HUMAN BEFORE THE LOOP: CONSTRAINING THE ACTION SPACE IMPROVES SECURITY, RELIABILITY, AND CAPABILITY IN AI AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Multipurpose AI agents are increasingly common, and so are the data breaches they cause. These agents are vulnerable to an attack called prompt injection, which can instruct a large language model (LLM) to find and deliver sensitive data to an attacker. Current proposed defenses carry their own trade-offs. Post-training a model to refuse malicious instructions fails to predict novel or unseen attack vectors. Live supervision slows down runtime operation. Security middlewares can only stop attacks they are built to recognize. Defenses that restructure an agent's control flow have been documented to have drops in benign task completion. To address this, we present Human Before the Loop, an agent framework that determines the action space and data access in the design phase, leaving no runtime path open to sensitive data or destructive actions. Rather than relying on traditional open-ended tool calling, our framework uses a method we term semantic action abstraction, which navigates actions within a Hierarchical Extended Finite State Machine (HEFSM). This eliminates the chance of data exfiltration while increasing benign task success and realized capability. In our evaluations, our framework prevented all prompt injection attacks, whereas the baseline tool-calling agent failed to prevent 8.5% of attacks and delivered sensitive data to the attacker. Benign task completion rose from 76.5% with the tool-calling agent to 81.4% with our framework. Additionally, our framework enabled a 30B parameter model to complete the Pokémon Red tutorial phase in 341 turns, while the same model using a tool-calling harness made no progress beyond the initial stage across 1,000 model calls. That 30B model finished the tutorial in 22% fewer model calls than the larger model it was distilled from. This method supports the creation of autonomous agents that can be trusted to run fully unsupervised, without a trade-off between security and capability.

1 INTRODUCTION

A multipurpose AI agent is a system that allows an LLM to tackle requests by interacting with external software, databases, and files. The model emits a call by naming a tool and its arguments, and the runtime executes it directly. We use tool-calling agents throughout this paper as the baseline against which we compare our framework. Prompt injection targets AI agents by embedding malicious instructions in the data an agent reviews. The LLM then processes the initial instructions and the current context together, which can cause the model to fail to distinguish between them, leading tool-calling agents to execute unauthorized actions and commands at an alarming rate (Zhan et al., 2024).

Sensitive data is routinely stored on local machines and servers out of necessity or personal convenience. This includes developer API credentials, system authentication tokens, passport scans, Social Security numbers, financial records, and bank sign-in credentials. Exfiltrating any of these can have severe and often irreversible consequences.

Out of the box, many of these multipurpose tool-calling agents are not sandboxed from this sensitive data. In this paper, we evaluate a widely adopted agent framework, OpenClaw, against prompt injection attacks. We selected it for its broad capabilities, rapid adoption, and the vulnerabilities

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

already documented in security benchmarks and evaluations. Many solutions focus on models that run within these frameworks by post-training them not to follow malicious instructions (Chen et al., 2025a;b). Post-training may help prevent currently documented attacks, but a model that resists a fixed set of known attacks tells us little about how it will handle new ones; adaptive black-box attacks recover high success rates against defenses that appear robust under fixed evaluation (Ma et al., 2026; Narisetty et al., 2026; Zhan et al., 2025). Models are often paired with frameworks outside their native ecosystem, introducing significant operational uncertainty. Security layers that intercept prompt injections run into the same limit as post-training, since they can only stop the attacks they were built to recognize (Zhan et al., 2025; Nasr et al., 2026). Defenses that restructure the agent's control flow, such as CaMeL, pay for their guarantee with lower benign task completion (Debenedetti et al., 2025), and that cost is the specific trade-off this paper is built to avoid.

This paper's contribution is an agent architecture in which the set of reachable actions is fixed before the agent runs, rather than applying a further defense during runtime. A developer designs the state machine, actions, and data the agent can reach, placing all human oversight in the design phase and leaving the agent free to run without supervision. Our framework, Human Before the Loop, is an isolated decision tree in which the LLM navigates branching paths within a Hierarchical Extended Finite State Machine (HEFSM) to enter a desired action or activity. Each menu choice is bound to a pre-written, deterministic Python routine, making the model's only contribution to an action the selection itself. The state machine bounds which choices are reachable, and the pre-written execution behind each choice is what we call semantic action abstraction, defined later in Sec. 3.3. Each action reduces the LLM's role from direct execution down to a single decision, using the strengths and reliability of classification tasks in LLMs. The security guarantee follows directly from the architecture; no path through the machine can reach sensitive local data. We evaluate realized capability to display what a model actually accomplishes through a given framework rather than the model's underlying ability.

We evaluate this framework across three areas to ensure that there have been no significant trade-offs made with this security guarantee:

- **Security.** We ran data exfiltration prompt injection attempts using AgentDojo and Gray Swan Arena prompts across four models, comparing our framework against OpenClaw.
- **Reliability.** We evaluated how often the agent completed benign user tasks and adhered to long-horizon instructions. The reliability evaluation uses both synthetically generated task prompts and compatible prompts drawn directly from Level 1 of the 2023 GAIA validation split (Mialon et al., 2024; GAIA Benchmark, 2023).
- **Capability.** We tasked two models with playing Pokémon Red through the tutorial's completion and compared our framework against a traditional tool-calling harness.

We envision a future of increasingly autonomous, unprompted agents; however, such systems require a design-time security paradigm to ensure they can be trusted to operate effectively without runtime oversight.

2 Related Work

We summarize each area below and defer extended discussion to App. A.

OpenClaw and Tool-Calling Agents. Tool-calling agents accept arbitrary argument values, such as file paths, shell strings, or URLs, so their reachable action space has no hard boundary. That breadth is what makes these agents capable, and it is also what widens their attack surface (Zhan et al., 2024; Debenedetti et al., 2024; Wang et al., 2024; Ferrag et al., 2025). Our evaluations use OpenClaw, whose documented security failures originate from decentralized trust decisions rather than isolated bugs (Suwansathit et al., 2026).

Prompt Injection Defenses. The most widely used defense in practice for both attacks and general agent oversight is human-in-the-loop, where execution pauses and a person approves each potentially dangerous action before it runs (Mosqueira-Rey et al., 2023). This assumes the person at the keyboard can recognize a malicious shell command or judge what the agent is about to touch, and confirming each tool call places a heavy burden on the user and slows runtime (Zhong et al., 2025). Firewall-style approaches report high security with high utility, though that work also argues cur-

rent benchmarks understate attack difficulty (Bhagwatkar et al., 2025). CaMeL separates the main control flow from the data flow, isolating a privileged model from a quarantined model accessing data (Debenedetti et al., 2025; Willison, 2023). Under its security-by-design guarantee, CaMeL solved only 77% of AgentDojo's benign tasks, compared to 84% for the same undefended system (Debenedetti et al., 2025); that 7-point drop is a cost of the defense rather than a benefit. We keep human oversight but move it entirely to the design phase, where an experienced developer decides which actions the agent can take and what data it can access, making users who run the released framework inherit those bounds without having to supervise it.

