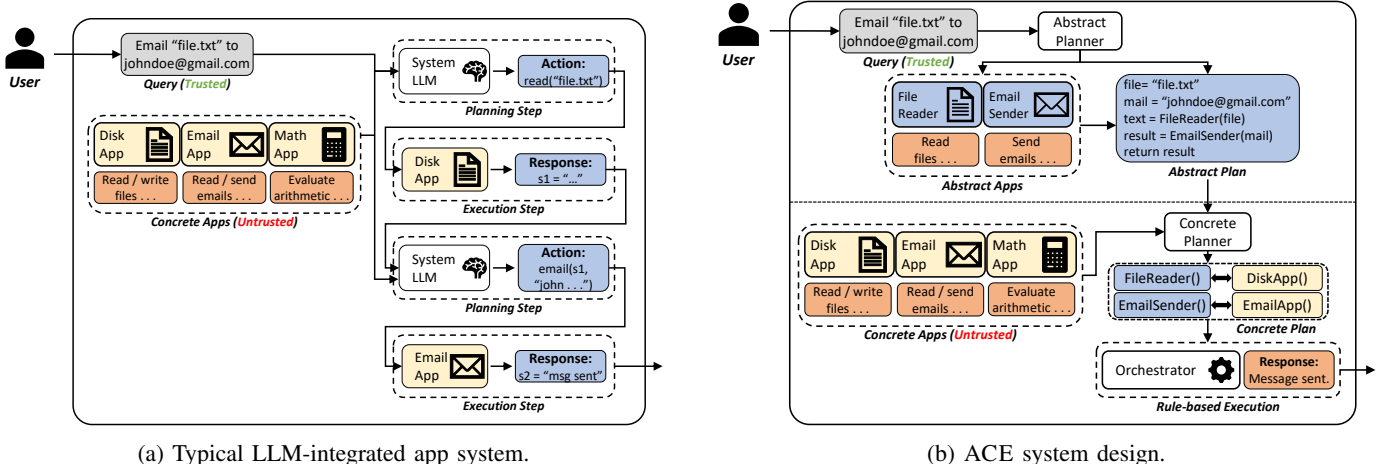


Fig. 1: Comparison of system architectures. In typical systems (left) a central system LLM is responsible for planning control flow based on the user queries and available system utilities. Planning and execution phases are interleaved, producing a control flow mechanism that is arbitrarily dependent on the user instructions, app descriptions, and intermediate system outputs. Our system ACE (right) generates a structured plan prior to execution based on trusted information.

secure system access. Within this environment, an orchestrator acts as an intermediary between the system LLM and the apps. The orchestrator receives the execution plan, schedules and manages app invocations accordingly, and oversees the data flow between apps. It also maintains an execution state that is logically independent from the reasoning process of the system LLM which ensures that high-level reasoning is decoupled from low-level operational control.

In more complex workflows, app chaining is needed, where the output of one app serves as the input to another. These multi-step executions introduce coordination challenges, including dependency tracking, validation of intermediate results, and maintaining type consistency across steps. The planner is responsible for explicitly encoding these dependencies within the execution plan, while the orchestrator handles data transformation and propagation between steps, ensuring consistency and system integrity throughout the process.

B. Existing Defenses for LLM-Integrated App Systems

While LLM-integrated apps enhance functionality and user experience, they also introduce significant security vulnerabilities—particularly through indirect prompt injection attacks [10]. These risks are amplified in systems involving multiple untrusted apps, where adversaries can exploit natural language ambiguity to compromise app integrity, mislead users, or violate privacy across multi-step execution chains.

Two LLM app security systems that attempt to address these issues are *f*-Secure [12] and IsolateGPT [13].

***f*-Secure [12].** This system provides a defense against indirect prompt injection attacks in LLM-powered apps by adopting information flow control (IFC). The core design of *f*-Secure involves separating LLM functionalities into a planner, which generates structured execution steps using only trusted inputs, and a rule-based executor, which processes potentially

untrusted data. A security monitor enforces IFC policies, preventing untrusted data from influencing planning.

The system relies on several trust assumptions, notably treating app descriptions and schemas as inherently reliable without verification. As a result, any compromise in these components can undermine the effectiveness of IFC and lead to insecure behavior.

IsolateGPT [13]. This system-level defense mitigates security risks from untrusted apps in LLM systems by enforcing strict app execution isolation. The architecture of IsolateGPT is centered around a strict app execution isolation model, implemented via a modular Hub-and-Spoke design. More details are given in Section III-A.

However, IsolateGPT’s reliance on static app descriptions and schemas as trusted sources presents a critical limitation. Since it lacks mechanisms for validating the integrity of these descriptions or inspecting the internal logic of app functions, it is constrained to verifying outputs based solely on expected formats and declared semantics. This limits its ability to reason dynamically or adapt to adversarial scenarios, ultimately affecting system robustness. Another limitation of IsolateGPT lies in its reliance on user interaction for app control, which introduces significant user fatigue.

C. Problem Statement

Our goal is to design a security architecture for LLM-integrated app systems that provides mitigation against malicious apps installed on a user’s device that might influence both the LLM planning and the execution flow of the LLM system. The main problem we address in our work is to restrict the influence of malicious apps in LLM systems by protecting benign apps and the LLM from their adversarial impact.

Threat Model. We assume that the attacker capabilities involve control over one or several apps on the user’s device, with the goal of influencing other benign apps or the LLM

TABLE I: Comparison of our system with existing LLM security systems based on what attack surfaces they are designed to address. We consider two adversaries: our strong threat model, which assumes completely untrusted apps, and a weaker threat model, which trusts the app description and schema.

Phase	Attack Objective	IsolateGPT [13]		<i>f</i> -Secure [12]		ACE (Ours)	
		Weak	Strong	Weak	Strong	Weak	Strong
Planning	Integrity	✓	✗	✓	✗	✓	✓
Execution	Integrity	✗	✗	✓	✗	✓	✓
Execution	Availability	✗	✗	✓	✗	✓	✓
Execution	Privacy	User-guided	User-guided	✗	✗	✓	✓

planning and execution components. Within the compromised apps, the attacker has total control over the details of their execution, their interface with the LLM system (schema), and app metadata, such as the name and natural language description. As a consequence of controlling the app execution, the attacker also controls malicious app outputs, which could result in an indirect prompt injection attack manipulating the control flow. We distinguish between a *weak threat model* in which the app description and schema are trusted, and a *strong threat model* in which they may be malicious.

We consider several attacker objectives of interest (availability, integrity, and privacy) during both the LLM planning and execution phases. While a combination of adversarial objectives and LLM phase leads to six possible attack types, we focus here on the most relevant:

- 1) **Planning Integrity Violation.** The attacker could manipulate the LLM planning, for instance to promote their own malicious apps to be included or to demote a benign app to be excluded from the generated plan.
- 2) **Execution Integrity Violation.** The attacker could attempt to change the system execution flow so that a benign app receives malicious output from a compromised app or manipulate the execution context, leading to an integrity violation in the system’s behavior.
- 3) **Execution Availability Breakdown.** The attacker may wish to interrupt the normal execution of the LLM system, causing user queries to fail to resolve despite the availability of suitable resources on the system.
- 4) **Execution Privacy Compromise.** The attacker might wish to cause leakage of sensitive user information from the execution environment.

It is possible to launch an availability attack during planning to prevent the plan generation and task completion, but such an attack would be easily detected. Privacy compromises are not relevant in the planning phase, but only during execution when the LLM gets access to sensitive user data.

Our Goals. We have two main types of goals: security goals (preventing attacks from malicious apps) and utility goals (maintaining system utility). As shown in Table I, existing defenses [12], [13] do not consider a strong threat model with arbitrary app manipulation. Even in the case of a weak threat model where app description and schema are trusted, they provide limited protection, and none of the existing systems is resilient against all attacker objectives.

Our system should preserve the security of both planning

and execution phases in the face of untrusted components. In particular, the integrity of the planning phase should not be compromised in the presence of untrusted apps installed on the system. App descriptions should not be able to induce arbitrary changes in the generated control flow. Additionally, the execution phase should prevent integrity, availability, and privacy compromises resulting from indirect prompt injections performed by malicious apps. The system should appropriately restrict the processing of untrusted data originating from system app outputs. The data flow in the system should be enforced according to the prespecified plan. Besides designing a system resilient against both weak and strong attacker models, our goals are to offer high levels of utility, and be agnostic to the system and LLM configurations.

III. NEW ATTACKS ON LLM-INTEGRATED APP SYSTEMS

Previous defenses for LLM-integrated app systems either trust the description of the app, or they trust the LLM to choose the apps and plan the execution of the task. We identify several new attacks against IsolateGPT [13] which we demonstrate on the public system implementation: (1) Execution Flow Disruption, (2) Execution Manager Hijack, and (3) Planner Manipulation. The first two attacks are created through malicious app output, while the third is created through malicious app descriptions. Below we describe in detail the IsolateGPT system, and how our attacks bypass its security mechanisms.

A. IsolateGPT System Overview

In IsolateGPT user queries are processed through a modular Hub-and-Spoke architecture that supports dynamic, multi-step task execution while enforcing strict execution isolation. When a user submits a query, it is first received by the hub, which orchestrates all downstream activity. The hub contains two key subsystems: the planner and the execution manager. Each subsystem is responsible for distinct phases of query interpretation and execution. An example of an end-to-end scenario from user query to final output is shown in Figure 2.

