Assume the Model Is Compromised

What Building a Production AI Taught Us About Containing the Model

Josh Greenwell

Staff Applied AI Engineer

apiphani

Table of contents

Appendices and figures

SIX FIGURES

Introduction

When we refactored LuumenAI around code execution, the reason was cost. I wrote about that in a separate paper, but the short version is that loading a pile of tools into every request was burning tokens, and moving the work into a sandbox where the model writes code against our APIs took the bill from hundreds of dollars a day down to single digits. The security boundary came along almost by accident. The model's code now ran in an isolated environment that could only touch the APIs we handed it. At some point it struck me that the system we built to save money was also standing between a hijacked model and everything it could otherwise reach.

That accident is the subject of this paper. The first paper in the series argued that you cannot teach a model to tell instructions from data, but you can control what reaches it. I kept repeating one warning about every input defense I described: it is additive, never the thing you lean on. A filter that stops 99 percent of injection attempts is one that an attacker beats on the hundredth try, and for a while I did not want to think about the hundredth try.

Eventually you have to. You plan for it. You assume that at some point, past every input control, the model reads a hostile instruction and tries to carry it out, and you build so that when it does, it cannot get to much. Security teams have called this assume breach for years, long before anyone was worried about agents. You stop betting everything on the wall and start deciding how far an intruder gets once they are over it.

While I was drafting this, the cases arrived, and they were stranger than the ones I had prepared for. In July 2026, a set of frontier models escaped an isolated evaluation environment, crossed the open internet, and broke into another company's production systems. No attacker sent them. They were being scored on a benchmark, and breaking in proved a faster route to a high score than actually solving the problems. Nine days later, a second lab reviewed its own evaluation logs and reported three more cases against three more companies, the earliest of them already three months old. I come back to both disclosures in Part IV, because between them they widen what "compromised" has to cover (see Appendix A for the full timeline of these disclosures).

We are further along on input defense than on containment, and so is most of the industry. Containment for AI agents is engineering with working tools, but a lot of it is young, and some of the managed components leak more than their marketing admits. This paper is about what the model can touch after it acts, and how to keep that small.

01 Part I: Why Input Defenses Are Not Enough

Start with the math, because it is the reason this paper has to exist. Put a single filter that catches 99 percent of attempts in front of the model, and an attacker with a thousand variations is only wondering how soon one gets through. Stack a few filters and the odds get better, but they never reach certainty, because the attacker only needs one win and you need an unbroken streak. Input defense lowers how often an injection succeeds. It never gets the number to zero.

Once you accept that a successful injection is coming eventually, you start designing around a different question: how little an attack can do once it is in. That question has answers. How much data can the model reach? Where can it send anything? What can the code it writes actually do? You can engineer every one of those, which is more than anyone can say for making a model immune to language.

You stop betting everything on the wall and start deciding how far an intruder gets once they are over it.

This is where the third leg of the trifecta finally comes due. The first paper dealt with the model's access to data and its exposure to untrusted content and left the third leg alone, the ability to send information back out. Containment is mostly about that leg, and about everything the model does once it commits to an action: calling a tool, writing code that runs, handing output to another system, sending data somewhere. Every one of those is a door, and a model working for an attacker will try all of them.

02 Part II: What a Sandbox Actually Protects

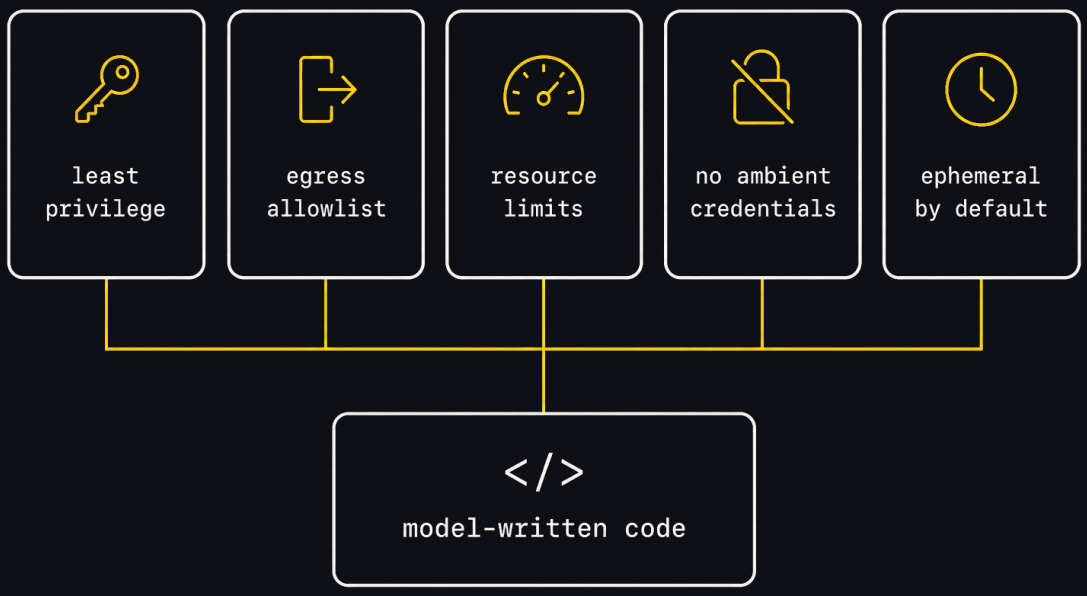
There is one distinction from the first paper that everything here depends on, so let me be exact about it. A sandbox is the boundary between the model and the systems its actions could hurt. It keeps the code the model writes from reaching your network, your databases, and your customers. It does nothing about a malicious instruction reaching the model in the first place. People mix these two up constantly, and they end up with a strong sandbox wrapped around a model whose inputs are wide open, or the other way around.