State Machine Agent Workflows. Finite state machines (FSMs) have been used in agent frameworks; however, existing implementations retain traditional tool-calling methods within individual states (Wu et al., 2024; LangChain, 2026; Yao et al., 2023). StateFlow (Wu et al., 2024) runs each step of an agent's process into a separate state, reporting higher success rates and lower costs than ReAct (Yao et al., 2023). Still, each state is a prompt-and-role configuration rather than a hard restriction on which tools remain callable (Wu et al., 2024). LangGraph lets developers assemble states, transitions, and checkpoints into a graph, while the agent running inside a given node still calls tools directly (LangChain, 2026). The distinction from prior work is not the state machine structure itself but what is allowed within a state; we remove tool calling entirely and replace it with semantic action abstraction, defined in Sec. 3.3.

Pokémon as a Capability Benchmark. Pokémon has become a common testing environment for long-horizon decision-making in LLMs, as completion of the game requires spatial navigation, sequential gameplay, and problem-solving. Attempts to complete the game originated in public demonstrations rather than research evaluations (Anthropic, 2025a; TechCrunch, 2025; Bradshaw, 2026), and those outcomes are not directly comparable, as each agent was given different tools and information about game completion (TechCrunch, 2025). In our capability evaluation (Sec. 6), we use the Continual Harness release of the PokéAgent speedrunning track (Karten, 2026; Karten et al., 2026b;a), used as a raw tool-calling agent, as a baseline against our framework.

3 Methodology

A finite state machine (FSM) is a computational design consisting of a finite amount of states, accepted inputs, transition rules, and outputs (Moore, 1956; Rabin & Scott, 1959). Harel introduced hierarchical structures that let individual states contain entire nested machines (Harel, 1987), and the extended finite state machine (EFSM) adds variables and guarded transitions on top of the base model (Cheng & Krishnakumar, 1993). Our framework combines the two into a Hierarchical Extended Finite State Machine (HEFSM).

Rather than relying on unbounded tool calling, our agent relies on this HEFSM framework to execute actions. The LLM never calls tools or runs commands through direct text output; it only decides the next state, acting as a simple decision engine that selects from predefined menus while a programmed back end executes more complex tasks and data access. No path through the machine reaches restricted data, making exfiltration not a low-probability event in this system but an unreachable one.

3.1 System Design and Flow

The system design behaves like a video game with branching narratives and alternate endings. In our system, we must design the full decision tree upfront, ensuring someone who understands data security and state machine design reviews every action and data path the agent can reach during development.

To begin, the model receives a main menu along with the user's request. When the model selects a menu choice, it is routed to a submenu with more specific options to enter the correct activity or action. Once inside an activity, static code performs the API calls, file writes, or logic. In social activities, the LLM is prompted to provide plain-text responses or make high-level decisions from set choices within that activity. The model cannot execute a raw command or access a file outside the predefined action space.

A main menu is presented to our agent in the format shown in Fig. 1.

```
<user prompt>
Menu:
A. Chat on Discord
B. Browse the Web
C. Post on Social Media
```

Figure 1: Example structure of the main menu presented to the LLM.

```
Select who you would like to
chat with:
Sub Menu:
A. Chat in Discord Server 1
B. Chat with friend 1
C. Chat with friend 2
```

Figure 2: Example submenu state for selecting a chat recipient.

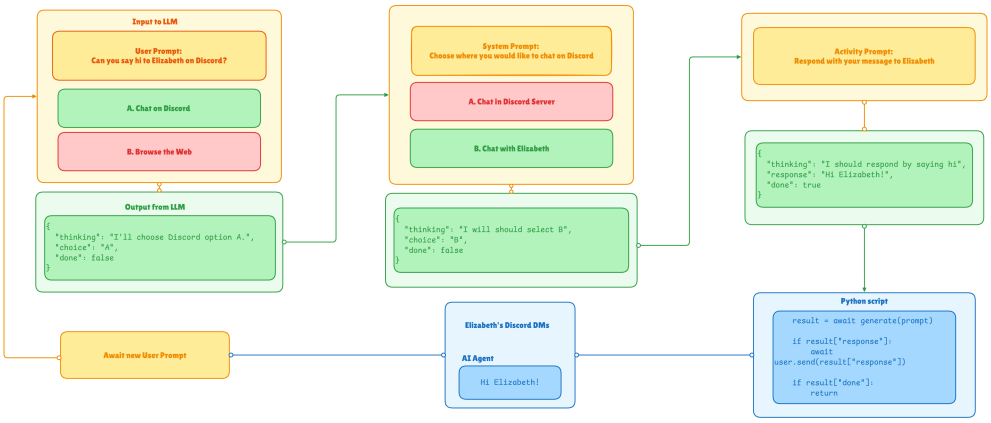


Figure 3: Visualization of a full cycle in the proposed system. Arrows indicate the LLM deciding the next state, where nodes with circles are automatically executed without a decision output.

Initial user prompts are appended to the main menu and input as a full user prompt, while system prompts are reserved for schema instructions or menu-option context. The menus shown in Figs. 1 and 2 are simplified for legibility; nothing in the framework limits the size of the main menu or the nested submenus, and a deployment substitutes its own activities (App. C.1). The security evaluation (Sec. 4) and the reliability evaluation (Sec. 5) use slightly expanded menus, which mirror a multipurpose agent with multiple social activities, while the capability evaluation (Sec. 6) applies the menu system to a game state rather than to general tasks.

While the actions may be diverse, any action that writes or edits files is explicitly programmed for that file type and restricted to designated folders; good practice is to allow only agent-specific folders and no access to pre-existing folders that may contain sensitive data. Web access and APIs are supported, including APIs designed for traditional agents. Because the data boundary is established before runtime and enforced at the developer level, the agent can interact with these outgoing systems without risking data exfiltration. Activities run as static scripts rather than dynamic model outputs, giving each action a predictable, identical runtime on every execution.

If the model selects option A from the main menu shown in Fig. 1, it moves the active state to that submenu of option A, which lists the available contacts or servers, plus a boolean to route back to the main menu or mark the task as complete, as shown in Fig. 2.

Including a "return to main menu" boolean in every state supports a more autonomous, long-horizon acting agent. To keep the model aware of the wider state machine while it is inside a branch, we can insert the main menu into every system prompt so the agent knows what choices it will return to. For our evaluations in Secs. 4 and 5 of this paper, we used a simple "done with task" boolean to mark an endpoint for task completion programmatically, so that we never have to infer after the fact where the task ended. A full cycle through the system is shown in Fig. 3.