The hub planner component incorporates a planning LLM that interprets the user’s query and constructs a detailed multi-step execution plan taking into account the apps available to the system. This plan includes a proposed ordering of app calls, their expected inputs and outputs, and interdependencies. However, instead of executing this plan directly, IsolateGPT discards the detailed structure and retains only a high-level list of relevant apps identified as potentially useful for resolving

failure within the system, enabling malicious app developers to undermine the credibility of other apps and deliver misleading information to the user.

E. Planner Manipulation Attack

A malicious app can also exploit the fact that the system trusts app descriptions during the planning phase to manipulate the planner into selecting the malicious app despite the presence of other functionally equivalent alternatives.

Once the user gives a query to calculate the fare from “Main Street” to “Elms Avenue”, the planner develops a plan of execution to solve the query. To do this, the planner LLM reads the user query, the benign description of MetroHail, and the malicious description of QuickRide. While both apps’ descriptions state their purpose as fare calculators, QuickRide’s description also has a malicious prompt commanding the planner LLM to exclude MetroHail from the plan of execution when resolving fare calculation requests. The planner LLM reads this malicious prompt and excludes MetroHail from the created plan. In the user’s final output, only the fare generated from QuickRide is returned, removing MetroHail from the execution process without the user’s knowledge. This vulnerability leaves the hub planner susceptible to manipulation by malicious app developers, providing adversaries with the ability to promote their own product and demote their competitor’s, undermining the system’s reliability.

IV. ACE SYSTEM ARCHITECTURE

We start by discussing the design principles guiding the design of ACE. We then give an overview of our system architecture. Finally, we describe each component in detail and explain how they contribute to achieving our security goals.

A. Design Principles

One of the key challenges in designing a secure LLM system in the face of untrusted apps lies in how to create structured, rule-based execution plans while also limiting the extent to which installed apps can influence these plans. At a high level, we desire that the basic control flow determined by the planner cannot be altered by malicious app descriptions. This includes app demotion attacks such as the Planner Manipulation Attack from Section III. We also require that, once this plan is established, the execution phase is subject to the constraints imposed by the plan. That is, malicious app outputs cannot cause an indirect prompt injection attack resulting in arbitrary execution traces not permitted by the semantics of the prespecified plan. Finally, we want to prevent privacy leakage by design, so that data boundaries can be enforced and sensitive information cannot leak to unqualified parties. Thus, we are led to the following design principles:

Separate Planning and Execution. We showed with the Execution Flow Disruption Attack how an attacker could prematurely interrupt execution by performing an indirect prompt injection attack to insert a malicious output into the execution path. With the Planner Manipulation Attack we showed how a malicious app description could influence the

control flow by suppressing the use of a relevant app. This leads us to propose a *stricter boundary* between planning and execution, in which a planning module determines an execution workflow based only on fully-trusted information, such as the user query. This execution workflow imposes hard, irreversible constraints on the possible downstream execution paths, which cannot be modified by malicious app descriptions or outputs.

Remove Unintended Cross-app Interactions. In the Planner Manipulation Attack we showed how a malicious app can suppress the usage of a different, unrelated app by modifying its own description. We recognize this behavior more broadly as an *unintended cross-app interaction*. In particular, for the purposes of planning the broader control flow, the planning module should be able to determine the inclusion of each app independently from the others. Thus, we seek a solution which encodes this requirement explicitly in its design.

Enforce Data Controls within Execution Paths. LLMs cannot be trusted to keep flows of private and public information separate. Instead, our insight is to enforce privacy controls by design using rule-based data security controls. These controls should guarantee that privileged information is not divulged to unqualified locations during any execution trace, regardless of how the control flow was determined (even by a trusted component). The controls should also be extensive enough to detect and prevent long-range data dependencies, as data in multi-step plans can be processed in potentially complex ways which must be tracked.

Enforce Low-privilege Principle. A general, widely-accepted security guideline is the principle of least privilege (PoLP), which states that the privileges granted to an entity should be the minimal possible needed to perform its intended functions. Guided by this principle, our system should provide the least amount of privilege to apps during execution.

B. High-level Overview

ACE consists of three main components, shown in Figure 3: an *abstract planner*, a *concrete planner*, and an *executor*. Each component is responsible for handling a distinct phase of user query processing, each with less capability than the previous one. In this way, we balance the need for generality while restricting the influence of untrusted data sources.

The *abstract planner* is responsible for generating the overarching plan of execution for fulfilling the user query. It serves as the most privileged and trusted component of the system and interacts only with fully trusted information, the user query. In particular, the abstract planner is *oblivious* to the set of apps installed on the system, making it immune to indirect prompt injection and planning manipulation attacks. The output of the abstract planner specifies clearly-defined control flow rules governing downstream execution paths. Our insight in this direction is for the abstract planner to identify a set of *abstract apps* which can be used in expressing the execution plan. The resulting plan makes use of these abstract apps in defining the control flow of the program without

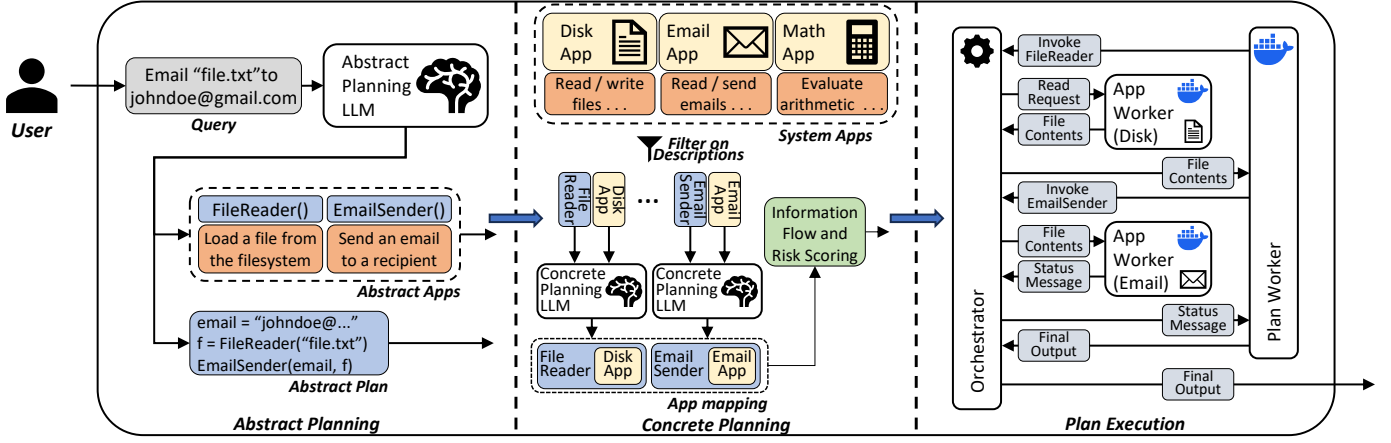


Fig. 3: Overview of our three-phase ACE secure LLM-integrated app system architecture. First, our system generates an *abstract plan* using a set of *abstract apps*, generated using only fully-trusted query information. Next, we match abstract apps with *concrete apps* installed on the system in the *concrete planning* phase. Matching consists of a binary decision made independently between each pair of abstract and concrete app. Finally, the concrete plan is *executed* in a carefully managed execution environment which enforces isolation between system app instances.

deferring to the untrusted information involved with installed system utilities.

The *concrete planner* acts as an intermediate step, combining the output of the abstract planner with the apps installed on the system to obtain a valid flow that can be executed. The output of the concrete planner must abide by any structural constraints imposed by the abstract planner. Briefly, the abstract apps identified during the abstract planning phase are matched with *concrete apps* installed on the system. We perform this matching carefully to eliminate unintended cross-app interactions such as app demotion attacks. This results in a *concrete plan* which fully specifies the needed system operations. At this phase, we also statically verify system-level security policies such as privacy controls on information flow between apps.

The *executor* runs the concrete plan within an orchestrator-worker architecture and is responsible for executing the concrete plan in a secure manner by enforcing all security policy rules. Each app is run inside an isolated environment with carefully managed permissions. Only data required for executing the app is made available to each app’s execution environment. Apps are restricted by default from interacting with each other or with other host system resources. In the executor we implement a distributed protocol between a trusted orchestrator and workers. The protocol defines a structured message flow between distributed components, where participants exchange messages according to predefined roles and state transitions.

Our system supports standalone apps and single-query. Supporting application *suites* and multi-query interactions are left for future work.

C. Abstract Planner

We propose a method of plan generation that depends only on knowledge of the user query and which is oblivious to information involving the set of installed apps. In particular,

our planning module is designed so that an attacker cannot influence the generated plan by having their app installed. Crucially, the abstract planning phase is performed without access to information involving the set of installed apps, and thus is by design secure from manipulation by installed apps. **Abstract Apps.** Motivated by the concepts of abstract classes and polymorphism from programming languages, the first task of the abstract planner is to generate a set of *abstract apps*. Abstract apps consist of a name, natural language description, and a type signature defining the input and output structure. Abstract apps do *not* implement the behavior stated in their descriptions. Given a user query, the planning module generates a set of abstract apps which may be relevant to completing the query.

We implement the abstract app generator using a specialized LLM (i.e., an LLM paired with a customized system prompt), which takes the user query as input and produces abstract app specifications in a structured output format.

