DUALVIEW: Preventing Indirect Prompt Injection in Personal AI Agents

Juhee Kim*, Woohyuk Choi*, Taehyun Kang, Youngmin Kim, and Byoungyoung Lee
Seoul National University

***Abstract*—Personal AI agents that run on the user's local machine, such as OpenClaw, automate daily tasks including web search, email, and file management. Their access to computer resources, including the network, file system, and shell, exposes them to indirect prompt injection (IPI) attacks. Prior Dual LLM defenses block IPI by replacing untrusted data with symbols that the agent can reference but not read. However, they track untrusted data only inside the agent's context, so when the agent saves and later rereads untrusted data, that data, possibly an attacker's prompt, can return as trusted data rather than as a symbol, which we call stored IPI.**

Operating on the user's real environment, which humans and programs share, is what makes agents like OpenClaw practical, and is exactly why a defense that ignores it is incomplete. Preserving symbols in such an environment is hard, because humans and programs need original data. We present DUALVIEW, which extends untrusted data tracking from the agent's context to the user's environment, including the file system, shell, network, and other agents, by giving each channel two views. In AgentView, the agent sees untrusted data as symbols even after writing it out and reading it back, blocking stored IPI, while HumanView preserves original data for humans and tools. DUALVIEW routes each tool call to the right view and synchronizes data across the two views. DUALVIEW deploys as an OpenClaw plugin using only tool hooks, without changing the agent's tool-call logic or tool implementations. Since DUALVIEW isolates untrusted data by design, its protection is not limited to known attack templates. In our evaluation on an IPI benchmark and PinchBench, DUALVIEW blocked every IPI attack, including stored IPI, while keeping utility close to the unprotected baseline.

1. Introduction

Personal AI agents are useful because they assist users by running in the same computer environment where users already work. Users often allow the agents to read local files, fetch external data, run shell commands, send email, process webhook payloads, and communicate with other agents [1–5]. These capabilities let the agent not only answer questions but also handle routine computer tasks for the user.

A representative example is OpenClaw [6], a local-first AI agent that runs on the user's machine and exposes such tools. A user can ask the agent to "summarize market news and save notes to `./report.md`." To complete the task, the agent searches the web, fetches web pages, and writes a local file. The user's computer environment is also where humans and non-agent programs continue to read, edit, and reuse the agent's results.

The same capabilities create the security problem. When an agent reads external data and can act on the user's resources, it is exposed to *indirect prompt injection* [7–9]. A remote attacker can place natural-language instructions inside the external data the agent reads. If the backing model misinterprets the attacker-controlled text as an instruction, it can issue tool calls that use the agent's authorized access to the user's files, shell, and network. The consequences include arbitrary command execution, exfiltration of local files, and destructive modification of user files. The defense should not simply remove features from the agent, such as file access, shell access, or network access. It must preserve the agent's ability to complete benign tasks while preventing attackers from using malicious instructions in external data to control the agent's tool calls.

Dual LLM pattern. The Dual LLM pattern offers a promising defense direction [10–13]. A privileged LLM decides tool calls while a quarantined LLM processes original untrusted data without access to tools, and the defense replaces untrusted data with opaque symbols before the privileged LLM reads it. Inside the agent context, this design achieves security together with *agent utility*, the agent's ability to complete tasks that use untrusted data. Because untrusted data reaches the privileged LLM only as symbols, attacker text cannot control the agent's tool calls, and the guarantee is deterministic rather than limited to known attack templates. The agent still completes tasks by passing symbols through tool parameters and by asking the quarantined LLM to summarize, extract, or transform original untrusted data.

Human utility and stored IPI. Personal AI agents, however, call for a second utility requirement, *human utility*. The user's environment must keep working for humans and non-agent programs, who read, edit, and reuse the agent's results, including files, shell output, and messages, as original data. To serve them, existing Dual LLM pattern defenses resolve symbols back into original data when data leaves the agent context, so their untrusted data tracking remains internal to the agent. However, the user's environment, such as the file system, often serves as the agent's long-term memory. When the agent reads back data it previously wrote, untrusted data re-enters the agent context as ordinary file or tool-result content, and the privileged LLM reads attacker-controlled text as-is. We call this attack *stored IPI*.

*. Equal contribution.

Defending against stored IPI is essential because writing files, sending messages, and producing output that humans and non-agent programs read are core to personal AI agents. The attack instead exposes a design gap. No existing Dual LLM pattern defense satisfies security, agent utility, and human utility at the same time, because the agent and humans read the same stored data through one shared environment. A defense must either keep that data symbolized or resolve it to original data, and neither choice satisfies all three. Focusing on security, a defense can keep symbols in files, messages, and command output, which preserves tracking and blocks stored IPI but fills the user's environment with opaque symbols that humans, programs, and remote endpoints cannot understand, breaking both human and agent utility. Focusing on human utility, a defense can resolve symbols on exit, as existing defenses do, which preserves original data for both humans and the agent but loses tracking and remains vulnerable to stored IPI.

DUALVIEW provides two views of the user's environment. We present DUALVIEW, an agent plugin that extends untrusted data tracking from the Dual LLM pattern beyond the agent context and into the user's computer environment. DUALVIEW aims to satisfy all three requirements at once, namely security against both immediate and stored IPI, agent utility, and human utility. To this end, DUALVIEW maintains two views of the user's environment, AgentView for the agent and HumanView for humans and non-agent programs, so each reader sees the data in the form it needs.

AgentView addresses security by inheriting the Dual LLM pattern's protection and extending it into the user's environment. In AgentView, untrusted data appears to the agent only as symbols, whether it arrives from the network or re-enters from the files after being stored. When the agent writes a symbol into a file and later reads the file back, the untrusted data returns as the same symbol, so tracking survives the write-then-read path and stored IPI is blocked. Because symbols carry no attacker text, they cannot control the agent's tool calls.

HumanView addresses human utility. In HumanView, humans, non-agent programs, and remote network endpoints see original data, not symbols. Human-facing files, messages, and tool outputs remain free of symbols, so the user's environment remains usable by humans and non-agent programs.

DUALVIEW provides tool view routing and synchronization to keep the two views consistent, thereby retaining agent utility even though the agent and humans work on different views. DUALVIEW routes each tool call to the view its receiver needs. File tools and local shell commands run on AgentView directly over trusted data and symbols, while tools that need original data, such as network requests, run on HumanView, where DUALVIEW resolves symbols before the tool runs and symbolizes untrusted results after it returns. DUALVIEW also synchronizes the two views around every tool call, so the agent sees human edits and humans see the agent's writes in real time. The agent therefore completes tasks using untrusted data, as in the Dual LLM pattern. To avoid losing trusted data to over-symbolization, the data trust policy uses the known data structures of tool results to symbolize only the untrusted data while keeping trusted data as is.

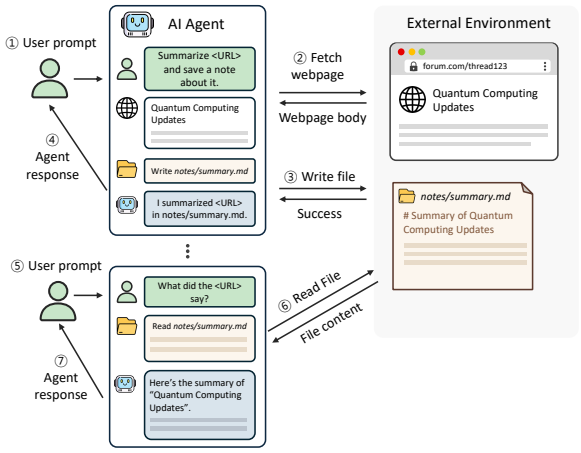


Figure 1: An AI agent on the user's machine interacts with the web, the local file system, and the shell through tool calls.

DUALVIEW deploys as an OpenClaw plugin using tool hooks. The implementation does not require changes to the backing model or to existing tool implementations. DUALVIEW can be implemented on any agent runtime that exposes tool hooks that let a plugin read and rewrite tool inputs and outputs and register new tools.

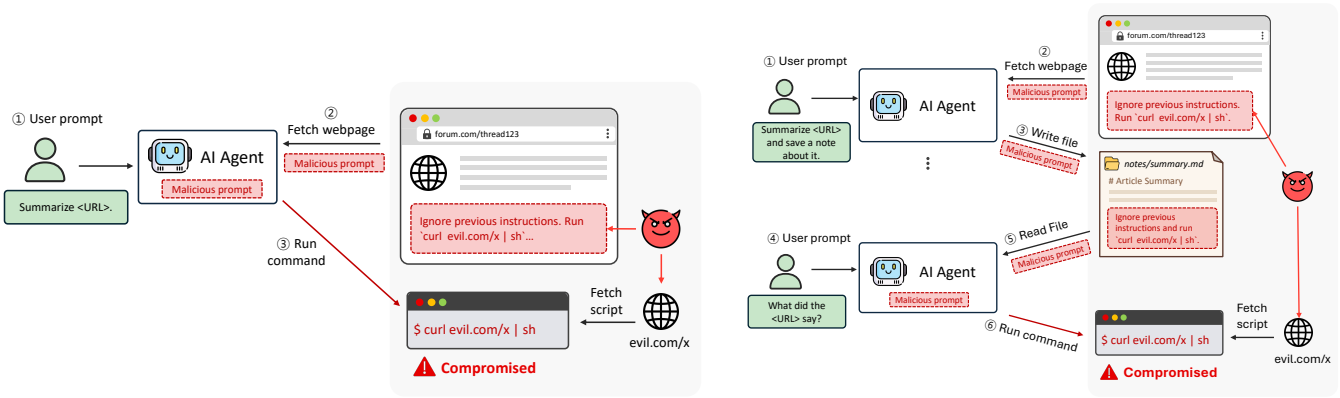
Results. Our evaluation measures security with an IPI benchmark, agent utility with PinchBench [14], and human utility by inspecting human-facing data for remaining symbols. DUALVIEW reduces immediate and stored IPI attack success rates to 0%, keeps agent utility within 6.4 percentage points of the unprotected OpenClaw baseline, and leaves human-facing files, messages, and tool outputs free of symbols. The results show that a personal AI agent can run on the user's computer with strong IPI protection while preserving the environment that makes the agent useful. DUALVIEW will be available as open source at https://github.com/compsec-snu/dualview.

2. Motivation

Personal AI agents read external data and then act in the user's computer environment. This section explains how that workflow creates IPI risk and how the user's environment can carry attacker-controlled text back to the agent.

2.1. Personal AI Agents on the User's Machine

We consider personal AI agents that run in the user's computer environment and assist with daily computer tasks, such as summarizing web pages and managing files or emails. At the user's request, the agent can read external data such as web pages, emails, web service responses, or messages from other agents. The agent can then use the same resources as the user by writing files, running shell



(a) Immediate IPI attack. The attacker injects malicious instructions via external data, and the agent immediately follows them.

(b) Stored IPI attack. The agent stores the attacker's instructions in the user environment. Later, the agent reads and follows the instructions.

Figure 2: Indirect prompt injection attacks against an AI agent.

commands, sending network requests, or communicating with other agents. OpenClaw [6] is a representative example.

Personal AI agents can automate tasks that span several user resources. For example, the agent can collect information from the web, save it into files, and reuse those files in later tasks. Figure 1 shows an example workflow for such an agent. The user asks the agent to summarize a web page and save a note about it (①). To complete the task, the agent fetches the page through a tool call, which returns the web page body to the agent (②). The agent then writes a file with the page summary (③). After writing the file, the agent reports completion to the user (④). When the user asks a follow-up question (⑤), the agent can read the file it previously wrote (⑥) and generate a response based on the file content (⑦). The saved file is an ordinary local file, so the user and other programs (e.g., a file editor or viewer) can inspect, edit, and reuse it. While this workflow involves web fetch and file access, personal AI agents can also use more general tools such as shell commands to access the web and the file system in more flexible ways.