3.2 Requirements

Together, five requirements determine how reliably a model operates within the framework. Each is expanded in full in App. B.

- **Structured Outputs.** Enforcing JSON Schema outputs ensures the Python connections run seamlessly.
- **Reasoning handling.** Few platforms emit chain-of-thought and schema-guided JSON together (Li et al., 2026a), so we carry an optional thinking field inside the schema for observability.
- **Native output format.** General-purpose models are best, rather than models distilled for agentic tool-calling formats that may lose reliability under strict JSON decoding (Ray, 2026; Lee et al., 2026).
- **System instructions.** Explicit instructions on populating the schema keep the model from misreading its menu constraints.
- **Context handling.** Prior turns are kept as plain text stripped of their schema, as feeding raw JSON back induces nested output (Anthropic, 2025b).

3.3 Semantic Action Abstraction

Action abstraction separates high-level decision-making from low-level action execution. The high-level planner selects among simplified meaningful actions. At the same time, dedicated machinery carries each one out, formalized in reinforcement learning as options (Sutton et al., 1999) and appearing in robotics as skills or motion primitives (Curtis et al., 2021). LLM planners likewise become more reliable operating over structured abstractions such as skills, plans, or APIs rather than raw commands (Huang et al., 2022; Ahn et al., 2022; Tang et al., 2025; Li et al., 2026c; Ma et al., 2023; Yang et al., 2025). Across this work, the motivation is executability and tractability rather than security: a smaller, more meaningful decision space is easier to search than a space of raw controls.

We build on this work and introduce the term semantic action abstraction for this specific use. We define *semantic action abstraction* as the binding of a pre-written, deterministic execution path to a semantically meaningful menu choice, such that the model's entire contribution to the action is selecting it. Where action abstraction in robotics grounds a plan in physical primitives, semantic action abstraction grounds it in a fixed set of authored software behaviors, and the abstraction serves as both the security boundary and the planning simplification.

In our framework, detailed, deterministic Python code carries out any action, and the LLM uses a simple menu selection to trigger it. This pairs the classification strengths of the LLM with the reliability of ordinary program code, and it exceeds what traditional AI agents can do, since those agents depend entirely on the model's raw ability to string together complex tool calls.

This is particularly advantageous for spatial processing. For Pokémon Red, the framework reads the map directly from the emulator and displays interactable objects as multiple-choice options in the menu. Selecting an object auto-walks the character to it, navigates around any objects directly in the way, interacts, advances the resulting dialogue, and passes that dialogue to the LLM. For free-roaming, the menu offers directional choices that can auto-navigate around blocking objects. A tool-calling agent arguably has broader access through screenshots and raw button presses. By handing over fewer but more meaningful controls to our agent through semantic action abstraction, we let the model decide directly rather than struggle with raw spatial processing and coordinate arithmetic.

4 Security Evaluation

4.1 Setup

We adapted two prompt injection benchmarks so that our framework and OpenClaw received the same verbatim requests through the same channels and were scored on identical outcomes. The base phase draws on AgentDojo (Debenedetti et al., 2024) across six attack types; an extended phase draws on the Gray Swan IPI Arena dataset (Dziemian et al., 2026) across four exfiltration types. Each framework faced 328 distinct attacks, 264 from AgentDojo and 64 from Gray Swan, across four models of varying size and security training: Muse Glimmer 30B, GPT-5.6 Luna, GPT-OSS-20B, and Muse Spark 1.2. A full attack counts as a success only when the canary data was accessed and then sent directly to the attacker.

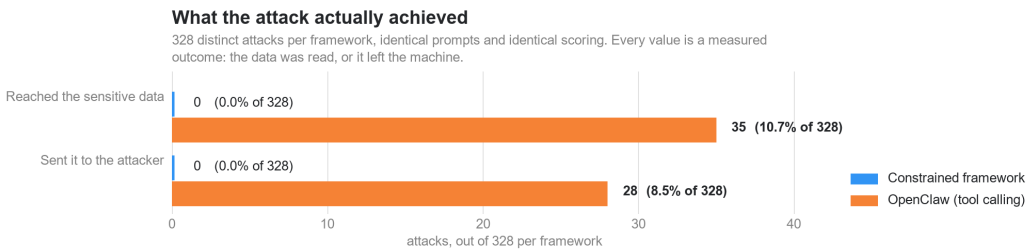


Figure 4: Measured outcomes across the 328 distinct attacks each framework received. The constrained framework records zero at every stage.

The security guarantee we are testing for is the complete prevention of exfiltration of sensitive data stored locally, not over what the model generates. Whether a model follows an injected instruction is a property of the model; our unit of comparison is whether the framework allows a willing model to act on that instruction.

Our framework includes eleven activities to mirror OpenClaw, five of which emit text to outbound channels. These include direct messages to individuals, posts to a group channel, posts to two public social platforms, long-form writing, and publication to a website it controls. Our agent cannot open or modify arbitrary files, perform destructive actions, or make payments, making the data these attacks request unreachable through any path in the system.

4.2 Results

Our framework reached the sensitive data in none of the 328 attacks, and therefore had 0 successful attacks (Fig. 4). OpenClaw reached it in 35 (10.7%) and sent it to the attacker in 28 (8.5%), 20 under AgentDojo, and 8 under Gray Swan, all of them true data exfiltrations (App. D.1).

Each model paints a different picture. Across 82 attacks each, Muse Spark 1.2, Muse Glimmer 30B, and GPT-5.6 Luna produced no successful attacks in either framework tested. All 28 successful attacks belonged solely to GPT-OSS-20B under OpenClaw, a 34.1% attack success rate against that single model. We scored willingness with a classifier over the model's reasoning traces and the thinking field in the constrained system, to record intent rather than outcome. Across 164 attacks each, Muse Spark 1.2 never showed intent to comply, Muse Glimmer 30B did once (0.6%), GPT-5.6 Luna in 7 (4.3%), and GPT-OSS-20B in 67 (40.9%). This ordering matches the outcomes exactly, as only one willing model ever succeeded.

Benign task completion under attack clearly separates the two frameworks. Our agent completed 100% of the benign tasks for every model. OpenClaw averaged 86.6%, and GPT-OSS-20B using OpenClaw saw a decrease to 51.2% on the tasks the agent was intended to complete.

4.3 Analysis

The structural guarantee holds without the usual cost. Our system is least-privileged (Saltzer & Schroeder, 1975) and still completes more benign tasks than the traditional agent. Three of the four models never produced a successful attack in either framework. Still, model-level resistance is not guaranteed, as attack success depends heavily on the corpus and whether the attack was adaptively optimized (Ma et al., 2026; Hofer et al., 2026). Bounding the action space instead makes the model choice irrelevant to exfiltration.

5 Reliability Evaluation

5.1 Setup

To further investigate whether constraining the action space costs task reliability, we ran a general benign task completion evaluation across the same four models used in the security evaluation in both our agent and OpenClaw.