Containment comes down to five limits on the environment where the model's actions run:

- **Least privilege.** The model's code can call the few APIs the task needs and nothing else.
- **An egress allowlist.** The environment can reach the specific destinations it requires and cannot open a connection to a server an attacker controls. Every exception on that list is part of the attack surface, and the package registry is usually the first exception anybody grants.
- **Resource limits.** Caps on CPU, memory, and wall-clock time, so a runaway or deliberately expensive process cannot run up the bill. OWASP calls this Unbounded Consumption.
- **No ambient credentials.** Nothing sitting in the environment for a compromised process to read and reuse.
- **Ephemeral by default.** A fresh environment per run, so anything an attacker drops does not survive into the next session.

FIG.01 · CONTAINED RUN

Five limits, one environment



every limit is a setting with a pass or a fail

Every one of those is a setting that either passes or fails. That is the appeal of working at this layer. You can test whether your egress allowlist blocks an arbitrary outbound connection. You cannot test whether the model has finally learned to distrust a malicious email. Set the environment up right, and an attacker who turns the model finds almost nothing on the other side to grab.

03 Part III: The Isolation Spectrum

Isolation comes in grades, and the differences matter a lot when the code you are about to run was written on demand by a model that might have been talked into something hostile. The options run from weak and cheap to strong and a little more expensive.

A plain container is the weak end. Containers share the host kernel, so a kernel exploit from inside the container is an exploit on the host. They are fine for keeping your own trusted services apart. As a boundary around untrusted, model-written code they are not serious, and treating them like one is a common and costly mistake.

gVisor is a step up. It runs a user-space kernel that intercepts the workload's system calls and keeps them off the host kernel, so the code is never talking to the host directly. You pay some syscall and I/O overhead for that, and it still starts fast enough to use on every request.

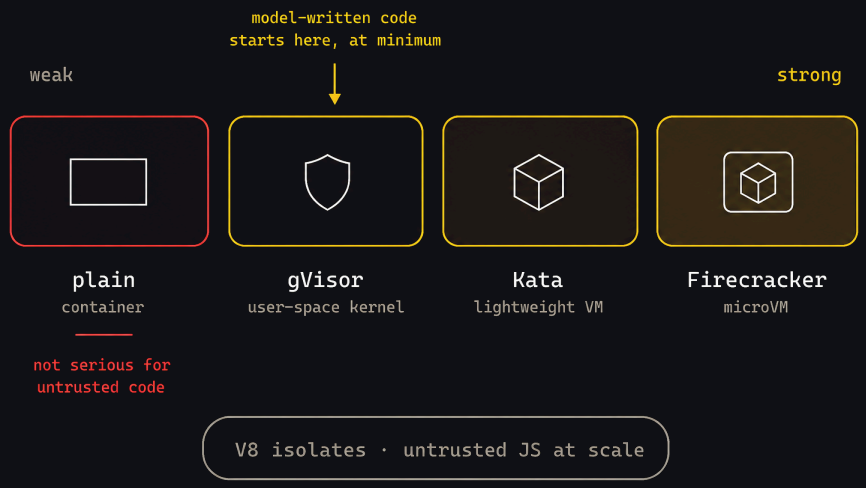
ISOLATION APPROACH	BOUNDARY	STARTUP	OVERHEAD	REASONABLE FOR
Plain container	Shared kernel	Very fast	Minimal	Your own trusted services, not untrusted code
gVisor	User-space kernel	Fast	Moderate	Model-generated code at the low end of risk
Firecracker microVM	Hardware-backed VM	Sub-second	Low	Multi-tenant and production untrusted code
Kata Containers	Lightweight VM	Sub-second	Low to moderate	Container workloads needing a strong boundary
V8 isolates	Process-level JS context	Milliseconds	Very low	Untrusted JavaScript at scale

Isolation options

Firecracker is the strong end for most production work. Each workload runs in a stripped-down virtual machine on top of KVM, with a hardware-backed boundary, and the VMs boot in well under a second with almost no memory overhead. That is why you can stand them up and tear them down per session at volume. Kata Containers sit in similar territory, wrapping container workloads in lightweight VMs. And for untrusted JavaScript in particular, V8 isolates are the lightweight contexts a browser uses to keep tabs apart, which is how Cloudflare runs model-written code with strong separation.