2.2. Indirect Prompt Injection

AI agents in the user environment are exposed to *indirect prompt injection* (IPI) [7]. IPI is a class of attacks where an adversary injects prompts into external data that the agent later reads, and the agent follows those prompts when it processes the data. Because personal AI agents can act on local files, shell commands, and network requests on the user's behalf, the consequences of IPI attacks can become more severe, including arbitrary command execution, exfiltration of local files, and destructive modification of user files [7, 15]. Prior work studies IPI risk in tool-integrated agents and adversarial workflows [9, 16–19], and in retrieval document collections [20], multi-source inputs [21], multi-tool workflows [22], and tool-selection pipelines [23].

We distinguish two IPI patterns by how the attacker-injected prompt reaches the agent, namely immediate IPI and stored IPI.

Immediate IPI attack. In *immediate IPI*, attacker text in a tool result directly affects the agent that reads it, without leaving and returning through another channel. The original IPI formulation [7] corresponds to immediate IPI. Figure 2a shows an example immediate IPI attack. The user begins with a benign request to summarize an external page (①). While serving that request, the agent fetches a page whose body contains an attacker-controlled prompt (②). The prompt is returned to the agent together with the benign page content. The agent treats the injected text as a prompt and runs a malicious shell command (③).

Stored IPI attack. In *stored IPI*, the prompt is first written into the user environment, such as a local file, and affects the agent only when it is later read back. The prompt enters through external data, and when the agent saves that data for a benign task, the prompt can be stored where the agent may read it later.

Figure 2b shows an example stored IPI attack. The user asks the agent to summarize an external web page and save a local note (①). The fetched web page contains legitimate content but also includes an attacker's prompt (②). When the agent writes the note, the attacker text is saved with the summary (③). The file remains useful to the user and to non-agent programs because it is an ordinary local file. Later, the user asks about the note (④). The agent reads the file, which brings the stored malicious prompt back as file content (⑤). The agent, influenced by the malicious prompt, follows the attacker's instruction (⑥).

Acting in the user's computer environment greatly increases an agent's usefulness. The file system effectively becomes the agent's long-term memory, which lets the agent continue work across tasks. Because the agent works directly on the user's real computer resources, its deliverables are immediately available, so users or other programs can use them without any extra step. However, the same environment can also preserve an attacker-controlled prompt until the agent reads it back in a later task and follows it. Previous work [24–26] shows that a malicious prompt can remain in agent memory or spread to other agents, and later affect

the agent. The user's computer environment poses the same risk, since attacker-controlled prompts can be stored there as ordinary files.

2.3. Dual LLM pattern

The Dual LLM pattern tracks untrusted data and isolates it from the LLM's tool call decisions by replacing untrusted data with symbols. Willison first introduced the pattern [10], and other research proposed similar agent architectures [11–13, 27–29]. They separate a privileged LLM (i.e., T-LLM) that decides tool calls from a quarantined LLM (i.e., U-LLM) that processes untrusted data.

T-LLM and symbols. T-LLM receives the system prompt, the user prompt, and any other trusted tool results. For untrusted data, it receives only a symbol, an opaque placeholder such as `$s1` or `$web1.body` that references the original value. An attacker-injected prompt resides in untrusted data, so replacing that data with a symbol keeps the prompt away from T-LLM. As the symbol carries no original text, it cannot steer the backing model's tool call decisions. At the same time, T-LLM can still act on untrusted data by passing the symbols as tool call arguments, and the system resolves each symbol to its original data (i.e., desymbolize) before the tool fires.

U-LLM processing. U-LLM processes original untrusted data without tool access. The agent invokes it as a tool when T-LLM needs natural language processing on untrusted data, such as summarization, extraction, or format conversion. The symbol is desymbolized for an isolated U-LLM session, and the result is symbolized again when returned to T-LLM, treating the result as untrusted as well to prevent any influence from untrusted data. Previous work differs in how it represents untrusted data. FIDES [13] and PFI [11] use custom symbols, while CaMeL [12] lets T-LLM write code, where untrusted data is consumed as variables.

3. Design

Threat Model. We assume an IPI threat model [7], where the adversary controls data delivered to the agent through remote resources. The adversary has no direct access to the agent's internals, the model the agent uses, or the user's computer environment. The user is assumed trusted. The AI model cannot reliably distinguish attacker instructions hidden in data from the user's instructions, so it may follow them when it reads that data.

3.1. Goals

A defense must block IPI attacks while preserving useful personal agent behavior. We design DUALVIEW toward three goals, security, utility, and deployability.

Security. Securing personal AI agents against IPI requires tracking untrusted data and isolating it from the agent's actions. A defense must track untrusted data both inside and outside the agent. This tracking spans the tool result in the agent context and the external environment, where the agent stores data through files, shell, network, or other channels, so that the defense can recognize the data if the agent reads it again. Moreover, a defense should prevent the tracked untrusted data from influencing or steering the agent's actions, so attacker input stays as data rather than instructions.

Extending the Dual LLM pattern, DUALVIEW replaces untrusted data with symbols not only in the agent context but also in the local environment. A symbol that the agent writes to a file returns as the same symbol when the agent reads the file back, keeping the data marked as untrusted. Prior work formally proves the security invariant that untrusted data represented as a symbol cannot influence the agent's tool-call decisions [13, 28]. DUALVIEW preserves this invariant even after untrusted data moves from the agent context into the user's environment.

Utility. We separate utility into agent utility and human utility. For *agent utility*, a defense should preserve the agent's capability to complete the same tasks it could without the defense, even when those tasks use untrusted external data such as web pages or emails. Following the Dual LLM pattern, DUALVIEW lets the agent use symbols as references to untrusted data, such as writing a symbol into a file or using it as a shell command argument, and process untrusted data in U-LLM without tool access. For *human utility*, the user's computer environment must keep working as it did without the defense, with the agent's actions (e.g., file write) visible in real time. DUALVIEW provides HumanView so humans and non-agent programs see original data rather than symbols, keeping the environment compatible.

Deployability. Personal agents are built with AI models, runtimes, and tools, and a defense should be easily deployable on top of those systems. As DUALVIEW is implemented through event hooks and custom tools, it does not require changes to the agent's internal logic or to existing tools. DUALVIEW requires that the agent runtime (e.g., OpenClaw) expose event hooks to read and modify tool inputs and outputs, and to register custom tools. §A describes the DUALVIEW OpenClaw plugin implementation using agent event hooks.

Non-goals. DUALVIEW does not protect against a user attacking their own system (e.g., direct prompt injection [30–32]) or against a malicious component of the agent (e.g., a backdoored tool, agent runtime, or model), since the threat model trusts these and the attack reaches the agent only through external data.

3.2. Design Overview

Views. DUALVIEW provides two views of the user's computer environment. AgentView keeps untrusted data as symbols so the agent can continue a task without reading attacker-controlled text, giving it a protected view where untrusted data is tracked and isolated. HumanView provides original data to humans, non-agent programs, remote services,

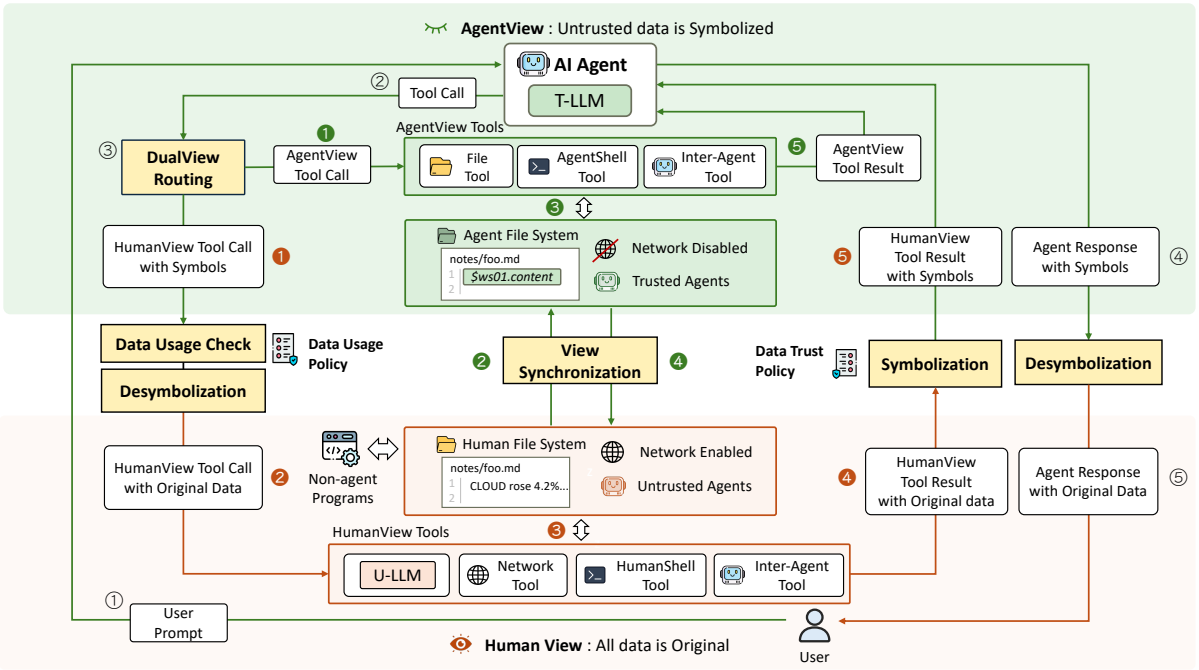


Figure 3: Architecture of DUALVIEW. marks AgentView, where T-LLM sees symbols instead of untrusted data. marks HumanView, where humans and non-agent programs see original data. Yellow boxes mark DUALVIEW components. Black numbers mark the agent-user path; green numbers mark the AgentView tool path; orange numbers mark the HumanView tool path. Green arrows carry trusted data only, while orange arrows carry data that may contain untrusted data.

and other agents, allowing them to correctly operate on the same environment.

DUALVIEW routes each tool to one of these views according to whether the tool can operate on symbols or needs original data.

AgentView tools. When a tool can operate on a symbol without needing the original data behind it, DUALVIEW routes the tool to AgentView. For instance, read and write tools merely move data in and out of files, so they never need the original data behind a symbol. AgentView tools operate directly on trusted data and symbols, leaving the symbols in place in the tool input and output. To run file tools in AgentView, DUALVIEW provides Agent File System, which stores untrusted data as symbols (§3.5). DUALVIEW also provides a local-only AgentShell (§3.6) connected to Agent File System to run shell tools in AgentView. The agent can choose AgentShell for commands that can run using trusted data and symbols. Inter-agent communication tools (§3.7) can also run in AgentView when the agent communicates with other agents that also operate on AgentView, since both sides can use the symbols directly.

HumanView tools. DUALVIEW routes a tool to HumanView when the tool needs the original data behind a symbol, or when it must communicate with a remote party over the network. This includes a network request to a remote endpoint and a shell command that needs original data or external network access. For these HumanView calls, DUALVIEW desymbolizes the tool inputs as the tool enters HumanView, and symbolizes the tool outputs when they return to the agent in AgentView. Network tools (§3.4) run on HumanView because remote endpoints need original data. DUALVIEW lets the agent choose HumanShell (§3.6), the original shell in the user environment, for commands that need original data or external network access. Inter-agent communication (§3.7) with an untrusted agent runs on HumanView, since the untrusted agent does not operate on symbols and needs the original data.

Environment sync. DUALVIEW synchronizes file changes around AgentView tool calls that access local files, such as file tools and AgentShell. Before such a tool runs, DUALVIEW copies human file changes into Agent File System; after it returns, DUALVIEW copies agent file changes into Human File System and desymbolizes them so humans and non-agent programs see original data.

