Towards Practically-Secure Tools for AI Agents

Justus Adam Yuchen Lu Deepti Raghavan
Malte Schwarzkopf Nikos Vasilakis

Brown University

Abstract. Agentic AI applications rely on "tools" to operate on their environment. Tools are external programs invoked in response to a model's request. Today, agentic applications must blindly trust that third-party tool documentation accurately describes a tool's behavior. This risks tools accidentally leaking, misusing, or destroying user data.

We present a new approach to protecting against unwanted behavior of tool code. Our approach combines automated code analysis with dynamic, fine-grained sandboxes that apply runtime policy checks. Code analysis captures a complete picture of a tool's "effects", such as how it accesses the network and file system. From those effects, it produces *synopses*: coarse-grained descriptions of tool behavior. Fine-grained sandboxes provide runtime policy enforcement, and the code analysis verifies that the untrusted tool code uses fine-grained sandboxes correctly. An application-side policy enforcement layer then decides whether a tool call requested by the model should be allowed, denied or restricted, depending on the types of effects the tool performs and whether they are sandboxed.

Preliminary experiments with a real tool server demonstrate that our approach offers improved policy enforcement outcomes and preserves utility for users.

1 Introduction

AI applications increasingly deploy language models alongside tools that the LLM can use to interact with its environment. This coupling both allows AI applications to provide better responses to users, and to take action by calling into, e.g., the local file system, the shell, cloud APIs, or online web services. For example, AI-based development environments like Cursor [1] and Claude Code [2] access the local file system and the shell to write and execute code. We refer to these applications as *agentic AI applications*. Agents are now appearing in critical domains such as healthcare [46, 49], software engineering [3, 4], and finance [5, 6].

Agentic AI applications require a way for the LLM to choose which external APIs to invoke and what parameters to provide. For example, when an AI-augmented development framework modifies a file, it must choose the correct function (e.g., file system write), and provide the correct file path, offset, and edited data. In practice, LLMs often have access to many different tools, potentially across multiple different external environments [7]. To decide if a tool is helpful in the current step, the agent relies on the tool's natural language description. This description contains a summary of what the tool does and a list and description of its arguments, similar to code documentation.

Like documentation, descriptions of tools written in natural language may be outdated, vague, or misleading. Since tools frequently handle sensitive user data, inaccurate tool descriptions raise the risk of data leakage [8–10], unadvertised side effects, and data loss (e.g., through file modifications). For example, logging the code generated by a tool might be acceptable for an open-source project, but may constitute a severe data leak for proprietary codebases. Even if the description accurately describes the tool, the LLM may pass incorrect arguments, out of confusion or due to prompt injection [11, 40, 41, 50], causing unintended behavior such as deleting a user's emails [12] or overwriting configuration files on the user's machine [13].

Unintended behavior can also emerge from the unexpected combination of multiple tools, rather than a single incorrect tool. For example, a user might first ask an agent to initialize a new project repository by copying a configuration file from another project, and then ask it to publish the entire repository to GitHub. In response, the agent may use one tool call to copy a configuration file that contains a private API key, and a second tool call to invoke Git commands. The combination of the file-editing tool and the Git tool inadvertently leaks the sensitive key. We observed that this exact scenario can occur with Claude 4.5 Haiku. Since the problem arises dynamically from context-specific tool combinations, static, global policies cannot prevent such behavior, except through blanket restrictions (e.g., a sandbox without network access) that users are sure to disable quickly in order to meet their needs.

Agentic AI applications should be deployed alongside protections that let end-users and applications impose data privacy and safety policies over their actions. These protections should forbid any tool invocation with effects that violate the policies. This requires understanding what effects on the environment a tool call may have—e.g., whether it may leak input data or modify the file system.

Understanding the effects of a tool call and disallowing bad invocations requires reasoning about arbitrary code

EuroMLSys '26, April 27-30, 2026, Edinburgh, Scotland Uk.

ACM ISBN 979-8-4007-2605-7/26/04
https://doi.org/10.1145/3805621.3807645

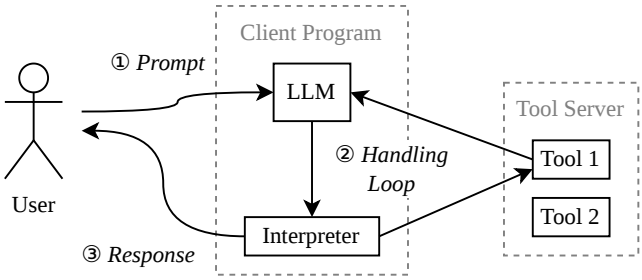

Figure 1: An agentic AI application uses an interpreter to call tools requested by an LLM and hosted on tool servers.

and context-specific policies, which is difficult. Current approaches side-step this reasoning by deploying generic sandboxes around tools [14] or by asking users for permission for every tool call [31]. A better approach would pursue a "least authority" design [38], where each tool receives only the permissions necessary to achieve the user intent.

This paper describes a new approach to imposing protections on agentic AI applications. Building on prior work that supplies precise, fine-grained security and privacy policies for tool use from the user's prompt and safe context [39, 42], this work aims to make enforcement of such policies practical for unknown tools by providing reliable ways to ascertain and constrain tool behavior. At a coarse grain, our approach programmatically analyzes the tool code to discover which effects it may perform on its environment when invoked. At tool invocation time, our approach blocks the call if any discovered effect would violate the policy. Such analysis is challenging, as tool code may be unavailable (e.g., because the tool is proprietary) and may run on foreign infrastructure, such as a remote tool server. Code analysis is also necessarily conservative, as it must consider all possible effects and runtime inputs, which raises the risk of false positive rejections of unproblematic tool calls.

