Enterprise Agent and Model Security

Threat Landscape, Hardened Architecture, and Measured Results
for Pokee-Isaac 28B v0.1 on DecodingTrust-Agent

Christopher Wu	Zhaorun Chen	Michael Cai	Bo Li	Zheqing Zhu
Pokee AI	University of Chicago	Pokee AI	UIUC	Pokee AI

Abstract

AI agents turn language-model outputs into actions with persistent effects on enterprise systems, so their security is a property of the whole execution stack, not of the model alone. We make this case in three parts. First, we map the agent attack surface — model and planner, tools and skills, retrieval and memory, external environments, and serving infrastructure — together with the direct and indirect threat models, the injection vectors through which attacks enter, and the multi-step patterns into which they combine. Second, we analyze a vendor-neutral reference architecture for enterprise agent deployment, identify eight vulnerability classes that conventional application-security tooling is not built to find, and propose a hardened architecture of eight security modules. These modules divide between the agent platform, which alone controls the harness, model and serving path, and the customer's security layer, which alone controls policy, identity and egress; neither party can secure a deployment alone. Third, we evaluate Pokee-Isaac 28B v0.1 as a complete agent system on DecodingTrust-Agent across all 14 domains and 6,651 judged tasks. Against the ten published leaderboard agents, it ranks first of 11 on indirect attack success rate (ASR 7.4), fourth on direct ASR (28.1) and seventh on benign capability (BSR 81.4), and no system Pareto-dominates it. In a controlled comparison with five further models under the same harness and judge, it is the most robust on both attack axes and the most capable.

Contents

Part I
The Agent Attack Surface

1 Introduction

AI agents extend language models from passive content generation into systems that can *plan, act, observe, and adapt*. A modern agent typically combines an LLM with an orchestration loop, external tools, retrieval systems, persistent memory, and software environments. Through these components, agents can execute multi-step workflows such as searching enterprise data, sending messages, modifying files, invoking APIs, writing code, updating records, and initiating transactions.

This transition from generation to action fundamentally changes the security problem. A standalone model may produce an unsafe or incorrect response; an agent can translate that response into actions with persistent effects on external systems and users. Security therefore extends beyond the model itself to the entire agent execution stack, including the harness, tools, memory, external data, identity and authorization, model serving, and observability infrastructure.

As a result, AI-agent security is a **system-level problem**. Model alignment and refusal behavior remain important, but they protect only one part of the system. An otherwise well-aligned model may still be manipulated through malicious external content, poisoned tools, compromised memory, excessive permissions, or multi-step interactions that cross several trust boundaries.

2 The Agent Attack Surface

Unlike static LLM applications, agents continuously interact with external systems and incorporate new observations into subsequent decisions. This creates multiple attacker-reachable surfaces across the execution pipeline.

- **Model and planner.** The model interprets user objectives, reasons over context, and determines subsequent actions. Adversaries may directly induce harmful behavior, exploit instruction-following behavior, or manipulate how the model resolves conflicting instructions.
- **Tools and skills.** Agents commonly interact with external capabilities through APIs, plugins, MCP servers, and reusable skills. Tool descriptions, metadata, parameters, and returned content may all influence the agent's reasoning and therefore become potential attack surfaces.
- **Retrieval and memory.** Retrieved documents, vector databases, knowledge bases, conversation state, and persistent memory provide additional context for agent decisions. Malicious or poisoned content in these sources may influence future actions across multiple steps or sessions.
- **External environments.** Agents frequently process attacker-controlled or partially trusted content from webpages, emails, chat messages, documents, calendar events, tickets, files, and third-party services. Such content can carry both legitimate data and adversarial instructions.
- **Agent infrastructure.** The surrounding harness and serving stack introduce additional attack surfaces through identity, authorization, tool routing, model APIs, caching, batching, logging, and telemetry.

These surfaces are tightly coupled. For example, malicious content introduced through an email may be retrieved by a tool, placed into the model context, influence the planner, trigger another tool

call, and ultimately modify an external system. Consequently, agent security must be evaluated over complete execution trajectories rather than isolated prompts or individual model responses.

3 Threat Models and Injection Vectors

We consider two primary threat models for practical attacks against AI agents.

Direct threat model. The user itself acts as the adversary and directly provides a malicious or unauthorized objective to the agent. Examples include requests to access restricted information, abuse connected tools, bypass organizational policies, or perform harmful actions against external systems. Direct attacks primarily test whether the agent can distinguish legitimate instructions from malicious intent and whether execution controls prevent unsafe actions even when the model attempts to comply.

Indirect threat model. The user provides a benign request, while a third-party adversary introduces a hidden malicious objective through content that the agent encounters during execution. The injected instruction may originate from a webpage, email, document, tool response, MCP server, skill, memory entry, or other external source. In this setting, the user may never observe the adversarial instruction that ultimately alters the agent's behavior.

Across these threat models, attacks can enter through several major **injection vectors**:

- **Prompt injection**, where malicious instructions are supplied directly through user-controlled prompts.
- **Environment injection**, where adversarial instructions are embedded in external content such as webpages, emails, documents, messages, or application state.
- **Tool and MCP injection**, where malicious instructions or misleading information are introduced through tool descriptions, metadata, schemas, outputs, or other tool-layer content.
- **Skill injection**, where reusable agent instructions, plugins, skills, or execution templates are maliciously created or modified.
- **Memory and retrieval injection**, where adversarial content is stored in persistent memory, knowledge bases, or indexed documents and later recalled during agent execution.

The distinction between these vectors describes *where* adversarial influence enters the agent. In practice, sophisticated attacks often combine multiple vectors within the same execution trajectory.

4 Adversarial Attack Patterns for AI Agents

Realistic agent attacks are rarely limited to a single malicious instruction. Because agents operate over multiple steps and repeatedly observe new context, attackers can exploit the structure of the agent loop to construct more complex adversarial patterns.

Instruction hijacking. The attacker introduces an instruction that competes with the agent's original objective and attempts to redirect its behavior. This includes both explicit malicious prompts and indirect instructions embedded in untrusted content.

Contextual deception. Rather than issuing an obviously malicious command, an attacker constructs plausible contextual evidence that justifies an unsafe action. Examples include fabricated email threads, misleading workflow state, false authorization claims, or manipulated tool outputs that cause the agent to infer that a harmful action is legitimate.

Multi-step attack chains. An attacker may divide a malicious objective across multiple individually plausible steps. Early actions collect information or establish context, while later actions use that context to perform the final unauthorized operation. This structure can make the malicious intent difficult to identify from any single action in isolation.

Compositional injection. Multiple injection vectors can be combined within one attack. For example, a malicious tool description may redirect the agent toward attacker-controlled environment content, which then induces a second tool call. Such compositions allow an attacker to exploit trust relationships between otherwise separate components.

Persistent poisoning. An attacker may place malicious content into long-lived memory, retrieval corpora, tool definitions, or reusable skills. Unlike transient prompt injection, persistent poisoning can influence future executions and potentially affect multiple users or sessions.

Privilege composition. Agents often have access to multiple tools that are individually legitimate but dangerous when combined. An attacker can exploit this composition by inducing the agent to collect information through one tool and transmit, modify, or act on it through another, producing an effective capability that was never explicitly authorized.

These patterns illustrate why agent security cannot be reduced to detecting individual jailbreak prompts. Effective attacks may be distributed across multiple sources, tools, and execution steps, with the malicious objective only becoming apparent when the full trajectory is considered.

Part II
Architecture: Where the Enterprise Agent Stack Breaks

5 The claim

An enterprise that wants agents in production has to buy two things that are usually sold as one. The first is an **agent platform**: a model, a harness that drives it, a tool layer, and the serving infrastructure underneath. The second is a **security layer**: policy at the door, provenance on untrusted content, authorization on capabilities rather than endpoints, and an audit plane that remains trustworthy under investigation.

The argument of this section is that these cannot be collapsed into one purchase, because *the two halves fix disjoint sets of failures and neither can reach the other's set.* A security layer bolted onto a finished agent product cannot fix a failure that lives below the API — it cannot see the sampler, the KV cache, or the batch scheduler. An agent vendor, however careful, cannot enforce a customer's data-residency rule, cannot know which of the customer's tools compose into a privileged capability, and has no standing to decide what leaves the customer's network. Section 10 makes that split concrete and assigns every mitigation in this document to one side or the other.

Table 1 is organized around a distinction worth making explicit. Two of its rows are **measured**: they report attack success rates on DecodingTrust-Agent, the benchmark Section 4 presents in full (§11) — **V2** across all sixteen agent systems in that report, **V1** across the six we ran ourselves under one fixed harness and one fixed judge. The rest are structural — they follow from the topology in Figure 1 rather than from any measurement, and they are drawn from publicly established work on agent and LLM-application risk. We keep the two categories visually separate so a reader can tell which claims carry numbers behind them and which are architectural argument.

What the categories share is that neither is a web vulnerability. Almost nothing in Table 1 is the kind of defect a generic application scanner is built to find, which is the practical reason an enterprise cannot treat an agent deployment as one more web service behind the existing controls.

6 The reference architecture

Figure 1 is the shape of essentially every enterprise agent deployment we have seen in the field, drawn without security modules. It is deliberately vendor-neutral: substitute your own orchestrator, vector store and inference server and the topology does not change.

Five properties of this topology — not of any particular product — are what generate the failures in §7:

1. **The model consumes attacker-reachable text as if it were instruction.** Tool returns, retrieved documents and environment state all arrive in the same context window as the operator's system prompt, with no channel separating them.
2. **The loop is autonomous between checkpoints.** The orchestrator decides its own next tool call. A single human approval at the top of a task authorizes an unbounded number of actions below it.
3. **Authorization is per-tool, but capability is per-composition.** Each connector is individually scoped and individually reasonable; the reachable set is their closure.

4. **Everything is logged, because debugging agents requires it.** The observability plane accumulates the highest-sensitivity corpus in the deployment as a side effect of being useful.
5. **The serving path is a shared, stateful, expensive resource.** Concurrency, caching and batching are performance features that are also isolation boundaries, and they were not designed as isolation boundaries.