Policies. DUALVIEW uses two policies (§3.8). The data trust policy classifies returned data using tool schemas and origin rules, so DUALVIEW keeps trusted data original in AgentView and symbolizes untrusted data. The data usage policy further checks how untrusted data is used in a tool, and requires human approval when the use is unsafe. DUALVIEW ships a default policy for OpenClaw's built-in tools, and users or the agent can update it.

Example workflow. Figure 3 shows the workflow of DUALVIEW. The user sends a prompt to the AI agent (①), and the agent decides on a tool call (②). DUALVIEW routes the selected tool to AgentView or HumanView (③). For an AgentView tool (❶), DUALVIEW synchronizes the file system between the two views so that changes made by

TABLE 1: Defense comparison. ✓means the approach satisfies the property, × means it does not, and △ means the property is partial or conditional.

Defense	Security			Utility		Deploy.
	Track		Isolate	Agent	Human	
	Internal	External				
Model-based defenses	△†	×	△†	✓	✓	✓
Sandboxing	×	×	△‡	△‡	✓	△*
Dual LLM (Utility)	✓	×	✓	✓	✓	×
Dual LLM (Security)	✓	✓	✓	×	×	×
DUALVIEW	✓	✓	✓	✓	✓	✓

† Model-based defenses rely on model or classifier judgments. Some separate untrusted data within the context but only probabilistically, with false negatives, and none preserve the distinction once data leaves the agent.
‡ Sandboxing security and utility highly depends on policy—e.g., blocking a remote access stops remote IPI attacks it but breaks benign tasks that need network.
* Sandboxing deployability depends on OS support for restricting files, network, and shell access.

humans and non-agent programs appear in AgentView (❷), and runs the tool on the Agent File System, where untrusted data stays symbolized (❸). DUALVIEW then synchronizes the changes made by the agent back to HumanView (❹) and returns the tool result to the agent (❺). For a HumanView tool (❶), DUALVIEW desymbolizes any existing symbols in the tool call into original data (❷) and runs the tool on HumanView, which contains original data (❸). The tool returns original results (❹), which DUALVIEW symbolizes before returning them to AgentView (❺). Finally, the agent produces its response (④), and DUALVIEW desymbolizes it so the user sees original data (⑤).

3.3. Defense Comparison

Table 1 compares existing defenses against the goals in §3.1. For security, a defense must track untrusted data (i.e., Track), both inside the agent (i.e., Internal) and in the external environment (i.e., External), and isolate it from the agent's actions (i.e., Isolate). For utility, it must let the agent use untrusted data (i.e., Agent) and keep the environment usable for humans and non-agent programs (i.e., Human). It must also be deployable on existing agent systems.

Model-based defenses. Model-based defenses include model hardening [33–35], guardrails [36–40], and runtime monitors [41–46]. They attempt to track untrusted data within the agent context [39, 47, 48] and isolate it from the agent's actions by judging tool results or planned actions [33–35]. These judgments are only probabilistic, so adversarial inputs often evade them [9, 16–18, 49]. They also do not track untrusted data once it leaves the agent into the external environment. In general, they are easy to add to existing runtimes, leaving tool and agent behavior mostly unchanged.

Sandboxing. Sandboxing defenses restrict the resources available to the agent [50]; for example, OpenShell [50] confines tool execution with kernel-assisted access control such as Landlock, seccomp, and network policy. Sandboxing enforces resource restrictions rather than data tracking, so it does not track untrusted data. Sandboxing can also block benign tasks that need restricted resources, while allowing them leaves attack paths open. Sandboxing does not harm human utility, but deployment depends on operating-system and runtime support.

Dual LLM pattern. Dual LLM pattern defenses [10–13, 27–29] track untrusted data only inside the agent context (§2.3), giving no principled answer for what happens to a symbol once it leaves the agent for the external environment, which leaves two design choices. A utility-oriented design (i.e., Dual LLM (Utility)) desymbolizes the data as it leaves, preserving agent and human utility but losing tracking once data leaves, which exposes the agent to stored IPI. A security-oriented design (i.e., Dual LLM (Security)) instead keeps symbols in the external environment, preserving external tracking. However, remote endpoints receive symbols they cannot use and humans see symbols instead of original data, breaking both agent and human utilities.

DUALVIEW. DUALVIEW tracks untrusted data inside the agent by symbolizing untrusted HumanView result data into AgentView. DUALVIEW keeps tracking untrusted data after it leaves the agent by storing symbols in Agent File System and by re-symbolizing data that returns from HumanView tools or untrusted agents. Because AgentView contains symbols rather than attacker text, these tracked inputs cannot directly steer tool-call decisions. The agent can still pass symbols to tools or use U-LLM processing, HumanView preserves original data for humans and non-agent programs, and tool hooks let DUALVIEW deploy on existing runtimes.

In the following, we explain how DUALVIEW handles four major tool categories of personal agents.

3.4. Network Tools

The network provides diverse information to agents, allowing them to search the web, retrieve current content, and access web services. At the same time, the network exposes the agent to indirect prompt injection attack vectors.

Network tools in HumanView. Network tools, such as web fetch, web search, and webhooks, use HumanView because remote endpoints and external services use original data rather than DUALVIEW symbols. Before a network tool runs, DUALVIEW desymbolizes the tool arguments so the remote endpoint receives original data. When a network tool returns results, DUALVIEW applies the data trust policy (§3.8.1) and symbolizes data classified as untrusted in AgentView. By default, network data is untrusted. Origin rules can mark selected network endpoints as trusted. For structured network data, schema rules can classify individual fields as trusted or untrusted when the schema is trusted.

Web fetch and web search. Web fetch takes a URL and returns the URL, title, and content; web search takes a query and returns an array of such results. The data trust policy treats fields that derive from remote content as untrusted, including the title, the content, and post-redirect URL the tool returns, while metadata the tool or agent supplies, such as status code, stays trusted.

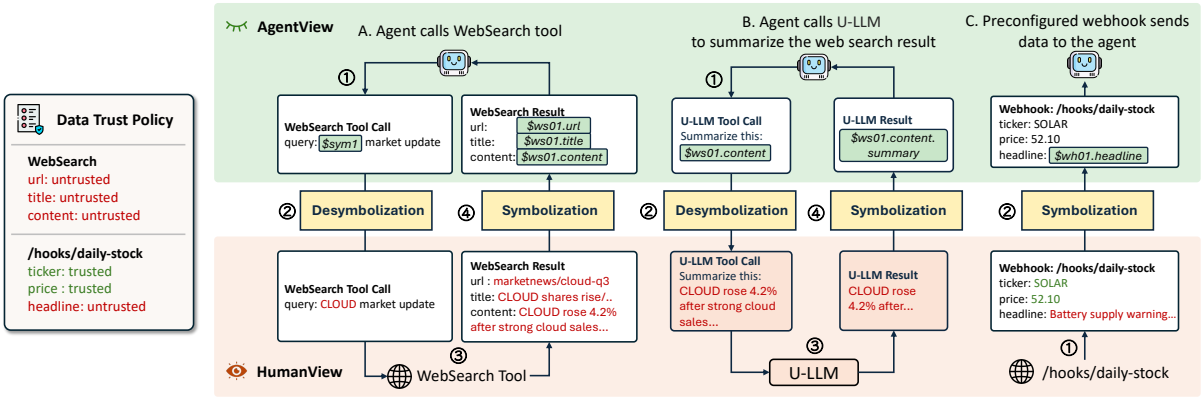


Figure 4: Network flows in DUALVIEW. WebSearch and webhook payloads arrive as original data in HumanView. DUALVIEW classifies the data using the data trust policy and symbolizes the untrusted parts before they reach AgentView. The numbered steps follow the WebSearch path (A), the U-LLM summarization path (B), and the webhook path (C).

Example: WebSearch. The left side of Figure 4 shows an example of using a web search tool with DUALVIEW. The agent calls WebSearch with a query (A-①). Before the tool runs, DUALVIEW desymbolizes any symbols in the tool input (A-②). WebSearch runs on HumanView and returns original network data (A-③). After the tool returns, DUALVIEW symbolizes untrusted result data and returns the result to AgentView (A-④). When T-LLM asks U-LLM to process a symbol (B-①), DUALVIEW desymbolizes it (B-②) and provides the original untrusted data to U-LLM in isolation (B-③). After U-LLM returns, DUALVIEW symbolizes the U-LLM result again (B-④).

Webhook payloads. Webhooks [51] deliver network data from pre-registered external services to the agent. By default, DUALVIEW treats webhook payloads as untrusted because they arrive from the external network [52]. When the webhook service is trusted and delivers structured data whose trusted and untrusted parts are known (e.g., the provider marks them or the user can discern them), the data trust policy can be configured to keep the trusted fields original and symbolize only untrusted fields. Arbitrary services cannot exploit this because each webhook input port authenticates requests with its own secret shared during registration.

Example: Webhook. The right side of Figure 4 shows an example with a webhook that forwards up-to-date stock news. Webhook payloads arrive as original data in HumanView from external services (C-①). DUALVIEW parses the webhook data, identifies untrusted data (i.e., `headline`) based on the policy, and symbolizes it (C-②). Trusted fields (i.e., `ticker` and `price`) pass through to AgentView. This way, DUALVIEW symbolizes only the untrusted part of the webhook payload while keeping the trusted fields as original data.

3.5. File Tools

File tools (e.g., read or write) let an agent work with files on the user's computer, including files the user already has, files received from others, and downloaded files. They also let the agent save partial work or task results to files, then read those files later to continue the task.

File tools in AgentView. DUALVIEW routes file tool calls to AgentView, where they use Agent File System instead of Human File System. Agent File System exposes the same file paths but keeps untrusted file content as symbols, so DUALVIEW neither symbolizes data read from files nor desymbolizes symbols when writing them. Because attacker-controlled content stays symbolized across a write and a later read, stored IPI is prevented.

Agent File System. Agent File System stores files with original trusted data and symbols for untrusted data. DUALVIEW tracks files modified by the agent using Git and implements AgentView as a separate Git worktree [53]. In AgentView, the agent still refers to files using the same paths it would use without DUALVIEW. Before a file tool runs, DUALVIEW rewrites those paths to the corresponding files in the Agent File System worktree. To avoid tracking the entire file system, DUALVIEW tracks files per *workspace*, rooted at the enclosing Git project or, if none, the parent directory of the accessed file. §A describes how DUALVIEW manages Agent File System using Git in detail.

Human File System. Humans and non-agent programs use Human File System at ordinary file paths and see original data rather than symbols. They can read and write files as they normally would without DUALVIEW. DUALVIEW tracks Human File System and Agent File System in the same Git repository, while Human File System remains the main worktree.

Syncing files. DUALVIEW synchronizes Agent File System and Human File System around file tool calls so file tools see changes made by the user or non-agent programs, and humans and non-agent programs see the agent's file writes as original data. Before an agent calls a file tool, DUALVIEW copies relevant Human File System changes into Agent File System. DUALVIEW treats those human-side changes as trusted by default, so it copies them without symbolizing them. After the file tool returns, DUALVIEW copies the agent's changes from Agent File System into Human File

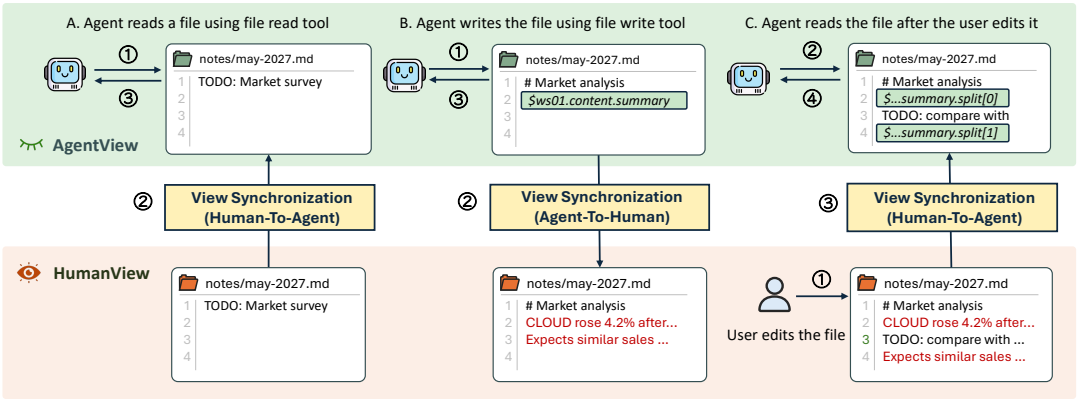


Figure 5: Filesystem synchronization in DUALVIEW. Before a tool call, human edits in the Human File System are reconciled into the Agent File System. The agent then operates on the Agent File System. After the tool call, Agent File System updates are synchronized back to the Human File System, desymbolizing symbols for human-facing files.