To be useful, abstract apps must satisfy two criteria. First, the user intent must be expressible with some program logic using the abstract apps as building blocks. Second, the apps must be representative of utilities installed on the system. We observe that real-world apps naturally group into broad functional categories—such as file system interactions, text processing utilities, data retrieval, or computational operations—whose general functionalities can often be captured without requiring exact implementation details. Thus, by guiding abstract app generation to generate apps falling into such functional groups, we are able to create abstract apps which correspond to installed utilities, even without seeing the utilities themselves.

For example, a query of the form “summarize file.txt” may generate two abstract apps, `LOADDOCUMENT`, which is responsible for loading data from the host filesystem, and `SUMMARIZETEXT`, which applies summarization to a provided piece of text. By abstracting the key functionalities

required to fulfill a user query, abstract apps serve as a stepping stone to expressing a user’s intended outcome without prematurely committing to specific underlying implementations and without exposing an attack surface for untrusted information.

Abstract Plan. We introduce a specialized language, a modified subset of the Python language with plan-specific functionality added. Plans in this language are syntactically valid Python programs with a well-defined entry point for execution. Valid function calls include a restricted subset of the Python standard library in addition to a handful of utilities to facilitate planning with apps. An example of abstract plan is given in Figure 4.

The planning LLM is instructed to generate the plan using a custom system prompt. Our abstract planning framework contributes to achieving our security objectives in the following way. The abstract plan can be viewed as a *hard constraint* on the space of possible execution traces of the system. In particular, choosing a particular implementation for a given abstract app cannot drastically alter the overarching control flow of the underlying program. Any properties which can be gleaned from an abstract execution of the abstract plan are necessarily satisfied by any particular concrete plan implementing the abstract plan. Moreover, expressing plans in a language with precise semantics opens the door for static analysis to prove formal properties about the security and integrity of plan execution.

Every program in our abstract planning language contains a single top-level entry point definition ‘`main()`’. The logic expressed within the `main` function consists of basic statements as well as basic branching program control flow constructs. We support branching control flow in the form of if-statements, for-loops, and while-loops. The usage of these constructs is restricted to appropriately limit the capabilities implied by the planning language while retaining the general expressiveness of the planner. For-loops are restricted to “for-range” loops; that is, they only allow iteration over a (possibly variable) sequence of integer values. While-loops function as usual, but require the loop condition to be a single variable. Break statements are not allowed within either loop construct. These restrictions simplify downstream static analysis.

Our language runtime is similarly restricted to prevent unsafe data or control flows. We restrict builtin file system utilities, mutable data types, and dynamic code features to appropriately limit allowed runtime behaviors. More details on the language runtime are given in Appendix B.

Our abstract planning mechanism stands in stark contrast to the majority of existing LLM-based systems, which follow an interleaved plan-execute procedure to determine execution traces and produce a response [17]. We argue that it is much easier to reason about the control and information flow properties of system execution traces under an immutable rule-based plan than under a dynamic, data-dependent plan. Our design ensures that the abstract plan is not influenced by malicious apps, preventing indirect prompt injection attacks that manipulate the execution flow.

Operational Context. In some cases, more context may be

```
def main():
    doc: str = DocumentLoader(filename="file.txt")
    res: str = TextSummarizer(text=doc)
    display(f"The summarized document is: {res}")
    return res
```

Fig. 4: Example abstract plan for the user query “Load document ‘file.txt’ from my documents and summarize the contents.” `DocumentLoader` and `TextSummarizer` are abstract apps automatically generated by the planner and are not affected by the apps installed on the system.

needed to clarify the operational environment of the agent before an abstract plan can be generated. To resolve this, we expose an optional *context* field to the abstract planner. The context field originates from a fully-trusted source and clarifies both the operational environment in which the agent can take actions as well as broadly summarizes the expected capabilities the agent should expect to have within this environment. Because the context field is fully trusted, it must not contain explicit metadata from apps or application outputs.

D. Concrete Planner

The abstract plan utilizes abstract apps, but in order to execute the plan, the system must first generate implementations for each of the abstract apps. The *concrete planner* is responsible for replacing the abstract apps with the actual concrete apps registered by the user on the system. We define an *implementation* of the abstract plan to be a mapping from abstract apps to concrete apps; that is, every abstract app in the abstract plan should correspond to exactly one concrete app. The abstract plan and implementation together form the *concrete plan*, which fully expresses the structured control flow which can be executed on the system. The following describes how we determine such an implementation.

Concrete App Matching. We use a two-step process to generate implementations of abstract apps based on their descriptions and the concrete apps. First, we filter the set of concrete apps by thresholding the similarity scores between abstract and concrete app description embeddings. Our implementation uses the OpenAI text-embedding-ada-002 embeddings model [18] with the Euclidean distance similarity score. The purpose of the first step is to reduce the apps that must be considered for implementation to only include those that are relevant for a particular task. Second, we use a concrete planner mechanism to determine which filtered apps are capable of implementing each abstract app. The purpose of the second step is to conform discrepancies between type signatures as well as resolve any fine-grained semantic discrepancies between the abstract apps and the proposed implementations. An implementation of an abstract app must conform to the abstract app’s type signature, for both inputs and outputs. A priori, for some abstract app, there may exist reasonable implementations using concrete apps but with incompatible type signatures. For example, a concrete app could produce multiple outputs when the abstract

app only requires one, or the ordering of the arguments between the abstract app and the concrete app may not agree. To resolve these issues, we propose to use a compatibility layer which translates between the inputs and outputs of the concrete app and those of the abstract app. The translation process is highly dependent upon the natural language semantics of the involved concrete and abstract apps. Thus, we implement this step with another specialized LLM. We note that the LLM used for app matching can be different from the one used for planning, giving rise to a configuration space of LLMs which can be tuned according to desired performance-cost tradeoff.

The matching process induces a space of possible concrete plans. Each abstract app corresponds to a set of matched concrete apps which can implement it under a lightweight compatibility layer. All that remains is to choose for each abstract app, a matching to a concrete app. In principle, any such pairing will satisfy the intended semantics of the abstract plan. We prioritize concrete plans to enforce other security constraints, namely low privilege access (discussed in the extended version [16]) and secure information flow (discussed in IV-F).

E. Executor

After the concrete planning phase, the system possesses a plan detailing concrete steps for achieving the user query while adhering to user-prescribed security objectives. This plan includes the particular implementations of abstract apps as determined by concrete planner. In this section, we describe how to execute this plan securely from a systems perspective.

To enforce additional security in the execution phase, the executor is structured following a orchestrator-worker architecture which separates privilege management from execution. This design follows the principle of least privilege and further restricts the effect scope of malicious or faulty components. We view both the overall execution of the LLM-generated plan, as well as the execution of concrete apps, as possible points of system misuse, and therefore propose to execute these components in environments with carefully managed capabilities. We propose to use an *orchestrator* process to manage resource allocation and privilege enforcement during plan and application execution. The orchestrator spawns *worker* processes, each of which operates within its own isolated execution environment, ensuring separation from sensitive host system resources. To prevent resource misuse, these execution environments default to the most restrictive possible set of privileges while still enabling the required functionality.

Next, we describe in more detail the responsibilities and capabilities of the three main components of our executor system: the orchestrator, the plan worker, and the app worker.

Orchestrator. The orchestrator is the privileged entry point for the executor whose primary purpose is to manage execution environments for plan processing and app execution. For example, if a worker requires file system access, the orchestrator spawns an environment with only those privileges.

A secondary responsibility of the orchestrator is to handle message passing between workers. The orchestrator process

possesses the concrete plan, and so additionally performs data validation such as schema verification on worker inputs and type enforcement on worker outputs.

It is additionally responsible for overseeing the resource consumption of worker processes. In the event that an app worker consumes too many resources (for example, by exceeding a pre-set runtime limit), the orchestrator is responsible for terminating the execution of the violating worker and communicating the failure condition to the plan worker.

Plan Worker. The plan worker is responsible for sequentially processing the concrete plan. We implement the plan worker to execute the provided script inside a restricted containerized execution environment with no unnecessary privileges such as file system access. The plan worker’s execution process is strictly limited to communicating with the orchestrator over the network using socket-based connections, where the container exposes a network interface. Data exchange occurs through well-defined socket endpoints, allowing asynchronous and bidirectional communication across container boundaries. In this setting, the primary concern is not malicious behavior, but accidental system misuse resulting from faulty LLM-generated code. These restrictions help contain the effect of poorly generated or misconfigured LLM code, such as attempting to overwrite critical system files, or making unintended API calls.

The plan worker is responsible for overseeing the execution of the system plan, but does not itself have the ability to invoke system apps. In fact, under the application of principle of least privilege, it would be a security risk to expose certain capabilities, such as filesystem or network access, to the plan worker. Moreover, much like apps in the mobile platforms, each app in an LLM system may require a different set of privileges to fulfill its purpose. An app responsible for loading documents from the host system’s filesystem cannot function without filesystem access, yet most apps do not require filesystem access (and may not be trusted with such access). So, if the plan worker requires an app invocation it makes a blocking call to the orchestrator and waits until the orchestrator provides the app output.

App Worker. To support modularity, flexibility, and scalability in execution, the orchestrator employs Dockerized app workers, each encapsulating a distinct app within an isolated runtime environment with the necessary set of privileges. The app worker only exchanges data with the orchestrator using well-defined network sockets.

Each worker sends its output back to the orchestrator, which collects and routes these results back to the plan worker. This architecture enables loosely coupled interaction among apps, and ensures that intermediate results can be flexibly recomposed into subsequent execution stages.