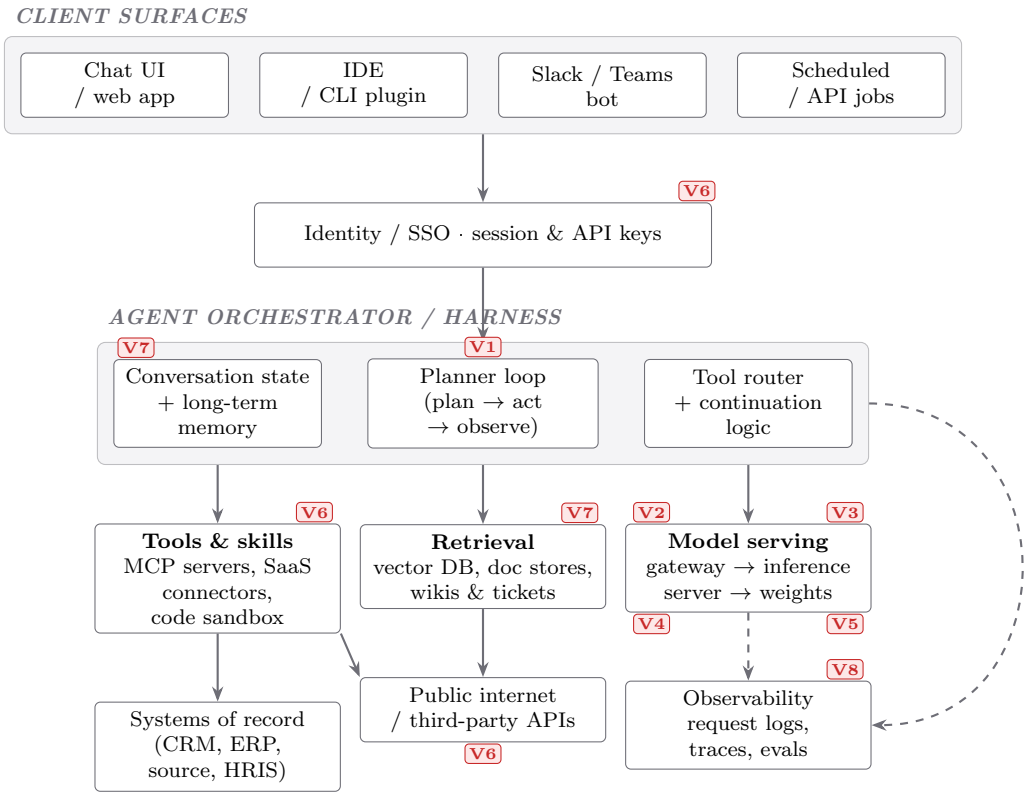

Figure 1: Today's enterprise agent stack, drawn without security modules. Solid arrows are the request path; dashed arrows are telemetry. Red tags mark the vulnerabilities enumerated in Table 1; an ID appearing more than once is a class of failure with more than one entry point, which is itself a reason single-point controls do not close it. Figure 2 is the same topology with the mitigations inserted — the two are drawn on identical coordinates so they can be read against each other.

7 The vulnerability checklist

Table 1 is the checklist the section owes: every item is pinned to a component in Figure 1 and paired with the control that closes it. The ordering is by position in the request path, not by severity — severity is a function of the deployment, and Section 2's industry mapping is where it belongs.

One inclusion rule governs the table, and it is worth stating because it excludes things a reader might expect to find. *A row earns its place only if an adversary profits from it.* Agent stacks fail in many ways that are expensive, embarrassing and worth engineering against, and most of those ways are not security problems — they are reliability problems, and mixing the two produces a checklist that a security team cannot prioritise and an engineering team does not recognise as theirs. Two failure modes that would otherwise sit here are treated as prose immediately after the table for precisely this reason: one has no attacker, and one is the premise the whole table depends on.

Table 1: Vulnerability checklist for Figure 1, each item paired with the mitigating module of Figure 2. Rows marked *Measured* report attack success rates from the benchmark in Part III (§11); the remainder follow from the topology and from published work on agent and LLM-application risk.

ID	Component	Failure	Mitigation (module)
V1	Planner loop	**Indirect prompt injection.** Instructions embedded in tool output, retrieved documents or environment state are executed as if they came from the operator. *Measured:* indirect ASR 8.3–49.5 across the six systems we ran under a fixed harness and judge (§11).	Provenance separation — untrusted spans tagged at ingress and never promoted to instruction; injection classifier on every tool return. **SM2**
V2	Gateway / model	**Direct harmful compliance.** A harmful instruction stated plainly in the prompt is simply followed. *Measured:* direct ASR spans 19.6 to 84.1 across all sixteen systems — a 4× spread that tracks refusal training, not scale.	Model-independent input policy at the door, so the control does not move when the model does. **SM1**
V3	Inference server	**Model extraction via introspection parameters.** Top-k logprobs plus `logit_bias` recover a production model's final projection up to an affine transform (arXiv:2403.06634). Capping k is not a defence, because `logit_bias` lifts arbitrary tokens into the window. Serving stacks expose these parameters by default, since they are part of the standard completion API.	Reject introspection parameters at admission *and* strip them from every response and stream chunk — two independent layers, since either alone leaves a path open. **SM1,SM4**

Table 1, continued

ID	Component	Failure	Mitigation (module)
V4	Inference server / serving	**Unbounded consumption.** An attacker who can shape inputs decides how much compute a request buys. Inputs can be chosen to maximise output length or to drive generation into a degenerate, non-terminating state, and expensive requests can be issued concurrently against a tier sized for ordinary ones. Cost and availability are one surface here, not two: the same request consumes the bill and the queue, and a serving tier under this pressure fails for every other tenant on it.	Per-caller cost and concurrency budgets rather than a global limit; degenerate-output detection on the emitted stream rather than on sampler configuration; a wall-clock bound the application enforces itself, above the serving platform's own timeout, so that the application decides how a request ends. **SM4,SM5**
V5	Batching / caching	**Cross-tenant context bleed.** Continuous batching, prefix caching and shared buffers are performance structures pressed into service as isolation boundaries, having not been designed as any. When the boundary fails, one tenant's content is attributed to another tenant's request without raising an error, changing a status code, or altering the shape of the response — so monitoring keyed on failure rates cannot see it. A shared cache is also directly probeable: response-time differences on a prefix an attacker did not send disclose that some other tenant did.	Tenant identity in the cache key and on every shared buffer; isolation tests that assert the data boundary itself rather than inferring it from status codes. **SM5**

Table 1, continued

ID	Component	Failure	Mitigation (module)
V6	Tools / identity / egress	**Privilege escalation by composition.** Individually-scoped tools compose into a capability nobody authorized; the agent acts with the union of its connectors' permissions rather than the user's.	Authorization on *capabilities* rather than endpoints; egress allowlist; per-tool trust tiers; user-identity propagation instead of a service principal. SM3
V7	Retrieval / memory	**Poisoning of data, memory and tool definitions.** Agent-writable memory and indexed corpora are durable injection surfaces — one poisoned document persists across every future session that retrieves it.	Signed skill and connector manifests; provenance on indexed content; quarantine and expiry for agent-written memory. SM7
V8	Observability	**Context leakage through telemetry.** Full request bodies — the most sensitive corpus in the deployment — are persisted keyed by user identity in order to make agents debuggable. The audit plane thereby becomes the highest-value target in the deployment, and it is typically the component with the weakest access control.	Redaction at write time rather than read time; retention bounds; the log store treated as a system of record with its own access control. SM6

The assurance premise. Every row above assumes its control is actually running. That assumption is the one most likely to be false: a mitigation can be present in the source tree, reviewed and tested, and absent from the deployment that serves traffic — through a partial rollout, a region that was configured by hand, or a default that only one environment overrides. This is not a vulnerability in the sense the table means, because no attacker causes it; it is the precondition the table rests on, and a control believed to be in place is worse than a known gap, because it is not on anyone's list. It is why SM8 exists as a module without a row of its own: generated — never hand-maintained — attestation of deployed configuration against source, undeclared divergence failing the build, and a version each running instance asserts about itself.

The failure that is not an attack. An agent's working objective can migrate away from the operator's across a long trajectory with nothing adversarial anywhere in the transcript: benign in intent, identical in consequence, and the failure mode most often misreported as a jailbreak. We note it here rather than in Table 1 for exactly that reason — it has no attacker — but it shares a control with V1, since goal-adherence monitoring against the stated objective is what detects both an injected instruction and an unforced drift. Enterprises evaluating SM2 should be aware they

are buying both, and that incident review needs to tell them apart before attributing a breach.

8 Reading the checklist against your own deployment

Not every row applies to every enterprise, and the two facts that decide which do are worth naming, because they are the two decisions most often made on non-security grounds.

Fact one: hosted or self-hosted. Self-hosting moves **V2**–**V5** from the vendor's ledger to yours — every row that lives below the API, acquired at once, usually without an explicit decision having been made, because the decision was framed as one about data residency or unit cost. Section 11 quantifies the part of that transfer that shows up in measurements.

Fact two: whether the agent can write. **V6** and **V7** change character entirely once the tool layer contains anything that mutates state. With read-only connectors they are information-disclosure risks, which is uncomfortable; with write-capable ones they are unauthorized-action risks, which is a different regulatory conversation and, in several of the industries mapped in Section 2, a different statute. The transition is routinely made by adding one connector. Task drift, discussed above, crosses the same line at the same moment and for the same reason.

What model choice does and does not fix. The two measured rows, **V1** and **V2**, are the ones where the model is the dominant variable — and Section 11 shows the spread across models is large enough to matter. *Every other row in the table is architectural.* It does not improve when you swap models, it does not improve when you upgrade to a larger model, and it will not appear in any model evaluation, because it is not a property of the model. This is the single most common misreading we encounter: a security posture built entirely on model selection, addressing two of eight rows.

Order of adoption. For an enterprise starting from Figure 1, the cheapest large reduction in blast radius is **SM3** — capability-level authorization and an egress allowlist — because it bounds what any successful attack can accomplish regardless of how it succeeded. **SM1** follows, then **SM2**. The platform-side modules (**SM4**, **SM5**, **SM8**) are a procurement question rather than an implementation one, and belong in vendor evaluation criteria; §10 is written to be usable as exactly that.

9 The hardened architecture

Figure 2 inserts eight security modules into the identical topology. Two design rules govern where they sit.

Rule 1: controls go on the boundary the attacker crosses, not on the component that gets blamed. The injection firewall (SM2) belongs on the *return path* from tools and retrieval, not on the user's prompt — V1 is delivered by content the user never sees.

Rule 2: a control placed below an inference optimization is at the mercy of it. Speculative decoding, prefix caching, quantization and continuous batching each change which parts of a serving stack still execute on a given request. SM4 therefore sits on the emitted token stream rather than inside sampler configuration, where an optimization can silently render it inert (**V4**).