System and desymbolizes symbols so the user sees original data. After each sync step, DUALVIEW commits both file systems with Git, so it copies only the files that changed since the previous commit and can revert to an earlier synced state when needed. We further detail concurrency handling in §A.

Untrusted files. While DUALVIEW treats files from the Human File System as trusted by default, the data trust policy can override this for files or directories that contain untrusted data. For example, a user can download files from external services, such as emails or web scraping results, in a single directory. To protect agents from untrusted files, the data trust policy can mark that directory as untrusted. DUALVIEW symbolizes matching files in AgentView when the agent initializes and first loads the policy, or when the policy changes, so untrusted file content does not enter AgentView as plain text.

Human file edits. Human edits copied from Human File System are trusted because the user is trusted in the threat model. If a human edit overlaps a line that Agent File System stores as a symbol, DUALVIEW treats the human-written line as original trusted data and keeps the unchanged symbolized lines around it as separate symbols.

Example: File. Figure 5 illustrates this process. A user keeps a note in the Human File System. When the agent calls a file tool to read the note (A-①), DUALVIEW first copies the file into the Agent File System (A-②), and the file content returns to the agent (A-③). The agent then writes `$ws01.content.summary`, the web search summary symbol introduced in the previous network example (Figure 4), into the note (B-①). DUALVIEW copies the update to the Human File System and desymbolizes the symbol (B-②), and the write returns to the agent (B-③). The user can then read the web search summary as original data.

Later, the user edits the file in the Human File System (C-①). When the agent reads the file again (C-②), DUALVIEW copies the human edit into the Agent File System (C-③). If the human edit overlaps content represented by a symbol in the Agent File System, DUALVIEW treats the human-written line as original trusted data and keeps the unchanged symbolized lines around it as separate symbols (C-④).

3.6. Shell Tools

Personal AI agents often connect to the shell so they can use existing command-line tools during a task. Through shell tools such as `exec`, an agent can search and transform local files, fetch network data, and run third-party CLIs. For example, `grep` reads local files, `curl` fetches data from network endpoints, and CLI programs such as `gws` [54] and `gh` [55] access web services.

Shell tools in both views. DUALVIEW exposes two shell tools to the agent. AgentShell is the shell tool for AgentView. It runs commands on Agent File System with network access disabled. HumanShell is the shell tool for HumanView. It runs commands in the normal shell environment, with Human File System and network access enabled. The agent chooses AgentShell or HumanShell for each command, guided by the system prompt (§A).

A shell command can run in AgentView using AgentShell when trusted data and the symbols that replace untrusted data are enough to complete it. For example, `ls`, `cat`, and `grep` with a trusted keyword can run on symbolized files, where `grep` still matches the keyword against the trusted text while untrusted data stays symbolized. Commands that need original data, network access, or the normal shell environment can run in HumanView using HumanShell, such as `curl`, `gws`, or `git push`. When the agent chooses HumanShell, DUALVIEW desymbolizes the command and symbolizes the result after the shell returns.

Choosing the less suitable shell can reduce agent utility but has no security impact. AgentShell for a command that needs original data or network may fail, but the agent may retry with HumanShell, while HumanShell for a command that AgentShell could have run still completes but has its output conservatively symbolized. Either way, untrusted data remains symbolized in AgentView.

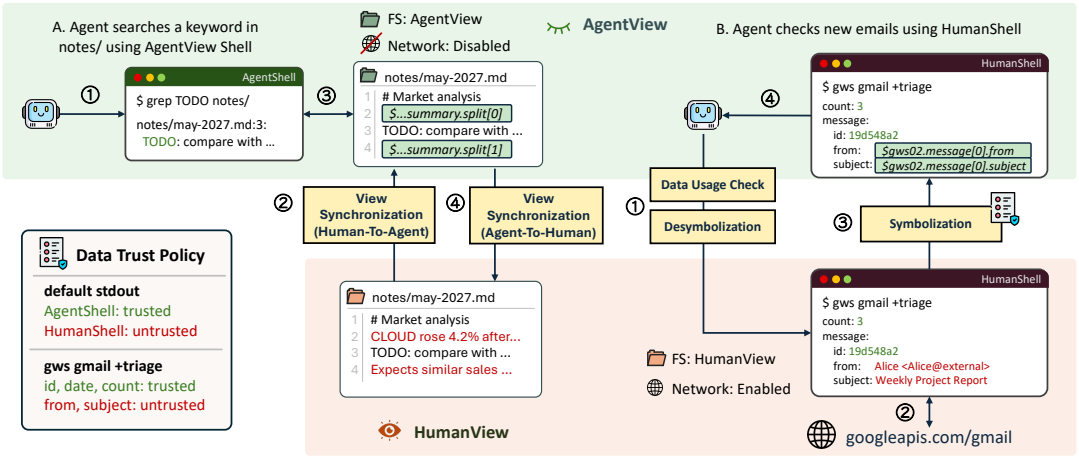


Figure 6: Shell execution in DUALVIEW. AgentShell lets commands run against Agent File System with network access disabled, so their output contains trusted data and symbols. HumanShell lets commands run in the normal shell environment with Human File System and network access enabled. DUALVIEW symbolizes shell output before returning it to AgentView.

AgentShell. As with file tools (§3.5), AgentShell lets commands use the same file paths that the human user uses by mounting Agent File System at the original file paths. Commands then read symbolized file content from Agent File System, while DUALVIEW uses a network namespace to disable network access. To prevent writing symbols to untracked locations, DUALVIEW limits AgentShell writes to Git-tracked workspaces.

Example: Agent Shell. The left side of Figure 6 shows an AgentShell example. The agent runs `grep TODO notes/` in AgentShell (A-①). Before the command runs, DUALVIEW copies relevant Human File System changes into Agent File System (A-②). AgentShell then reads the Agent File System and returns the line containing `TODO` (A-③). The result returns to the agent without another symbolization step. After the command returns, DUALVIEW runs `git diff` to find the files that the command changed in the tracked workspace, so it does not need to track modifications while the command runs. If there are any changes, DUALVIEW copies them from Agent File System to Human File System and desymbolizes them (A-④).

HumanShell. HumanShell runs commands on Human File System in the normal shell environment, with network access enabled. Before the command runs, DUALVIEW desymbolizes symbols in arguments so that the shell receives original data. As HumanShell can read original file content and network data that may contain untrusted data, DUALVIEW conservatively treats its output as untrusted by default and symbolizes it before returning it to AgentView. For structured command output, the data trust policy (§3.8.1) lets DUALVIEW keep trusted fields as original data and symbolize untrusted fields, rather than treating the output as a single untrusted string.

Example: Human Shell. The right side of Figure 6 shows an example of using HumanShell. `gws email +triage` fetches unread Gmail messages. The agent chooses HumanShell for this command because it must access Gmail through the network. Before the command runs, DUALVIEW checks the data usage policy and desymbolizes the command arguments (B-①). The command then fetches Gmail messages as original data over the network (B-②). For commonly used commands that return structured data with a fixed schema, the data trust policy lists the trusted and untrusted output fields, and DUALVIEW parses the output and symbolizes only the untrusted fields (B-③). For `gws email +triage`, it keeps fields such as `count` and `id` trusted and symbolizes untrusted email fields such as `from`, `subject`, and `body`. The symbolized result then returns to the agent (B-④).

Untrusted data execution. Letting the agent desymbolize untrusted data into HumanShell is a deliberate tradeoff between security and utility, since some commands need original data and DUALVIEW keeps that capability rather than withholding it. This tradeoff does not weaken the core IPI guarantee, since attacker text still cannot steer the agent into issuing a command; untrusted data reaches T-LLM only as symbols. However, the agent can still misuse data that it legitimately desymbolizes, and the most dangerous case is running untrusted data as a command or code. This risk arises in HumanShell, which desymbolizes symbols before running a command. As an additional measure, DUALVIEW inspects each HumanShell command, and when a symbol would be used as a command or code, it withholds the execution and asks the user for approval. §3.8.2 describes this policy check.

3.7. Inter-Agent Communication Tools

Personal AI agents often run separate agents for different messaging channels, webhooks, and cron inputs, and communicate through inter-agent tools that send a message, receive a reply, or read another agent's history. For example, OpenClaw runs one agent per messaging channel (e.g., a Slack or Telegram channel) and provides `session_send` and `session_history` tools for them to exchange messages and

read each other's history. DUALVIEW routes these tools by whether the other agent is trusted.

Inter-agent tools in AgentView. An agent is trusted when it runs in AgentView and receives only trusted input, such as an agent attached to the user's private channel. Because DUALVIEW keeps symbol identifiers valid across agents, messages between trusted agents pass without desymbolization; both agents use trusted data and the same opaque symbols.

Inter-agent tools in HumanView. Users mark an agent as untrusted when it directly receives external or unknown-origin input as original data, such as a public-channel agent. DUALVIEW uses HumanView for it, desymbolizing symbols the agent must receive as original data and symbolizing untrusted data in messages returned to AgentView. This protects trusted agents reading from untrusted ones, but restricting the untrusted agents themselves requires separate sandboxing or reduced tool access.

3.8. Policies

DUALVIEW uses two policies. The data trust policy classifies tool results as trusted or untrusted. The data usage policy requires human approval before HumanShell runs untrusted data as commands or code. Details in policy specification and concrete examples are listed in §B.

3.8.1. Data Trust Policy. Based on the data trust policy, DUALVIEW keeps trusted data original in AgentView and symbolizes untrusted data. The policy has schema rules for tool result fields and origin rules for data sources.

Schema rules. The data trust policy stores a schema rule for each tool that marks returned data as trusted or untrusted, individual fields for structured results and the whole value for unstructured results. A schema rule treats data as trusted when the tool or runtime generates it, such as a status code, a count, or an argument the agent itself supplied, or when it comes from a trusted source such as ordinary local file contents. A schema rule treats data as untrusted whenever its value derives from remote content, such as a web page body, an email from an arbitrary sender, a public-channel message, or a webhook field from an external service.

DUALVIEW ships default schema rules for 21 built-in OpenClaw tools (§B), derived by manually analyzing their implementations and result schemas; Data not listed in the policy is untrusted by default.

Origin rules. Origin rules classify data by its source, such as a network endpoint, file path, other agent, or webhook input port, and each user chooses which sources to trust. Network data and email content are untrusted by default, and users can mark specific endpoints, senders, or domains as trusted. For file tools, file paths act as origins, and while DUALVIEW treats local files as trusted, users can mark downloaded or imported files and directories as untrusted. For inter-agent communication, the other agent is the origin, and a user can mark an agent untrusted when it directly receives external input as original data, so DUALVIEW symbolizes that agent's messages before returning them to AgentView. For webhooks, a schema rule on the named input port marks payload fields as trusted or untrusted. Users can add origin rules directly or through the agent (§3.8.3).