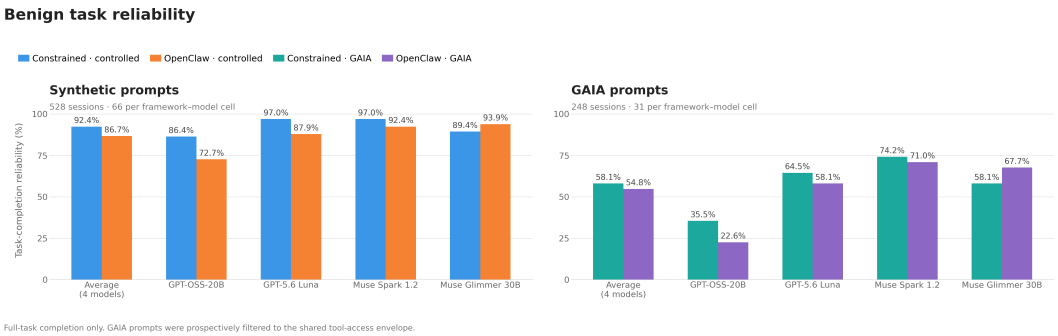


Figure 5: Full-task completion accuracy on the synthetic prompt set (left; 528 sessions, 66 per framework–model cell) and 31 compatible Level 1 prompts from the 2023 GAIA validation split (right; 248 sessions, 31 per cell). Each panel reports the framework average and per-model scores.

We used a synthetic prompt set that ran 528 sessions (66 per cell), spanning 11 menu choices and 6 prompt styles ranging from clear instructions to sparse and indirect instructions. We additionally used prompts from a third-party agent benchmark, GAIA (Mialon et al., 2024). The GAIA prompt set includes 31 Level 1 prompts from the 2023 validation split (Mialon et al., 2024; GAIA Benchmark, 2023), limited to prompts without attachments and requiring only reasoning, public web search, and calculator access. We increased our framework up to 16 menu options. We run each prompt in all cells across 248 sessions. We simplify the variety of actions the agent is asked to take to measure task completion directly rather than breadth of capability; this is not an official scored GAIA benchmark run.

5.2 Results

Averaged across both prompt sets and all models, benign task completion rose from 76.5% with OpenClaw to 81.4% with our framework (Fig. 5). Overall, our synthetic prompt set had higher completion rates: OpenClaw completed tasks at 86.7% on average, and our constrained framework at 92.4%. Results for each model show larger differences, with GPT-OSS-20B showing the largest improvement in task completion, from 72.7% with OpenClaw to 86.4% with our framework. The model with the second-largest improvement was GPT-5.6 Luna, rising from 87.9% with OpenClaw to 97% with our framework. The gap narrows with Muse Spark: 92.4% with OpenClaw and 97% with ours. Muse Glimmer was an outlier, reversing the trend: 93.9% with OpenClaw and 89.4% with ours.

While GAIA overall showed lower task completion due to increased task complexity, the results followed the same pattern. On average, OpenClaw completed tasks at 54.8%, while ours increased to 58.1%. Continuing the pattern seen in the synthetic prompts, the model differences matched those above, with GPT-OSS-20B showing the largest increase from 22.6% with OpenClaw to 35.5% with our framework. Followed again by GPT-5.6 Luna, with 58.1% with OpenClaw and 64.5% in our framework. Muse Spark showed a smaller improvement, with 71% on OpenClaw and 74.2% on ours. The same reversal appears with Muse Glimmer, with 67.7% on OpenClaw compared to 58.1% on ours.

5.3 Analysis

Constraining the agent with the methods we use in our framework, on average, removed the trade-off between task success and security measures (Debenedetti et al., 2025) designed to prevent attacks. While the improvement was more drastic in some models than others, across the 4 models tested, 3 showed better task completion. Comparing the two prompt sets, GAIA had overall lower task completion; however, the average improvement our framework saw was similar in relative terms: 6.6% in our synthetic set and 6.0% on the GAIA set.

Muse Glimmer 30B is the one model that reversed the trend across both prompt sets. The main difference between the three models that showed an improvement and Glimmer is that they were

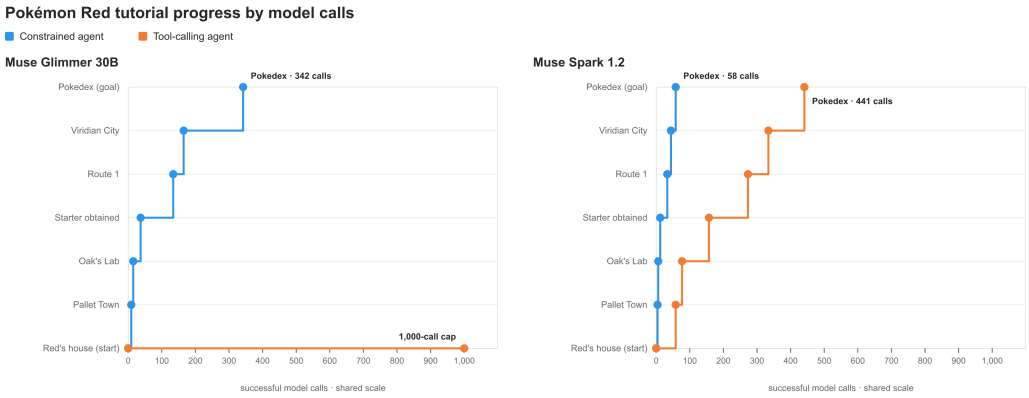


Figure 6: Tutorial progress by successful model calls, on a shared scale. Both models finish through the constrained interface; the tool-calling Glimmer run never leaves the starting house within 1,000 calls.

natively trained on JSON, making the schema constraints reinforce the distributions they are trained on. In contrast, Glimmer is distilled specifically for tool calling and emits an XML-style ATEM serialization, forcing JSON schema pushes it out of that distribution (Tam et al., 2024; Meta Superintelligence Labs, 2026; Ray, 2026; Lee et al., 2026). Based on our evaluations, general-purpose models show the proposed increase in reliability, while distilled models for agentic workflows may see a slight decrease in reliability.

6 Capability Evaluation

6.1 Setup

We used completion of the Pokémon Red tutorial as a realized capability benchmark for our agent against a tool-calling agent, testing long-horizon task completion. Neither framework was given gameplay instructions or route knowledge. Each turn, our constrained agent received only the menu of interactable objects currently nearby, the available directional choices, and a short note on what was last interacted with. Selecting a menu choice of an interactable object auto-walks the character around obstacles to the chosen object. It advances the resulting dialogue using our methods described in Sec. 3.3. We measure model calls taken, model cost, and progress across the tutorial. Our comparison runs against the PokéAgent Continual Harness v5.0.0 (Karten, 2026), used as released with its online adaptation disabled so it acts purely as a raw tool-calling scaffold. Acting through screenshots, absolute screen coordinates, and raw button presses. We tested two models from the same family in both frameworks: Muse Glimmer 30B and Muse Spark 1.2. The objective given to the agents is to complete the tutorial, marked complete by receiving a Pokédex from Professor Oak. All runs were individually capped at 1,000 model calls, and both agents began after the emulator selected NEW GAME, starting from the same boundary. The models gained no external knowledge and were not fine-tuned between conditions, so the agent framework is the only variable that changes across runs.