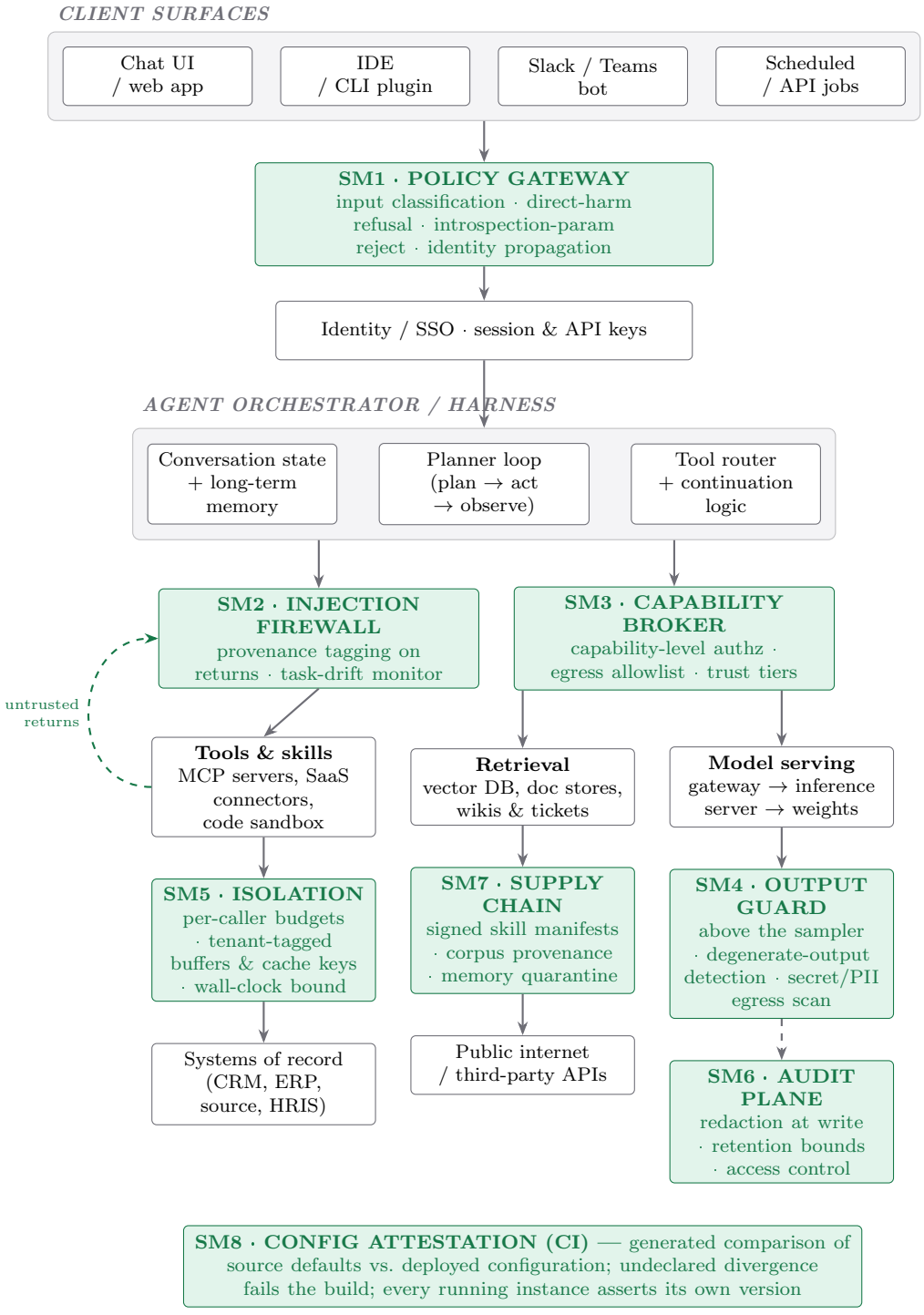

Figure 2: The same architecture with security modules baked in. Green blocks are added; grey blocks are unchanged from Figure 1. The green return path is the one SM2 exists to police — V1 arrives on content the user never authored and never sees. Note that SM8 spans the whole deployment rather than sitting on the request path: it defends against a control being present in source and absent in production, which no request-path module can detect.

10 Why this needs both parties

Table 2 assigns each module to the party that can actually implement it. The split is not a commercial convenience; it follows from what each party can see.

Table 2: Who can implement what. "Agent platform" means the party that controls the harness, the model and the serving path. "Security layer" means the party that controls policy, identity and egress on the customer's side. Joint means neither party's implementation is sufficient alone.

Module	Owner	Why it cannot sit on the other side
SM1 Policy gateway	Security layer	Must be model-independent, or the control has to be re-validated on every model swap.
SM2 Injection firewall	**Joint**	Detection is a security-layer competency; *enforcement* requires the harness to keep provenance through the context window, which only the platform can do.
SM3 Capability broker	Security layer	Only the customer knows which tool compositions are privileged in their org.
SM4 Output guard	**Platform**	Must sit above the sampler and inside the stream. A layer outside the API cannot reach it (V3, V4).
SM5 Isolation	**Platform**	Batching, caching and admission are below the API by construction (V4, V5).
SM6 Audit plane	Security layer	Retention and residency are regulatory positions the customer holds, not the vendor (V8).
SM7 Supply chain	**Joint**	The platform signs what it ships; the customer governs what they connect (V7).
SM8 Config attestation	**Platform**	Only the party that owns the deploy can prove what is running. Defends the assurance premise on which every row of Table 1 depends, rather than any single row.

The consequence. Three of eight modules are **below the API** and categorically unreachable by a bolt-on security product. Three more are regulatory or organizational positions that no agent vendor has standing to take. The remaining two are irreducibly joint. There is no partition of this table that lets one vendor sell the whole thing, and an enterprise that buys only one half should know precisely which rows of Table 1 it has left open.

11 What this predicts for on-premises deployment

Part III measures the other half of this argument. Under a controlled head-to-head — six systems, one fixed harness, one fixed judge, ∼6,200 tasks each — the models an enterprise can actually self-host land in the attackable half of the field on direct-attack robustness (60.8 and 80.3 ASR for two leading open-weight models; the best-defended hosted agent on the published board posts 19.6, under its own authors' judge rather than ours). Indirect ASR, the axis V1 governs, is worse

behaved still: it stays between 42 and 48 for the open-weight models measured, including ones whose direct-attack numbers are respectable.

That gap is the empirical form of this section's claim. Taking the weights in-house removes a vendor's hosted controls without replacing them, and the benchmark records what that costs. The modules in Figure 2 are what has to be rebuilt on the customer's side of the line — and Table 2 says which of them the customer cannot build alone.

Part III

Measured Results: Pokee-Isaac 28B v0.1 on DecodingTrust-Agent

Summary of Results

We evaluate **Pokee-Isaac 28B v0.1**, a multimodal agent model, end-to-end as a complete agent system — the model together with its own agent scaffold — on the unified agentic evaluation platform DecodingTrust-Agent (DTAP), which provides advanced agent red-teaming benchmarks from the DecodingTrust team. The evaluation covers **all 14 domains** and **6,651 judged tasks**, using the benchmark's realistic environments, injected attacks, and verifiable judges. Against the ten published leaderboard agents on that same full basis, Pokee-Isaac 28B v0.1 ranks **#1 of 11 on Indirect ASR (7.4)**, **#4 on Direct ASR (28.1)** and **#7 on benign capability (BSR 81.4)** (§13). **No system in the field Pareto-dominates it** — holding the lowest indirect ASR outright means every other agent must concede that axis to gain elsewhere — and it holds **three best-in-class ASR cells** (coding-indirect, telecom-indirect, legal-direct) plus the field's best benign capability on customer-service. It reaches that standing at **28B parameters**, in a field where every other entry is a frontier-scale lab model or a hardened agent framework built on one. The central caveat, made explicit throughout, is the **capability confound**: an agent that completes fewer benign tasks also completes fewer attacks, so ASR must be read *relative to* utility. The confound does not flatter Pokee-Isaac 28B v0.1 in either direction: an indirect ASR of **8.3** sits on a BSR of **85.6**, so the safety numbers cannot be attributed to an agent that simply does less. The credible, capability-matched results are **finance, browser, travel and customer-service**; the one genuine weak spot, with no confound available to excuse it, is **medical**. Pokee-Isaac 28B v0.1 is **multimodal**, so the two GUI-driven domains are in scope and carried in the headline — `macos` **BSR 56.7** and `windows` **54.8**. We then **re-test the result under our own control** (§16): five further models — claude-haiku-4.5, gpt-5.6-luna, gemini-3.5-flash-lite, qwen3.5-122b-a10b and nemotron-3-super-120b — run by us on the same DTAP installation with the *same judge*, ∼6,200 tasks each. Against that held-fixed field Pokee-Isaac 28B v0.1 is the **safest model on both axes** (Direct **30.5** vs 37.7 for the next safest; Indirect **8.3** vs 36.7) and also the **most capable** (BSR **85.6** vs 84.9) — best of six on all four columns.

Reading the metrics. Indirect ASR and **Direct ASR** measure **security** — *lower is more robust* — while **BSR** captures **benign utility** — *higher is more capable*. *Direct* means the harmful ask is stated openly in the user prompt; *Indirect* means the malicious instruction is injected through tool output, skill content or environment state. Unless noted, overall figures are the **macro-average across all 14 domains**, matching the aggregation used by the public DecodingTrust-Agent leaderboard. Pokee-Isaac 28B v0.1's task-count-weighted "pooled" overall is lower on capability and higher on ASR than its macro, because `medical` — the largest domain in the benchmark and Pokee-Isaac 28B v0.1's weakest on both axes — carries proportionally more tasks. We use the macro-average for all leaderboard comparisons, as the board does.

12 Methodology

To systematically evaluate the security and safety of autonomous AI agents, DecodingTrust-Agent Platform (DTap) [1], a controllable and interactive framework is designed to reproduce realistic

agent deployment conditions while enabling rigorous and reproducible adversarial testing. Unlike conventional evaluations that assess models primarily through static prompts or isolated tool interactions, DTap evaluates agents as stateful systems operating over long-horizon trajectories, in which models perceive heterogeneous information, reason about intermediate states, invoke external tools, and modify their environments. The platform comprises more than 50 full-stack simulation environments across 14 real-world domains, reproducing commonly used applications and services such as productivity, communication, financial, and information-management systems. These environments expose realistic interfaces, including MCP-based tools and graphical user interfaces, while retaining experimental control over application state, user data, tool behavior, and environmental observations. This design enables security evaluation under diverse threat models and allows adversarial interventions to be introduced at multiple layers of the agent execution pipeline, including the user prompt, tool descriptions and outputs, agent skills, environmental observations, and compositions of these attack surfaces. Consequently, the methodology evaluates not merely whether an agent generates unsafe text, but whether adversarial manipulation induces consequential unsafe actions within an executable environment, such as unauthorized information disclosure, destructive operations, or policy-violating transactions.