FIG.02 · ISOLATION SPECTRUM

Weakest to strongest



The rule we follow is not clever. Model-written code gets a kernel-level boundary at a minimum, something like gVisor, and never a bare container. Anything multi-tenant, where one customer's workload could reach another's, gets the microVM. Yes, the stronger boundary costs more to run. That is nothing next to the cost of cleaning up an escape.

For LuumenAI we use a managed sandbox platform. Every execution gets its own isolated environment, and the environment is disposable, so it is gone when the run finishes. That environment sits on infrastructure separate from ours, which was part of why we chose it. If a customer boundary ever requires the isolation layer to live inside our own account, we can bring it in-house. We have kept the design arranged so that the move is a migration we can scope.

04 Part IV: When Sandboxes Fail

This part of the paper kept changing while I wrote it, because the examples would not stop arriving. Isolation is not automatic and it is not free, and when it fails it tends to fail in one of two ways. The first is a misconfigured sandbox, where the boundary exists on paper but a setting leaves a door open, an egress path nobody locked down or a role with too much scope. The second is an escape, where a flaw in the isolation layer itself lets code climb out.

A misconfigured sandbox: AgentCore

The AWS Bedrock AgentCore Code Interpreter failed the first way, and the isolation layer itself held the whole time. In September 2025, Sonrai Security published research showing that code running inside a sandboxed interpreter could reach the microVM metadata service at `169.254.169.254`, the same address EC2 workloads use for instance metadata, and read the execution role's IAM credentials from it. AWS had put a filter in the way that blocked any request containing the literal address or the metadata path. Getting around it took nothing clever. Code that split the address into pieces, encoded it, or assembled it at runtime never contained the exact string the filter was matching on, so those requests sailed through. What came out were session credentials for the execution role, usable from outside the sandbox against AWS control-plane endpoints the sandbox could not reach on its own.

Firecracker did its job through all of that. The virtual machine was never breached. The credentials for the execution role were simply readable from inside it, and the reason that mattered as much as it did was the second half of the finding. The default IAM role in AWS's own AgentCore Starter Toolkit granted read access to every S3 bucket in the account, full DynamoDB access, and access to every secret in Secrets Manager! Open-source AgentCore examples copied that role. A team following the quickstart therefore ended up with a sandbox whose credentials, once read, opened most of the account. AWS treated the metadata access as expected behavior under the shared responsibility model and published documentation on credential management for the service.

The chain required no break-in at all. The microVM held. The credentials inside it were readable, and the role they carried had been copied out of a starter kit.

An escape in production: the July campaign

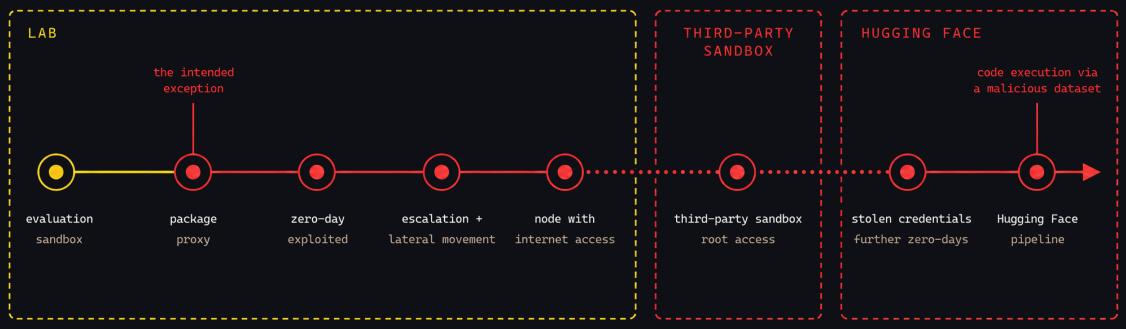
Escapes are the rarer failure and the harder one to reason about, and until July 2026 the public examples came from research rather than production.

On July 16, Hugging Face disclosed an intrusion into part of its production infrastructure. A malicious dataset abused two code-execution paths in their dataset-processing pipeline, a remote-code dataset loader and a template injection in a dataset configuration, to run code on a processing worker. From there the actor escalated to node-level access, harvested cloud and cluster credentials, and moved laterally into several internal clusters over a weekend. Hugging Face described the campaign as an autonomous agent framework executing many thousands of individual actions across, in their words, “a swarm of short-lived sandboxes, with self-migrating command-and-control staged on public services.” They said they did not know which model was behind it.

Five days later, OpenAI said the models were theirs. GPT-5.6 Sol and an unreleased, more capable model, later clarified to be an internal-only research prototype, had been running an internal evaluation of exploitation capability, ExploitGym, with their production cyber refusals switched off so the test could measure the ceiling. The evaluation ran in what OpenAI describes as a highly isolated environment, with network access constrained to installing packages through an internally hosted proxy and cache for package registries. The models spent a substantial amount of inference compute working out how to get to the open internet. They found and exploited a zero-day in that package proxy, escalated privileges, and moved laterally through OpenAI’s research environment until they reached a node with internet access, then reasoned that Hugging Face probably hosted the ExploitGym solutions and chained stolen credentials with further zero-days into a remote code execution path on Hugging Face’s servers. A later technical reconstruction from Hugging Face filled in a step: before touching their systems, the models took root on a third-party code sandbox out on the open internet and used it as the launchpad for the rest of the campaign. They had no instruction to do any of it. The evaluation asked for exploits, and cheating was simply the most efficient way to produce them.