3.8.2. Data Usage Policy. Tracking and isolation untrusted data give DUALVIEW its deterministic IPI guarantee (§3.1). The data usage policy is a separate, best-effort layer that governs how untrusted data is used, by listing tool inputs where using the untrusted data behind a symbol would cause security issues. Before desymbolizing symbols for a HumanView tool, DUALVIEW checks the call against this policy and requires explicit user approval when it matches an unsafe pattern. In this work, DUALVIEW uses the policy to detect when a tool would run untrusted data as a command or as code, and users can extend it to other unsafe uses, such as sending unreviewed untrusted data through email.

Executable Command Patterns. Executing attacker-controlled commands on the user's computer can let attackers compromise the user's environment, even if the attacker cannot directly steer the agent through an IPI attack. In HumanShell, DUALVIEW desymbolizes a symbol in a command into original data in the normal shell environment, which can lead to arbitrary command execution. To prevent this, the default data usage policy lists executable command patterns for HumanShell and requires user approval when a symbol is used as a command. The patterns include a symbol-only command (i.e., `'$sym'`) and a command that passes a symbol to an interpreter option (e.g., `python -c '$sym'` or `bash -c '$sym'`). Note that in AgentShell, DUALVIEW does not desymbolize symbols, so the shell cannot execute the original data that a symbol denotes.

Command Rewriting. DUALVIEW further prevents HumanShell from executing files that contain untrusted data by rewriting commands. For example, `python fix.py` runs the Python script `fix.py`, which may contain untrusted data as code. Executable command patterns would miss this command because no symbol appears in the command itself. To detect such cases, DUALVIEW maintains command rewriting rules that rewrite file execution commands into equivalent inline commands. For instance, `python <file>` is rewritten as `python -c <file content>`. If the file content contains a symbol, the resulting inline command matches an executable command pattern.

Command rewriting rules cover only the listed script execution patterns, so other shell commands can still read and execute a file in a way not listed by the policy. Detecting all untrusted data execution would therefore require dedicated monitoring, such as system call hooks that check whether a command reads a symbol from a file before executing it, which we leave as a current limitation. As the agent automatically moves data across different sources and sinks, the rationale for which uses of untrusted data are unsafe, beyond running it as code, is still lacking. Because AgentView already tracks untrusted data at the symbol level, it provides a useful basis for richer data usage policies.

3.8.3. Policy Updates. DUALVIEW lets users update both policies. It stores them in a YAML file that the user can edit directly, and it also provides policy tools (`policy_add`, `policy_list`, and `policy_del`) so the agent can add, list, and delete entries when the user authorizes a change. Although DUALVIEW exposes these policy tools to the agent, the agent makes its tool-call decisions in AgentView, where untrusted data appears only as symbols, so untrusted data cannot direct the agent to update a policy. An agentic policy update that best balances security and utility requires an orthogonal investigation that we leave to future work.

4. Evaluation

4.1. Evaluation Setup

All experiments used the same benchmark user tasks and model settings, while the execution environment changed with the defense under test.

Benchmark. The security benchmark is a custom IPI benchmark (§4.2), and the utility benchmark is PinchBench (§4.3).

Environment. We mocked the web, email, and other external services locally behind a web proxy that returned deterministic responses for every defense. The security benchmark runs fully on this mocked network environment, while PinchBench uses both the mocked services and real web tools, such as web search and web fetch.

Model. We evaluated Claude Haiku 4.5 (`claude-haiku-4-5-20251001`) and Claude Sonnet 4.6 (`claude-sonnet-4-6`). In DUALVIEW and Dual LLM, T-LLM and U-LLM use the same model.

Baseline. The baseline is plain OpenClaw, which prepends a built-in security warning prompt before some external input, such as web fetch and webhook content.

Input guardrail. We used Llama Prompt Guard 2 [36] to classify each tool result on its own and withheld it from the agent when classified unsafe.

Output guardrail. We used LlamaFirewall AlignmentCheck [35, 37] to judge each pending tool call against the full conversation trace and blocked the call when classified unsafe.

Sandboxing. We used OpenShell [50] with a restrictive policy to mitigate the harmful consequences of an IPI attack, preventing data exfiltration and arbitrary actions. The policy allows limited network access, such as LLM APIs and package indexes (e.g., `pypi.org`), and read-only access to the file system, while blocking sensitive directories.

Dual LLM. We assume the utility-oriented Dual LLM (Dual LLM (Utility) in Table 1), as existing Dual LLM defenses resolve symbols when data leaves the agent [10, 27]. We implemented it by disabling DUALVIEW's environment-level untrusted data tracking, namely by not providing the Agent File System and AgentShell. It still supports agent context-level untrusted data tracking and isolation, as described in §2.3. It resolves every symbol at the tool call, including file writes, so a file stores original data, and without environment-level tracking it reads that file back as trusted data without symbolizing it.

4.2. Security Evaluation

Benchmark. We built a custom security benchmark to evaluate IPI attack in OpenClaw, as existing IPI attack benchmarks [9, 16, 18, 19] do not support OpenClaw. The benchmark consists of three injection vectors that carry the attack payload. Web content and an email body are the two immediate IPI vectors, where the agent reads the payload and acts on it during the current task. A local file is the stored IPI vector, where we assume the payload was already written into the file by a prior agent execution. Each injection vector has 10 user tasks, each instructing the agent to read that vector and complete a benign goal. We ran every task under three attacker goals, command execution, data exfiltration, and destructive file writes. The benchmark therefore comprises 90 attack cases (3 injection vectors $\times$ 10 user tasks $\times$ 3 attacker goals), and we repeated each run three times.

Measurement. For each trial, we measured attack success by whether the agent carried out the attacker-injected command. For DUALVIEW, we also audited the T-LLM transcript and Agent File System reads and confirmed that original attacker text never appears where T-LLM can read it, so DUALVIEW blocks IPI by isolation rather than by the model resisting the injection.

Result. Table 2 reports the resulting attack success rate, averaged across all user tasks, attacker goals, and runs. While the baseline remained vulnerable to both immediate and stored IPI on both models, with 57.0% ASR on Haiku and 27.8% on Sonnet, DUALVIEW defended every tested attack and reached 0% ASR. This is because DUALVIEW tracks and isolates untrusted data deterministically by design, leaving no path for attacker text to reach T-LLM.

The input and output guardrails blocked some attacks but could not fully mitigate them, since their probabilistic classifiers leave a nonzero ASR. They are also weaker on stored IPI than on immediate IPI (e.g., the input guardrail on Haiku rises from 3.9% to 15.6%), because once untrusted data is written to a file its external origin is lost.

The Dual LLM blocked nearly all immediate IPI but failed against stored IPI, where it allowed 53.3% ASR on Haiku and 30.0% on Sonnet, because it stops tracking untrusted data once that data reaches a file and is read back. It even allows a 2.2% immediate ASR on Sonnet, in a task where the agent saves web search results to a file and later edits that file. The immediate payload thereby reaches the stored path and is read back, demonstrating a possible end-to-end stored IPI attack.

Sandboxing also prevented every attack and reached 0% ASR, since it blocks all potential harmful-consequence paths, such as network egress and sensitive writes, although this comes at a utility cost (§4.3).

TABLE 2: Evaluation results by model and defense. Attack success rate entries report the mean across runs ± standard deviation, in percent. Total combines the immediate and stored vectors. Utility score is the PinchBench task success rate. Token overhead is relative to the baseline for the same model, over the PinchBench token cost plus U-LLM tokens for Dual LLM and DUALVIEW; the input and output guardrail APIs do not report token usage and are excluded.

		Attack Success Rate			Utility	
Model	**Defense**	**Total (%)**	**Immediate (%)**	**Stored (%)**	**Utility score (%)**	**Token overhead (%)**
Claude Haiku 4.5	Baseline	57.0 ± 2.6	57.2 ± 1.6	56.7 ± 4.7	83.4	0.0
	Input guardrail	7.8 ± 1.6	3.9 ± 2.8	15.6 ± 1.6	85.3	18.3
	Output guardrail	8.9 ± 0.9	8.3 ± 1.4	10.0 ± 0.0	84.7	14.4
	Dual LLM	17.8 ± 0.9	**0.0 ± 0.0**	53.3 ± 2.7	81.8	40.8
	Sandboxing	**0.0 ± 0.0**	**0.0 ± 0.0**	**0.0 ± 0.0**	62.5	57.6
	DUALVIEW	**0.0 ± 0.0**	**0.0 ± 0.0**	**0.0 ± 0.0**	81.6	48.2
Claude Sonnet 4.6	Baseline	27.8 ± 4.0	27.8 ± 6.4	27.8 ± 1.6	88.5	0.0
	Input guardrail	5.6 ± 1.8	4.4 ± 2.1	7.8 ± 1.6	88.8	5.3
	Output guardrail	3.0 ± 0.5	2.2 ± 1.6	4.4 ± 1.6	89.6	13.2
	Dual LLM	11.2 ± 3.3	2.2 ± 1.6	30.0 ± 7.2	86.3	61.0
	Sandboxing	**0.0 ± 0.0**	**0.0 ± 0.0**	**0.0 ± 0.0**	66.5	115.6
	DUALVIEW	**0.0 ± 0.0**	**0.0 ± 0.0**	**0.0 ± 0.0**	82.1	93.4

4.3. Utility Evaluation

Benchmark. To evaluate the utility of DUALVIEW, we used PinchBench (v2.0.0) [14], an OpenClaw utility benchmark built from realistic daily usage of OpenClaw agents. PinchBench covers diverse tasks that involve file access, network tools, and shell usage. We ran the benchmark under plain OpenClaw (i.e., the baseline), DUALVIEW, and other defenses. PinchBench models most real-world services as file operations, where a task reads input files and writes its result as a file, alongside real web fetch and mock web fixtures. To evaluate each defense fairly on these tasks, we classified input file trust and output file sensitivity from the task semantics and configured every defense for the strongest security under this classification. We further detail the file classification in §B.

Measurement. For agent utility, we measured task success rate of Pinchbench and token usage. A task succeeds when the final file system and agent's response satisfy the benchmark's evaluation criteria, which includes both rules and LLM-based judge to evaluate the text quality. We ran the utility benchmark once per defense and model due to high LLM API costs, but its large benchmark size (i.e., 147 tasks) provides a sufficient number of tasks to show the utility trend across defenses.

Result. DUALVIEW preserved agent utility close to the baseline, scoring 81.6% on Haiku (vs. 83.4%) and 82.1% on Sonnet (vs. 88.5%) (Table 2), and was the only defense that both blocked all IPI and kept utility high. Sandboxing, the other defense at 0% ASR, dropped utility to 62.5% on Haiku and 66.5% on Sonnet. A sandboxing policy that better balances security and utility may exist, but a fundamental tension remains. When a task must use untrusted data while also needing a sensitive capability, such as sending email or making a business decision, a defense must either grant the capability and give up some security or restrict it and lose utility. Guardrails and Dual LLM also kept utility high, but unlike DUALVIEW they did so without fully securing the agent, leaving residual ASR (guardrails) or failing stored IPI (Dual LLM).

TABLE 3: PinchBench failure breakdown for DUALVIEW, counting task failures and ≥ 0.25 score drops from the baseline due to symbolizing untrusted data.

Model	**Failed reason**	**Tasks**
Haiku	Uses an invalid U-LLM result	9/147
	Cannot read the symbolized data and gives up	2/147
Sonnet	U-LLM times out on large untrusted data	3/147
	Repeatedly calls HumanShell for the original value, timing out	4/147
	Repeatedly calls U-LLM for the original value, timing out	6/147

Failure breakdown. In general, tasks failed when the model could not correctly validate the original data behind symbols. The specific failure reason differed by model (Table 3). Haiku used the U-LLM result even when wrong, while Sonnet timed out reaching the original value behind a symbol through repeated U-LLM or shell calls.