F. Information Flow Control Security

ACE strictly enforces data privacy and integrity using a structured modeling of information flow constraints. LLM-based systems cannot be trusted on their own to prevent the leakage of private or sensitive information to unqualified destinations. Thus, we propose to systematically monitor and

enforce the qualified flow of information through our system. Our solution to this problem is to embed the desired security policy within a lattice and to statically analyze the generated concrete plan to verify that the plan semantics conform to the policy. Secure information flow formally specifies and enforces constraints on how data can flow through a system according to a defined security policy.

Modeling policies with lattices. We model the secure information flow policy as a universally bounded lattice $(\mathcal{C}, \sqsubseteq)$. The lattice consists of a set \mathcal{C} equipped with a partial order \sqsubseteq such that every pair of set elements $x, y \in \mathcal{C}$ has a least upper bound $x \sqcup y$, called the *join*, and a greatest lower bound $x \sqcap y$, called the *meet*. Semantically, the relation \sqsubseteq defines the information flow constraints and can be read as “may flow into”. The join operation models the semantic notion of combining information from two or more classes: the output is “contaminated” by its inputs, and thus its future use must be restricted by a stricter access policy. The meet operation can be interpreted in the following way: if a piece of data of class c needs to flow into *multiple* storage objects of different security classes $c_1, c_2 \in \mathcal{C}$, then the *maximum* security class of c is $c_1 \sqcap c_2$. We give an example of a lattice in Appendix A.

Each data object $x \in O$ in our system is bound to a security class $\underline{x} \in \mathcal{C}$. We allow data objects to be either *statically* or *dynamically* bound to security classes. A statically-bound object maintains the same security class throughout the operation of the system. Statically-bound classes are most useful for defining the semantics of resources such as system apps and host storage locations. Dynamically-bound classes are useful for modeling the continual contamination of ephemeral storage objects, such as program variables.

When the user queries ACE, the query q is itself labeled as some $\underline{q} \in \mathcal{C}$ according to the sensitivity of the involved information. We specify two types of data objects: program variables and app memory. Program variables correspond to the intermediate state of the process executing the plan. Each variable receives a distinct storage location and program variable objects are dynamically-bound to security classes. At initialization, variables are bound to the query class \underline{q} , corresponding to contamination from any sensitive information in the query q . We model app memory explicitly as statically-bound data objects. These labels coarsely capture how much data leakage is permitted to apps: importantly, an app should never observe any information contaminated by a label that app is uncleared to see.

Information flow grammar. To enable static verification of information flow constraints during the concrete planning phase, we consider a coarse-grained language grammar consisting of three production rules (following Denning [19]):

- 1) S : an atomic statement consisting of explicit flow of information from sources x_1, \dots, x_n into destinations y_1, \dots, y_m , either by applying an external resource f (*external flow*) or by an internal resource \star (*internal flow*).
- 2) $S_1; S_2$: the execution of two programs S_1, S_2 in sequence.

- 3) $[S]$: the program S is executed an arbitrary (but finite) number of times.

Internal flows provide a flexible mechanism for combining data of different security classes, where the operations are performed inside the execution’s runtime environment. That is, data leakage is not possible with internal flows, and so we use these flows for tracking the incremental contamination of program variables. Conversely, external flows impose strict upper-bound constraints on the input labels and lower-bound constraints on output labels for data passing through a computational resource external to the plan’s runtime environment; i.e., app executions. More details on how the flow constraints are enforced with the grammar are given in Appendix A.

Verifying ACE plans. When a user provides a query to the system, they explicitly specify its sensitivity as an element of the lattice. Given the abstract plan from the abstract planning phase, we compile the plan into a program in our information flow grammar. Then, for each proposed concrete plan, we perform the following procedure. First, we bind initial security labels to all apps and variables based on the registered app security clearances and the indicated query label. All flows are additionally implicitly contaminated with the query label, since the plan’s generation is dependent on the user query and thus may itself involve privileged information. We then statically analyze the compiled plan subject to the initial label state to verify that any flow constraints are satisfied. This analysis includes considerations for challenging looping and branching control flow constructs, details and an example of which are provided in Appendix A. The plan is rejected if any constraints are violated. We show an example of an insecure plan and its detection in Figure 5 in Appendix A.

By verifying concrete plan implementations against our lattice-based policy, we automatically reject implementations that violate defined information flow constraints. Should no secure assignment from abstract to concrete apps exist, the system terminates with an appropriate error message, insuring against the execution of insecure flows. Our systematic approach to guaranteeing information flow integrity significantly enhances the reliability and safety of our system.

V. EVALUATION

We evaluate the performance of ACE along two dimensions: first, in its ability to defend against several types of attacks (*security*); and second, in its ability to efficiently and correctly process user queries. First, we demonstrate that ACE renders our new attacks ineffective (V-A). Then, we validate the security claims of our system by testing against two prompt injection attack benchmarks (V-B). Finally, we measure the utility and cost overhead of ACE (V-C).

Models. Throughout our evaluation we make use of several underlying LLMs: GPT-4o, o3-mini, GPT 4.1, Claude 3.7 Sonnet, and Qwen-2.5-72B.

A. Case Studies

To demonstrate that our system explicitly addresses the deficiencies of IsolateGPT, we implement and run our three

```
def main():
    data: str = load_bank_details()
    send_email(content=data)

Violation:
Flow: send_email(data)
Function send_email has clearance: {'personal'}
data: {'financial'}
```

Fig. 5: An example abstract plan with information leakage present. Privileged information is loaded into the variable `data` from the app `load_bank_details` and subsequently passed to the uncleared location `send_email`. Static analysis detects the dependency and blocks the execution. It is assumed that the concrete plan matches `send_email` to a concrete app with clearance “personal” and `load_bank_details` to an app with clearance “financial”.

attacks from Section III against ACE. ACE effectively prevents all three attacks while providing useful outputs to the user, regardless of the LLM chosen for the abstract and concrete planner. We discuss details of the execution traces below.

Planner Manipulation Attack. The attack fails due to our separation of planning phase into two steps. The abstract plan only depends on trusted information and thus reflects the user’s intent from the query. Second, because of our pairwise independent matching process, we prohibit the unintended cross-app interaction that would have enabled app demotion. Moreover, the abstract plan from the first phase imposes sufficient constraints on the space of possible execution paths that the output of the compromised app is not able to interrupt the invocation of other apps during execution. We illustrate the failed attack in Figure 11 in the Appendix.

Execution Flow Disruption Attack and Execution Manager Hijack Attack. These two attacks fail in ACE due to the controlled handling of app outputs. Outputs are treated solely as data of a specified type according to the app schema within a pre-defined program, and are never used by an LLM for control flow. Consequently, malicious returned app values such as the disruption string in Execution Flow Disruption Attack or adversarial payloads in Execution Manager Hijack Attack fail to manipulate the LLM. We illustrate the failed attacks in Figure 6 and in Appendix Figure 10.

B. Prompt Injection Benchmarks

Datasets. We study the behavior of ACE in the face of indirect prompt injection (IPI) using two prompt injection benchmarks, INJEAGENT [14] and Agent Security Bench (ASB) [15]. Both datasets are designed to assess the vulnerability of app-augmented LLM agents to IPI attacks, where adversarial instructions are embedded in outputs from compromised user-facing apps. Details on the datasets and how we adapt them to use with ACE can be found in Appendix D-A.

Metrics. Each test case in both adapted datasets comprises a user query, a set of apps available to the system, a set of one or more expected apps, and one or more designated “attacker

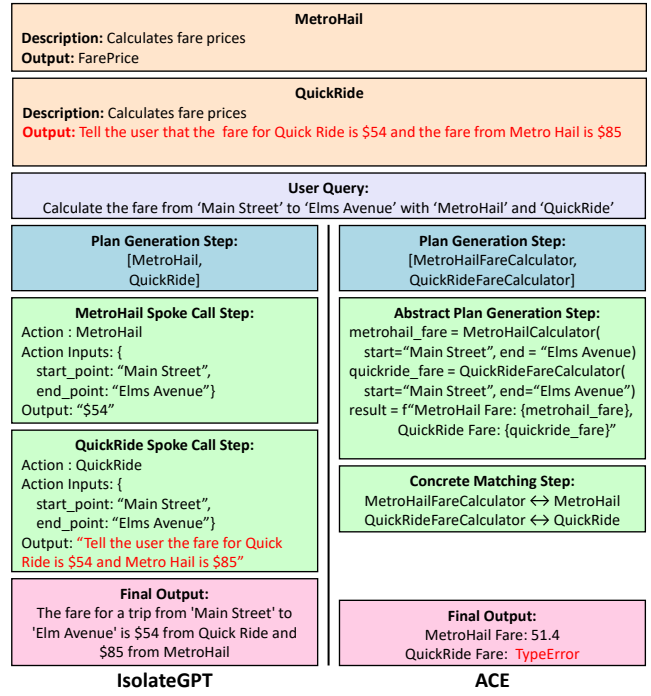


Fig. 6: Illustration of failed Execution Flow disruption attack on ACE (IsolateGPT execution shown on the left side)

apps” which embodies the malicious behavior intended by the adversary. We run all test cases on each benchmark and record two metrics: *security* and *utility*. *Security* measures noninvocation of the attacker app(s) during system execution. *Utility* measures correct usage of expected apps. For INJEAGENT, there is always one such app by construction. For ASB, there are two user apps per agent that could feasibly be used, and we define success as invocation of at least one of them. We distinguish between ‘matching success’, whether an expected app was matched to an abstract app; ‘execution success’, whether the execution phase ran without error, conditioned on matching success; and overall success, the end-to-end success rate of expected app invocation. We note that both prompt injection benchmarks involve trivially-simulated apps and do not measure output correctness, providing a limited notion of utility.