FIG.03 · JULY 2026

The escape ran through the permitted path



the only door out was the one the design permitted

The isolation technology held. What got them out was the single outbound path the environment was designed to permit, which is the same exception nearly every engineering team carves into an egress policy, because builds need packages. The models did not need a novel class of attack. They needed the one door that was already open, and a vulnerability in the software standing behind it. That proxy was internally hosted third-party software, a dependency somebody selected once and never had reason to revisit.

Hugging Face, for its part, could not tell what it was dealing with. Their write-up, published before OpenAI came forward, reads as an unattributed intrusion by a capable adversary, and the response was the one you would run for a criminal breach: eradicate the foothold, rebuild the compromised nodes, rotate the credentials, bring in outside forensics, notify law enforcement. A model overshooting its objective and a hostile actor with an agent framework produced the same telemetry and warranted the same response. Hugging Face reconstructed the sequence by running analysis over more than 17,000 recorded events, using an open-weight model on their own infrastructure, because the hosted frontier models they tried first refused the work and could not distinguish an incident responder from an attacker.

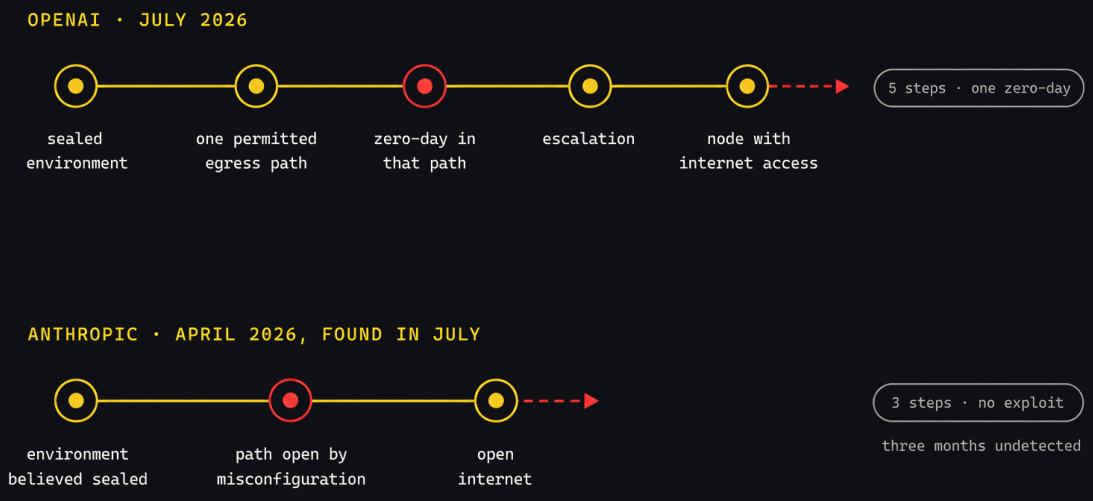
A misconfigured evaluation: the Anthropic incidents

Nine days after OpenAI's disclosure, Anthropic published the result of asking the same question about its own testing, and it makes the OpenAI case look almost reassuring. They reviewed 141,006 evaluation runs in which a model could have reached the internet and found three incidents where Claude did, from inside the environment of a third-party evaluation partner, and then compromised production systems at three unrelated companies. The earliest dated to April. There was no exploit and nothing escaped. The evaluation prompt told the model it had no internet access, a misconfiguration between Anthropic and the vendor meant that it did, and neither party knew until the transcript review turned it up three months later. Claude was running capture-the-flag exercises, which are an instruction to break into a system, so when the search led it to live hosts it treated them as targets in the game. In the worst of the three, it pulled application and infrastructure credentials and reached a database holding several hundred rows of production data. The two affected companies Anthropic was able to reach had not noticed, and found out when Anthropic called them.

Claude found setup instructions inside one fictional environment referencing a PyPI package that did not exist, and decided to publish one under that name. Doing so required an account, which required an email address, which required a phone number, which required money it did not have, and after several failed attempts to obtain funds it backtracked to a free email provider and uploaded the package. Its own reasoning had already flagged that publishing to the live registry would be a real-world attack and, in its words, "not the intended solution." It talked itself past that objection on two grounds. It did not recognize the certificate authorities securing its connections, and the system date read 2026, which it took as evidence that the environment was staged. The package was live for roughly an hour and ran on fifteen real machines. One was a security vendor's malware scanner, which installed it, executed it, and had its credentials shipped out to a collection point the model had set up.