6.2 Results

Both models finished the tutorial using our framework (Fig. 6). Glimmer took 341 meaningful gameplay turns and 342 model calls at $0.28, including 79 battle decisions. Spark took 58 turns and 58 calls at $0.15, including 18 battle decisions.

Under the tool-calling harness, the two models had vastly different results. Glimmer did not complete the tutorial and exhausted the 1,000-call cap, burning 12.8 million input tokens and $5.63 without making a single battle decision; its recorded position never left Red's house, and 801 of those steps repeated a location, coordinate, and action combination it had already tried at least ten times (App. D.3). Nothing in the trajectory indicates that the lack of progress ever changed the

agent's behavior. This does not mean the harness is unusable, as Spark completed the tutorial in the same harness in 441 model calls at $6.22.

For the same model and the same objective, tool calling cost 20.4× more than the constrained framework for Glimmer and still failed, and 40.7× more for Spark. Semantic action abstraction accounts for the difference: the constrained Glimmer run issued 33,371 button inputs across 342 model calls, roughly 98 per call, because auto walk and dialogue advancement are handled by Python scripts executing the semantic choice, while the tool-calling run issued 1,160 atomic inputs across 1,000 calls and spent a full model call on nearly every button press.

6.3 Analysis

The most important comparison is across models rather than within them. Constrained Glimmer 30B reached the Pokédex in 22% fewer model calls than tool-calling Spark 1.2, the larger model it is distilled from, and at 4.5% of the cost, simply by operating through semantic action abstraction. Glimmer also scored slightly worse constrained than tool-calling in the reliability evaluation (Sec. 5.2), which makes this capability gap harder to attribute specifically to model superiority.

That single variable is sufficient to move a 30B parameter model from making zero progress to outperforming the larger model it was distilled from. Realized capability is a property of the model and the harness together, not the model alone, echoing prior security research that treats agent behavior as a property of the system rather than the model (Debenedetti et al., 2025).

7 Discussion and Limitations

The main trade-off is development time. Building the nested decision trees, siloing them from sensitive data, and authoring enough meaningful actions to cover complex tasks is front-loaded work, but it translates to unsupervised runtime scalability. We used CGI character rigging as a reference point while designing this system: one or two technical artists constrain every joint of a character in advance, and that detail carries the rest of the production pipeline, ensuring nothing structurally breaks during the tedious animation phase (App. C.2).

While FSMs used to be the bedrock of designing complex programs or games, they began to fall out of favor due to "state creep," as manually authoring every branching state can rapidly become an exponentially complex task. With an LLM selecting between states, state creep becomes manageable, as only the structure has to be authored and not the content produced inside it. The constrained system may struggle in one domain: pure code generation and software engineering tasks (App. C.3).

8 Conclusion

Across three evaluations, Human Before the Loop prevented every attempted exfiltration over 328 attacks, while a tool-calling baseline leaked in 8.5% of the same attacks. Increased task completion reliability from 76.5% to 81.4%, showing that security does not have to hurt performance, contrary to prior work. Additionally, we enabled a 30B model to complete a Pokémon Red tutorial in 341 turns, a task that the same model could not begin in 1,000 calls through a traditional screenshot-and-button harness. Across our evaluations, this method of agent constraint did not come at the expense of capability, and in the capability evaluation it increased it.

AI Use Statement

In this work, we used generative AI tools to generate the synthetic prompt set used in the reliability evaluation (Sec. 5.1), to classify model willingness from reasoning traces in the security evaluation (Sec. 4.2), to assist in generating the result figures (Figs. 4–6), and to organize the raw results released in the reproducibility repository. The authors wrote the text of this paper. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

Reproducibility Statement

The full compiled results from every evaluation, together with the evaluation harnesses for both the constrained framework and the tool-calling baselines, are available at: https://anonymous.4open.science/r/hbtl-D921

References

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, et al. Do as I can, not as I say: Grounding language in robotic affordances. In *Conference on Robot Learning (CoRL)*, 2022. arXiv:2204.01691.

Meysam Alizadeh, Zeynab Samei, Daria Stetsenko, and Fabrizio Gilardi. Simple prompt injection attacks can leak personal data observed by LLM agents during task execution. *arXiv preprint arXiv:2506.01055*, 2025.

Anthropic. Claude 3.7 Sonnet and Claude Code. Anthropic News, February 2025, https://www.anthropic.com/news/claude-3-7-sonnet, 2025a. Accessed August 2026.

Anthropic. Effective context engineering for AI agents. Anthropic Engineering Blog, https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents, 2025b. Accessed August 2026.

R. Bhagwatkar, K. Kasa, A. Puri, G. Huang, Irina Rish, Graham W. Taylor, Krishnamurthy Dvijotham, and Alexandre Lacoste. Indirect prompt injections: Are firewalls all you need, or stronger benchmarks? *arXiv preprint arXiv:2510.05244*, 2025.

J. Bradshaw. A year late, Claude finally beats Pokémon. LessWrong, May 2026, https://www.lesswrong.com/posts/sehJYg5Yny9fvpbpt/a-year-late-claude-finally-beats-pokemon, 2026. Accessed August 2026.

Sizhe Chen, Julien Piet, Chawin Sitawarin, and David Wagner. StruQ: Defending against prompt injection with structured queries. In *34th USENIX Security Symposium (USENIX Security 25)*, pp. 2383–2400, 2025a. arXiv:2402.06363.

Sizhe Chen, Arman Zharmagambetov, Saeed Mahloujifar, Kamalika Chaudhuri, David Wagner, and Chuan Guo. SecAlign: Defending against prompt injection with preference optimization. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 2833–2847, 2025b. arXiv:2410.05451.

Kwang-Ting Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th ACM/IEEE Design Automation Conference (DAC)*, pp. 86–91, 1993.

Aidan Curtis, Tom Silver, Joshua B. Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Discovering state and action abstractions for generalized task and motion planning. *arXiv preprint arXiv:2109.11082*, 2021.

Edoardo Debenedetti, Jie Zhang, Mislav Balunović, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. In *NeurIPS Datasets and Benchmarks Track*, 2024. arXiv:2406.13352.