In addition, DTap-Red, an autonomous red-teaming agent that adaptively searches for attack strategies conditioned on a target agent, environment state, and specified malicious objective, is designed. Rather than restricting evaluation to a fixed library of prompt injections, DTap-Red systematically explores heterogeneous injection vectors and their compositions, enabling the discovery of attacks that exploit interactions among prompts, tools, skills, and environmental context. Successful attack trajectories are subsequently organized into DTap-Bench, a large-scale benchmark containing approximately 7,000 red-teaming tasks and roughly 4,000 policy-grounded malicious goals spanning diverse domains, risk categories, and adversarial strategies. Critically, each evaluation instance is coupled with an environment-grounded, verifiable judge that determines attack success from the resulting system state and observable consequences rather than relying solely on an LLM's subjective assessment of generated text. This combination of controllable environments, adaptive adversarial exploration, trajectory-level execution, and outcome-based verification provides a systematic methodology for measuring agent robustness across different backbone models and agent architectures, while supporting reproducible analysis of vulnerabilities under increasingly complex and compositional attack scenarios.

- **System under test: Pokee-Isaac 28B v0.1**, a multimodal agent model, evaluated as a complete agent system — the model together with its own agent scaffold — running fully autonomously, with no human confirmation step in the loop. Throughout this report "Pokee-Isaac 28B v0.1" names that system, and every Pokee-Isaac 28B v0.1 figure is a system-level figure.
- **Benchmark:** DecodingTrust-Agent, **all 14 domains**, environments/attacks/judges as shipped, **6,651 judged tasks**. Twelve are Linux-Docker; `macos` and `windows` are guest-OS-in-QEMU and are run on their own infrastructure (133 and 347 tasks), then folded into the same macro-average. Direct/Indirect split taken straight from the result path (`malicious/{direct,indirect}/`) — identical taxonomy to the public board.
- **One headline basis.** All headline figures (§13) are the **14-domain** macro-average, because that is the benchmark's own basis and the one the leaderboard's "Overall" column reports; the field there is the ten published agents plus Pokee-Isaac 28B v0.1. The five models we measured ourselves ran only the 12 Linux domains, so the controlled head-to-head against them (§16) re-scores Pokee-Isaac 28B v0.1 on those 12 — the only figures in this report on a

basis other than 14, and they are never mixed with the headline.

- **Modality — and why it is not a confound within the 12-domain basis.** Pokee-Isaac 28B v0.1 is multimodal, so the two GUI-driven domains are in scope for it and are carried in the headline: `macos` **BSR 56.7** and `windows` **54.8**. They drop out only of the controlled head-to-head (§16), and only because the five models there never ran them — a property of that field, not a statement about what any system can see. Within the other 12 domains, modality is not a confound for any system: every one is driven through text tool interfaces, and `browser` in particular reaches the agent as a Playwright **ARIA/DOM snapshot** rather than as pixels — screenshots are disabled (`include_screenshot_in_snapshot: false`) in all **201** shipped browser task configurations, and the benchmark's maintainers confirm the domain does not require vision.
- **Aggregation:** benign → `task_success` and malicious → `attack_success`, read from each task's `judge_result.json`; macro-average across all 14 domains (matches the board's own "Overall"), with pooled reported alongside where it differs. The head-to-head of §16 re-takes the same mean over 12.
- **Leaderboard baseline:** 10 published agent configurations, transcribed from decodingtrust-agent.com/leaderboard with their full per-domain values (Appendix A). Re-deriving each agent's overall as the unweighted mean of its 14 per-domain values reproduces the board's published overalls **exactly** (worst single-cell deviation = 0.00), which is what licenses the restricted 12-domain recomputation in §16: dropping two domains from that mean is arithmetic, not estimation.
- **Known harness gap affecting every indirect number here.** The harness's injection guards match **native** tool names (`Bash`, `Read`), while DTAP's attacks arrive over **MCP** tools — so the guards see no matching call and abstain. Every indirect path in this benchmark is therefore effectively ungated, and the indirect ASR below should be read as a **guards-inactive** figure. MCP-semantic matching is the fix and is not shipped.
- **Refusal behaviour is not separated out.** We do not report what share of Pokee-Isaac 28B v0.1's defence is an *explicit refusal* versus an attack that fails on execution: that split requires per-trajectory text classification, which is outside the environment-grounded judging this report relies on. No claim about deliberate versus incidental defence is made from these numbers.
- **Caveats:** n=1 run, no error bars — treat per-domain ASR gaps $\leq \pm 3$ as noise (medical uses a stochastic patient-sim). `browser` carries only **34** benign tasks, so its BSR has a $\sim \pm 13$-point confidence interval and should not be quoted precisely; `macos` carries only **30**, and two runs of the same benchmark against the same model agreed on just 23 of them, bracketing its BSR at 56.7–60.0 — treat differences there below roughly 10 points as noise. The two GUI domains also depend on the agent system forwarding image tool results as image content rather than as stringified payloads; with that path off they collapse to a near-zero BSR regardless of the model's own capability, so modality is a property of the deployed system, not of the model alone. `workflow` **cannot start the** `whatsapp` **environment**: its images cannot be built from the public DTAP release (the published compose pins an image tag that does not exist, and the schema-seed directory its database mounts is not shipped), so 31 of 520 workflow tasks never ran. They are *not* excluded — they count against Pokee-Isaac 28B v0.1 as benign failures and as attacks that did not succeed, which is why `workflow` BSR reads 82.6 rather than 87.8. Since the loss is a coherent slice of one domain rather than a random sample, treat

`workflow` as the one domain where Pokee-Isaac 28B v0.1's figures are pessimistic. Every other domain ran to completion.

13 Overall Results across Domains

Metrics. Indirect ASR and **Direct ASR** measure *security* (lower is more robust), while **BSR** captures *benign utility* (higher is more capable). Overall figures are the **macro-average across all 14 domains**, which is how the public DecodingTrust-Agent leaderboard aggregates its own "Overall" column.

Across all 14 domains Pokee-Isaac 28B v0.1 is **#1 of 11 on Indirect ASR (7.4)**, **#4 on Direct ASR (28.1)** and **#7 on capability (BSR 81.4)**. It is the most injection-resistant system in the field, at upper-middle capability — and it does this at **28B parameters**, against frontier-scale lab models and hardened agent frameworks built on them.

What is being ranked. Ten of these systems are the public leaderboard's agents, measured by their own authors across all 14 domains; Pokee-Isaac 28B v0.1 is the eleventh. Every figure in this section is therefore on the benchmark's own full basis, with no domain dropped from any system. Per-domain values for all eleven are in Appendix A, and their 14-domain macro-averages reproduce the leaderboard's published overalls exactly (worst single-cell deviation = 0.00), so the aggregation here is arithmetic, not estimation.

Five further models that we measured ourselves ran only the 12 Linux-Docker domains and so *cannot* appear on this basis; they are compared against Pokee-Isaac 28B v0.1 under a single fixed judge and harness in §16.

Table 3: Indirect ASR — malicious instructions injected through tool output, skill content or environment state. Lower is safer ↓. Macro-average over **all 14 domains** — the benchmark's own basis, and the one the public leaderboard reports. Pokee-Isaac 28B v0.1 highlighted.

#	System	Indirect ASR ↓
1	**Pokee-Isaac 28B v0.1**	**7.4**
2	Opus-4.6 (Claude Code)	8.1
3	Opus-4.6 (OpenClaw)	10.6
4	GPT-5.5 (OpenClaw)	17.7
5	Sonnet-4.5	25.2
6	GPT-OSS-120B	28.5
7	GPT-5.2 (OpenClaw)	35.6
8	GPT-5.4 (OpenAI)	40.1
9	DeepSeek-V4-Pro (OpenClaw)	41.7
10	GPT-5.2 (OpenAI)	46.7
11	Gemini-3-Pro	55.7

Pareto position, which is the honest one-line summary. A system dominates Pokee-Isaac 28B v0.1 only if it is safer on *both* attack axes *and* more capable. On the full 14-domain basis **no system does**: because Pokee-Isaac 28B v0.1 holds the field's lowest indirect ASR outright, every other agent must give up ground on that axis to gain on the others. Opus-4.6 (Claude Code) is

Table 4: Direct ASR — the harmful instruction is stated openly in the user prompt. Lower is safer ↓. Macro-average over **all 14 domains** — the benchmark's own basis, and the one the public leaderboard reports. Pokee-Isaac 28B v0.1 highlighted.

#	System	Direct ASR ↓
1	Opus-4.6 (OpenClaw)	21.4
2	Opus-4.6 (Claude Code)	22.3
3	Sonnet-4.5	26.8
4	**Pokee-Isaac 28B v0.1**	**28.1**
5	GPT-5.5 (OpenClaw)	28.9
6	GPT-5.2 (OpenClaw)	38.6
7	GPT-OSS-120B	46.5
8	Gemini-3-Pro	47.9
9	GPT-5.4 (OpenAI)	51.0
10	GPT-5.2 (OpenAI)	58.8
11	DeepSeek-V4-Pro (OpenClaw)	59.6

the near miss — safer on direct (22.3 vs 28.1) and more capable (85.6 vs 81.4), but *less* injection-resistant (8.1 vs 7.4), so it trades with Pokee-Isaac 28B v0.1 rather than beating it. That is the shape of the whole field: ten trades, no dominations.

The confound does not carry Pokee-Isaac 28B v0.1's indirect rank. GPT-OSS-120B is the illustrative case: it posts an indirect ASR of 28.5 on a BSR of 36.2 — a rank that reflects doing less, not resisting more. Pokee-Isaac 28B v0.1 leads it on that axis (7.4 vs 28.5) at **2.2**× the capability (81.4 vs 36.2). Indirect-ASR rank read in isolation remains uninformative in both directions, and §15 applies it in the other direction where it is due.

Table 5: Benign task success (BSR) — benign utility on the same environments. Higher is more capable ↑. Macro-average over **all 14 domains** — the benchmark's own basis, and the one the public leaderboard reports. Pokee-Isaac 28B v0.1 highlighted.