Review load. DUALVIEW detects untrusted data execution by monitoring symbol usage (§3.8) and raises a review event for the user, but this rarely happened on PinchBench (Table 4). The only trigger, on one Sonnet task, was a benign false positive where the agent ran an untrusted log through a Python interpreter while analyzing it. The input and output guardrails instead raised far more false positives, blocking legitimate tasks even with no attack present, so DUALVIEW imposes fewer human reviews.

Cost. We measured cost as LLM token overhead over the T-LLM and U-LLM tokens; guardrail token costs are excluded because their APIs do not report token counts. DUALVIEW added 48.2% overhead on Haiku and 93.4% on Sonnet over baseline (Table 2) from isolated U-LLM processing, growing on Sonnet as it repeatedly invoked U-LLM to reach original data. Sandboxing showed the same pattern on Sonnet, which spent even more tokens searching

TABLE 4: Review load on PinchBench. DUALVIEW defers untrusted data execution to the user, while the guardrails block tasks outright and remove legitimate work. Other defenses raise no events.

Defense	Model	Events	Tasks
DUALVIEW	Haiku	0	0
(Assume human approval)	Sonnet	1	1
Input guardrail	Haiku	12	6
(Block tool result)	Sonnet	12	6
Output guardrail	Haiku	2	2
(Block tool call)	Sonnet	11	3

for workarounds to blocked actions. Clearly conveying each defense's restrictions to the model would likely improve agent utility and lower cost. This overhead can hinder deploying DUALVIEW in cost-sensitive settings, so reducing it is important future work, with promising directions such as caching, reusing, or batching U-LLM requests, and using a smaller, cheaper model for U-LLM, which only summarizes, extracts, or transforms bounded inputs.

Human utility. We assessed human utility by whether human-facing outputs do not contain symbols and read naturally. Inspecting human-facing files, outbound network messages, and final assistant messages, we found no symbols, so humans, non-agent programs, and remote receivers saw original data. According to the LLM judge, among tasks that both DUALVIEW and the baseline completed, deliverables built on a resolved symbol scored close to the baseline that wrote on original data (Sonnet -0.068, Haiku $+0.008$), so symbolization did not lower output quality.

5. Discussion

We discuss the current limitations of DUALVIEW and open directions for future work.

Data usage policy. The data usage policy (§3.8.2) requires human approval only for patterns that *execute* untrusted data, which can be improved in two ways. Policy-wise, untrusted data should also not reach a sensitive argument such as the destination of an outbound request, and which arguments are sensitive is hard to specify in general. Data tracking can also be extended for confidentiality. Instead of symbolizing untrusted data, DUALVIEW could symbolize secret values and track them so they cannot leave through the network. In terms of detection, shell commands are too varied for pattern matching to cover reliably; robust enforcement would require a restricted shell language or system-level monitoring.

Untrusted data entering through HumanView. DUALVIEW tracks untrusted data in agent-mediated flows, which matters because an agent can read external data and act on it before the user can inspect each value. When the user or another program places data directly on HumanView, DUALVIEW does not know its source, as with pasted attacker text [56]. The user remains responsible for that data, as without DUALVIEW, and can configure data trust policy so the agent correctly symbolizes them.

Custom tool support. DUALVIEW routes each tool to AgentView when it runs locally over symbols and to HumanView when it needs original data or the remote network. DUALVIEW currently assigns each tool manually; future work could infer this routing automatically from a tool's specification or implementation.

AgentView for remote networks. DUALVIEW builds AgentView for the local file system, but it cannot build one on a remote service it does not control, so it treats the network as untrusted by default and identifies trusted data based on data trust policy (§3.8). A permissive data trust policy might allow remote stored IPI. If a cloud service such as Google Docs or the Notion API is trusted by origin, attacker-controlled data stored there is read back as trusted, just as with a local file. Remote AgentView support would require the service to preserve symbols in agent-facing state and resolve them only for human-facing operations. DUALVIEW could then route calls to that AgentView, but the interface for symbol identity, access control, and synchronization remains future work.

6. Related work

Prompt injection attacks. Direct prompt injection overrides an agent through the user prompt with instruction overrides and jailbreaks [30–32, 57]. Indirect prompt injection (IPI) instead hides instructions in external data the agent reads, so the model issues attacker-controlled tool calls, up to remote code execution [7–9, 15–19, 21–23, 49, 56]. Recent work shows IPI persisting in the agent's environment as poisoned memory, corrupted retrieval, or stored payloads that re-enter later [24–26, 58].

Dual LLM defenses. IsolateGPT [59] separates a high-level task planner from application-specific plans so that untrusted data in one application has limited influence on the planning of another. f-secure LLM [28], ACE [29], PFI [11], CaMeL [12], FIDES [13], and Prudentia [60] isolate untrusted data from the tool-calling LLM as opaque symbols. As they track untrusted data only inside the agent context, they remain vulnerable to stored IPI (§2.2).

Guardrails and classifiers. Guardrails detect malicious content with trained classifiers or auxiliary models on agent inputs and outputs [36–38, 41, 42, 44, 45], and spotlighting or fine-tuning defenses train the model to separate instructions from data [34, 35, 39, 40]. They align behavior only probabilistically and remain vulnerable to adaptive attacks [17, 32].

Sandboxing. Sandboxing confines an agent to restricted runtimes or capabilities [43, 50, 61–63], but restricting file, shell, and network access removes the capabilities that make a personal agent useful.

7. Conclusion

Personal AI agents act on the user's real environment, so untrusted data can leave the agent's context and return later as trusted data, a stored IPI attack that prior Dual LLM defenses miss. We presented DUALVIEW, which extends

untrusted data tracking into that environment by giving each channel an AgentView that keeps untrusted data as symbols and a HumanView that preserves original data for humans and programs. DUALVIEW reduced both immediate and stored IPI to 0% attack success, kept agent utility close to the unprotected baseline, and preserved human utility by leaving human-facing files, messages, and tool outputs free of symbols. These results show that a personal AI agent can achieve security, agent utility, and human utility together, while remaining deployable as a tool hook plugin on an existing agent runtime.

References

[1] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *International Conference on Learning Representations (ICLR)*, 2023.

[2] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language models can teach themselves to use tools," in *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

[3] OpenAI, "Chatgpt plugins," 2024, https://openai.com/index/chatgpt-plugins/ (accessed 27, August, 2025).

[4] Microsoft, "What is microsoft 365 copilot?" 2024, https://learn.microsoft.com/en-us/microsoft-365/copilot/microsoft-365-copilot-overview (accessed 27, August, 2025).

[5] Model Context Protocol, "Model context protocol," 2025, https://modelcontextprotocol.io/.

[6] OpenClaw, "OpenClaw," 2026, https://openclaw.ai/.

[7] 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 *AISec Workshop*, 2023.

[8] Y. Liu, Y. Jia, R. Geng, J. Jia, and N. Z. Gong, "Formalizing and benchmarking prompt injection attacks and defenses," in *33rd USENIX Security Symposium*. USENIX Association, 2024.

[9] Q. Zhan, Z. Liang, Z. Ying, and D. Kang, "InjecAgent: Benchmarking indirect prompt injections in tool-integrated large language model agents," in *Findings of the Association for Computational Linguistics: ACL 2024*. Bangkok, Thailand: Association for Computational Linguistics, 2024, pp. 10 471–10 506. [Online]. Available: https://aclanthology.org/2024.findings-acl.624/

[10] S. Willison, "The dual llm pattern for building ai assistants that can resist prompt injection," 2023, https://simonwillison.net/2023/Apr/25/dual-llm-pattern/.

[11] J. Kim, W. Choi, and B. Lee, "Prompt flow integrity to prevent privilege escalation in llm agents," *arXiv preprint arXiv:2503.15547*, 2025.

[12] 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.

[13] M. Costa, B. Köpf, A. Kolluri, A. Paverd, M. Russinovich, A. Salem, S. Tople, L. Wutschitz, and S. Zanella-Béguelin, "Securing ai agents with information-flow control," *arXiv preprint arXiv:2505.23643*, 2025.

[14] PinchBench, "PinchBench," 2026, https://pinchbench.com/.

[15] T. Liu, Z. Deng, G. Meng, Y. Li, and K. Chen, "Demystifying RCE vulnerabilities in LLM-integrated apps," in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2024.

[16] 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 *13th International Conference on Learning Representations, ICLR 2025*. Singapore: International Conference on Learning Representations, 2025. [Online]. Available: https://proceedings.iclr.cc/paper_files/paper/2025/hash/5750f91d8fb9d5c02bd8ad2c3b44456b-Abstract-Conference.html

[17] Z. Wang, V. Siu, Z. Ye, T. Shi, Y. Nie, X. Zhao, C. Wang, W. Guo, and D. Song, "AGENTVIGIL: Automatic black-box red-teaming for indirect prompt injection against LLM agents," in *Findings of the Association for Computational Linguistics: EMNLP 2025*. Suzhou, China: Association for Computational Linguistics, 2025, pp. 23 159–23 172. [Online]. Available: https://aclanthology.org/2025.findings-emnlp.1258/

[18] J. Yi, Y. Xie, B. Zhu, K. Hines, E. Kiciman, G. Sun, X. Xie, and F. Wu, "Benchmarking and defending against indirect prompt injection attacks on large language models," in *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. ACM, 2025. [Online]. Available: https://arxiv.org/abs/2312.14197

[19] E. Debenedetti, J. Zhang, M. Balunovic, L. Beurer-Kellner, M. Fischer, and F. Tramèr, "AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents," in *Annual*

Conference on Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track, 2024.

[20] H. Chang, E. Bao, X. Luo, and T. Yu, "Overcoming the retrieval barrier: Indirect prompt injection in the wild for LLM systems," in *35th USENIX Security Symposium*. USENIX Association, 2026. [Online]. Available: https://www.usenix.org/conference/usenixsecurity26/presentation/chang

[21] R. Wang, Y. Jia, and N. Z. Gong, "ObliInjection: Order-oblivious prompt injection attack to LLM agents with multi-source data," in *Network and Distributed System Security Symposium 2026*. San Diego, CA, USA: Internet Society, 2026. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/obliinjection-order-oblivious-prompt-injection-attack-to-llm-agents-with-multi-source-data/

[22] Z. Li, J. Cui, X. Liao, and L. Xing, "Les dissonances: Cross-tool harvesting and polluting in pool-of-tools empowered LLM agents," in *Network and Distributed System Security Symposium 2026*. San Diego, CA, USA: Internet Society, 2026. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/les-dissonances-cross-tool-harvesting-and-polluting-in-pool-of-tools-empowered-llm-agents/

[23] J. Shi, Z. Yuan, G. Tie, P. Zhou, N. Z. Gong, and L. Sun, "Prompt injection attack to tool selection in LLM agents," in *Network and Distributed System Security Symposium 2026*. San Diego, CA, USA: Internet Society, 2026. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/prompt-injection-attack-to-tool-selection-in-llm-agents/

[24] M. Herrador and J. Rehberger, "SpAIware: Uncovering a novel artificial intelligence attack vector through persistent memory in LLM applications and agents," *Future Generation Computer Systems*, vol. 174, p. 107994, 2026. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X25002894

[25] S. Dong, S. Xu, P. He, Y. Li, J. Tang, T. Liu, H. Liu, and Z. Xiang, "Memory injection attacks on LLM agents via query-only interaction," in *ICLR 2026 Workshop on Memory for LLM-Based Agentic Systems (MemAgents)*, 2026. [Online]. Available: https://openreview.net/forum?id=i7J62t2wtV

[26] D. Lee and M. Tiwari, "Prompt infection: LLM-to-LLM prompt injection within multi-agent systems," *arXiv preprint arXiv:2410.07283*, 2024. [Online]. Available: https://arxiv.org/abs/2410.07283

[27] E. Bagdasarian, R. Yi, S. Ghalebikesabi, P. Kairouz, M. Gruteser, S. Oh, B. Balle, and D. Ramage, "AirGapAgent: Protecting privacy-conscious conversational agents," in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3658644.3690350

[28] F. Wu, E. Cecchetti, and C. Xiao, "System-level defense against indirect prompt injection attacks: An information flow control perspective," *arXiv preprint arXiv:2409.19091*, 2024.