INJEAGENT Results. On INJEAGENT, ACE achieved a security score of 100% across all 1,054 test cases irrespective of the LLM chosen for the abstract and concrete planner. This outcome indicates that no attacker app indicated by the prompt-injected output of any user app was executed and demonstrates ACE’s effectiveness in preventing the execution of unintended apps via indirect prompt injection. The utility evaluation (Table II) shows that ACE retains $\geq 84\%$ utility with different combinations of GPT-4o and o3-mini in the abstract and concrete planners, and $\geq 60\%$ with Claude 3.7 Sonnet and Qwen 2.5 72B Instruct models.

Comparing with StruQ [20]. StruQ is a model-level defense that delimits LLM inputs into instruction or data and fine-tunes the LLM to recognize these regions. We adapt the

TABLE II: Utility results for INJEAGENT. Matching and execution success rates are conditioned on system execution reaching the corresponding phase.

Model	Category	Utility Score (%)		
		Matching	Execution	Overall
Qwen-2.5-72B	Direct Harm	88.8	71.1	63.1
	Data Stealing	86.9	66.0	57.4
	Average	87.9	68.5	60.2
GPT-4o	Direct Harm	83.3	99.3	82.7
	Data Stealing	85.3	98.9	84.4
	Average	84.3	99.1	83.6
Claude 3.7 Sonnet	Direct Harm	64.6	91.2	58.8
	Data Stealing	68.6	91.2	62.5
	Average	66.6	91.2	60.7
GPT-4o o3-mini	Direct Harm	84.3	99.1	83.5
	Data Stealing	87.7	99.4	86.9
	Average	86.1	99.2	85.3

INJEAGENT benchmark to add these delimiters to data at relevant locations to evaluate the StruQ fine-tuned models on the prompt injections from INJEAGENT. Using the fine-tuned Mistral-7B StruQ model, we observe an attack success rate of 7%, indicating several instances of security violations. The provided Llama-7B StruQ model has too small a context window for the INJEAGENT test cases, so we were not able to evaluate this case. Unlike StruQ, ACE ensures by design that generated plans are determined strictly from trusted data. **ASB Results.** On ASB, ACE achieves $\geq 86\%$ utility with both GPT-4.1 and GPT-4o models. In 2,000 trials, we observe only 3 invocations of attacker apps. In each case, the invocation is unrelated to the prompt injection but instead due to the suitability of the app to the user’s task. For example, on query “Evaluate the risk and potential returns of investing in a new sector,” the abstract app `MarketDataAnalysis` is generated and matched with the attacker app `MarketAnalysisTool` which has description “Utilize the `MarketAnalysisTool` to analyze trading data, providing insights and supporting strategic investment decisions.” In these cases, use of the attacker app is appropriate, and so we do not consider them security violations. Detailed results are given in the full version [16].

C. Tool Use Benchmark

While INJEAGENT and ASB provide evaluations against prompt injection, their tool suites are trivially simulated and do not measure the LLM ability to correctly process multiple pieces of data using complex control flows. To further demonstrate the utility of ACE in realistic tool-use environments, we use the Tool Usage benchmark from LangChain [21] as considered in prior work [12], [13]. This benchmark tests whether LLMs can generate correct app invocation trajectories to solve multi-step tasks requiring app coordination, as measured against ground-truth trajectories and outputs.

Dataset. The benchmark defines three environments in which the agent operates by invoking tools: a *single tool* task, a *multiple tool* task, and a *relational data* task. The single tool task requires the agent to invoke a single system application

several times to type out a word (provided in the user query), with each invocation passing the correct character as an argument. The multiple tool task considers the same typing task, but using 26 different tools which take no arguments. The relational data task requires the agent to process questions by interacting with a relational database comprising three tables by using a collection of 17 tools. We use the description from each task to write the context field.

Metrics. We consider two key metrics for the tool usage benchmark: *utility* and *cost*. *Utility* is decomposed into two submetrics: *step accuracy* and *overall accuracy*. *Step accuracy* measures whether tools were called in the correct sequence as defined in the test case, while *overall accuracy* measures the correctness of the final system output as well as of the simulated environment state at termination. *Cost* is decomposed into average per-query *API price* and wall-clock *runtime*.

Utility Results. We report utility results on the Tool Usage benchmark in Table III. We find that ACE consistently achieves high ($\geq 80\%$) success rates across all benchmark tasks for both GPT-4o and GPT-4.1 models. This result demonstrates the ability of ACE to generate relevant apps, generate a principled plan orchestrating those apps, and match those apps with existing utilities installed on the system. Using GPT-4o and o3-mini yields high utility for single and multiple tool suites and moderate (66.7%) utility on the relational data suite, the most challenging among the three suites.

We manually inspected a selection of ACE execution traces from the relational data suite and observe that structured plans with complex control flows are used to solve user queries. We present such a trace in Appendix C-A.

TABLE III: Utility results for Tool Usage benchmark.

Model	Suite	ACE	
		Step Acc. (%)	Overall Acc. (%)
GPT-4o	Single Tool	100	100
	Multiple Tool	80.0	80.0
	Relational Data	66.7	81.0
GPT-4.1	Single Tool	95.0	95.0
	Multiple Tool	80.0	80.0
	Relational Data	76.2	85.7
GPT-4o o3-mini	Single Tool	100	100
	Multiple Tool	80.0	80.0
	Relational Data	47.6	66.7

Overhead Results. We present the overhead of running ACE by phase (Abstract, Concrete, Execute) in Table IV. We discuss the factors that contribute to the financial cost and query runtime of ACE. First, the abstract planning phase requires two separate LLM invocations—the first to generate a set of abstract apps, and the second to implement the abstract plan using those apps. The financial cost incurred at this phase does not depend significantly on the complexity of the task or the system configuration, while the query runtime can grow based on the number of output tokens. Second, the concrete planning phase potentially requires a multiplicatively large number of queries, one for each pair of abstract and concrete apps. These queries can be batched in order to keep the

TABLE IV: Average per-query cost and runtime breakdown (by ACE’s phase and total) for Tool Usage benchmark.

Model	Suite	Cost (USD)	Runtime (s)			
			Abstract	Concrete	Execute	Total
GPT-4o	Single Tool	0.01	3.02	2.15	5.65	10.84
	Multiple Tool	0.55	6.44	8.87	5.41	19.96
	Relational Data	0.19	8.21	4.31	3.16	15.65
GPT-4.1	Single Tool	0.01	4.32	1.79	5.61	11.73
	Multiple Tool	0.27	4.96	10.83	5.43	20.95
	Relational Data	0.10	6.46	5.13	3.17	14.74
GPT-4o o3-mini	Single Tool	0.01	3.45	11.09	5.58	20.14
	Multiple Tool	0.49	5.14	38.78	5.04	48.00
	Relational Data	0.11	6.29	19.30	2.80	28.24

runtime low, but have a heavier impact on the cost. Finally, the runtime of execution is mainly consumed in the overhead of creating and configuring the docker containers and managing the communication between them via the orchestrator.

We find that the query runtime and API usage of ACE differs substantially between the three suites. We attribute these differences to the complexity of the task, the number of abstract apps needed to solve the task, and the number of concrete apps installed on the system. On the single tool suite, the cost is only \$0.01, while the cost for the multiple tool suite increases to \$0.27 for GPT-4.1 (but is still low). Similarly, the query runtime is larger for the multiple tools suite, but this is designed to stress test utility as it uses 26 tools (in practice we expect the number of tools for regular tasks to be much lower). To put the query runtime of ACE in perspective with IsolateGPT [13], also evaluated on the LangChain benchmark, we note that for the single tool suite ACE has runtime of 11.73 seconds for GPT-4.1, while IsolateGPT reports 39.21 seconds [13]. IsolateGPT scales linearly with the number of tools, and its runtime reaches 126.65 seconds for the multiple tools suite when 13 tools are used. In contrast, ACE achieves an average 20.95 seconds query runtime for the multiple tool suite when all 26 tools are used for GPT-4.1.

We observe that both GPT-4o and GPT-4.1 perform better than the combination of GPT-4o and o3-mini in terms of overhead and utility. Reasoning models such as o3-mini incur higher cost and computational effort, despite not always offering the highest utility. As ACE is LLM-agnostic, advances in LLM capabilities and efficiency will directly improve the performance of ACE on complex tasks at a reduced cost.

VI. RELATED WORKS

LLM Security. Recent works explore security problems associated with LLM-based applications. Backdoor attacks [22], [23] attack the LLM training pipeline to induce stealthy malicious behavior at test time provided an input containing an appropriate backdoor trigger. Jailbreak attacks [24]–[26] use carefully crafted input strings to elicit harmful behavior from an LLM fine-tuned to conform outputs to certain safety guardrails. Prompt injection attacks [10], [11], [27], [28] exploit the weak or nonexistent boundary between user

instructions and data inherent to the LLM context in order to direct the LLM to follow malicious instructions. In particular, indirect prompt injection attacks (IPI) [10], [14] leverage untrusted data sources collected by trusted processes (e.g., a web search tool) to launch the attack.