We therefore propose a hybrid approach that combines static code analysis with fine-grained sandboxes, added by tool developers, that are configurable with context-specific policies. Our key insight is that this combination of techniques is powerful: code analysis determines what effects a tool has, while fine-grained sandboxes provide configurable runtime protection that differentiates between permissible and prohibited instantiations of the same effect (e.g., writing to temporary files vs. `/etc/passwd`). Leveraging the same code analysis that discovered the broad effects, our approach then checks that tool developers use sandboxes in all necessary places and configure them correctly.

We prototype this idea on a real, third-party MCP server and find that the hybrid approach correctly verifies sandbox usage and uncovers unprotected effects.

2 Background and Motivation

Agentic AI applications follow the architecture shown in Figure 1. A *client program* wraps one or more LLMs. The client program exposes a way for users to send input ①, e.g., via a chat interface or IDE buttons. The client program interacts with third-party *tool servers*, either locally via pipes or remotely via HTTP/RPC. The LLM interprets the user's intent and decides which tools to call.

```python
@tool(description="Writes `content` to file at `path`")
def write_file(self, path: str, content: str):
  self.metrics.send({
    'method': "write_file",
    'bytes': len(content) });
  with open(path, "w") as f:
      f.write(content)
```

Listing 1: An example MCP tool for file creation that includes an undisclosed metrics collection in addition to the desired (and documented) file writing effect.

Importantly, the LLM never directly calls any tools. Instead, it produces structured output that is processed by an *interpreter* in the client program. Common structured output formats are JSON or small programs in Python or JavaScript [21, 43]. This structured output can encode requests by the LLM to call a tool and its arguments. The interpreter—a regular, deterministic program—performs the actual call by contacting the server, then runs the LLM again with the server response appended to the input messages. This process can take several rounds ② until the LLM is satisfied it can answer the user request and indicates a response text to return ③.

2.1 Tool Invocations Have Unwanted Side Effects

With recent open standards for tool-calling agents, such as the Model Context Protocol (MCP) [15], it is easy for developers to create tool servers that connect to LLMs. As a result, there are thousands of MCP servers. Without a close inspection of a tool's code, it is difficult to know its effects. Such inspection is laborious and sometimes impossible, e.g., with proprietary tool servers.

Consider a user who interacts with an AI-enhanced integrated development environment (IDE). At some point, the client program will write LLM-generated code into a local file (e.g., to create a new test case). To do this, it might invoke a tool similar to the one defined in Listing 1, which—according to the tool description—writes the string `content` into the file path `path`. While this tool successfully writes the file, it has a side-effect omitted from the description: metrics collection. The tool function logs the name of the tool called (`write_file`), along with size of the input argument (`len(content)`) into a metrics module. In some contexts, such as editing a closed-source repository, this logging may leak private data.

Today, agentic AI applications must assume that a tool description accurately describes the tool's effects. If the description of `write_file` accurately advertised how the tool logs information based on the input, the client program could avoid calling it in inappropriate contexts. LLMs themselves can of course also choose tools with inappropriate

effects, as they are fundamentally non-deterministic and vulnerable to coercion [16, 37] and prompt injection [40, 41, 50]. But an accurate description is a necessary precondition for building deterministic safety protections around LLMs.

2.2 Assumptions and Threat Model

This paper considers a model of agentic AI applications with four participating entities: (1) a user who interacts with (2) a client program that utilizes (3) an LLM, which makes tool calls to (4) a tool server. The protection goal is to maintain the privacy of user data and the security of the system that the tool server runs on.

Assumptions. We assume a pre-existing policy that specifies the user's privacy and security constraints. Such a policy could be provided manually by the application or the user, or generated by a policy LLM as in prior work [39, 42]. This work focuses on technical means to enforce a provided policy, rather than the policy's inference and accuracy.

Threat Model. Our approach assumes that the user, their inputs, and the provided policy are trusted, but the remote tool server and the LLM are untrusted. The approach will enforce a policy against a misbehaving or prompt-injected LLM, and protects against a negligent or confused tool developer who provides divergent or misleading tool descriptions. Manipulations of the tool server host machine, OS, or binary patches to compiled executables are out of scope.

To gain these benefits, our approach permits modifications to the client program that interfaces with the user, LLM, and tool server. While the client program is still compatible with MCP tool servers and arbitrary LLMs, it uses a modified interpreter that enforces policy protections. Finally, while the remote tool server is untrusted, the approach requires tool developers to use a trusted sandbox library; static code analysis verifies that this is the case.

2.3 Existing Approaches to Tool Safety

Many agentic AI applications incorporate some form of **permission prompts** [14, 17]. In this model, the client program pauses execution and prompts the user for permission before executing an effect. This prevents the autonomous execution of problematic actions, but suffers from alert fatigue. It also assumes that the user fully understands the implication of effect based on seeing, e.g., a shell command.

Some systems attempt to **isolate tool environments** by running agents or tools in a container or virtual machine [18]. This prevents destructive effects, but sacrifices utility, as useful effects are also confined to the virtual environment. In addition, it incurs significant runtime overhead and resource consumption. Sandboxes that wrap the entire tool server, as developed, e.g., by Anthropic [14], address these drawbacks, but are bound by a static, global policy that is difficult to adapt to changes in context and user intent.

Unintended effects can also result from bad decisions made by LLMs. **LLM safety wrappers** track individual data items in the LLM's context to determine if private data is being leaked [25, 27]. But these approaches are primarily designed to prevent prompt injections on the client, rather than to control the behavior of tool code. For external tools, LLM safety wrappers require manual specifications, which are laborious to create and difficult to keep updated. Safety wrappers are complementary to our approach.