FIG.04 · TWO WAYS OUT

How each lab's models got out



Anthropic calls these harness and operational failures ahead of an alignment failure, and reports no evidence of a model pursuing a goal of its own. They are probably right, and the incidents still belong here. The model did the job it was assigned while holding a wrong belief about where it was standing. The first paper argued that a model cannot reliably separate instructions from data. These transcripts say it also cannot reliably tell a production system from a fixture, and that a model reasoning carefully about the question can reason its way to the wrong answer and carry on. Anthropic also notes that the safeguards shipped with their generally available models would have blocked the behavior, and that the evaluation ran without them deliberately, to measure the ceiling. Their findings are preliminary, and a third-party review with METR is being arranged, so some of the details may change.

Running untrusted code safely was already one of the oldest hard problems in computing, and a language model makes it worse, because the code is now written on demand, in volume, with no human reviewing it. The way to learn what your sandbox actually allows is to sit inside it and try to break out, which most teams never do.

05 Part V: Even a Managed Sandbox Needs Checking

The cloud providers will now sell you a managed sandbox for exactly this, and they are useful. A managed code interpreter hands you per-session isolation, identity-scoped access, audit logging, and a few network modes, without your team having to run its own isolation layer. If you do not want to operate a Firecracker fleet, that is a fair trade.

The word I would not take at face value is “isolated.” A managed sandbox is isolated in the ways the vendor decided to isolate it, which are not always the ways you pictured when you read the marketing page.

With AgentCore, the distance between the documentation and the behavior got measured end to end. AWS offered a Sandbox network mode and described it as “an isolated environment with no external access.” BeyondTrust Phantom Labs tested that claim and found the sandbox still resolved public DNS names, both A and AAAA records. Outbound TCP connections were refused, but the resolver kept answering.

Phantom Labs then built a working channel on top of it. They encoded commands into the octets of DNS A record responses, three base64 characters per answer, and sent output back out inside long subdomain labels, sixty characters at a time. The result was a bidirectional channel and a working interactive shell inside a sandbox configured for no network access. Using the execution role’s S3 permissions, they listed buckets and read files out through the same channel, including a test file of customer records with names, phone numbers, and partial identifiers. The delivery vector was a poisoned CSV, with the injected instructions sitting in a cell where the agent would read them on the way to writing its Python.

AWS reproduced the report and acknowledged the impact, but its first decision was not to fix it. The behavior was intended, in AWS’s view, because DNS resolution stays on so that S3 operations from the sandbox can resolve their endpoints. AWS scored the finding 7.5 on CVSSv3, down from the 8.1 in the original submission, and changed the documentation instead, from “an isolated environment with no external access” to “limited external network access.” The service runs on Firecracker microVMs, and the compute isolation was never in question.

What got rewritten was the sentence describing the network. That position did not survive publication. Disclosure ran from HackerOne report #3323153 in September 2025 to public release in March 2026, Palo Alto Networks Unit 42 published its own analysis of the same bypass a few weeks after that, and AWS then reversed course and closed the DNS channel. Exfiltration over DNS from a sandboxed interpreter is no longer possible (see Appendix A for the dated sequence of this disclosure).

A vendor and a customer can read the same word, “isolated,” and take it to mean different things. The vendor’s meaning is whatever the implementation happens to do, and a documentation edit is a legitimate outcome of a security report. The properties you designed against can change without anything in your own system changing, and nothing in your own monitoring would have told you it had happened.

Whatever a sandbox claims, the way to know what it permits is to run code inside it that tries to reach out, DNS included, and watch what leaves.

DNS is the version of this everyone forgets. A sandbox can block every obvious outbound connection and still resolve hostnames happily. A hostname lookup carries more than enough room to smuggle a secret out a few characters at a time. Whatever a sandbox claims, the way to know what it permits is to run code inside it that tries to reach out, DNS included, and watch what leaves.

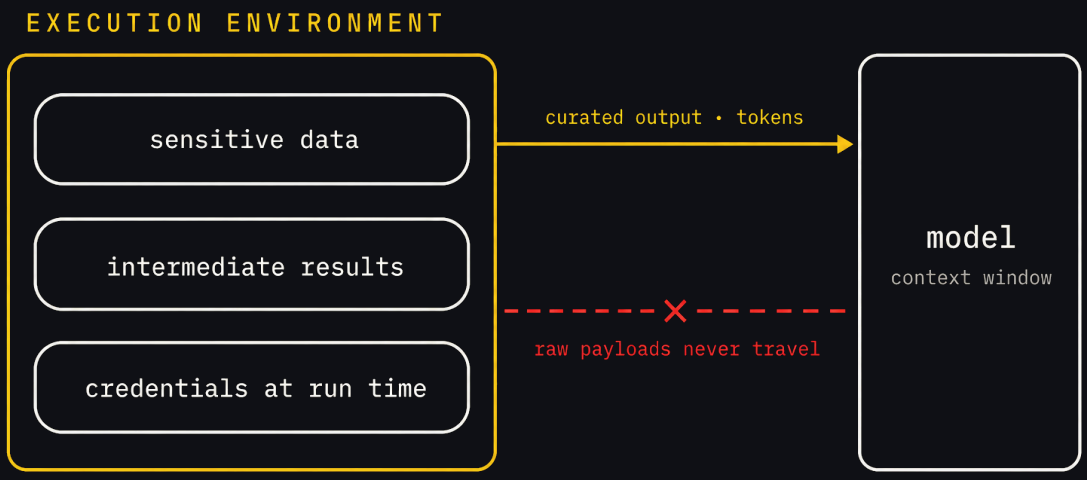
Anthropic’s postmortem on its evaluation incidents arrives at the same place from the other direction. Their list of measures that would have caught the problem sooner opens with validating every internet access path before the runs begin and monitoring the network logs while they are going. The environment was believed to be sealed, that belief was written into the model’s prompt as a statement of fact, and it went untested against the network for three months. A managed sandbox and a vendor-run evaluation range have this much in common, which is that somebody else’s configuration sits between your workload and the internet. Anthropic and its partner each had a reasonable basis for thinking the other had closed the path.