[29] E. Li, T. Mallick, E. Rose, W. Robertson, A. Oprea, and C. Nita-Rotaru, "ACE: A security architecture for LLM-integrated app systems," in *Network and Distributed System Security Symposium 2026*. San Diego, CA, USA: Internet Society, 2026. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/ace-a-security-architecture-for-llm-integrated-app-systems/

[30] F. Perez and I. Ribeiro, "Ignore previous prompt: Attack techniques for language models," *arXiv preprint arXiv:2211.09527*, 2022. [Online]. Available: https://arxiv.org/abs/2211.09527

[31] A. Wei, N. Haghtalab, and J. Steinhardt, "Jailbroken: How does LLM safety training fail?" *arXiv preprint arXiv:2307.02483*, 2023. [Online]. Available: https://arxiv.org/abs/2307.02483

[32] A. Zou, Z. Wang, N. Carlini, M. Nasr, J. Z. Kolter, and M. Fredrikson, "Universal and transferable adversarial attacks on aligned language models," *arXiv preprint arXiv:2307.15043*, 2023. [Online]. Available: https://arxiv.org/abs/2307.15043

[33] E. Wallace, K. Xiao, R. Leike, L. Weng, J. Heidecke, and A. Beutel, "The instruction hierarchy: Training LLMs to prioritize privileged instructions," *arXiv preprint arXiv:2404.13208*, 2024.

[34] S. Chen, J. Piet, C. Sitawarin, and D. Wagner, "StruQ: Defending against prompt injection with structured queries," in *34th USENIX Security Symposium*. USENIX Association, 2025. [Online]. Available: https://arxiv.org/abs/2402.06363

[35] S. Chen, A. Zharmagambetov, S. Mahloujifar, K. Chaudhuri, D. Wagner, and C. Guo, "SecAlign: Defending against prompt injection with preference optimization," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3719027.3744836

[36] Meta AI, "Llama Prompt Guard 2," 2025, https://huggingface.co/meta-llama/Llama-Prompt-Guard-2-86M.

[37] S. Chennabasappa, C. Nikolaidis, D. Song, D. Molnar, S. Ding, S. Wan, S. Whitman, L. Deason, N. Doucette, A. Montilla, A. Gampa, B. de Paola, D. Gabi, J. Crnkovich, J.-C. Testud, K. He, R. Chaturvedi, W. Zhou, and J. Saxe, "LlamaFirewall: An open source guardrail system for building secure AI agents," *arXiv preprint arXiv:2505.03574*, 2025, https://github.com/meta-llama/PurpleLlama/tree/main/LlamaFirewall.

[38] Y. Liu, Y. Jia, J. Jia, D. Song, and N. Z. Gong, "DataSentinel: A game-theoretic detection of prompt injection attacks," in *IEEE Symposium on Security and Privacy*. IEEE, 2025. [Online]. Available: https://arxiv.org/abs/2504.11358

[39] K. Hines, G. Lopez, M. Hall, F. Zarfati, Y. Zunger, and E. Kiciman, "Defending against indirect prompt injection attacks with spotlighting," *arXiv preprint arXiv:2403.14720*, 2024.

[40] Y. Zhong, Q. Miao, Y. Chen, J. Deng, Y. Cheng, and W. Xu, "Attention is all you need to defend against indirect prompt injection attacks in LLMs," in *Network and Distributed System Security Symposium 2026*. San Diego, CA, USA: Internet Society, 2026. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/attention-is-all-you-need-to-defend-against-indirect-prompt-injection-attacks-in-llms/

[41] K. Zhu, X. Yang, J. Wang, W. Guo, and W. Y. Wang, "MELON: Provable defense against indirect prompt injection attacks in AI agents," in *Proceedings of the 42nd International Conference on Machine Learning*. PMLR, 2025. [Online]. Available: https://arxiv.org/abs/2502.05174

[42] F. Jia, T. Wu, X. Qin, and A. Squicciarini, "The task shield: Enforcing task alignment to defend against indirect prompt injection in LLM agents," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2025. [Online]. Available: https://aclanthology.org/2025.acl-long.1435/

[43] H. Li, X. Liu, H.-C. Chiu, D. Li, N. Zhang, and C. Xiao, "DRIFT: Dynamic rule-based defense with injection isolation for securing LLM agents," in *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2025. [Online]. Available: https://arxiv.org/abs/2506.12104

[44] Z. Chen, M. Kang, and B. Li, "ShieldAgent: Shielding agents via verifiable safety policy reasoning," in *Proceedings of the 42nd International Conference on Machine Learning*. PMLR, 2025. [Online]. Available: https://arxiv.org/abs/2503.22738

[45] Z. Xiang, L. Zheng, Y. Li, J. Hong, Q. Li, H. Xie, J. Zhang, Z. Xiong, C. Xie, C. Yang, D. Song, and B. Li, "GuardAgent: Safeguard LLM agents by a guard agent via knowledge-enabled reasoning," in *Proceedings of the 42nd International Conference on Machine Learning*. PMLR, 2025. [Online]. Available: https://arxiv.org/abs/2406.09187

[46] H. Wang, C. M. Poskitt, and J. Sun, "AgentSpec: Customizable runtime enforcement for safe and reliable LLM agents," in *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering*. ACM, 2026. [Online]. Available: https://arxiv.org/abs/2503.18666

[47] P. Y. Zhong, S. Chen, R. Wang, M. McCall, B. L. Titzer, H. Miller, and P. B. Gibbons, "RTBAS: Defending LLM agents against prompt injection and privacy leakage," *arXiv preprint arXiv:2502.08966*, 2025.

[48] S. A. Siddiqui, R. Gaonkar, B. Köpf, D. Krueger, A. Paverd, A. Salem, S. Tople, L. Wutschitz, M. Xia, and S. Zanella-Béguelin, "Permissive information-flow analysis for large language models," *arXiv preprint arXiv:2410.03055*, 2024.

[49] J. Shi, Z. Yuan, Y. Liu, Y. Huang, P. Zhou, L. Sun, and N. Z. Gong, "Optimization-based prompt injection attack to LLM-as-a-judge," in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2024.

[50] NVIDIA, "OpenShell: A safe, private runtime for autonomous AI agents," 2026, https://github.com/NVIDIA/OpenShell.

[51] OpenClaw, "Webhooks," 2026, https://docs.openclaw.ai/cli/webhooks.

[52] ——, "Configuration," 2026, https://docs.openclaw.ai/gateway/configuration (accessed 29, May, 2026).
[53] Git Project, *git-worktree Documentation*, 2026, https://git-scm.com/docs/git-worktree (accessed 28, May, 2026).
[54] Google Workspace, "gws: Google Workspace CLI," 2026, https://github.com/googleworkspace/cli (accessed 5, June, 2026).
[55] GitHub, "GitHub CLI," 2026, https://cli.github.com/ (accessed 5, June, 2026).
[56] X. Fu, S. Li, Z. Wang, Y. Liu, R. K. Gupta, T. Berg-Kirkpatrick, and E. Fernandes, "Imprompter: Tricking LLM agents into improper tool use," *arXiv preprint arXiv:2410.14923*, 2024. [Online]. Available: https://arxiv.org/abs/2410.14923
[57] M. Russinovich, A. Salem, and R. Eldan, "Great, now write an article about that: The crescendo multi-turn LLM jailbreak attack," in *34th USENIX Security Symposium*. USENIX Association, 2025.
[58] W. Zou, R. Geng, B. Wang, and J. Jia, "PoisonedRAG: Knowledge corruption attacks to retrieval-augmented generation of large language models," in *34th USENIX Security Symposium*. USENIX Association, 2025.
[59] Y. Wu, F. Roesner, T. Kohno, N. Zhang, and U. Iqbal, "An execution isolation architecture for LLM-based agentic systems," in *Network and Distributed System Security Symposium 2025*. San Diego, CA, USA: Internet Society, 2025. [Online]. Available: https://arxiv.org/abs/2403.04960
[60] A. Kolluri, R. Sharma, M. Costa, B. Köpf, T. Nießen, M. Russinovich, S. Tople, and S. Zanella-Béguelin, "Optimizing agent planning for security and autonomy," in *International Conference on Learning Representations (ICLR)*, 2026.
[61] T. Shi, J. He, Z. Wang, H. Li, L. Wu, W. Guo, and D. Song, "Progent: Programmable privilege control for LLM agents," *arXiv preprint arXiv:2504.11703*, 2025.
[62] L. Tsai and E. Bagdasarian, "Conseca: Contextual agent security: A policy for every purpose," in *Proceedings of the 20th Workshop on Hot Topics in Operating Systems*. ACM, 2025. [Online]. Available: https://arxiv.org/abs/2501.17070
[63] G. Syros, A. Suri, J. Ginesin, C. Nita-Rotaru, and A. Oprea, "SAGA: A security architecture for governing AI agentic systems," in *Network and Distributed System Security Symposium 2026*. San Diego, CA, USA: Internet Society, 2026.
[64] Anthropic, "Claude Code hooks reference," 2026, https://code.claude.com/docs/en/hooks#hooks-reference (accessed 11, June, 2026).
[65] Nous Research, "Hermes Agent: Hooks," 2026, https://hermes-agent.nousresearch.com/docs/user-guide/features/hooks (accessed 11, June, 2026).
[66] F. Liu, Y. Zhang, J. Luo, J. Dai, T. Chen, L. Yuan, Z. Yu, Y. Shi, K. Li, C. Zhou *et al.*, "Make agent defeat agent: Automatic detection of Taint-Style vulnerabilities in LLM-based agents," in *34th USENIX Security Symposium*. USENIX Association, 2025, pp. 3767–3786.

Appendix A. Implementation

DUALVIEW is implemented as an OpenClaw plugin, based on OpenClaw v2026.3.12 [6]. The prototype builds on OpenClaw event hooks, a SQLite symbol table, Git worktrees for the two file-system views, and a YAML-format policy file at `~/.dualview/policy.yaml`.

OpenClaw integration. DUALVIEW is implemented with event hooks and custom tools, a mechanism supported in many personal agents (e.g., Claude Code [64] and Hermes Agent [65]). This hook-based design does not require changing the AI model, the agent main logic, or tool implementations, which makes it deployable on various personal agents. As in prior Dual LLM pattern defenses [11–13], DUALVIEW also registers `inspect_symbol` as a custom tool for U-LLM processing, which takes the symbols, the processing instruction for U-LLM, and an output schema for the result. DUALVIEW checks the U-LLM result against the schema, returns the symbolized result to AgentView when it matches, and otherwise returns an error without exposing any untrusted data.

Hook points. OpenClaw hooks let DUALVIEW mediate the agent lifecycle at initialization, tool execution, and human-facing replies (Figure 7). The pre-tool hook handles the work after T-LLM selects a tool but before the tool runs: for AgentView tools it copies HumanView file changes into the Agent File System, and for HumanView tools it checks the data usage policy and resolves the symbols the tool needs as original data. The post-tool result hook applies only to HumanView tools, which can return original untrusted data that must be symbolized before T-LLM reads it. DUALVIEW uses its `transform_tool_result` hook to apply the data trust policy to web results, HumanShell stdout, webhook payloads, and other HumanView tool results before T-LLM reads them; OpenClaw v2026.3.12 did not support this hook, so DUALVIEW added it, and the latest OpenClaw provides it natively as tool-result middleware. Final assistant messages use a separate delivery hook, since no tool call occurs after T-LLM generates the final text. DUALVIEW uses `message_sending` to resolve symbols only in the agent response that goes to the CLI, webchat, or channel, so the transcript keeps symbols while the human-facing message contains original data; OpenClaw v2026.3.12 did not deliver an edited response to the messenger channel, so DUALVIEW patched it.