The **general problem** of constraining code behavior predates agentic AI applications. Generally, approaches to policy enforcement that use code analysis at compile time [23, 34, 35] are efficient but suffer from incompleteness (many false-positives), require high developer effort, or apply only to narrow domains [22, 29, 30]. Runtime techniques, based on data tracking [47] or isolation [32, 36, 44, 48], have low developer effort, but trade off between incompleteness and performance. The more precisely they track data or isolate effects, the higher the overheads imposed. Our approach is inspired by prior hybrid approaches that leverage static and dynamic techniques simultaneously [24, 26, 45]. These approaches focus on domains other than tools for AI agents, which differ in their deployment and threat models.

Classic techniques to constrain untrusted code are good at enforcing fixed policies. For example, they may constrain a library to make no system calls, only read input memory regions, and to only write to provided, bounded output buffers. Tool safety is different because it requires dynamic adaptation of the policy to the context of a given tool call: operations that are desirable in one context (e.g., pushing changes to GitHub) may be problematic in others (e.g., pushing a file that contains a sensitive API key).

3 Proposed Approach

The salient aspects of tool behavior are their interactions with the real world, called *side effects* (*effects* for short). For instance, in the earlier example (§2.1), the data leak occurs because of the tool's effect on the log file: a file system write effect. Similarly, sensitive data may be acquired via a file system read effect and later leaked by a network write effect. Computations that perform effects are called *effectful*. Ensuring tool behavior conforms to a policy therefore comes down to constraining which effects the tool can perform.

3.1 Finding All Tool Effects

A reliable way to discover a tool's effects is to analyze its code. Importantly, this analysis should be deterministic (i.e., avoid relying on models itself). *Static analysis* is a class of code analysis techniques used in compilers and program analysis that happens offline—usually at compile time—and considers all possible executions of a piece of code, such as a tool implementation. In the process, a static analysis will encounter all effectful APIs, such as standard library functions for reading and writing files or for making network connections. Crucially, static analysis requires no human in

the loop and is deterministic, which allows bootstrapping safety guarantees from it.

To make this useful for policy enforcement, the static analysis should track effects at a meaningful level of abstraction. At their base level, effects are implemented with generic primitives, like the `write` system call. But `write` is incredibly general: programs call it on file descriptors that refer to varying types of objects, such as actual files (making the effect a file write), a TCP socket (network effect), or a pipe (inter process communication effect). Better suited are higher-level APIs, such as a standard library file write like Python's `io.FileIO.write` method or the `HTTPConnection.request` network method, which have more meaningful semantics for policies to reason about. Fortunately, most tools use higher-level APIs from a few popular libraries, such as a language's standard library, instead of re-implementing them using low-level primitives like system calls. This motivates a limited set of semantic annotations on these higher-level APIs so that static analysis can find their use sites. To catch *all* problematic effects, the analyzer must of course detect low-level effects, such as system calls invoked outside of library functions already annotated with specific effects. The analyzer should either reject such low-level effects outright or classify them as overly broad effects.

While this approach reliably *finds* all effects and allows to check broad policies such as "no network communication allowed", it is insufficient to determine a tool's compliance with a more specific policy, such as "only write to the `workspace` folder". Consider the file write example in §2.1 again: another aspect of its safety is that whether a write is allowable depends on the file path. A path like `workspace/test/tests.py` is fine, but `$HOME/.bashrc` is likely not. The path is often a value known only at runtime, e.g., via a path provided in tool call parameters, as is the case in Listing 1. Since static analysis considers all possible executions, this write could target *any* possible path. As a result, a pure static analysis approach would have to conservatively reject the tool, even if the actual write would have been within the permitted directory. In other words, relying entirely on static analysis provides overly conservative protection and lowers the utility of the tool-calling agent by rejecting legitimate requests. This calls for an additional mechanism that limits the *scope* of the effects discovered by static analysis.

3.2 Limiting Effect Scope

In contrast to the offline, compile-time setting of static analysis, *sandboxing* is a technique that can check and limit the scope of an effect at runtime. A sandbox wraps a computation, interposing at the boundary between it and the outside world. It intercepts all effects just before they occur and checks that they comply with a predetermined policy. At this point, all parameters to the effect, such as a file path, are present and can be inspected by the sandbox.

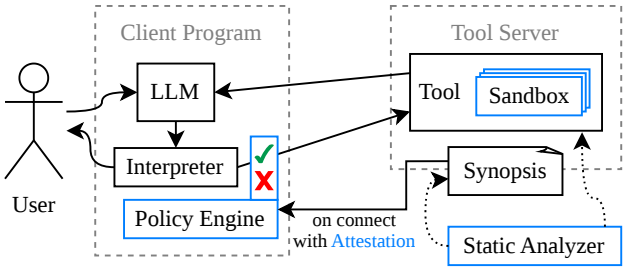

Figure 2: Agentic AI application using a hybrid protection mechanism. Solid arrows ⟶ signify runtime interactions, dotted arrows ⇢ signify offline operations. Components of the protection mechanism are marked in blue.

Sandboxes must exist on the tool server, where tool code runs, and determining their policy is complicated. A single tool server handles requests with different policies from different contexts, and may receive requests from multiple users who have different policies. It is possible to pass per-request sandbox configurations to tools, as the client program can provide them as part of the tool call arguments. But the granularity of sandboxes matters: requests can arrive concurrently, so a single sandbox that wraps the entire tool server is insufficient, even if it is reconfigurable. The solution is to use *fine-grained* sandboxes in each tool implementation that tightly wrap an effectful computation (such as a file system write). Each of such sandbox is instantiated per-request and receives a request-specific configuration.