Edoardo Debenedetti, Ilia Shumailov, Tianqi Fan, Jamie Hayes, Nicholas Carlini, Daniel Fabian, Christoph Kern, Chongyang Shi, Andreas Terzis, and Florian Tramèr. Defeating prompt injections by design. *arXiv preprint arXiv:2503.18813v2*, 2025.

M. Dziemian, M. Lin, X. Fu, M. Nowak, N. Winter, E. Jones, A. Zou, L. Ahmad, K. Chaudhuri, S. Chennabasappa, X. Davies, L. Deason, B. L. Edelman, T. Emek, I. Evtimov, J. Gust, M. Hamin, K. He, K. Krawiecka, R. Patana, N. Perry, T. Peterson, X. Qi, J. Rando, Zifan Wang, Zihan Wang, S. Whitman, E. Winsor, A. Zharmagambetov, M. Fredrikson, and Z. Kolter. How vulnerable are AI agents to indirect prompt injections? Insights from a large-scale public competition. *arXiv preprint arXiv:2603.15714*, 2026. Evaluation kit: https://github.com/GraySwanAI/ipi_arena_os.

M. Ferrag, N. Tihanyi, D. Hamouda, L. A. Maglaras, and M. Debbah. From prompt injections to protocol exploits: Threats in LLM-powered AI agents workflows. *ICT Express*, 12:353–383, 2025.

GAIA Benchmark. GAIA dataset card, 2023 validation split. https://huggingface.co/datasets/gaia-benchmark/GAIA, 2023. Accessed September 2026.

Z. Garby, Andrew D. Gordon, and David Sands. The LLMbda calculus: AI agents, conversations, and information flow. *arXiv preprint arXiv:2602.20064*, 2026.

David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

D. K. Hofer, Edoardo Debenedetti, and Florian Tramèr. Assessing automated prompt injection attacks in agentic environments. *arXiv preprint arXiv:2606.10525*, 2026.

Sihao Hu, Tiansheng Huang, and Ling Liu. PokéLLMon: A human-parity agent for Pokémon battles with large language models. *arXiv preprint arXiv:2402.01118*, 2024.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning (ICML)*, 2022. arXiv:2201.07207.

Seth Karten. Continual harness: Reset-free agentic harness refinement through online in-context learning. v5.0.0, commit 62bf6f6, https://github.com/sethkarten/continual-harness, 2026. Accessed August 2026.

Seth Karten, Andy Luu Nguyen, and Chi Jin. PokéChamp: An expert-level minimax language agent. In *International Conference on Machine Learning (ICML)*, 2025. arXiv:2503.04094.

Seth Karten, J. Zhang, T. Upaa, Jr., R. Feng, W. Li, C. Shi, Chi Jin, and K. Vodrahalli. Continual harness: Online adaptation for self-improving foundation agents. *arXiv preprint arXiv:2605.09998*, 2026a.

Seth Karten et al. The PokéAgent challenge: Competitive and long-context learning at scale. *arXiv preprint arXiv:2603.15563*, 2026b.

LangChain. LangGraph documentation. https://langchain-ai.github.io/langgraph/, 2026. Accessed August 2026.

J. Lee, W. Song, J. Han, H. Pyun, and Y. Jo. Don't adapt small language models for tools; adapt tool schemas to the models. In *Proceedings of ACL 2026 (Volume 1: Long Papers)*, pp. 20695–20719, 2026. arXiv:2510.07248.

F. Li, A. Zhang, and C. Lv. Constraint tax in open-weight LLMs: An empirical study of tool calling suppression under structured output constraints. *arXiv preprint arXiv:2606.25605*, 2026a.

L. Li, Y. Dong, G. Wang, Z. Xu, A. Jiang, and T. Chen. XGrammar-2: Dynamic and efficient structured generation engine for agentic LLMs. In *ACM CAIS '26*, 2026b. arXiv:2601.04426.

X. Li, G. Tian, and Y. Cui. Semantically guided task planning: Supervised vision-language-action model by large language models. *IEEE Transactions on Circuits and Systems for Video Technology*, 36:7411–7425, 2026c.

Kaixin Ma, Hongming Zhang, Hongwei Wang, Xiaoman Pan, and Dong Yu. LASER: LLM agent with state-space exploration for web navigation. *arXiv preprint arXiv:2309.08172*, 2023.

X. Ma, T. Li, C. Xiao, Z. Yu, N. Zhang, and Y. Vorobeychik. AutoDojo: Adaptive black-box attacks reveal the limits of IPI defenses and task-specification effects in LLM agents. *arXiv preprint arXiv:2606.15057*, 2026.

Meta Superintelligence Labs. Muse Glimmer 30B model card and ATEM tool-call format, 2026. August 2026. See also the vLLM muse_glimmer tool-call and reasoning parsers.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: A benchmark for general AI assistants. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2311.12983.

Edward F. Moore. Gedanken-experiments on sequential machines. In *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pp. 129–153. Princeton University Press, 1956.

Eduardo Mosqueira-Rey, Elena Hernández-Pereira, David Alonso-Ríos, José Bobes-Bascarán, and Ángel Fernández-Leal. Human-in-the-loop machine learning: A state of the art. *Artificial Intelligence Review*, 56(4):3005–3054, 2023.

P. Narisetty, S. Kore, U. K. R. Kattamanchi, and J. Kumarapu. Adaptive evaluation of out-of-band defenses against prompt injection in LLM agents. *arXiv preprint arXiv:2606.26479*, 2026.

Milad Nasr, Nicholas Carlini, Chawin Sitawarin, Sander V. Schulhoff, Jamie Hayes, Michael Ilie, Juliette Pluto, Shuang Song, Harsh Chaudhari, Ilia Shumailov, Abhradeep Thakurta, Kai Yuanqing Xiao, Andreas Terzis, and Florian Tramèr. The attacker moves second: Stronger adaptive attacks bypass defenses against LLM jailbreaks and prompt injections. In *35th USENIX Security Symposium (USENIX Security 26)*, 2026. arXiv:2510.09023.

Michael O. Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

B. Ramakrishnan and A. Balaji. Securing AI agents against prompt injection attacks. *arXiv preprint arXiv:2511.15759*, 2025.

J. Ray. The constraint tax: Measuring validity-correctness tradeoffs in structured outputs for small language models. *arXiv preprint arXiv:2605.26128*, 2026.

Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

Richard S. Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1–2):181–211, 1999.

S. Suwansathit, Y. Zhang, and G. Gu. A systematic taxonomy of security vulnerabilities in the OpenClaw AI agent framework. *arXiv preprint arXiv:2603.27517v1*, 2026.

Zhi Rui Tam, Cheng-Kuang Wu, Yi-Lin Tsai, Chieh-Yen Lin, Hung-yi Lee, and Yun-Nung Chen. Let me speak freely? A study on the impact of format restrictions on performance of large language models. In *Proceedings of EMNLP 2024: Industry Track*, pp. 1218–1236, 2024. arXiv:2408.02442.