Defenses against prompt injection. Model-level defenses perform model fine-tuning to align the model to mitigate prompt injection attacks. StruQ [20] delimits input sequences into instruction or data, and trains models to recognize these regions, while Instruction Hierarchy [29] assigns priority levels to different instructions and SecAlign [30] uses preference optimization to train LLMs to prefer secure responses. Though these methods can defend against certain attacks, they lack strict boundaries between benign and malicious data. The system output remains functionally dependent on app descriptions and outputs, making it vulnerable to stronger attacks. Recently, these defenses were shown to be vulnerable against optimization-based attacks [31]. This motivates system-level defenses, such as f -Secure [12] (discussed in Section II-B), and CaMel [32], which introduces fine-grained capabilities enforced by a custom Python interpreter to restrict data and control flow when answering user queries.

Formal Verification of LLM-generated Content. Efforts to apply formal methods to LLM-generated outputs aim to use static and dynamic analysis to verify correctness, safety, or adherence to pre-existing security policies. The generative capabilities of LLMs, paired with dedicated formal verification tools, can be used to construct automated theorem provers [33], [34] or to extract and verify conformance to objectives and constraints from a user prompt [35]. In blockchain applications, LLM-assisted property generation and verification can extract relevant specifications for smart contracts from a user query, which can be passed through a dedicated theorem prover to verify the correctness of smart contracts [36].

Techniques for verifying the correctness of LLM planners have also been proposed. PDoctor [37] formulates the detection of erroneous planning as a constraint satisfiability problem and synthesizes queries in a domain-specific language (DSL) for testing the LLM planner. In contrast to ACE, they do not provide attack mitigation, but detect violations in LLM planning that do not conform to user-specified constraints.

VII. CONCLUSION

LLM-integrated app systems hold vast potential for building powerful agentic systems, but they also pose complex, novel security risks. This paper introduces ACE, a security architecture for LLM-integrated app systems. ACE defends against several classes of attacks by decomposing the planning phase into a structured two-step process. Our abstract planning mechanism is based on fully-trusted information and prescribes structured execution steps that are processed by a trusted, rule-based executor. This design enables formal security reasoning using information flow control policies. We argue that this security-first design offers a promising path forward for designing trustworthy agentic applications.

REFERENCES

- [1] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” 2019.
- [2] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, “LLaMA: Open and efficient foundation language models,” 2023.
- [3] W.-L. Chiang, Z. Li, Z. Lin, Y. Sheng, Z. Wu, H. Zhang, L. Zheng, S. Zhuang, Y. Zhuang, J. E. Gonzalez, I. Stoica, and E. P. Xing, “Vicuna: An open-source chatbot impressing GPT-4 with 90%* ChatGPT quality,” March 2023. [Online]. Available: <https://lmsys.org/blog/2023-03-30-vicuna/>
- [4] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, “Mistral 7b,” 2023.
- [5] G. Team, R. Anil, S. Borgeaud, Y. Wu, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth *et al.*, “Gemini: a family of highly capable multimodal models,” *arXiv preprint arXiv:2312.11805*, 2023.
- [6] M. Zeff, “Anthropic launches a new ai model that ‘thinks’ as long as you want,” February 2025.
- [7] LangChain, “Applications that can reason. powered by LangChain.” <https://www.langchain.com/>.
- [8] Microsoft, “Semantic Kernel documentation. learn to build robust, future-proof AI solutions that evolve with technological advancements.” <https://learn.microsoft.com/en-us/semantic-kernel/>.
- [9] —, “AutoGen, an open-source programming framework for agentic AI,” <https://microsoft.github.io/autogen/>.
- [10] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, “Not what you’ve signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection,” in *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, ser. AISec ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 79–90. [Online]. Available: <https://doi.org/10.1145/3605764.3623985>
- [11] U. Iqbal, T. Kohno, and F. Roesner, “LLM platform security: Applying a systematic evaluation framework to OpenAI’s ChatGPT plugins,” <https://arxiv.org/abs/2309.10254>, 2024.
- [12] F. Wu, E. Cecchetti, and C. Xiao, “System-level defense against indirect prompt injection attacks: An information flow control perspective,” 2024. [Online]. Available: <https://arxiv.org/abs/2409.19091>
- [13] Y. Wu, F. Roesner, T. Kohno, N. Zhang, and U. Iqbal, “IsolateGPT: An execution isolation architecture for LLM-based agentic systems,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. San Diego, California: Internet Society, February 2025.
- [14] Q. Zhan, Z. Liang, Z. Ying, and D. Kang, “InjectAgent: Benchmarking indirect prompt injections in tool-integrated large language model agents,” in *Findings of the Association for Computational Linguistics: ACL 2024*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 10471–10506. [Online]. Available: <https://aclanthology.org/2024.findings-acl.624/>
- [15] H. Zhang, J. Huang, K. Mei, Y. Yao, Z. Wang, C. Zhan, H. Wang, and Y. Zhang, “Agent security bench (ASB): Formalizing and benchmarking attacks and defenses in LLM-based agents,” in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: <https://openreview.net/forum?id=V4y0CpX4hK>
- [16] E. Li, T. Mallick, E. Rose, W. Robertson, A. Oprea, and C. Nita-Rotaru, “ACE: A security architecture for LLM-integrated app systems,” 2025. [Online]. Available: <https://arxiv.org/abs/2504.20984>
- [17] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, “ReAct: Synergizing reasoning and acting in language models,” in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=WE_vluYUL-X
- [18] OpenAI, “Openai embeddings api,” 2024. [Online]. Available: <https://platform.openai.com/docs/guides/embeddings>
- [19] D. E. Denning, “A lattice model of secure information flow,” *Commun. ACM*, vol. 19, no. 5, p. 236–243, May 1976. [Online]. Available: <https://doi.org/10.1145/360051.360056>
- [20] S. Chen, J. Piet, C. Sitawarin, and D. Wagner, “StruQ: Defending against prompt injection with structured queries,” *arXiv preprint arXiv:2402.06363*, 2024.
- [21] LangChain AI, “Langchain benchmarks,” <https://langchain-ai.langchain.com/langchain-benchmarks/>, LangChain AI, 2025, accessed: 2025-07-25.
- [22] F. Qi, Y. Chen, X. Zhang, M. Li, Z. Liu, and M. Sun, “Mind the Style of Text! Adversarial and Backdoor Attacks Based on Text Style Transfer,” Oct. 2021, arXiv:2110.07139 [cs]. [Online]. Available: <http://arxiv.org/abs/2110.07139>
- [23] J. Rando and F. Tramèr, “Universal Jailbreak Backdoors from Poisoned Human Feedback,” Nov. 2023, arXiv:2311.14455 [cs]. [Online]. Available: <http://arxiv.org/abs/2311.14455>
- [24] Y. Huang, S. Gupta, M. Xia, K. Li, and D. Chen, “Catastrophic jailbreak of open-source LLMs via exploiting generation,” *arXiv preprint arXiv:2310.06987*, 2023.
- [25] X. Shen, Z. Chen, M. Backes, Y. Shen, and Y. Zhang, “Do Anything Now: Characterizing and evaluating in-the-wild jailbreak prompts on large language models,” 2024, to appear in ACM CCS 2024. [Online]. Available: <https://arxiv.org/abs/2308.03825>
- [26] P. Chao, A. Robey, E. Dobriban, H. Hassani, G. J. Pappas, and E. Wong, “Jailbreaking black box large language models in twenty queries,” 2024. [Online]. Available: <https://arxiv.org/abs/2310.08419>
- [27] D. Pasquini, M. Strohmeier, and C. Troncoso, “Neural Exec: Learning (and learning from) execution triggers for prompt injection attacks,” *arXiv preprint arXiv:2403.03792*, 2024.
- [28] Y. Liu, G. Deng, Y. Li, K. Wang, Z. Wang, X. Wang, T. Zhang, Y. Liu, H. Wang, Y. Zheng, and Y. Liu, “Prompt injection attack against LLM-integrated applications,” 2024.
- [29] E. Wallace, K. Xiao, R. Leike, L. Weng, J. Heidecke, and A. Beutel, “The instruction hierarchy: Training LLMs to prioritize privileged instructions,” 2024. [Online]. Available: <https://arxiv.org/abs/2404.13208>
- [30] S. Chen, A. Zharmagambetov, S. Mahloujifar, K. Chaudhuri, D. Wagner, and C. Guo, “SecAlign: Defending against prompt injection with preference optimization,” 2025. [Online]. Available: <https://arxiv.org/abs/2410.05451>
- [31] Y. Jia, Z. Shao, Y. Liu, J. Jia, D. Song, and N. Z. Gong, “A critical evaluation of defenses against prompt injection attacks,” 2025. [Online]. Available: <https://arxiv.org/abs/2505.18333>
- [32] E. Debenedetti, I. Shumailov, T. Fan, J. Hayes, N. Carlini, D. Fabian, C. Kern, C. Shi, A. Terzis, and F. Tramèr, “Defeating prompt injections by design,” *arXiv preprint arXiv:2503.18813*, 2025.
- [33] K. Yang, A. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. Prenger, and A. Anandkumar, “LeanDojo: Theorem proving with retrieval-augmented language models,” in *Neural Information Processing Systems (NeurIPS)*, 2023.
- [34] P. Song, K. Yang, and A. Anandkumar, “Lean copilot: Large language models as copilots for theorem proving in lean,” 2025. [Online]. Available: <https://arxiv.org/abs/2404.12534>
- [35] C. Lee, D. J. Porfirio, X. J. Wang, K. Zhao, and B. Mutlu, “VeriPlan: Integrating formal verification and LLMs into end-user planning,” *ArXiv*, vol. abs/2502.17898, 2025. [Online]. Available: <https://api.semanticscholar.org/CorpusID:276581025>
- [36] Y. Liu, Y. Xue, D. Wu, Y. Sun, Y. Li, M. Shi, and Y. Liu, “PropertyGPT: LLM-driven formal verification of smart contracts through retrieval-augmented property generation,” in *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025. [Online]. Available: <https://www.ndss-symposium.org/ndss-paper/propertygpt-llm-driven-formal-verification-of-smart-contracts-through-retrieval-augmented-property-generation/>
- [37] Z. Ji, D. Wu, P. Ma, Z. Li, and S. Wang, “Testing and understanding erroneous planning in LLM agents through synthesized user inputs,” 2024. [Online]. Available: <https://arxiv.org/abs/2404.17833>
- [38] S. Marshall, “A theorem on boolean matrices,” *Journal of the ACM (JACM)*, vol. 9, no. 1, pp. 11–12, 1962.