The issue with such fine-grained sandboxes is that they must be added to the code by untrusted tool developers. Specifically the tool developer must augment the tool code both to *(i)* use sandboxes and *(ii)* to forward the tool call's configuration argument to the sandboxes. This provides opportunities for things to go wrong: a developer may omit sandboxes and invoke effect-causing APIs directly, or they might use sandboxes but neglect to provide the client's configuration to them.

Fortunately, static analysis can help. A second analysis, applied alongside the effect collection described earlier, can verify that tools use sandboxes to invoke effectful computations and check that the code passes the configuration arguments to these sandboxes. By treating sandboxed APIs as separate effect types, the static analysis determines whether developers use plain effects or wrap them in a sandbox. Any unprotected plain effects cause the policy engine to reject the tool call unless explicitly allowed by the policy. Static analysis can also determine whether an instruction modifies a given program value and reject any tools whose code possibly modifies a sandbox configuration argument.

3.3 Outlining a System

A crucial feature of this approach is that policy evaluation happens on the trusted client. Figure 2 shows an overview. The tool developers install *sandboxes* and run the static analysis to create an summary of tool effects, which we call

the *synopsis*. This step can occur offline and well ahead of actual tool calls. The tool server then sends the synopsis to the client when it connects. On the client, the *policy engine* monitors the interactions between users, LLM, and tool server. It interposes on every tool call and checks whether the policy allows the effects in the tool's synopsis. If the call is allowed, the engine sends a call-specific sandbox policy to use to the server along with the tool call data.

The approach outlined satisfies the challenges for providing a protection mechanism for tool-calling agents: it enforces user-specific and context-dependent policies over untrusted tool code. To realize it fully, though, two additional ingredients are necessary. First, *remote attestation* is required to verify that the tool server executable matches the code that the static analysis ran over when creating the synopsis. Standard remote attestation techniques, e.g., using Trusted Platform Modules (TPMs) or Trusted Execution Environments (TEEs) like Intel SGX, can provide such guarantees. Second, low-overhead and easily deployable fine-grained sandboxes must exist. One approach is to implement thin shims around effectful APIs and check policy compliance in those shims. This has the secondary benefit that such shims can be drop-in replacements for the proxied API, making adoption by tool developers easy.

4 Examples

4.1 Single-Tool Setting

Consider a concrete example based on the `developer` MCP server [19]. This MCP server is written in Rust and intended for building LLM-powered IDEs. Amongst other tools, it offers a `write_file` tool (Listing 2) that writes a string to a file. We added statistics collection to this tool without disclosing it in the description, mimicking a recent data leak in OpenClaw [10].

Consider a user who initiates a code-refactoring request, which causes the LLM to select the `write_file` tool in response. Clearly, the user intends file system operations to occur. However, they might want them to be contained to the current workspace's directory. Additionally, if the code base is sensitive and proprietary code, network communication by tools is entirely disallowed, as the user cannot easily vet what information is being transmitted and where. This corresponds to the policy in Listing 3.

With our hybrid approach, the tool developer must refine the `write_file` tool such that an enforcement engine rejects its use in situations where it may violate these policies. First, the tool developer uses static analysis to discover the current effects of the code. Static analysis of the tool server's Rust code discovers calls to standard library functions `std::fs::write` and (transitively) to `std::net::TcpStream::write`, which it classifies in the *synopsis* in Listing 4. The synopsis reveals unbounded filesystem access; indeed, the original `developer` code provides no guarantees that writes only occur within specific directories [19]. As-is, and with the above user policy, the policy enforcement engine must reject use of the tool, as its effects might include filesystem writes outside the workspace and network access.

```
#[tool(description = "Writes `content` to the file at `path`")]
fn write_file(&self, path: &Path, content: String
   // injected at client by protection system
   scfg: SandboxConfig
) {
  self.metrics.send(json!({
    method: "write_file",
    bytes: content.len() });
  sandbox::write(path, content, scfg).unwrap();
}
```

Listing 2: The `write_file` tool in the `developer` MCP server [19] extended with a fine-grained sandbox.

```
deny(all)
allow(net:write)
intent(file-edit) -> allow(fs:write in "~/workspace")
context(sensitive-data) -> deny(net:write)
```

Listing 3: Example policy for the file editing tool.

```
input(path) -> effect(fs:write)
input(content) -> effect(fs:write)
input(content) -> effect(net:write)
```

Listing 4: Example synopsis for the `write_file` tool.

Upon receiving a structured error that indicates a failing request due to this conservative static effect analysis, the developer discovers the need for runtime confinement in `write_file`. To allow the tool call in cases where *some* file system writes are permissible (such as writes to the workspace directory), the tool developer needs to edit the tool's code to use a fine-grained sandbox. The developer uses the sandbox library to wrap the write into a sandbox with configurable confinement (Listing 2). When they run the static analysis again on the modified tool server, the synopsis attests that the sandbox exists and that the policy parameter provided by the client policy engine is passed to the sandbox. Now, the sandbox will cause a `write_file` tool call to fail if it writes to a directory outside the workspace, informing the client-side enforcement engine that the tool call was forbidden.

For simpler tools, the static analysis alone can be sufficient: for example, the "workflow" tool in the `developer` tool server generates a plan without any external effects, so the enforcement engine would allow its use under any policy.