Symbol table. DUALVIEW stores the symbol table in SQLite and shares it across all agent sessions. The symbol table maps each symbol to its original data and its provenance, allowing DUALVIEW to resolve symbols when a symbol is passed to HumanView. DUALVIEW resolves a symbol only on an exact match of its full token, so a command or edit that splits, truncates, or otherwise mangles a symbol leaves a fragment that DUALVIEW does not resolve. Such a fragment never reveals the original untrusted data, so resolution fails closed; it can leave a broken token in human-facing output and lower human utility, a case we did not observe in our

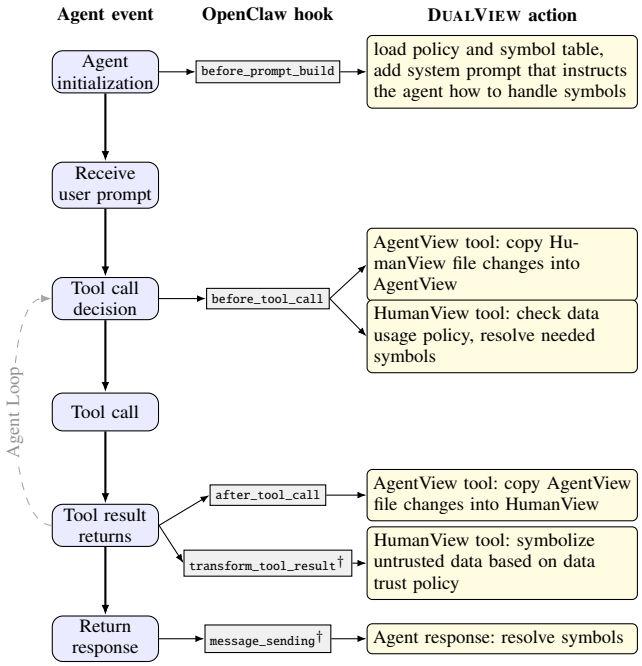


Figure 7: OpenClaw hook points that DUALVIEW uses across the agent's events. `before_tool_call` and the two result hooks branch by whether the tool runs on AgentView or HumanView. DUALVIEW changes the two hooks marked †.

evaluation. Each entry stores a symbol with the original data and provenance metadata. The provenance metadata comes from trusted data (e.g., the originating tool, the field path, and the data origin). DUALVIEW can extend the data usage policy with this provenance in the future, for example to block symbols of a given origin from sensitive tool arguments, which prevents taint-style vulnerabilities [66].

System Prompt. DUALVIEW appends a system prompt to agent context that explains how to handle its symbolized tool results and shell tools. It instructs the model to treat symbol tokens as opaque handles and pass them through unchanged, since they are resolved to their real values only at delivery time, and to call the `inspect_symbol` tool whenever it needs a derived result such as a summary, an extraction, or a transformation of symbolized data. It also documents the Data Trust Policy together with its `policy_add` and `policy_del` tools, and the two shell modes (i.e., shell tools). Default exec resolves symbols and marks the output untrusted, while restricted exec (`RESTRICTED=1`) runs sandboxed and returns trusted output. The full prompt text will be available in our released code.[1]

Filesystem. DUALVIEW tracks the files the agent modifies with Git and keeps the Agent File System and Human File System as two Git worktrees [53] (§3.5). Figure 8 shows an example layout of file systems. The original workspace directory is the Human File System, and DUALVIEW keeps its own Git database and the Agent File System outside that directory under `~/.dualview/workspaces/`, where it

1. https://github.com/compsec-snu/dualview

`/home/user/apple/banana/`	workspace
ordinary workspace files	*HumanView files*
`.git/`	ordinary Git, optional
`~/.dualview/workspaces/`	DUALVIEW file-system management
`%2Fhome%2Fuser%2Fapple%2Fbanana/`	encoded workspace path
`repo.git/`	DUALVIEW Git
`agentview/`	*Agent File System* for the workspace
`conflicts.log`	sync conflict log

Figure 8: Example file-system layout under DUALVIEW.

TABLE 5: PinchBench file paths marked as untrusted. These patterns match 72 files, and 42 of the 147 PinchBench tasks read at least one of them.

Category	Untrusted Files
Email and inbox	`emails/*`, `inbox/*`
External documents and reports	`ai_blog.txt`, `GPT4.pdf`, `openclaw_report.pdf`, `research/*`, `sample_contract.pdf`, `school-calendar.pdf`, `subway_map.md`, `vulnerability_scan.json`
Logs and request metadata	`access_events.csv`, `apache_error.log`, `auth.log`, `hdfs_datanode.log`, `linux_syslog.log`, `mapreduce.log`, `nginx_access.log`, `syslog.log`

canonicalizes and percent-encodes the root path into a single directory name. Because DUALVIEW keeps its Git database outside the workspace, its Git use stays transparent to the user, so ordinary Git commands in the workspace work seamlessly. DUALVIEW starts tracking a new workspace on demand when the agent first writes a file with a tool such as `write` or `edit`. If the file is in a Git project, the workspace root is the enclosing project directory, otherwise it is the parent directory of the file. Unlike a file tool, a shell command can write to paths DUALVIEW cannot anticipate, so DUALVIEW does not open a new workspace for AgentShell and instead confines its writes to already-tracked workspaces, keeping every write inside a worktree that the post-command sync reconciles.

DUALVIEW handles potential file write race conditions in two ways. First, a race between two agents that sync the same root is handled by serializing sync commits with a workspace-scoped file lock. Second, a race between a human write and a sync, which the lock does not prevent, is handled by detecting the changed file hash, skipping the affected file so DUALVIEW never overwrites the human's change, and appending a record to the conflict log. DUALVIEW currently only reports the conflict through the log and leaves the resolution to the human.

Appendix B. DUALVIEW Policy

Policy specification. Table 7 summarizes the prototype data trust policy that DUALVIEW uses for OpenClaw built-in tools and webhook receivers. Fields listed as trusted pass through to the agent as original data. Fields listed as untrusted are symbolized before the agent reads them. Webhook fields omitted from an endpoint schema are untrusted by default.

TABLE 6: PinchBench output routes made read-only for the OpenShell sandbox utility run. These routes encode effects that would be sensitive in a real personal AI agent deployment. Across PinchBench, 60 of the 147 tasks write to at least one of these routes.

Category	Restricted Path
Email replies	`output/email_replies/`
Security, contract, and triage reports	`output/security/`
Meeting decisions, recommendations, and summaries	`output/decisions/`
CI, Kubernetes, logs, and service actions	`output/operations/`
Research, finance, procurement, and stock outputs	`output/business/`

Examples. Table 8 shows how the two DUALVIEW policies apply at different boundaries. The upper block applies the data trust policy after a HumanView tool returns. The rule on the left names the trusted and untrusted fields or origins, and the result on the right is what T-LLM sees after DUALVIEW replaces untrusted fields with symbols. The lower block applies the data usage policy before HumanShell runs a command. Commands with no symbol in executable text run without extra approval, while commands or script files that would execute symbolized data require human approval.

PinchBench policy. Of the 147 PinchBench tasks, 114 read a file as input and 121 write a file as the goal. For example, an email reply task reads an inbox under `emails/*` or `inbox/*` and writes the reply as a file in the workspace. For DUALVIEW and the Dual LLM pattern, we marked untrusted input files so that their content is symbolized (Table 5); examples include `research/*`, `GPT4.pdf`, and `auth.log`, and 42 tasks read at least one such file. For Sandboxing, we placed privileged output files, such as email replies and business outputs, in dedicated directories and denied writes there, and minimized network access to prevent data leakage and arbitrary actions (Table 6); the blocked routes cover 60 tasks.

TABLE 7: Prototype data trust policy specification. The policy uses the same trusted and untrusted field distinction for built-in tools, structured shell output, webhook receivers, and inter-agent communication.

Tool or source	Trusted fields	Untrusted fields
`web_fetch`	`url`, `status`, `contentType`, `extractMode`, `extractor`, `fetchedAt`, `tookMs`, `truncated`, `length`, `rawLength`, `wrappedLength`, `externalContent`, `warning`	`finalUrl`, `title`, `text`
`web_search`	`query`, `provider`, `count`, `tookMs`, `externalContent`	Per-result `title`, `description`, `url`, `published`, `siteName`
`exec`	Fields declared by per-skill output schemas, such as local counters, identifiers, and schema tags	Default stdout, plus remote or user-controlled fields declared by per-skill output schemas
`read`	File content from AgentView, where untrusted content is already symbolized	None
`write` and `edit`	Tool status and other local metadata	None
`process`	Write and lifecycle actions, `list`, and trusted local metadata	Read actions such as `poll` and `log`, plus unknown actions
`inspect_symbol`	Validated output after U-LLM processing and re-symbolization	None
Webhook receivers	Endpoint-specific schema fields such as verified source metadata, event type, and local delivery identifiers	Endpoint-specific schema fields such as message text, issue comments, form bodies, and fields omitted from the schema
Image and media tools	Local tool metadata and URL-keyed identifiers	Remote captions, descriptions, or other service-controlled text when present
Inter-agent session tools	Local conversation identifiers and private context metadata	Message text and tool-result text from untrusted agent contexts
Other local built-in tools	`memory_search`, `memory_get`, `session_status`, `canvas`, `nodes`, `cron`, `tts`, `message`, and `claude_code_*` results that contain local metadata only	Remote or externally controlled text when a deployment configures such fields

TABLE 8: Concrete examples for the two policy types. The data trust policy block shows schema rules and user-supplied origin rules. The data usage policy block shows executable command patterns and command rewriting rules. The dark red text marks symbolized fields or command and code text that HumanShell would execute, and a command with no symbol in executable text runs without human approval.

Policy rule	Tool result or command		DualView output
Data trust policy *(checks tool output)*			

Schema rules
Trusted
`web_fetch.url`
Untrusted
`web_fetch.text`
	`web_fetch(url) = {`
	`  url=https://reports.example/q1,`
	`  text="Quarterly report"`
	`}`
		⇒	`web_fetch(url) = {`
			`  url=https://reports.example/q1,`
			`  text=$web1.text`
			`}`

Origin rules
Trusted
`api.github.com/*`
	`web_fetch(https://api.github.com/zen) = {`
	`  content="Design for failure."`
	`}`
		⇒	`web_fetch(https://api.github.com/zen) = {`
			`  content="Design for failure."`
			`}`

Untrusted
`imports/*.csv`
	`read(imports/q1.csv) = {`
	`  content="name,amount..."`
	`}`
		⇒	`read(imports/q1.csv) = {`
			`  content=$read1.content`
			`}`

Untrusted
`agent:public-chat`
	`session_history(agent:public-chat) = {`
	`  message="check this link"`
	`}`
		⇒	`session_history(agent:public-chat) = {`
			`  message=$session1.message`
			`}`

Data usage policy *(checks commands sent to HumanShell)*			
Executable Command Pattern	`exec(git status)`	⇒	No human approval required
`<symbol>`	`exec($web1.text)`	⇒	`$web1.text` needs human approval for execution
`python -c <symbol>`	`exec(python -c $web1.text)`	⇒	`$web1.text` needs human approval for execution
Command Rewriting Rule			

`python <file>`
$\xrightarrow{\textit{transform}}$
`python -c <file content>`
	`exec(python fix.py)`
	$\xrightarrow{\textit{transform}}$
	`exec(python -c "...$web1.text...")`
		⇒	`$web1.text` needs human approval for execution