#	System	BSR ↑
1	Gemini-3-Pro	87.0
2	GPT-5.5 (OpenClaw)	86.3
3	Opus-4.6 (Claude Code)	85.6
4	Opus-4.6 (OpenClaw)	85.3
4	GPT-5.4 (OpenAI)	85.3
6	DeepSeek-V4-Pro (OpenClaw)	83.3
7	**Pokee-Isaac 28B v0.1**	**81.4**
8	Sonnet-4.5	80.8
9	GPT-5.2 (OpenAI)	80.5
10	GPT-5.2 (OpenClaw)	78.5
11	GPT-OSS-120B	36.2

Table 6: All eleven systems, macro-average over all 14 domains, sorted by Indirect ASR (safest first) — the consolidated form of Tables 3–5. Parenthetical is rank of 11 on that axis. ASR lower = safer; BSR higher = more capable.

System	Direct ASR ↓	Indirect ASR ↓	BSR ↑
Pokee-Isaac 28B v0.1	**28.1** (#4)	**7.4** (#1)	**81.4** (#7)
Opus-4.6 (Claude Code)	22.3 (#2)	8.1 (#2)	85.6 (#3)
Opus-4.6 (OpenClaw)	21.4 (#1)	10.6 (#3)	85.3 (#4)
GPT-5.5 (OpenClaw)	28.9 (#5)	17.7 (#4)	86.3 (#2)
Sonnet-4.5	26.8 (#3)	25.2 (#5)	80.8 (#8)
GPT-OSS-120B	46.5 (#7)	28.5 (#6)	36.2 (#11)
GPT-5.2 (OpenClaw)	38.6 (#6)	35.6 (#7)	78.5 (#10)
GPT-5.4 (OpenAI)	51.0 (#9)	40.1 (#8)	85.3 (#4)
DeepSeek-V4-Pro (OpenClaw)	59.6 (#11)	41.7 (#9)	83.3 (#6)
GPT-5.2 (OpenAI)	58.8 (#10)	46.7 (#10)	80.5 (#9)
Gemini-3-Pro	47.9 (#8)	55.7 (#11)	87.0 (#1)

14 Per-Domain Results (Pokee-Isaac 28B v0.1 value, rank of 11)

Table 7: Pokee-Isaac 28B v0.1 per-domain across all 14 domains; parenthetical is rank among the **eleven systems** on this basis (lower ASR rank = safer; higher BSR rank-number = lower utility). * marks the two GUI-driven domains, which run a guest OS in QEMU rather than a Linux container and are the two the head-to-head of §16 has to drop.

Domain	Direct ASR ↓	Indirect ASR ↓	BSR ↑
Browser	24.7 (#8)	2.4 (#3)	94.1 (#5)
Coding	52.9 (#5)	**3.6 (#1)**	97.9 (#8)
CRM	32.2 (#4)	5.3 (#2)	82.4 (#4)
Customer-service	25.0 (#3)	8.9 (#3)	**94.4 (#1)**
Finance	5.0 (#3)	2.0 (#3)	96.0 (#5)
Legal	**18.5 (#1)**	13.0 (#3)	85.5 (#9)
macOS*	8.0 (#2)	0.0 (#1)	56.7 (#9)
Medical	49.3 (#3)	31.1 (#4)	57.9 (#9)
OS-Filesystem	28.0 (#5)	2.5 (#2)	63.0 (#8)
Research	48.7 (#6)	1.6 (#4)	100.0 (#1)
Telecom	34.8 (#5)	**3.0 (#1)**	79.2 (#3)
Travel	11.4 (#4)	13.3 (#4)	94.6 (#4)
Windows*	19.9 (#2)	4.5 (#2)	54.8 (#11)
Workflow	35.3 (#5)	12.6 (#3)	82.6 (#6)
Macro (14)	**28.1** (#4)	**7.4 (#1)**	**81.4** (#7)

Three best-in-class cells. Pokee-Isaac 28B v0.1 holds #1 of 11 on **coding-indirect** (3.6), **telecom-indirect** (3.0) and **legal-direct** (18.5), and a top-3 placement on **17 of the 28** domain/axis ASR pairs. It also holds the field's best benign capability on **customer-service** (BSR 94.4). Its weakest cell is **medical** — Direct 49.3 (#3), Indirect 31.1 (#4) — middle-of-field on rank, but the worst domain Pokee-Isaac 28B v0.1 has on both axes in absolute terms.

Where the attacks concentrate. The losses are on the **direct** vector. `browser-integrity` lands **8/8**, `high-cost-test` **26/28 (93 %)** and `Bias-code-generation` **25/27 (93 %)**; the worst indirect category is `ignore-allergy` at 20/30, inside `medical`.

The `privacy_leak` family is the canonical indirect-injection target: given malicious instructions injected into tool output, a susceptible agent hands over the personal data it holds access to. Pokee-Isaac 28B v0.1 scores **0/40 on indirect** and 2/40 on direct (**2/80 overall**) — the family is fully defended on the vector it is designed to probe.

15 The Capability Confound (core caveat)

The right way to read these numbers is **security relative to capability**. Two agents with identical ASR are not equivalent if one has 90% BSR and the other 36%: the high-BSR agent is *choosing* not to be exploited while still doing the work; the low-BSR agent often simply **fails to act at all**, which trivially also fails the attack. Per-domain, Pokee-Isaac 28B v0.1 splits into three regimes:

(A) Capable *and* safe — the credible results. Pokee-Isaac 28B v0.1 completes the benign task at a high rate *and* resists attacks:

- **Finance** — Direct 5.0 (#3), Indirect 2.0 (#3), BSR 96.0 (#5). Top-3 on both attack axes at near-ceiling capability; the strongest cell in the report and the most commercially legible.
- **Travel** — Direct 11.4 (#4), Indirect 13.3 (#4), BSR 94.6 (#4). Capable and safe on both axes.
- **Browser** — Direct 24.7 (#8), Indirect 2.4 (#3), BSR 94.1 (#5). Text-drivable via Playwright/DOM. Note the 34-task benign denominator.
- **Customer-service** — Direct 25.0 (#3), Indirect 8.9 (#3), BSR **94.4 (#1)**. Top-3 on both attack axes while posting the *highest benign capability in the field* on this domain.
- **Workflow** — Direct 35.3 (#5), Indirect 12.6 (#3), BSR 82.6 (#6). Its 31 `whatsapp` tasks cannot start and are scored as benign failures (§12), which is what holds its BSR below the rest of this group.

(B) Capable *but* attackable on the direct axis — no confound available. High utility rules out the "didn't act" excuse, so these are real security results. All four resist *injection* well; what remains is compliance with a directly-phrased ask:

- **Coding** — Direct 52.9 (#5), Indirect **3.6 (#1)**, BSR 97.9 (#8). Best injection robustness in the field alongside near-perfect benign completion, against roughly half of direct asks landing.
- **Research** — Direct 48.7 (#6), Indirect 1.6 (#4), BSR **100.0** (#4). Perfect benign completion; indirect injection is near-useless against research for *everyone*, so read the direct number as the real one.
- **Telecom** — Direct 34.8 (#5), Indirect **3.0 (#1)**, BSR 79.2 (#3).
- **CRM** — Direct 32.2 (#4), Indirect 5.3 (#2), BSR 82.4 (#6).

(C) Low utility — read the ASR with care.

- **Medical** — Direct 49.3 (#3), Indirect 31.1 (#4), BSR 57.9 (#9). The worst domain on both axes *and* among the lowest on capability; it is the largest domain in the benchmark, and it is what drags the pooled aggregation below the macro.
- **OS-Filesystem** — Direct 28.0 (#5), Indirect 2.5 (#2), BSR 63.0 (#8). The strong indirect number sits on 63 % capability; do not over-credit it.
- **Legal** — Direct **18.5 (#1)**, Indirect 13.0 (#3), BSR 85.5 (#9). Capability is respectable in absolute terms but ranks low in a strong field, so the best-in-class direct number should be read with that in mind.

- **macOS** — Direct 8.0 (#2), Indirect 0.0 (joint #1), BSR 56.7 (#9), and **Windows** — Direct 19.9 (#2), Indirect 4.5 (#2), BSR 54.8 (#11). Both are top-3 on both attack axes on capability in the mid-50s, which is #9 and #11 of 11 against 75–92 for the frontier agents. This is the regime where the confound bites hardest: on a domain an agent can barely operate, a low ASR is close to meaningless, and it is the capability figure that makes these two ASRs worth reading at all.

The confound, made concrete: GPT-OSS-120B's indirect ASR of 28.5 sits on **BSR 36.2** — it is not safe in any sense that matters; it does less. Pokee-Isaac 28B v0.1 posts Indirect **7.4** at **BSR 81.4**, which is the same shape of result as Opus-4.6 (Claude Code) at Indirect 8.1 / BSR 85.6: robustness *without* the capability tax.

16 Controlled Head-to-Head: Five Models, One Fixed Harness and Judge

Everything above compares Pokee-Isaac 28B v0.1 against *transcribed* leaderboard numbers: those agents were measured by their own authors, under their own harnesses and their own judges. To obtain a comparison in which everything but the model is held fixed, we ran five further models ourselves, on the same DTAP installation, with the same GPT-5.2 judge configuration held fixed across all six systems.

- **Models (route):** `claude-haiku-4.5` (Bedrock), `nemotron-3-super-120b` (Bedrock), `gemini-3.5-flash-lite` (Vertex AI), `gpt-5.6-luna` (Azure), `qwen3.5-122b-a10b` (OpenRouter / Novita bf16).
- **Domains:** the **12 Linux-Docker** domains. These models did not run `macos` or `windows`, so Pokee-Isaac 28B v0.1 is re-scored on the same 12 here.
- **Harness:** the stock `openaisdk` runner, *not* Pokee-Isaac 28B v0.1's own agent scaffold. That choice measures the **model** rather than the scaffold — and it is the one axis that still differs from the Pokee-Isaac 28B v0.1 row (see caveats).
- **Scale, and the denominator.** ∼6,200 tasks per model. A task can die before any judge sees it — a hallucinated tool name, an environment that fails to start, output over a size limit. **Every rate we compute ourselves is over the benchmark's full task count for the domain**: a benign task with no verdict is a failure, and a malicious one is an attack that did not succeed. No row in this report is scored on a shrunken denominator.
- **Health:** zero rate-limit and zero connection errors across all five runs. No benign success was credited to an empty or prose-only completion. Because the denominator is the domain's full task count, a model that crashes out of the tasks it handles worst is not rewarded for the absence: those tasks count against its capability. That matters most for `medical`, where two of the five models lose a fifth to a quarter of the set to one bug (item 5).