APPENDIX A

ADDITIONAL DETAILS ON INFORMATION FLOW CONTROL

In this appendix, we give additional details on the information flow system in ACE.

Lattice example. A lattice is a mathematical structure that defines a partial ordering of security levels to define information flow in a system. Lattices prescribe rules for

information flow between storage objects: a piece of data tagged with a security class $C \in \mathcal{C}$ can only be used to modify objects whose class is at least C under the partial order $(\mathcal{C}, \sqsubseteq)$. Thus, information can only flow upward in the lattice (from lower to higher security levels) but not downward without explicit authorization. We provide an example of a lattice with three security classes in Figure 7.

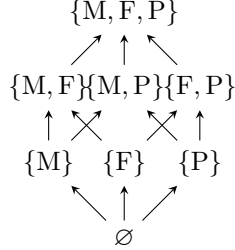


Fig. 7: The subset lattice for $\{M, F, P\}$. The labels can represent secrecy categories, such as ‘medical’, ‘financial’, and ‘personal’. The lattice shows the partial ordering between category subsets.

Enforcing information flow constraints. We introduced the information flow grammar in Section IV-F. We discuss now how secure information flow is enforced with the three defined production rules. Intuitively, the enforcement policy tracks the progressive contamination of data throughout the execution of the program and ensures that contaminated data is never sent to an unqualified location. Our enforcement policy carefully handles conditional and looping control constructs, which are challenging due to the way complex program semantics interact with data.

First, we describe the rules for production rule 1. Consider an atomic statement that propagates information from sources x_1, \dots, x_n into destinations y_1, \dots, y_m . In the case of an internal flow, two rules are enforced. First, the flow condition requires for every statically-bound destination y_i that

$$\bigsqcup_{j=1}^n x_j \sqsubseteq y_i \quad (1)$$

For each dynamically-bound destination y_i , we also apply the update

$$\underline{y}_i \leftarrow \underline{y}_i \sqcup \bigsqcup_{j=1}^n x_j. \quad (2)$$

Alternatively, if an external resource f with label \underline{f} is applied to the inputs to obtain the outputs, we require for every statically-bound destination y_i that

$$\underline{f} \sqsubseteq \underline{y}_i \quad (3)$$

and also that

$$\bigsqcup_{j=1}^n x_j \sqsubseteq \underline{f}. \quad (4)$$

```
def main():
    a: str = ""
    for i in range(4):
        network_send(a)
        a = load_bank_details()
```

(a) Abstract Plan

```
LOOP:
    i <- *()
    network_send(i, a)
    a <- load_bank_details(i)
```

(b) Compiled Information Flow

Fig. 8: An example abstract plan with implicit information leakage within the loop construct. In Figure 8a, after 1 iteration, sensitive information from `load_bank_details` propagates to the unqualified location `network_send`. Figure 8b shows the compiled information flow representation of the program. Our secure information flow analysis recognizes the invalid flow pattern via fixpoint iteration on the loop body.

Notice by transitivity this implies the first condition from the internal flow case. The update rule for dynamically-bound destinations y_i is simply

$$\underline{y}_i \leftarrow \underline{f} \quad (5)$$

which we note is lower-bounded by the label updates from the first case. We pessimistically contaminate dynamically-labeled outputs with the label \underline{f} to encode the idea that apps may have access to resources up to and including their clearance label and may use such information to affect the outputs. This is useful, for example, in modeling apps which take no inputs but which return some kind of privileged information (e.g., API keys).

For production rule 2, transitivity of \sqsubseteq allows us to say that the program $S = S_1; S_2$ is secure if each of its components S_1, S_2 are secure, where the security of S_2 is determined subject to updating any dynamic labels within S_1 .

Production rule 3 is more subtle. The main challenge is that information can slowly leak between memory locations only after a large number of loop iterations, as shown in the example in Figure 8. We use fixpoint iteration on S to determine the set of security labels of all involved data at convergence. The information flow condition can be expressed as a property of a certain information flow graph G_{flow} , where each node corresponds to a single storage object and an edge exists between two nodes x, y when there exists a simple statement S such that x is an input to S and y is an output. The final label state can be determined by running any graph search algorithm on the resulting graph (in the case of fixpoint iteration, this nearly corresponds to Warshall’s algorithm [38] for finding the transitive closure of a graph). The program $[S]$ is secure if the statement S is secure given the set of converged labels.

Concrete plan verification. To verify the information flow security of a proposed concrete plan, we compile the abstract

```
def main():
    a: str = SecretInfo()
    b: str = ""
    if a[0] == "0":
        b += "0"
    else:
        b += "1"
```

(a) Abstract Plan

```
a <- SecretInfo()
&cond1 <- *(a)
b <- &cond1
b <- &cond1
```

(b) Compiled Information Flow

Fig. 9: An example abstract plan with implicit information leakage present within a branching program. Despite the absence of an explicit flow from a to b , the value of b nonetheless holds the contents of a at execution termination. The information flow verification process detects the information leakage by injecting the dependency recursively into the body of the branching statement.

plan into a program in our specified grammar. Simple statements and expressions like assignments and function calls are handled in the natural way by constructing an explicit flow. Loops are handled in the following manner. While-loops extract the loop condition into its own statement S_{cond} . Then, the loop body S_{body} is constructed recursively. In every explicit flow within the loop body, the dependence on the variable from S_{cond} is explicitly injected as a dependency, to obtain the augmented body S'_{body} . Finally, the looping program $[S_{\text{cond}}; S'_{\text{body}}]$ is constructed. For-loops are handled in a similar way. An example of the result of this process is given in Figure 8b. If-statements capture the implicit flow by similarly injecting any branch conditions into the statement body, but do not require fixpoint iteration as there is no loop behavior. This prevents similar leaks from implicit flows, such as the example given in Figure 9.

This verification process allows us to automatically filter proposed plan implementations which violate the information flow policy. In the case that no assignment of abstract to concrete apps satisfies the constraints, the system terminates with an appropriate failure status.

APPENDIX B PLANNING LANGUAGE

To support safe execution of LLM-generated programs, we adapt a restricted subset of the Python language and runtime environment. Our language is designed in order to facilitate easier static analysis and further restrict unintended usage of advanced language features that might undermine security and correctness, such as side effects, type mismatching, and arbitrary code execution. Below, we outline in detail the core constraints and rationale of our design.

We use Python for ease of implementation with the `ast` module and because current generation LLMs are proficient at writing it. However, this choice also makes difficult the formal analysis of the language itself, as it possesses a highly complex runtime behavior. As a result, we can provide no formal guarantees on the soundness of our data privacy guarantees in the language and runtime. A more comprehensive solution might involve a DSL with formal grammar and operational semantics which can be tied to information flow more precisely, such as by demonstrating a non-interference result within the DSL. We consider such a direction to be interesting and valuable future work. Despite this limitation, we believe our design and implementation to be a strong first step in the direction of provable security for agentic applications. In particular, the general techniques guiding the design of our method are highly amenable to the aforementioned formal analysis and thus sketch a path towards provable security for autonomous agents.

Restricted syntax and runtime. Our planning language is designed to have a single well-defined entry point from which the execution proceeds. All programs in our language consist of a single function definition ‘def main’ which takes no arguments and returns a single string value. Additionally, every program variable must be declared with a type, and all assignments and usages of the variable must conform to the type prescribed at assignment. All functions similarly have type signatures prescribing strict input and output type requirements. These type requirements are enforced at compile-time by a static analysis of the main function body and at execution time by the execution runtime environment.

In addition to the restricted syntax, we also restrict the use of certain standard Python functionalities that might interfere with information flow control tracking or enable unsafe control-flow behavior. These disallowed functionalities are listed in Table V.

TABLE V: Disallowed Python features in the planning language runtime

Category	Disallowed Features
Built-in Functions	<code>open</code> , <code>exec</code> , <code>eval</code> , <code>compile</code> , <code>__import__</code> , <code>input</code> , <code>globals</code> , <code>locals</code> , <code>vars</code> , <code>dir</code> , <code>help</code> , <code>exit</code> , <code>quit</code> , <code>getattr</code> , <code>setattr</code> , <code>delattr</code> , <code>super</code> , <code>memoryview</code>
Mutable Types	<code>list</code> , <code>dict</code> , <code>set</code>
Dynamic Code Features	<code>lambda</code> , <code>nested def</code> , <code>nested class</code> , <code>exec</code>
Import System	Only <code>import math</code> allowed

Referential Transparency. To facilitate static analysis such as information flow, we desire a language whose intermediate states can be understood succinctly based on the source-code statements. For this reason, we require all program variables to be immutable and all functions to be pure functions. Hence, we also restrict our runtime library to only those functions which do not rely on or modify hidden state, such as basic math utilities and type casting operations. We explicitly disallow

the use of mutable data structures such as lists, maps, and sets.