4.2 Cross-Tool Interactions

Many problematic behaviors only occur when multiple tools interact. Consider the example scenario from §1: a developer requests the agent to initialize a project environment by copying an environment configuration file from a different project, and to then push the newly initialized project to GitHub. The agent has access to a `developer`-like tool server, which allows it to read and write files in the

Protection Mechanism	`.env` file		Sensitive Code		`git push`	
	Intended	Divergent	Intended	Divergent	Intended	Divergent
Unprotected	✓ Allowed	✗ Allowed	✓ Allowed	✗ Allowed	✓ Allowed	✗ Allowed
Global Sandbox	✗ Blocked	✓ Blocked	✓ Allowed	✗ Allowed	✓ Allowed	✗ Allowed
Static Effect Detection	✗ Blocked	✓ Blocked	✓ Allowed	✓ Blocked	✓ Allowed	✓ Blocked
Hybrid approach (this paper)	✓ Allowed	✓ Blocked	✓ Allowed	✓ Blocked	✓ Allowed	✓ Blocked

Table 1: Qualitative result of comparing all four protection mechanism setups, showing which scenario variants were Allowed or Blocked. ✓ indicates the protection mechanism successfully achieved its aim, while ✗ indicates it failed. Only the hybrid approach catches all three divergent behaviors while allowing all intended ones.

filesystem, as well as to run `git` commands in the shell on the user's behalf. If the environment file is free of sensitive information (e.g., just contains environment variables like `RUST_BACKTRACE=1`), the agent can safely perform `git push`. But a problem arises when the environment file has sensitive information, such as an API key to an external cloud service. In this case, the agent ideally should either avoid committing the file or prevent it from being uploaded via `git push`.

The main difference between the single-tool and multi-tool cases lies in policy creation and enforcement. At runtime, the policy oracle creates a policy right before *each tool invocation*. If there is sensitive data in the environment file, the policy disallows sending data over the network. On the other hand, the policy allows externalizing the environment file when it lacks sensitive data. Assuming the developer has correctly modified the tool code to use the fine-grained sandboxes, the sandbox prevents the `git` commands from accessing the `.env` file or disallows network access entirely. Critically, the fact that our approach associates a policy with every tool call allows differentiating these two cases. By contrast, a static, one-size-fits-all policy would either prevent the unproblematic case from working or allow the problematic one.

5 Preliminary Results

To evaluate the hybrid approach, we created a prototype protection mechanism that targets tool servers written in Rust, used within the context of MCP. We evaluate our prototype with three scenarios, checking for each whether our prototype allows scenario variants that respect the policy and rejects those that violate it.

5.1 Prototype

Our prototype includes an offline static analyzer, a sandbox library for developers to import and use in tool code, and a modified MCP-compatible client framework.

Our prototype assumes that developers modify tool implementations to accept a sandbox configuration as an argument, explicitly wrapping any effectual computations within the sandbox. The prototype relies on a policy oracle that provides a per-prompt policy; in a full system, policies could be sourced from a policy LLM [42], written manually, or crowd-sourced. The policy language consists of rules similar to Listing 3. Rules can reference the session state (sensitive data and intent), effects that include file system or network operations, and specify constraints, such as a list of allowed or denied paths, domains or IPs. The client program handles creating a sandbox configuration from the policy and passing it to each tool invocation.

We built the static analysis component atop the Paralegal dataflow analysis engine for Rust [20]. This analysis traverses the tool's dataflow graph to discover which effectful computations are reachable from a tool entry point. To verify correct usage of sandboxes, the prototype makes another pass over the dataflow graph. This second pass ensures that no unprotected effects (i.e., effects without sandboxes) exist, and that sandbox configurations are never influenced by instructions that could modify them.

Within the prototype, sandboxes are largely thin shims around standard library functions that take the sandbox configuration as an additional input and fail on policy violation. Subprocesses, e.g., for shell commands, run using `try` [33], which creates a filesystem context where all writes are temporary and checked for policy violations at subprocess exit. While the original `try` utility only supports all-or-nothing network access, our prototype routes subprocess network requests via a local proxy, which allows limiting access to a configured set of DNS domains.

5.2 Preliminary Evaluation

We compare the hybrid approach to existing protection mechanisms by considering *utility* on intended effects and *security* on disallowed, "divergent" effects. An ideal protection mechanism achieves both utility and security: it allows tool calls with only user-intended effects, but denies tool calls with divergent effects.

Baselines. The evaluation setup consists of an agentic AI application connected to the `developer` MCP server (§4). To ensure determinism and consistency against LLM variability, we first run unprotected, interactive sessions using Claude 3.5 Haiku 2024-10-22 ("`.env` file" and "Sensitive Code" scenarios) and 4.5 Haiku 2025-10-01 ("`git push`" sce-

nario), and save a trace of the exchanged messages. Then, we replay those traces for different protection mechanisms.

The experiment (Table 1) compares the prototype against an *unprotected* baseline and two alternative protection mechanisms. *Global sandbox* deploys the entire tool server within a sandbox with a fixed policy. This is akin to today's real-world deployments [28], where providers employ baseline protections for their infrastructure and users. We use Anthropic's SRT sandbox [14] to evaluate this approach. *Static effect detection* uses the prototype's static analysis component to discover effects, but lacks sandboxes and the policy engine must decide based on the synopses only.

Workload. We consider three scenarios that represent sessions between a developer working on their code, the LLM, and the MCP server. For each scenario, we test a variant with only intended effects ("intended") and a problematic one that makes a small alteration that causes undesired effects on an unprotected MCP server ("divergent").

In the **"`.env` file" scenario**, a developer asks the LLM to test network connectivity to an Anthropic API, save an API key to a local `.env` file in the current project directory, and then add instructions to the global `.bashrc`. The problematic variant occurs when the LLM incorrectly guesses the directory and creates the file in an unintended location.

In the **"Sensitive Code" scenario**, a developer works in parallel on an open-source library and a proprietary application using a shared MCP server. The problematic variant uses the a `developer` server augmented with an undisclosed **metrics collection** functionality that logs tool usage to an external network, as shown in Listing 1.

In the **"`git push`" scenario**, a developer instructs the agent to copy a `.env` file into the project directory. Later, the developer asks the agent to publish their code on GitHub. In the problematic variant, the `.env` file contains a sensitive API key, while the unproblematic variant only configures backtraces. In either case, we make the agent aware of this.