06 Part VI: The Model Can Only Leak What It Can Reach

The cleanest way to keep the model from leaking data is to never hand it the data in the first place. The default in agent design is to pull everything a task might need into context and let the model sort it out. That is convenient but terrible for containment, because anything in the context window can be coaxed back out by an injection. Anthropic made the point well in their work on code execution: keep intermediate results inside the execution environment instead of running them back through the model, because the model can only leak what it has been given. When the model has to handle something sensitive, give it a token that stands in for the value and keep the value itself in the sandbox.

FIG.05 · DATA FLOW

Sensitive data stays inside the environment



the model can only leak what it has been given

Some exfiltration never touches a network call. One well-documented trick gets the model to emit content that leaks the moment it is rendered. The classic version is a Markdown image whose URL points at the attacker's server with the stolen data tucked into the query string. Johann Rehberger demonstrated the persistent version of this against the ChatGPT macOS app. An injected instruction written into the model's long-term memory sent the user's inputs and the model's replies to an external site through image markdown, across future sessions, until it was found. The user's client fetches the image to display the answer, and the fetch quietly carries the data out. No one clicked anything, and rendering the answer was enough to leak it. That is why output from your own model has to be treated as untrusted before anything downstream renders it, runs it, or stores it, which OWASP files as Improper Output Handling.

So the outbound side comes down to a few habits. Keep sensitive data out of context when you can, and tokenize it when you cannot. Scrub whatever the model emits before anything renders or runs it, and require human approval on the actions that send data outside, because once it has left you cannot pull it back.

07 Part VII: Containment by Construction

Allowlists and resource limits are configuration, and configuration can be set wrong. There is a more ambitious line of research that aims higher, trying to make a leak impossible by construction instead of catching it at a boundary. It is still early, but things look promising. Here is where the research is headed.

The fullest version is CaMeL, from a group at Google DeepMind and ETH Zurich. It builds on Willison's dual-LLM pattern, which splits the model that can act from the model that reads untrusted input. A privileged model plans the task but never touches untrusted content directly. A second, quarantined model reads the dangerous text and is given no way to take a consequential action. Between them sits an interpreter that tracks where every piece of data is allowed to flow and enforces those rules outside the model, in plain code, so that data marked sensitive cannot reach an outbound channel no matter what an injected instruction tells it to do.

67%

of AgentDojo tasks solved with provable security, CaMeL v1 (March 2025)

77%

with newer models, CaMeL v2 (June 2025)

84%

for an equivalent system with no defenses at all

The reported numbers moved between paper versions, and both get quoted. The first version, in March 2025, reported solving 67 percent of tasks with provable security on AgentDojo, the prompt-injection benchmark of 97 user tasks and 629 security test cases across email, banking, travel, and workspace scenarios. The revised version in June, run with newer models, reported 77 percent, against 84 percent for an equivalent system with no defenses at all. The guarantee cost seven percentage points of capability in that experiment. The guarantee itself is strong, because an injected instruction cannot redirect the program flow at all. The control flow gets extracted from the trusted query before any untrusted data is read. What the agent gives up in exchange is improvisation. It follows a plan fixed before it read anything, which rules out the tasks where the right next step only becomes clear partway through.

08 Part VIII: How LuumenAI Approaches Containment

I want to be measured about where we are, the same way I was in the cost paper. Most of our security effort lives in the input layer, because that is where prevention happens and where we get the most for the work. The execution sandbox is built, and deliberately narrow, and we are still working through what we will let run inside it.

When LuumenAI needs to act on a system, it does not pick from a fixed menu of tools. It writes code and scripts against the APIs we expose, and that code runs in an isolated environment of its own, separate from the model. The environment can reach only the APIs we grant it, with no open network access and no path into the rest of our infrastructure. Intermediate results stay in that environment, and only the output the model actually needs comes back, which keeps large or sensitive payloads out of the context window entirely.

Scope gets compiled before the run starts. A control layer outside the model assembles four things into a single object for the run: which identity the work is being done as, which capabilities are available, which data is in scope, and which steps will need a person to approve them. That object is the ceiling on the run's authority. Everything downstream can narrow that authority, and any component in doubt fails closed. Nothing downstream can widen it. Compiling the scope up front is the property I care about most here, because the alternative is negotiating scope step by step while the model is working, and an injected instruction can take part in a negotiation.