J. Tang, Y. Yan, Z. Ye, Z. Song, Z. Zheng, and Y. Jin. Think small, plan smart: Minimalist symbolic abstraction and heuristic subspace search for LLM-guided task planning. *arXiv preprint arXiv:2501.15214*, 2025.

TechCrunch. Google's Gemini has beaten Pokémon Blue (with a little help). May 2025, https://techcrunch.com/2025/05/03/googles-gemini-has-beaten-pokemon-blue-with-a-little-help, 2025. Accessed August 2026.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. *arXiv preprint arXiv:2402.01030*, 2024.

Brandon T. Willard and Rémi Louf. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702*, 2023.

Simon Willison. The dual LLM pattern for building AI assistants that can resist prompt injection. https://simonwillison.net/2023/Apr/25/dual-llm-pattern/, 2023. Accessed August 2026.

Yiran Wu, Tianwei Yue, Shaokun Zhang, Chi Wang, and Qingyun Wu. StateFlow: Enhancing LLM task-solving through state-driven workflows. In *Conference on Language Modeling (COLM)*, 2024. arXiv:2403.11322.

Ke Yang, Yao Liu, Sapana Chaudhary, Rasool Fakoor, Pratik Chaudhari, George Karypis, and Huzefa Rangwala. AgentOccam: A simple yet strong baseline for LLM-based web agents. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2410.13825.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. arXiv:2210.03629.

Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. InjecAgent: Benchmarking indirect prompt injections in tool-integrated large language model agents. In *Findings of the Association for Computational Linguistics: ACL*, 2024. arXiv:2403.02691.

Qiusi Zhan, Richard Fang, Henil Shalin Panchal, and Daniel Kang. Adaptive attacks break defenses against indirect prompt injection attacks on LLM agents. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 7101–7117, 2025. arXiv:2503.00061.

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

A Extended Related Work

A.1 Additional Prompt Injection Context and Defenses

Human-in-the-loop approval, the most common defense in practice, assumes the person at the keyboard can recognize a malicious shell command or judge what the agent is about to touch (Mosqueira-Rey et al., 2023). Even short, unsophisticated injections can leak personal data that an agent happens to observe while carrying out an ordinary task (Alizadeh et al., 2025), and the attack surface now spans the full agent workflow rather than the prompt alone (Ferrag et al., 2025; Ramakrishnan & Balaji, 2025). Firewall-style approaches report high security with high utility, though that work also argues current benchmarks understate attack difficulty (Bhagwatkar et al., 2025). Model-level defenses instead fine-tune the LLM to ignore instructions that appear in the data portion of its input (Chen et al., 2025a;b), and defenses that appear robust under fixed evaluation have been bypassed once attacks are adaptively optimized (Ma et al., 2026; Zhan et al., 2025). We note that the original CaMeL paper argues for security by construction rather than providing a formal proof; a later calculus-based formalization does (Garby et al., 2026). We distinguish throughout between design-based guarantees, benchmark performance, and formal proofs. CaMeL ensures untrusted data cannot alter the original task or execution path by enforcing capability checks before calling any tool (Debenedetti et al., 2025; Willison, 2023). AgentDojo measures system security and model reliability (Debenedetti et al., 2024), and full datasets from red-teaming competitions such as Gray Swan Arena remain open-source and available for testing (Dziemian et al., 2026).

A.2 Action Abstraction in LLM Planners

Similar methods appear when LLMs are used as planners, which become more reliable operating over structured abstractions such as skills, plans, or APIs rather than raw commands. SayCan restricts a language model to a fixed library of learned robot skills and scores each one for feasibility before execution, so the plan is assembled only from actions the robot can actually perform (Ahn et al., 2022). Later work shows the same benefit from deliberately minimal symbolic abstractions (Tang et al., 2025) and from semantically guided action selection (Li et al., 2026c). For instance, AgentOccam introduces a simplified, minimal web observation and action space that improves zero-shot reliability on web navigation tasks purely by aligning what the agent sees and can do with what the model was trained on (Yang et al., 2025). LASER defines the web states that can be occupied and a set of permissible actions the agent may choose from; this rules out invalid actions by construction rather than correcting or retrying after the fact (Ma et al., 2023). Grounding a model's language in executable actions is a standard problem this method addresses. Huang et al. (2022) demonstrate in a simulated household, VirtualHome, that plans created by LLMs often include steps the environment cannot perform; they improved executability by semantically translating each step into the closest admissible action.

A.3 State Machine Foundations and Prior FSM Agent Frameworks

Moore established the deterministic variation of a state machine, where the next state depends solely on the current state and input, making the output entirely determined by the current state (Moore, 1956). Rabin and Scott further define the decision space for finite automation, formally distinguishing deterministic systems from non-deterministic ones (Rabin & Scott, 1959).

Our framework uses a Hierarchical Extended Finite State Machine (HEFSM), combining Harel's nesting (Harel, 1987) with the variable-carrying transitions of an EFSM (Cheng & Krishnakumar, 1993). The agent starts in a main menu that branches into submenus that decide the action state. The LLM acts as the decision engine, dictating the next state by selecting choices presented in the menus, and Python logic executes the action in the final state. The model cannot generate new states, alter branching, or supply menu choices the active state does not offer.

StateFlow runs each step of an agent's process into a separate state, reporting higher success rates and lower costs than ReAct (Yao et al., 2023) on Intercode SQL and ALFWorld (Wu et al., 2024). However, while the agent's state determines when a tool is called, the state itself is a prompt-and-role configuration rather than a hard restriction on which tools remain callable, so a state does not strictly bound the tool surface the way our activities do. LangGraph is a widely adopted framework for agent design and orchestration (LangChain, 2026); it is best understood as a modular authoring

layer, where developers assemble states, transitions, and checkpoints into a graph, while the agent running inside a given node still calls tools directly.

A.4 Structured Output Enforcement Mechanisms

Structured output is typically used to allow a model's output to be directly used by software without the use of a parsing step that can fail. Two enforcement options are available for general use. Constrained decoding masks any token that would violate the target grammar, which Willard & Louf (2023) implement by indexing the LLM's vocabulary against an FSM, allowing only schema-valid completions to remain available for generation; more recent grammar engines pursue the same guarantee at lower overhead (Li et al., 2026b). The alternative method involves requesting the format in the instruction prompt and then validating the result after generation, rejecting or retrying on parse failure (Tam et al., 2024).