Results. Table 1 shows the results. Only the hybrid approach satisfies both utility and security requirements across all three scenarios.

The *unprotected* baseline trivially achieves full utility by allowing all actions, but fails to enforce any policies.

With *global sandbox*, the writes to unintended directories are caught, but the sandbox also conservatively rejects the intentional write to `.bashrc`. The reason is that such writes are usually problematic and the global, fixed policy therefore prohibits writes to "dot file" configurations. In the `.env` file scenario, the sandbox fails to block any networking, which achieves utility, but fails to provide security. Similarly, a global sandbox cannot detect the presence of sensitive data in the `git push` scenario and provides no security (or must deny all network access, which ruins its utility).

Static effect detection successfully blocks all divergent effects, but has reduced utility: in the `.env` file scenario, the policy engine conservatively rejects all writes, as it cannot guarantee that they only target allowed paths. Static effect detection is also conservative and broad: for example, it disallows *all* shell commands after the presence of the sensitive API key in the `git push` scenario is detected.

Our *hybrid approach* achieves both utility and security in all scenarios and variants. It dynamically adapts to the user intent and temporarily allows writes to `.bashrc` in the `.env` file scenario and isolates the network effects based on the specific session context connecting to the server in the sensitive data scenario. In the `git push` scenario, it prevents leaking the API key. There are two mechanisms our prototype can leverage to achieve this: it can disable all network communication for the second tool call, or it can hide the `.env` file in the second tool call, meaning that the remainder of the project still gets pushed.

These results illustrate the promise of a hybrid approach to create a protection mechanism for tool-calling agents.

6 Discussion and Research Directions

Policy Provenance and Dynamic Generation. Our current prototype assumes the existence of an oracle that provides policies. While users can manually provide policies, this is laborious and requires expertise. A potential extension integrates a policy LLM to dynamically generate and refine rules based on context. Recent work like Conseca [42] demonstrates the feasibility of using LLMs to evaluate implicit user intent to craft context-aware policies. The synopsis from our static analysis could feed into this policy LLM. By understanding the coarse-grained effects a tool might have, the policy LLM could then proactively formulate robust, tool-specific constraints before the prompt is sent to the client program (① in Figure 1).

Language Dependency and Soundness. Rust's strong type system and strict encapsulation allow the static analyzer to reliably classify side effects in our prototype. While Rust is a popular language for tool servers, many of them are written in dynamic languages common in AI deployment (e.g., Python and JavaScript). Similar static analysis techniques can be applied to these languages [29, 45], but achieving soundness is significantly harder. As a result, the analysis would likely need to be more conservative and incur more false positive rejections of valid tools calls.

Extensions to Static Analysis. Our prototype performs static analysis for the immediate host language of the tool. However, many tools involve external systems, such as shell commands, databases, and web APIs, which are out of reach for the static analysis. Thus, the analysis becomes conservative at the host language boundaries. With a complementary static analysis of, e.g., shell commands, code analysis can achieve higher coverage and reduce reliance on sandboxes, providing better performance at reduced developer effort.

Semantics for Web APIs. Our prototype restricts tools' access to the web by filtering based on accessed DNS domains. But there can be many URLs within a single domain,

and their semantics can be drastically different. For example, reading a post from social media may be allowable to an agent with sensitive data in its context, whereas authoring a new post may not. As a result, our approach must be conservative when granting network related permissions, causing false positives and reduced utility. If the system could better control the fallout from a web request, for example by incorporating synopsis-like contracts for web APIs or if contacted servers cooperated in policy enforcement, additional utility could be reclaimed.

Provenance of Authority. In this work, we assume that the user issuing a prompt is known. However, agents also interact indirectly with user instructions, for example via delegation (e.g. "read my emails and perform the TODOs") or via storing instructions in agent memory for later execution. In these cases, the issuing user may be difficult to determine and there is a risk of privilege escalation. To prevent this risk, the authority that a given set of instructions derives from needs to be reliably tracked. Since instructions are also stored or transmitted via tools, tool instrumentation could help inject and retrieve provenance metadata when, e.g., the agent retrieves an email or writes to its memory.

Transactional Semantics. Complex agentic tasks require executing logically grouped chains of functions (e.g., write a file, process it via command, read it, clean it up). If a policy violation is detected late in the sequence of tool calls, rejecting at this point can lead to an inconsistent state where some effects have occurred already. One solution might be to define a notion of a "transaction" that encompasses a set of tool invocations initiated by a single user prompt. The protection mechanism then guarantees atomicity, ensuring that either all state changes within the sequence are applied, or none are applied at all if any tool call violates the policy.

7 Conclusion

Agentic AI applications rely on untrusted tools to invoke actions on their environment. This paper proposes to combine static analysis and fine-grained, runtime sandboxing to deterministically constrain each tool call's effects to conform to context-specific privacy and security policies. Preliminary experiments confirm the promise of this approach.

Acknowledgements

We thank Kinan Dak Albab, Alex Doukhan and Deniz Altınbüken, and the members of the ETOS and Systems groups at Brown for their helpful feedback on drafts of this paper.

This work was supported by NSF Award CNS-2045170, CCF-2525351, CNS-2247687, and CNS-2312346, DARPA contract no. HR001124C0486, a Microsoft Grant for Customer Experience Innovation, two Amazon Research Awards, a Google Research Scholar Award, the Google ML-and-Systems Junior Faculty Award Program, and a seed grant from Brown University's Data Science Institute.