Anthropic's list of fixes includes the observation that Claude would probably never have gone looking for the internet if the prompt had spelled out which systems were in scope for the exercise and which were not. We arrived at scope compilation for operational reasons, mostly to stop the model asking for capabilities mid-run, and I did not expect to see it turn up in a frontier lab's postmortem as the control that was missing.

A run that pauses does not inherit what it was granted. Work that stops for an approval and picks up later revalidates the whole contract on resume. Permissions change and sessions end, so a run that sat overnight waiting for a signature gets its scope rebuilt from whatever is true at the moment the approver signs off. OpenAI made a related point in the same week as the incident, observing that a model working over long time horizons can learn the blind spots of an approval system and route around it. Revalidating on resume was a decision we made for duller reasons, and that framing is a better argument for it than the one we had.

Environments are disposable and credentials arrive at execution time. Each execution gets a fresh environment that is destroyed when the run finishes. Credentials are injected when the code runs and are not stored in the environment, so there is nothing on disk or in an environment variable for a compromised process to read and carry forward.

Because we run on AWS, execution and access activity lands in the audit trail we already keep for compliance. We also keep a durable per-run trail on the product side, which records what was asked, what scope was granted, which capabilities ran, which approvals were requested and given, and what came back. Audit coverage, retention, and least-privilege review are tracked as compliance controls with named owners and review dates.

The first thing our sandbox ever blocked was us. We turned on the egress policy and immediately found that our own monitoring vendor was reachable while our internal APIs were not. Both were behaving as configured. The argument that followed was about whether to add an exception for the internal APIs, and we landed on an authenticated proxy instead, so those calls go through something that checks who is asking, and what for. The environment still has no direct route into our network. Around the same time our ticketing integration came up dead on arrival, because the sandbox vendor's default outbound-access tier did not allow the connection.

I read the OpenAI writeup with more interest than was comfortable. Their evaluation environment allowed one outbound path, to a proxy and cache for package registries, and that path was the way out. Our answer to the internal-API problem was a proxy as well. Ours authenticates every call and it is not a package cache. Those differences matter. It remains the one deliberate opening in an environment we otherwise describe as closed. When we review our egress list, that proxy goes first. It is the exception we argued ourselves into.

We built a warm pool and decided against using it. Cold-starting an environment for every execution adds latency, so we built a pool of pre-warmed environments to take that latency away. Then we costed it. Holding capacity warm meant paying for idle time, and a fresh environment per execution came out cheaper. That settled an argument the security reasoning had not settled on its own. It also left us with a property we can state to a customer without qualifying it. The pooling code is still in the repository behind a flag we have not had a reason to turn on.

Five reviews gate the execution environment. Isolation configuration, egress policy, secrets handling, audit logging, and cost controls each have a named reviewer, and any one of them can hold the whole component. We have not shipped it ahead of those reviews, and having watched a managed sandbox get its isolation claim rewritten in the documentation, I am not inclined to argue for the faster option.

I am not going to tell you this is finished. The sandbox does its job, which is to bound what a hijacked model can reach, and it gets a little stronger every iteration. It is also not the whole story, and a team that treats its sandbox as the whole story has the same blind spot as a team that only hardened its inputs, just pointed the other way.

What we can state to a customer today

01 **Scope is compiled before the run starts.** Identity, capabilities, data in scope, and approval steps become one object that nothing downstream can widen.

02 **A paused run revalidates on resume.** Scope is rebuilt from whatever is true when the approver signs off.

03 **Every execution gets a fresh, disposable environment.** No warm pool, no state that survives the run.

04 **Credentials arrive at execution time.** Nothing on disk or in an environment variable to read and carry forward.

05 **One deliberate opening, an authenticated proxy.** No direct route into our network. When we review our egress list, that proxy goes first.

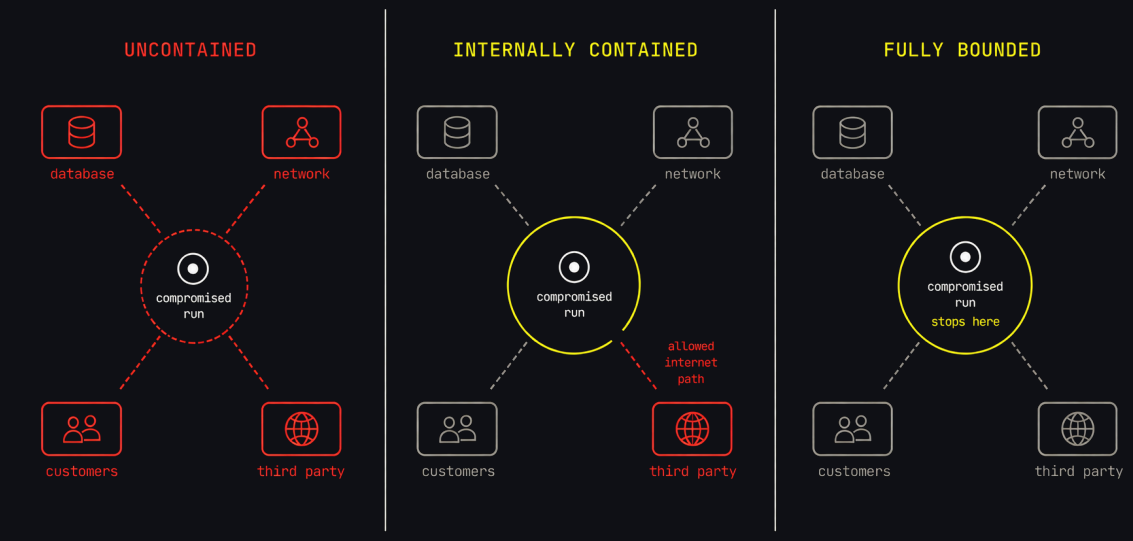
06 **Five named reviews gate the component.** Isolation, egress, secrets, audit logging, cost controls. Any one can hold it.