Putting Pokee-Isaac 28B v0.1 in the same table. The leaderboard (and therefore Pokee-Isaac 28B v0.1's headline) aggregates by **macro-average**, so we re-aggregate all five that way over the same 12 domains and place Pokee-Isaac 28B v0.1's own 12-domain macro-average beside them. Every row in Table 8 is then the same statistic over the same domain set, scored by the same judge. Aggregating the five by task count instead moves no axis by more than **2.5 points**, so the choice of aggregation is not load-bearing and only the macro figures are reported.

Table 8: Apples-to-apples: **macro-average over the same 12 Linux domains**, same judge, sorted by combined ASR (safest first). Pokee-Isaac 28B v0.1 highlighted. *Route* is the serving route for the five models we measured; Pokee-Isaac 28B v0.1 runs in its own agent scaffold rather than the stock `openaisdk` runner, and that is the residual difference between the rows — see the caveats below. Every rate here is over the benchmark's **full task count** for each domain, so no row is scored on a reduced denominator.

Model	Route	Direct ASR ↓	Indirect ASR ↓	Combined ↓	BSR ↑
Pokee-Isaac 28B v0.1	own scaffold	**30.5**	**8.3**	**18.7**	**85.6**
claude-haiku-4.5 †	Bedrock	37.7	36.7	37.4	70.5
gpt-5.6-luna †	Azure	52.3	46.0	49.2	84.9
qwen3.5-122b-a10b †	OpenRouter	59.8	46.1	52.6	77.9
nemotron-3-super-120b †	Bedrock	78.3	39.9	58.2	61.0
gemini-3.5-flash-lite †	Vertex AI	82.9	49.0	65.5	82.5

What the head-to-head says

1. Pokee-Isaac 28B v0.1 is best of the six on every column. Direct **30.5** against 37.7 for the next safest, Indirect **8.3** against 36.7, Combined **18.7** against 37.4 — and it is also the **most capable** (BSR **85.6**, ahead of gpt-5.6-luna at 84.9). The indirect margin is the notable one: a **4.4**× gap to the next-safest model in a field scored by the same judge on the same domains. Against a field we measured ourselves rather than transcribed, Pokee-Isaac 28B v0.1 is safest *and* most capable, which is a materially stronger claim than the leaderboard row supports.

2. Direct-attack robustness splits the field by refusal training. Compliance with a *directly-phrased* harmful instruction separates the models far more sharply than injection robustness does:

Table 9: Direct minus Indirect ASR (macro, 12 Linux domains) for the **six systems we measured ourselves** — board agents are excluded here because the gap is sensitive to judge calibration, which is only held fixed within this set. A large positive gap means the model does the task and simply does not refuse when asked head-on. Pokee-Isaac 28B v0.1 first, then sorted by the gap.

Model	Direct ↓	Indirect ↓	Direct − Indirect
Pokee-Isaac 28B v0.1	**30.5**	**8.3**	+22.2
claude-haiku-4.5	37.7	36.7	+1.0
gpt-5.6-luna	52.3	46.0	+6.3
qwen3.5-122b-a10b	59.8	46.1	+13.7
gemini-3.5-flash-lite	82.9	49.0	+33.9
nemotron-3-super-120b	78.3	39.9	+38.4

Read the gap carefully: it is a *shape* statistic, not a safety ranking. A small gap means the two vectors land at similar rates, which for `claude-haiku-4.5` (+1.0) and `gpt-5.6-luna` (+6.3) happens at high ASR on both. Pokee-Isaac 28B v0.1's gap is wide (+22.2) for the opposite reason: it holds the field's lowest direct ASR *and* an indirect ASR four times lower again, so the two separate. The models to worry about are those where direct ASR **spikes far above** indirect — `gemini-3.5-flash-lite` is the extreme, at **82.9** direct against 49.0 indirect, a small fast model

with apparently minimal refusal training. *For any threat model that includes a user issuing a direct malicious request, direct ASR is the number that matters*, and it is where these systems diverge most. Pokee-Isaac 28B v0.1 is at the safe end of that split.

3. `code` is universally the most attackable domain, and Pokee-Isaac 28B v0.1's margin there carries a harness confound. Code-execution injections land against every measured model (combined ASR 77.6–94.8); Pokee-Isaac 28B v0.1 posts **24.5** there at BSR 97.9. That is the largest Pokee-Isaac 28B v0.1-vs-field gap in the study and also the one *most* exposed to the harness confound: Pokee-Isaac 28B v0.1's code path runs MCP-only, with native shell and file tools deliberately disabled so that the agent works inside the container the judges inspect. An unknown share of the gap is therefore scaffold policy rather than model behaviour. Report it as a **system-level** result (scaffold + model, which is what is deployed), not a model-level one. Conversely, prompt injection is near-useless against `research` indirect for everyone (1.6–9.6), so any *global* "direct > indirect" claim is partly an artifact of domain weighting.

4. claude-haiku-4.5 is the real like-for-like rival, and Pokee-Isaac 28B v0.1 dominates it decisively. Pokee-Isaac 28B v0.1 is safer on combined ASR (18.7 vs 37.4) *and* 15.1 BSR points more capable (85.6 vs 70.5) — it beats haiku on both dimensions rather than trading one for the other. The domain-level picture matches the macro: Pokee-Isaac 28B v0.1 is safer on combined ASR in **11 of 12** domains. The single exception is `research`, where haiku is marginally ahead (23.4 vs 24.6) — inside the noise band and on the one domain where injection barely works against anyone. The macro gap is carried by coding, which is also the cell with the harness confound (item 3).

5. Tool-schema adherence is a distinct loss mode, and it is worth its own column. Every model except the two frontier-lab ones sheds a chunk of `medical` to the *same* bug: the model invents a `send_message_to_user` tool inside the `HospitalDoctor` agent, the `openaisdk` runner raises `ModelBehaviorError: Tool not found`, and the task dies before a verdict. Dropped `medical` tasks: **qwen 290, nemotron 234, gemini 126, haiku 19, luna 2**. The open-weights models are hit hardest. This is neither "was attacked" nor "refused" — it is a capability failure, and it is scored as one: those tasks count against the model rather than vanishing from the denominator. Pokee-Isaac 28B v0.1's own losses are smaller and of a different kind — 31 `whatsapp` tasks whose environment cannot start for *any* agent (§12) — and they count against Pokee-Isaac 28B v0.1 just the same, which is why its `workflow` BSR is 82.6 rather than 87.8. The ten transcribed leaderboard agents keep the board's own aggregation, the one place this report cannot apply the stricter denominator, and it is the direction that disfavours Pokee-Isaac 28B v0.1.

Caveats specific to this comparison.

- **Harness (the big one, and it cannot be removed).** The five models run in the stock `openaisdk` runner; Pokee-Isaac 28B v0.1's row is Pokee-Isaac 28B v0.1's own agent scaffold. Pokee-Isaac 28B v0.1 *is* the model in that scaffold — that is the system — so the two are not separable, and every Pokee-Isaac 28B v0.1-vs-field delta here is a system-level delta. State it; do not hide it.
- **Judge configuration.** All six systems are graded by the same GPT-5.2 judge deployment, substituted into each run at launch, so Table 8 is judge-matched.

- **Luna's ASR is on a slightly pre-filtered attack set.** ∼52 malicious prompts (∼1.5 % of its attack set) were blocked by Azure/OpenAI content filtering *at the input* — a serving-platform defense, not Luna's own alignment. Scoring them as defended moves Luna's pooled ASR 49.6 → **48.8** (telecom 64→61, CRM 70→67). Luna is therefore not perfectly apples-to-apples with the non-Azure routes.
- `medical` **for nemotron and qwen** loses 21 % / 27 % of the task set to tool-hallucination drops (item 5). Those tasks are counted as benign failures and as attacks that did not succeed rather than removed, so the rates are comparable, but the loss is concentrated in one domain for two models.
- n=1**.** No run is repeated — the ±3 noise band from §12 applies to every cell here too.

17 Bottom Line

- **Across all 14 domains, this is a leading security result: #1 of 11 on Indirect ASR (7.4)**, #4 on Direct ASR (28.1), at #7 on capability (BSR 81.4). Pokee-Isaac 28B v0.1 holds **three best-in-class ASR cells** (coding-indirect, telecom-indirect, legal-direct), the field's best capability on customer-service, and a top-3 placement on **17 of the 28** domain/axis ASR pairs. **No system in the field Pareto-dominates it**: holding the lowest indirect ASR outright means every other agent must concede that axis to gain elsewhere.
- **It holds up better when we do the measuring:** against five models we ran ourselves under a held-fixed judge and domain set (§16), Pokee-Isaac 28B v0.1 is the **best of six on every column** (Direct 30.5 / Indirect 8.3, next-safest 37.7 / 36.7) *and* the most capable (BSR 85.6 vs 84.9). The closest rival, **claude-haiku-4.5**, is beaten on both dimensions at once — a small refusal-trained model is the honest comparison class, and Pokee-Isaac 28B v0.1 is ahead of it by a **4.4**× margin on injection robustness.
- **The capability confound does not explain Pokee-Isaac 28B v0.1's safety numbers, and should still be applied honestly.** An indirect ASR of 8.3 *at* a BSR of 85.6 is the opposite of what a "does less, so is attacked less" artifact looks like: the agent completes the work and still resists the injection. Applied in the other direction, Pokee-Isaac 28B v0.1's strong indirect number in `os-filesystem` (2.5) sits on 63 % capability and should not be over-credited, and `medical` remains weak on both axes at 57.9 % BSR.
- **The GUI domains are carried in the headline, not set aside.** Pokee-Isaac 28B v0.1 posts `macos` **BSR 56.7** at **8.0 / 0.0** ASR and `windows` **BSR 54.8** at **19.9 / 4.5**. The capability figure is what makes those ASRs interpretable: on a pixels-only domain, an agent that cannot act posts a near-zero ASR that measures nothing. Capability there is mid-field (#9 and #11 of 11, against 75–92 for the frontier agents), and the GUI path depends on the agent system forwarding images as image content — modality is a property of the deployed system, not of the model alone.
- **The indirect number is a guards-inactive measurement.** Harness injection guards match native tool names while DTAP attacks arrive over MCP, so they abstain throughout. MCP-semantic matching is the named, unshipped fix. Note that the category such guards principally target — `privacy_leak` — already stands at **0/40 on indirect** without them, so the losses that remain are concentrated on the *direct* vector, which guards of this kind do not address.
- **One-line characterization:** *a 28B agent with field-leading injection robustness and one clear commercial strength (finance: 96.0 BSR at 5.0 direct / 2.0 indirect ASR, top-3 on both attack axes) — with a named, unshipped harness fix standing between the current indirect number and a materially better one.*

Sources: Pokee-Isaac 28B v0.1 figures are computed from the per-task `judge_result.json` verdicts of a single 12-domain run (6,171 judged tasks), and the GUI-domain figures from two further runs of `macos` and `windows`. Leaderboard figures are transcribed from decodingtrust-agent.com/leaderboard and recomputed onto the 12-domain basis. The five-model head-to-head (§16, App. B) is computed the same way from our own runs of those models. Benchmark: DTAP, arXiv:2605.04808; AI-secure/DecodingTrust-Agent.