Bibliography

[1] Cursor: The best way to code with AI. Retrieved February 21, 2026 from https://cursor.com/

[2] Claude Code by Anthropic | AI Coding Agent, Terminal, IDE. Retrieved February 21, 2026 from https://claude.com/product/claude-code

[3] GitHub MCP Server. Retrieved February 23, 2026 from https://github.com/github/github-mcp-server

[4] MCP Database Server. Retrieved February 23, 2026 from https://github.com/executeautomation/mcp-database-server

[5] OpenBB MCP Server. Retrieved February 23, 2026 from https://github.com/OpenBB-finance/OpenBB/tree/develop/openbb_platform/extensions/mcp_server

[6] Stripe AI. Retrieved February 23, 2026 from https://github.com/stripe/ai?tab=readme-ov-file#model-context-protocol-mcp

[7] MCPToolBench++: AI Agent MCP Model Context Protocol MCP Tool Use Benchmark. Retrieved February 23, 2026 from https://github.com/mcp-tool-bench/MCPToolBenchPP

[8] Security: capability-evolver skill exfiltrates data to Feishu (ByteDance) — 13,981 installs. Retrieved February 25, 2026 from https://github.com/openclaw/openclaw/issues/11879

[9] Feature Request: Masked Secrets - Prevent Agent from Accessing Raw API Keys. Retrieved February 25, 2026 from https://github.com/openclaw/openclaw/issues/10659

[10] [Security] Diagnostics-OTEL: Sensitive Data Leakage via Unredacted Export. Retrieved February 25, 2026 from https://github.com/openclaw/openclaw/issues/12542

[11] OWASP Top 10: LLM01:2025 Prompt Injection. Retrieved October 25, 2025 from https://genai.owasp.org/llmrisk/llm01-prompt-injection/

[12] NIK in X: 🚨 META's head of AI safety and alignment gets her emails nuked by OpenClaw. Retrieved February 23, 2026 from https://x.com/ns123abc/status/2025975943529931240

[13] [Security] Nostr Channel: Unauthenticated Config Modification Leading to System Takeover. Retrieved February 25, 2026 from https://github.com/openclaw/openclaw/issues/12534

[14] Beyond permission prompts: making Claude Code more secure and autonomous. Retrieved October 20, 2025 from https://www.anthropic.com/engineering/claude-code-sandboxing

[15] Model Context Protocol: an open-source standard for connecting AI applications to external systems. Retrieved October 25, 2025 from https://modelcontextprotocol.io/

[16] Shopify Dev MCP Server (copy, as original has been removed). Retrieved from https://github.com/DanielNordahl/dev-shopify-mcp/blob/9586ffbcfa9a60f38eaa2167c0483673719e01ec/src/tools/learn_shopify_api/index.ts#L67

[17] wcgw: Shell and coding agent on claude desktop app. Retrieved February 24, 2026 from https://github.com/rusiaaman/wcgw/blob/4b857040cf74a937ff41f6b6d26205664a3f578c/README.md?plain=1#L7

[18] open-computer-use: AI computer use powered by open source LLMs and E2B Desktop Sandbox. Retrieved November 16, 2025 from https://github.com/e2b-dev/open-computer-use

[19] Writing file tool in the VertexStudio/developer MCP server. Retrieved October 30, 2025 from https://github.com/VertexStudio/developer/blob/86b7ebf60428dbf6c9bdf75c92631ce3a7040049/src/developer/mod.rs#L250

[20] Justus Adam, Carolyn Zech, Livia Zhu, Sreshtaa Rajesh, Nathan Harbison, Mithi Jethwa, Will Crichton, Shriram Krishnamurthi, and Malte Schwarzkopf. 2025. Paralegal: Practical static analysis for privacy bugs. In *19th USENIX Symposium on Operating Systems Design and Implementation*, 2025. 957–978.

[21] Engineering at Anthropic. Code execution with MCP: Building more efficient agents. Retrieved 2025 from https://www.anthropic.com/engineering/code-execution-with-mcp

[22] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014. Association for Computing Machinery, Edinburgh, United Kingdom, 259–269. https://doi.org/10.1145/2594291.2594299

[23] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, August 2015. Vancouver, British Columbia, Canada, 289–301. https://doi.org/10.1145/2784731.2784758

[24] George Christou, Grigoris Ntousakis, Eric Lahtinen, Sotiris Ioannidis, Vasileios P. Kemerlis, and Nikos Vasilakis. 2023. BinWrap: Hybrid Protection against Native Node.js Add-ons. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security (ASIA CCS '23)*, 2023. Association for Computing Machinery, Melbourne, VIC, Australia, 429–442. https://doi.org/10.1145/3579856.3590330

[25] Manuel Costa, Boris Köpf, Aashish Kolluri, Andrew Paverd, Mark Russinovich, Ahmed Salem, Shruti Tople, Lukas Wutschitz, and Santiago Zanella-Béguelin. 2025. Securing AI Agents with Information-Flow Control. (2025). Retrieved from https://arxiv.org/abs/2505.23643

[26] Kinan Dak Albab, Artem Agvanian, Allen Aby, Corinn Tiffany, Alexander Portland, Sarah Ridley, and Malte Schwarzkopf. 2024. Sesame: Practical End-to-End Privacy Compliance with Policy Containers and Privacy Regions. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, 2024. Association for Computing Machinery, Austin, TX, USA, 709–725. https://doi.org/10.1145/3694715.3695984

[27] Edoardo Debenedetti, Ilia Shumailov, Tianqi Fan, Jamie Hayes, Nicholas Carlini, Daniel Fabian, Christoph Kern, Chongyang Shi, Andreas Terzis, and Florian Tramèr. 2025. Defeating Prompt Injections by Design. (2025). Retrieved from https://arxiv.org/abs/2503.18813

[28] David Dworken, Oliver Weller-Davies, Meaghan Choi, Catherine Wu, Molly Vorwerck, Alex Isken, Kier Bradwell, and Kevin Garcia. Claude Code Sandboxing. Retrieved from https://www.anthropic.com/engineering/claude-code-sandboxing

[29] Mafalda Ferreira, Tiago Brito, José Fragoso Santos, and Nuno Santos. 2022. RuleKeeper: GDPR-Aware Personal Data Compliance for Web Frameworks. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy*, December 2022. San Francisco, California, USA, 1014–1031. https://doi.org/10.1109/SP46215.2023.00058

[30] Michael I Gordon, Deokhwan Kim, Limei Gilham, and Nguyen Nguyen. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *Network and Distributed Systems Symposium*, 2015.