A.5 Pokémon as a Capability Benchmark

Models tested to complete the game originated in public demonstrations rather than research evaluations, starting with Anthropic launching the Claude Plays Pokémon livestream with Claude 3.7 Sonnet in February 2025 (Anthropic, 2025a), while Gemini 2.5 Pro completed Pokémon Blue in May 2025 using an independently built harness (TechCrunch, 2025). Claude did not complete the game until May 2026 with Opus 4.7 (Bradshaw, 2026). These outcomes are not directly comparable, as each agent was given different tools and information about game completion (TechCrunch, 2025). Researchers have also used narrower Pokémon environments to isolate battle reasoning (Hu et al., 2024; Karten et al., 2025). Two properties influence a model's outcome: the bottleneck from sprite interpretation rather than reasoning, and the fact that the game is documented well enough online that runs can surface memorized route knowledge.

Running these harnesses with different models has become more accessible. The PokéAgent Challenge (Karten et al., 2026b) provides a standard RPG speedrunning track, and the Continual Harness (Karten et al., 2026a) extends it with online in-context adaptation, letting a harness refine itself across runs for self-improving foundation agents.

B Full Requirements

- **JSON schemas as state options.** The model must return a valid structured output conforming to a supplied JSON schema, produced either by constrained decoding or by instruction-and-validation (Willard & Louf, 2023; Tam et al., 2024; Li et al., 2026b). Menu choices should be structured as enumerations, using the standard alphabet letters presented in the menu, so the model can only return a letter the active state offers. Freeform parameters, such as the message body in a social activity, are typed as validated strings. Task completion uses a single boolean flag, `done: true`, which lets the backend terminate or return to the main menu without interpreting text. Set `additionalProperties` to `false` in the schema to bind the output to the specified fields only.
- **Reasoning vs. structured outputs.** Many model platforms, APIs, and local systems do not support emitting chain-of-thought reasoning and strictly schema-guided JSON at the same time; where both constraints are applied together, the grammar mask can suppress the model's normal behavior outright (Li et al., 2026a). Reasoning must therefore often be turned off or handled in a custom parsing step. To preserve observability, we implement an optional thinking field inside the JSON schema itself. In some base models, imposing a constrained thinking field can artificially replicate reasoning training distributions and alter behavior, and format restrictions of this kind have been measured to reduce reasoning performance in their own right (Tam et al., 2024).
- **Model output distillation.** A key reliability factor is whether the model was natively trained on JSON structures or a proprietary tool-calling language. Models fine-tuned or distilled specifically for custom tool formats, such as XML-style tags or ATEM serialization, tend to perform less reliably when forced into strict JSON-guided decoding, as this pushes them out of their native training distribution (Ray, 2026; Lee et al., 2026). This affects general task reliability but not necessarily spatial or gaming capability, as Glimmer's performance in our Pokémon benchmark (Sec. 6) shows.

- **Custom system instructions.** Carefully authored system prompts that dictate exactly how to populate the structured JSON yield a more robust and predictable framework, helping prevent the model from misunderstanding its menu constraints.
- **Context handling.** Context windows must be manually reconstructed and maintained (Anthropic, 2025b). Strip raw JSON outputs from previous turns to plain-text logs before feeding them back into the context window. Feeding raw JSON outputs repeatedly back into the context window can induce the model to generate infinitely nested JSON structures during multi-turn interactions, drastically lowering overall reliability.

C Additional Discussion

C.1 Deployment Scale of the Menu

A deployment of this system can match a utility surface comparable to a multipurpose agent such as OpenClaw by authoring more menu options. The constrained space focuses on what each activity can execute and access, never limiting how large the overall system can become. The method is not specific to the example menu, and a deployment substitutes its own activities.

C.2 Design Effort as a Rigging Analogy

We used CGI character rigging as a reference point while designing this system. A multipurpose agent has a wide range of access, and hand-authoring every one of those actions is a large amount of upfront work, but it is not far off from what a rigger does when setting the attributes for every part of a character's body. Character riggers manually constrain every single joint in a character's body. They program the reasonable degree of rotation allowed, how far eyelids should blink or squash, and dictate precise blendshapes for every mouth position during speech. One or two technical artists typically complete this task per character. The detail established during this initial rigging phase directly benefits the remainder of the production pipeline, ensuring nothing structurally breaks during the tedious animation phase.

C.3 State Creep, Modularity, and Coding Agents

The distance to an activity remains short, such as navigating to send a message in a social channel. Unlike previous legacy state machine systems that had to define every conceivable message string rigidly, the LLM writes the message itself once it enters that state, and only the channels it can write to are authored in advance. This dynamically creates endless viable states purely from the text output, while keeping the structural framework constrained.

While crafting a system like this requires customization for individual deployment needs, publishing modular menu options and pre-packaged actions can alleviate this burden, allowing the menu to be expanded or kept minimal depending on the user's risk tolerance.

One domain where the constrained system may struggle is assisting in pure code generation or software engineering tasks. Autonomous coding agents excel precisely because of their ability to directly modify local files, install external packages, and run unrestricted bash scripts, actions that intrinsically violate the rigorous pathways and data silos enforced by our constrained architecture because our system focuses on multipurpose agents that achieve general tasks; a potential hybrid solution could involve allowing the constrained agent to interact with a heavily sandboxed coding agent via an isolated API, granting further capabilities without compromising the local host environment.

D Extended Results

D.1 Canary Breakdown (Sec. 4.2)

In the exfiltration attacks against OpenClaw, 28 successes spanned 11 canary types. The individual canaries exfiltrated most often were a private inbox (6), a passport scan (5), a bank account record

(4), two-factor recovery codes (2), and AWS credentials (2), with the Gray Swan corpus contributing model weights, codebase archives, and shared API keys.

D.2 Pytector Pass (Sec. 4.2)

We repeated the identical 328 attacks against both frameworks with Pytector, a middleware intended to detect prompt injection attacks, used here as a prompt warning to the model rather than as a blocking filter. The layer made no meaningful impact: the constrained agent replicated its previous results exactly, and OpenClaw produced the same 28 exfiltrations plus one additional successful destructive action. At the same time, its benign task completion fell from 86.6% to 83.2%. This is a narrow observation about one tool in one placement, and not a claim about detection middleware as a class.

D.3 Step-Level Breakdown of the Failed Tool-Calling Run (Sec. 6.2)

Across 1,000 steps, the tool-calling Glimmer agent issued 914 `press_buttons` calls and 85 `process_memory` calls, and the button groups averaged 1.27 presses per call. It visited 41 distinct map positions in the entire run and never left Red's house, spending 946 of 1,000 steps on the ground floor. Only the `GAME_RUNNING` milestone was ever recorded.

D.4 Reasoning-Action Agreement (Sec. 5.2)

Reasoning-action agreement is measurable in 100% of constrained sessions, as we include a thinking field regardless of the reasoning configuration. Still, in only 55.3% of OpenClaw sessions, because OpenClaw does not always expose reasoning: Muse Spark 1.2 produced no inspectable reasoning in any of its 66 OpenClaw sessions, and GPT-5.6 Luna in only 15. Where measurable, agreement was 97.3% for our framework against 86.3% for OpenClaw.