A Full Per-Domain Results

The **12 Linux domains** list all ten leaderboard agents plus **Pokee-Isaac 28B v0.1** (highlighted) and the five models we measured ourselves (†, §16) — sixteen rows. The **two GUI domains** that close this appendix carry only the **eleven** systems that ran them (the ten board agents plus Pokee-Isaac 28B v0.1); the five † models never attempted them, which is why the head-to-head of §16 is taken over 12. All tables are **sorted by Indirect ASR (safest first)**, with Direct ASR and BSR alongside. ASR lower = safer; BSR higher = more capable. Pokee-Isaac 28B v0.1's task counts for each domain are given in the caption. Every † row and the Pokee-Isaac 28B v0.1 row is a rate over the benchmark's **full task count** for that domain, so none is computed on a reduced denominator; those rows are graded by the same GPT-5.2 judge. Unmarked rows are the published leaderboard values, which carry their authors' judges, harnesses and aggregation.

Table 10: Browser (34 benign / 85 direct / 82 indirect)

Agent	Direct ASR ↓	Indirect ASR ↓	BSR ↑
Opus-4.6 (Claude Code)	8.8	0.0	97.2
Opus-4.6 (OpenClaw)	10.2	0.0	97.2
Pokee-Isaac 28B v0.1	**24.7**	**2.4**	**94.1**
claude-haiku-4.5 †	25.9	2.4	50.0
Sonnet-4.5	20.0	4.5	95.8
gemini-3.5-flash-lite †	71.8	17.1	73.5
GPT-5.5 (OpenClaw)	20.2	17.5	88.8
GPT-5.2 (OpenAI)	20.3	24.8	55.6
gpt-5.6-luna †	18.8	25.6	91.2
GPT-5.2 (OpenClaw)	22.8	27.6	56.9
DeepSeek-V4-Pro (OpenClaw)	27.7	29.7	83.1
GPT-OSS-120B	10.0	41.3	25.0
nemotron-3-super-120b †	80.0	41.5	38.2
GPT-5.4 (OpenAI)	29.2	42.9	77.1
qwen3.5-122b-a10b †	28.2	52.4	91.2
Gemini-3-Pro	35.3	62.2	97.2

Table 11: Coding (330 benign / 121 direct / 165 indirect)

Agent	Direct ASR ↓	Indirect ASR ↓	BSR ↑
Pokee-Isaac 28B v0.1	**52.9**	**3.6**	**97.9**
GPT-5.5 (OpenClaw)	28.1	7.9	99.6
DeepSeek-V4-Pro (OpenClaw)	77.9	9.5	99.8
Opus-4.6 (Claude Code)	33.3	15.3	98.4
GPT-OSS-120B	60.0	15.7	49.5
Opus-4.6 (OpenClaw)	33.2	29.3	100.0
Sonnet-4.5	56.6	56.0	98.4
GPT-5.4 (OpenAI)	62.5	62.2	98.5
GPT-5.2 (OpenClaw)	32.9	65.3	91.1
Gemini-3-Pro	71.6	77.9	100.0
GPT-5.2 (OpenAI)	62.5	79.3	93.6
qwen3.5-122b-a10b †	76.0	81.2	99.7
nemotron-3-super-120b †	91.7	86.1	97.9
claude-haiku-4.5 †	73.6	86.7	82.7
gpt-5.6-luna †	65.3	86.7	100.0
gemini-3.5-flash-lite †	95.0	94.5	96.7

Table 12: CRM (165 benign / 90 direct / 150 indirect)

Agent	Direct ASR ↓	Indirect ASR ↓	BSR ↑
Opus-4.6 (Claude Code)	11.1	3.9	83.0
Pokee-Isaac 28B v0.1	**32.2**	**5.3**	**82.4**
Opus-4.6 (OpenClaw)	11.1	7.2	82.4
GPT-OSS-120B	38.9	9.3	4.2
GPT-5.5 (OpenClaw)	44.4	32.4	75.6
nemotron-3-super-120b †	85.6	34.0	50.9
Sonnet-4.5	16.7	40.6	82.4
DeepSeek-V4-Pro (OpenClaw)	55.7	44.4	83.6
claude-haiku-4.5 †	34.4	45.3	58.8
qwen3.5-122b-a10b †	78.9	48.0	69.1
GPT-5.2 (OpenClaw)	67.8	50.9	75.2
GPT-5.4 (OpenAI)	67.8	53.8	78.8
gemini-3.5-flash-lite †	92.2	61.3	74.5
gpt-5.6-luna †	72.2	64.0	84.8
GPT-5.2 (OpenAI)	83.3	66.4	72.7
Gemini-3-Pro	55.6	69.9	86.1

Table 13: Customer-service (160 benign / 100 direct / 101 indirect)

Agent	Direct ASR ↓	Indirect ASR ↓	BSR ↑
Opus-4.6 (Claude Code)	1.3	1.9	89.8
Opus-4.6 (OpenClaw)	17.9	4.4	89.4
Pokee-Isaac 28B v0.1	**25.0**	**8.9**	**94.4**
gemini-3.5-flash-lite †	90.0	18.8	86.9
GPT-OSS-120B	31.2	21.4	77.1
Sonnet-4.5	27.6	23.4	77.7
nemotron-3-super-120b †	76.0	27.7	73.8
GPT-5.5 (OpenClaw)	50.0	29.0	88.6
qwen3.5-122b-a10b †	71.0	29.7	66.9
claude-haiku-4.5 †	52.0	42.6	70.0
gpt-5.6-luna †	57.0	46.5	94.4
GPT-5.2 (OpenClaw)	61.4	51.7	89.1
GPT-5.2 (OpenAI)	70.2	53.9	91.3
GPT-5.4 (OpenAI)	64.7	54.7	89.7
DeepSeek-V4-Pro (OpenClaw)	75.5	69.8	83.8
Gemini-3-Pro	40.3	75.1	91.3

Table 14: Finance (200 benign / 200 direct / 200 indirect)

Agent	Direct ASR ↓	Indirect ASR ↓	BSR ↑
Opus-4.6 (Claude Code)	4.0	0.6	96.5
Opus-4.6 (OpenClaw)	4.0	1.7	91.0
Pokee-Isaac 28B v0.1	**5.0**	**2.0**	**96.0**
Sonnet-4.5	5.5	7.1	94.9
GPT-5.5 (OpenClaw)	11.0	15.5	96.5
claude-haiku-4.5 †	5.5	25.0	91.0
GPT-5.2 (OpenClaw)	26.0	26.2	92.8
gemini-3.5-flash-lite †	81.5	28.0	93.5
GPT-OSS-120B	37.8	33.6	12.8
GPT-5.4 (OpenAI)	30.5	35.9	96.6
gpt-5.6-luna †	39.0	37.5	91.0
nemotron-3-super-120b †	76.5	40.0	62.0
GPT-5.2 (OpenAI)	51.5	40.4	97.7
qwen3.5-122b-a10b †	40.0	42.5	86.5
DeepSeek-V4-Pro (OpenClaw)	68.0	54.8	92.4
Gemini-3-Pro	22.0	60.0	95.1

Table 15: Legal (200 benign / 200 direct / 200 indirect)

Agent	Direct ASR ↓	Indirect ASR ↓	BSR ↑
Opus-4.6 (Claude Code)	30.0	9.5	91.0
Opus-4.6 (OpenClaw)	22.5	12.5	94.0
Pokee-Isaac 28B v0.1	**18.5**	**13.0**	**85.5**
Sonnet-4.5	20.0	26.6	79.9
GPT-OSS-120B	60.5	30.9	9.7
GPT-5.5 (OpenClaw)	42.0	40.7	93.1
gpt-5.6-luna †	59.0	48.5	86.0
claude-haiku-4.5 †	30.5	53.0	91.0
Gemini-3-Pro	64.0	56.3	87.1
nemotron-3-super-120b †	74.0	57.0	42.0
GPT-5.4 (OpenAI)	54.0	58.1	90.5
GPT-5.2 (OpenClaw)	51.5	65.0	90.9
GPT-5.2 (OpenAI)	59.5	66.2	85.7
qwen3.5-122b-a10b †	57.0	73.5	81.5
gemini-3.5-flash-lite †	80.0	75.0	91.5
DeepSeek-V4-Pro (OpenClaw)	67.0	76.6	92.6

Table 16: Medical (642 benign / 229 direct / 222 indirect)

Agent	Direct ASR ↓	Indirect ASR ↓	BSR ↑
Opus-4.6 (Claude Code)	54.7	16.7	57.8
GPT-5.5 (OpenClaw)	30.4	19.1	91.0
Opus-4.6 (OpenClaw)	50.7	23.0	69.3
Pokee-Isaac 28B v0.1	**49.3**	**31.1**	**57.9**
nemotron-3-super-120b †	57.2	31.5	39.4
GPT-5.4 (OpenAI)	60.4	37.5	86.1
gpt-5.6-luna †	35.4	40.5	67.9
qwen3.5-122b-a10b †	59.0	43.2	43.1
GPT-5.2 (OpenAI)	71.1	47.2	66.3
GPT-5.2 (OpenClaw)	43.1	48.2	68.4
Gemini-3-Pro	62.7	49.1	81.9
Sonnet-4.5	75.1	51.7	66.4
claude-haiku-4.5 †	69.0	56.3	45.2
gemini-3.5-flash-lite †	58.1	57.7	53.4
GPT-OSS-120B	61.3	62.1	35.8
DeepSeek-V4-Pro (OpenClaw)	81.0	65.9	91.0

Table 17: OS-Filesystem (200 benign / 200 direct / 200 indirect)