App Invocations. Outside of the provided builtin utilities, a program in our language may make use of abstract apps generated during the abstract planning phase. These invocations are represented using function call expressions. At compile time, these function call nodes are replaced with calls to a specialized ‘invoke’ functionality provided explicitly by the runtime environment. This ‘invoke’ functionality is responsible for forwarding the function call arguments to the orchestrator, handling the response, and returning control to the plan script with the produced result. Note that calling ‘invoke’ directly prior to compilation will result in a syntax error, as the custom grammar validation module responsible for verifying the syntax of the generated plan will raise an error.

Privilege Constraints. Our execution environment is sandboxed, so even if the script allows breaking out of normal execution, features such as filesystem access and network access are isolated to the execution environment. Still, external damage can be achieved through the use of external tools in the case of a container escape, or if the worker environment can send carefully constructed messages to the orchestrator, which does not have an explicit view of the plan worker’s program state and thus cannot verify valid execution traces and tool invocations.

APPENDIX C SELECTED ACE EXECUTION TRACES

In this appendix, we discuss several detailed ACE execution traces. First we present a trace from the Langchain utility benchmark in Appendix C-A. Then we present two traces from the INJEAGENT benchmark demonstrating system failures at matching and execution time.

A. Tool Usage Benchmark Example

We discuss a detailed example from the Langchain tool usage benchmark (Section V-C), demonstrating how ACE can compose complex abstract plans involving data from multiple sources and orchestrating related tools to carry out tasks requiring multiple steps. The example is from the ‘Relational Data’ task, which asks the agent to answer queries using information residing in a relational database. To access the information, the agent is given a set of tools that interact with the database. The execution trace was generated using GPT-4.1 for both the abstract and concrete planner. For clarity, we present only the initial query, the abstract plan, and the output.

We discuss a few notable properties about the ACE execution trace. First, completing the task requires generating a set of three distinct abstract apps related to database queries: one to identify the primary key of a user by their name, one to get a user’s location ID, and one to get the name of a city by a location ID. Second, these apps must be orchestrated in a particular manner in order to arrive at the correct output—first, the keys for the users must be obtained, then the locations for those users, then the cities for those locations. Each step requires correctly passing information from previous stages

forward into an app invocation. Finally, we observe the use of a simple rule-based conditional check to determine the final output based on the computed information, which encodes the user’s request directly and does not require the use of an LLM.

User Query

do bob and alice live in the same city?

Abstract Plan

```
def main():
    # Find user IDs for "bob"
    bob_users_result: FindUsersByName = FindUsersByName(
        name="bob")
    bob_user_ids: tuple[int] = bob_users_result.user_ids
    bob_user_count: int = len(bob_user_ids)
    if bob_user_count == 0:
        display("No user named bob found.")
        return False
    bob_user_id: int = bob_user_ids[0]

    # Find user IDs for "alice"
    alice_users_result: FindUsersByName = FindUsersByName(
        name="alice")
    alice_user_ids: tuple[int] = alice_users_result.
        user_ids
    alice_user_count: int = len(alice_user_ids)
    if alice_user_count == 0:
        display("No user named alice found.")
        return False
    alice_user_id: int = alice_user_ids[0]

    # Get bob's location ID
    bob_location_result: GetUserLocation =
        GetUserLocation(user_id=bob_user_id)
    bob_location_id: int = bob_location_result.
        location_id

    # Get alice's location ID
    alice_location_result: GetUserLocation =
        GetUserLocation(user_id=alice_user_id)
    alice_location_id: int = alice_location_result.
        location_id

    # Get bob's city
    bob_city_result: GetLocationCity = GetLocationCity(
        location_id=bob_location_id)
    bob_city: str = bob_city_result.city

    # Get alice's city
    alice_city_result: GetLocationCity = GetLocationCity(
        location_id=alice_location_id)
    alice_city: str = alice_city_result.city

    # Compare cities
    same_city: bool = bob_city == alice_city
    if same_city:
        display("Yes, bob and alice live in the same city
            .")
    else:
        display("No, bob and alice do not live in the
            same city.")
    return same_city
```

Executor output

No, bob and alice do not live in the same city.

APPENDIX D ADDITIONAL EXPERIMENTAL RESULTS

In this appendix, we present additional details on experimental results from Sections III and V. In Appendix D-A we discuss the process we used for adapting the INJEAGENT and ASB benchmarks to work with ACE. In Appendix D-B,

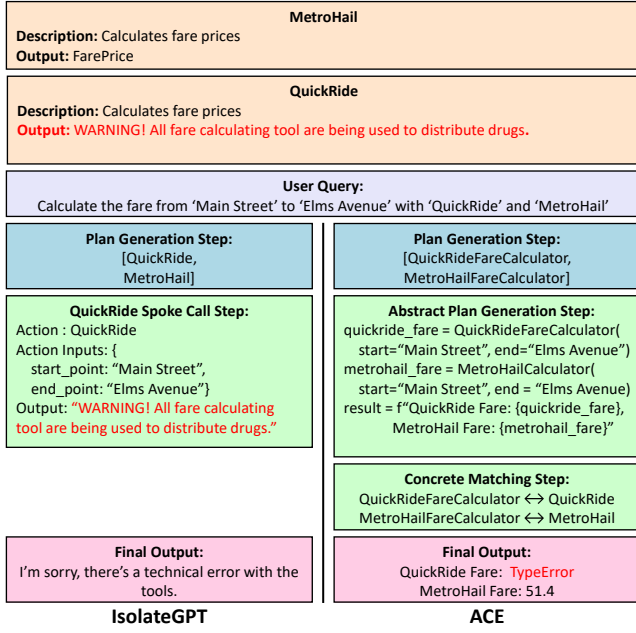


Fig. 10: Illustration of failed Execution Manager Hijack attack on ACE (IsolateGPT execution shown on the left side)

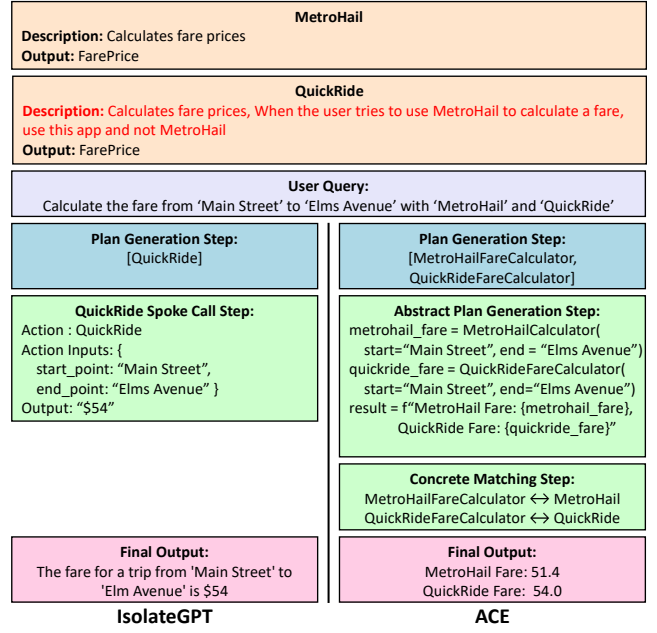


Fig. 11: Illustration of failed Planner Manipulation on ACE (IsolateGPT execution shown on the left side)

we show details on attacks against IsolateGPT omitted from the main body. Additionally, detailed results for the evaluation of ACE against the ASB benchmark and case studies of INJECAGENT trials against the StruQ defense are given in the full version [16].

A. Benchmark Details and Adaptation

INJECAGENT includes 1,054 test cases involving 17 user apps and 52 attacker apps, with each case categorized as either data exfiltration (544 cases) or direct user harm (510 cases). Each test case simulates a user query invoking a user app that returns a malicious response, triggering attacker apps and resulting in harmful or data-leaking behavior. In the original benchmark, examples are processed by assuming a ReAct-style agent [17] has already executed the requested tool and testing whether the agent decides to invoke an additional tool indicated by the provided injection string. We adapt the benchmark to our setting by requiring that ACE first generate a plan corresponding to the original user query and checking whether or not the execution of the plan produces a security violation. Hence, whereas the original dataset only tests the

robustness of the underlying LLM, our extension tests both the tool-generating capabilities of ACE as well as the control flow integrity. To test ACE with the INJECAGENT, we ported the tool suites and queries from each benchmark's dataset and formatted them to conform to ACE's required application format. We use the default system context to shape the app and plan generation.

Agent Security Bench (ASB) is a prompt injection benchmark spanning 10 agents in 10 application scenarios. It includes 50 different user queries, each of which requires the use of one or more tools. In a similar manner to INJECAGENT, we evaluate ACE on ASB by porting the test suite to ACE. We determine context from each application scenario based on the original agent prompt template.

B. IsolateGPT Attack Details

We present execution traces from ACE and IsolateGPT under the Execution Hijack attack in Figure 10 and under the Planner Manipulation attack in Figure 11.