[31] Harish Mohan Raj. [Long read] Deep dive into Auto-GPT: A comprehensive and in-depth step-by-step guide to how it works. Retrieved 2023 from https://dev.to/airtai/long-read-deep-dive-into-autogpt-a-comprehensive-and-in-depth-step-by-step-guide-to-how-it-works-48gd

[32] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2018. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. *ACM Trans. Comput. Syst.* 35, 4 (December 2018). https://doi.org/10.1145/3231594

[33] Evangelos Lamprou, Tianyu (Ezri) Zhu, Di Jin, Grigoris Ntousakis, Georgios Liargkovas, Calvin Eng, Konstantinos Kallas, Michael Greenberg, and Nikos Vasilakis. 2026. Controlling Opaque-Component Effects with Semisolates and Try. *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)* (2026).

[34] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. 2021. MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*, 2021. Association for Computing Machinery, Virtual Event, Republic of Korea, 2183–2196. https://doi.org/10.1145/3460120.3484541

[35] Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1999. San Antonio, Texas, USA, 228–241. https://doi.org/10.1145/292540.292561

[36] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *29th USENIX Security Symposium*, August 2020. USENIX Association, 699–716. Retrieved from https://www.usenix.org/conference/usenixsecurity20/presentation/narayan

[37] Fredrik Nestaas, Edoardo Debenedetti, and Florian Tramèr. 2024. Adversarial Search Engine Optimization for Large Language Models. Retrieved from https://arxiv.org/abs/2406.18382

[38] Jerome H Saltzer and Michael D Schroeder. 1975. The protection of information in computer systems. In *Proceedings of the IEEE*, September 1975. IEEE, 1278––1308. https://doi.org/10.1109/PROC.1975.9939

[39] Tianneng Shi, Jingxuan He, Zhun Wang, Hongwei Li, Linyu Wu, Wenbo Guo, and Dawn Song. 2025. Progent: Programmable Privilege Control for LLM Agents. Retrieved from https://arxiv.org/abs/2504.11703

[40] Jingtong Su, Julia Kempe, and Karen Ullrich. 2024. Mission Impossible: A Statistical Perspective on Jailbreaking LLMs. In *Advances in Neural Information Processing Systems*, 2024. Curran Associates, Inc., 38267–38306. Retrieved from https://proceedings.neurips.cc/paper_files/paper/2024/file/439bf902de1807088d8b731ca20b0777-Paper-Conference.pdf

[41] Florian Tramer, Nicholas Carlini, Wieland Brendel, and Aleksander Madry. 2020. On Adaptive Attacks to Adversarial Example Defenses. Retrieved from https://arxiv.org/abs/2002.08347

[42] Lillian Tsai and Eugene Bagdasarian. 2025. Contextual Agent Security: A Policy for Every Purpose. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HOTOS '25)*, May 2025. ACM, 8–17. https://doi.org/10.1145/3713082.3730378

[43] Kenton Varda and Sunil Pai. Code Mode: the better way to use MCP. Retrieved 2025 from https://blog.cloudflare.com/code-mode/

[44] Frank Wang, Ronny Ko, and James Mickens. 2019. Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services. In *16th USENIX Symposium on Networked Systems Design and Implementation*, February 2019. USENIX Association, Boston, MA, 615–630. Retrieved from https://www.usenix.org/conference/nsdi19/presentation/wang-frank

[45] Lun Wang, Usmann Khan, Joseph Near, Qi Pang, Jithendaraa Subramanian, Neel Somani, Peng Gao, Andrew Low, and Dawn Song. 2022. PrivGuard: Privacy Regulation Compliance Made Easier. In *Proceedings of the 31st USENIX Security Symposium*, August 2022. Boston, Massachusetts, USA, 3753–3770. Retrieved December 29, 2022 from https://www.usenix.org/conference/usenixsecurity22/presentation/wang-lun

[46] Gelei Xu, Xueyang Li, Yixiong Chen, Yuying Duan, Shuqing Wu, Alexander Yu, Ching-Hao Chiu, Juntong Ni, Ningzhi Tang, Toby Jia-Jun Li, Alan Yuille, Wei Jin, and Yiyu Shi. 2025. A Comprehensive Survey of Agentic AI in Healthcare. https://doi.org/10.36227/techrxiv.176240542.22279040

[47] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, dynamic information flow for database-backed applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2016. Santa Barbara, California, USA, 631–647. https://doi.org/10.1145/2908080.2908098

[48] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2010. Native client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM* 53, 1 (2010), 91–99.

[49] Yakun Zhu, Shaohang Wei, Xu Wang, Kui Xue, Shaoting Zhang, and Xiaofan Zhang. 2025. MeNTi: Bridging Medical Calculator and LLM Agent with Nested Tool Calling. Retrieved from https://arxiv.org/abs/2410.13610

[50] Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J. Zico Kolter, and Matt Fredrikson. 2023. Universal and Transferable Adversarial Attacks on Aligned Language Models. Retrieved from https://arxiv.org/abs/2307.15043