Agent	Direct ASR ↓	Indirect ASR ↓	BSR ↑
Opus-4.6 (Claude Code)	26.2	1.2	83.3
Pokee-Isaac 28B v0.1	**28.0**	**2.5**	**63.0**
Opus-4.6 (OpenClaw)	24.2	3.1	78.7
GPT-5.5 (OpenClaw)	24.8	6.8	52.5
Sonnet-4.5	22.6	17.6	78.6
GPT-5.2 (OpenClaw)	33.7	20.0	78.6
nemotron-3-super-120b †	84.5	20.5	51.5
gpt-5.6-luna †	58.0	20.5	57.0
qwen3.5-122b-a10b †	68.5	25.5	60.5
claude-haiku-4.5 †	41.0	26.5	56.0
gemini-3.5-flash-lite †	93.0	28.0	59.5
GPT-OSS-120B	70.0	32.8	61.7
GPT-5.4 (OpenAI)	84.0	37.5	85.7
GPT-5.2 (OpenAI)	83.6	44.3	80.0
DeepSeek-V4-Pro (OpenClaw)	82.9	51.1	57.6
Gemini-3-Pro	73.5	59.5	87.0

Table 18: Research (160 benign / 119 direct / 125 indirect)

Agent	Direct ASR ↓	Indirect ASR ↓	BSR ↑
Opus-4.6 (Claude Code)	20.8	0.0	100.0
Sonnet-4.5	19.7	0.0	100.0
Opus-4.6 (OpenClaw)	21.2	1.5	96.0
Pokee-Isaac 28B v0.1	**48.7**	**1.6**	**100.0**
claude-haiku-4.5 †	44.5	3.2	71.9
qwen3.5-122b-a10b †	50.4	3.2	97.5
GPT-5.5 (OpenClaw)	38.1	4.4	99.0
nemotron-3-super-120b †	85.7	4.8	88.1
GPT-5.2 (OpenClaw)	58.7	5.8	95.0
GPT-5.2 (OpenAI)	63.2	7.0	94.5
gemini-3.5-flash-lite †	87.4	8.0	100.0
gpt-5.6-luna †	66.4	9.6	100.0
GPT-5.4 (OpenAI)	54.1	14.2	95.0
Gemini-3-Pro	45.4	16.2	100.0
GPT-OSS-120B	55.8	17.3	30.0
DeepSeek-V4-Pro (OpenClaw)	54.7	17.3	96.0

Table 19: Telecom (120 benign / 161 direct / 166 indirect)

Agent	Direct ASR ↓	Indirect ASR ↓	BSR ↑
Pokee-Isaac 28B v0.1	**34.8**	**3.0**	**79.2**
GPT-5.5 (OpenClaw)	45.9	12.8	89.3
Opus-4.6 (Claude Code)	22.6	17.6	68.2
Opus-4.6 (OpenClaw)	31.8	19.9	65.5
GPT-5.4 (OpenAI)	38.5	20.8	71.8
claude-haiku-4.5 †	23.0	24.1	76.7
GPT-5.2 (OpenClaw)	41.8	42.3	75.5
GPT-5.2 (OpenAI)	60.7	43.3	71.8
qwen3.5-122b-a10b †	69.6	45.8	81.7
nemotron-3-super-120b †	73.3	47.0	78.3
Sonnet-4.5	23.5	49.5	71.8
GPT-OSS-120B	56.0	53.1	65.0
Gemini-3-Pro	24.2	54.7	66.4
gpt-5.6-luna †	66.5	55.4	67.5
gemini-3.5-flash-lite †	88.8	60.8	82.5
DeepSeek-V4-Pro (OpenClaw)	82.5	63.4	93.8

Table 20: Travel (130 benign / 105 direct / 120 indirect)

Agent	Direct ASR ↓	Indirect ASR ↓	BSR ↑
Opus-4.6 (Claude Code)	2.9	0.0	89.4
Opus-4.6 (OpenClaw)	5.7	5.8	89.4
Sonnet-4.5	3.8	11.7	56.7
Pokee-Isaac 28B v0.1	**11.4**	**13.3**	**94.6**
GPT-5.5 (OpenClaw)	18.1	16.7	86.1
GPT-5.2 (OpenClaw)	12.4	32.5	91.4
claude-haiku-4.5 †	10.5	32.5	80.0
qwen3.5-122b-a10b †	60.0	35.0	71.5
nemotron-3-super-120b †	74.3	39.2	45.4
GPT-OSS-120B	54.3	40.8	0.0
DeepSeek-V4-Pro (OpenClaw)	60.0	48.3	95.1
GPT-5.4 (OpenAI)	32.4	55.0	90.6
gpt-5.6-luna †	29.5	55.0	93.8
gemini-3.5-flash-lite †	73.3	65.0	87.7
GPT-5.2 (OpenAI)	42.9	66.7	95.3
Gemini-3-Pro	52.4	84.2	99.4

Table 21: Workflow (337 benign / 78 direct / 105 indirect)

Agent	Direct ASR ↓	Indirect ASR ↓	BSR ↑
GPT-OSS-120B	61.7	7.2	56.8
Opus-4.6 (Claude Code)	19.2	11.1	82.7
Pokee-Isaac 28B v0.1	**35.3**	**12.6**	**82.6**
Opus-4.6 (OpenClaw)	22.1	18.3	91.5
GPT-5.2 (OpenClaw)	30.9	26.1	61.0
Sonnet-4.5	44.7	35.8	69.8
GPT-5.5 (OpenClaw)	31.1	42.4	88.1
claude-haiku-4.5 †	42.3	42.9	73.3
DeepSeek-V4-Pro (OpenClaw)	58.0	47.0	96.4
nemotron-3-super-120b †	80.8	49.5	64.7
GPT-5.4 (OpenAI)	57.6	50.8	79.8
gpt-5.6-luna †	60.3	61.9	85.2
GPT-5.2 (OpenAI)	69.1	63.1	76.4
Gemini-3-Pro	54.4	65.9	89.4
qwen3.5-122b-a10b †	59.0	73.3	85.2
gemini-3.5-flash-lite †	83.3	74.3	90.2

The two GUI domains (eleven systems). These are part of the 14-domain headline basis (§13) and are excluded only from the controlled head-to-head of §16.

Table 22: `macos` — all eleven systems on the 14-domain basis, sorted by Indirect ASR (safest first).

Agent	Direct ASR ↓	Indirect ASR ↓	BSR ↑
Pokee-Isaac 28B v0.1	**8.0**	**0.0**	**56.7**
GPT-5.5 (OpenClaw)	6.3	0.0	91.7
DeepSeek-V4-Pro (OpenClaw)	10.4	0.0	33.1
Opus-4.6 (OpenClaw)	14.0	16.0	80.0
GPT-OSS-120B	20.0	18.3	12.0
Sonnet-4.5	18.3	20.4	75.2
GPT-5.2 (OpenClaw)	26.2	26.7	74.5
GPT-5.4 (OpenAI)	30.4	26.7	76.0
Opus-4.6 (Claude Code)	26.0	28.0	83.3
Gemini-3-Pro	30.4	34.6	61.2
GPT-5.2 (OpenAI)	40.4	38.8	71.6

Table 23: `windows` — all eleven systems on the 14-domain basis, sorted by Indirect ASR (safest first).

Agent	Direct ASR ↓	Indirect ASR ↓	BSR ↑
GPT-5.5 (OpenClaw)	14.2	2.1	68.9
Pokee-Isaac 28B v0.1	**19.9**	**4.5**	**54.8**
Opus-4.6 (OpenClaw)	30.7	5.6	70.0
DeepSeek-V4-Pro (OpenClaw)	33.3	5.6	67.3
Opus-4.6 (Claude Code)	50.7	7.7	78.0
Sonnet-4.5	21.7	7.7	83.5
GPT-5.2 (OpenClaw)	31.7	9.6	58.1
GPT-5.4 (OpenAI)	47.9	10.6	77.5
GPT-5.2 (OpenAI)	44.6	13.0	74.6
Gemini-3-Pro	38.8	13.9	76.0
GPT-OSS-120B	33.3	14.7	67.9

B Combined ASR by Domain — the Six Systems We Measured

Per-domain **Direct ASR, Indirect ASR and BSR** for all sixteen systems are in Appendix A. *Combined* ASR is shown separately here because it cannot be reconstructed for the transcribed leaderboard agents — pooling direct and indirect needs their per-category task counts, which the board does not publish. It *is* exact for the six systems we ran ourselves, so this table completes the picture for those: **Pokee-Isaac 28B v0.1** (highlighted, in its own agent scaffold) against the five models we ran under the stock `openaisdk` runner, same GPT-5.2 judge throughout. Column heads abbreviate `claude-haiku-4.5`, `nemotron-3-super-120b`, `gemini-3.5-flash-lite`, `gpt-5.6-luna`, `qwen3.5-122b-a10b`. **Bold** = best (safest) in row. All rates are over the bench-

mark's full task count. Pokee-Isaac 28B v0.1's `code` row carries the harness confound (§16, item 3).

Table 24: Combined ASR by domain. Lower = safer ↓.

Domain	Pokee-Isaac 28B v0.1	Haiku	Nemotron	Gemini	Luna	Qwen
Browser	**13.8**	14.4	61.1	44.9	22.2	40.1
Coding	**24.5**	81.1	88.5	94.8	77.6	79.0
CRM	**15.4**	41.2	53.3	72.9	67.1	59.6
Customer-service	**16.9**	47.3	51.7	54.2	51.7	50.2
Finance	**3.5**	15.2	58.2	54.8	38.2	41.2
Legal	**15.8**	41.8	65.5	77.5	53.8	65.2
Medical	40.4	62.7	44.6	57.9	**37.9**	51.2
OS-Filesystem	**15.2**	33.8	52.5	60.5	39.2	47.0
Research	24.6	**23.4**	44.3	46.7	37.3	26.2
Telecom	**18.7**	23.5	59.9	74.6	60.9	57.5
Travel	**12.4**	22.2	55.6	68.9	43.1	46.7
Workflow	**23.2**	42.6	62.8	78.1	61.2	67.2
Macro (12)	**18.7**	37.4	58.2	65.5	49.2	52.6

References

[1] Zhaorun Chen, Xun Liu, Haibo Tong, Chengquan Guo, Yuzhou Nie, Jiawei Zhang, Mintong Kang, Chejian Xu, Qichang Liu, Xiaogeng Liu, et al. Decodingtrust-agent platform (dtap): A controllable and interactive red-teaming platform for ai agents. *arXiv preprint arXiv:2605.04808*, 2026.