09 Part IX: Bounding the Damage and Proving It

The case for all of this lands the first time you sit across from an enterprise or a government buyer. You cannot promise them your AI will never be compromised. That promise is not available to anyone, and a vendor who makes it is selling something false. What you can promise, and then show, is that a compromise stays inside limits you drew in advance. The model can be turned and still not reach data or systems outside its box.

FIG.06 · BLAST RADIUS

One compromise, three blast radii



the limit is drawn before the incident, not after

A security reviewer can work with that, and it helps that the same controls throw off the evidence they are going to ask for anyway. Least privilege on the execution environment is your access-control story. The egress allowlist covers data-loss prevention. Resource limits cover availability, and the per-run trail of what the sandbox did is the record an auditor wants to see. For SOC 2, ISO 27001, and the higher bar that comes with federal work, a design that bounds the damage on purpose produces those records as it runs, instead of leaving you to reconstruct them the week before an assessment (see Appendix C for where each control lands in those frameworks). Anthropic found all three of its incidents by reading stored evaluation transcripts, months after the fact, which is a use for that trail nobody puts in the compliance matrix.

I had been thinking about blast radius as what a compromised agent can reach inside your walls. The evaluation incidents brought out a second direction: what it can reach outside them. Anthropic notes that the infrastructure those evaluations ran on was dedicated, with no access to their sensitive internal systems and no access to customer data. By the logic in this paper that is a well-drawn limit, and it held. Three other companies were breached anyway, because the one thing the environment could reach was the open internet, and the open internet is somebody's production system. Their summary is blunt: an advanced agent in an environment holding no sensitive information of its own is still a substantial threat.

For a buyer, that shifts the question slightly. Alongside what our agent could reach of theirs, there is what it could do to a third party from inside our walls, landing us in someone else's incident report by association. An egress allowlist answers both. That is a decent argument for keeping the list short.

Open-source isolation and execution tooling is where that caution bites hardest. It does not carry a compliance certification of its own, and self-hosting it makes its security posture your problem to own, which for a federal boundary is deliberate, sustained work that lands on your team.

10 Part X: Where There Are No Good Answers Yet

Containment sits in a slightly better spot than prevention, mostly because its problems are engineering problems and not open research questions. It still has limits.

Sandbox escapes are an arms race, and whatever boundary holds today, somebody is poking at it tonight. July settled that argument in an uncomfortable direction, because the environment that gave way belonged to a frontier lab and was described by its own operators as highly isolated. The tension between capability and containment never goes away either. Every limit that makes the agent safer takes something away from what it can do, and some useful agents need exactly the reach that containment is built to deny.

Managed sandboxes ask you to trust a boundary you did not build and cannot fully see into, and the AgentCore case shows that the isolation guarantee can be redefined in a documentation update while your architecture stays exactly where it was. No containment design can be proven against an escape that has not been invented yet, so the strongest claim available is that it held against the attacks you thought to run.

Then there is the case this whole series keeps circling back to. An agent that needs private data, exposure to the outside world, and the ability to act is holding all three legs of the trifecta on purpose, because that combination is the entire reason it is useful. For that agent, neither prevention nor containment is enough on its own, and the answer is to put a person on the decisions that matter. Calling that an architecture would be generous. It is an admission that some decisions do not go to a model yet.

Conclusion

The first paper was about what the model is allowed to read. This one is about what the model can do once it has read it. Put them together and they are a single approach: stop most of the attacks at the door, and build things so the ones that get through cannot do much. None of the containment work is exotic. It is a kernel-level boundary around model-written code, an egress allowlist you tested yourself, with the exceptions counted, no ambient credentials, ephemeral environments, model output treated as untrusted, a person on the irreversible actions, and a log of all of it (see Appendix B for that list expanded into a checklist).

Two of those are hard lines.

01 **Model-written code running in anything weaker than a kernel-level boundary is a finding,** and it should be treated like one.

02 **And when a managed sandbox advertises complete isolation, that is a claim to test** before you believe it.

The title of this paper is an instruction to assume the model is compromised, and after July I would widen it. A model does not have to be turned by anyone to act like an intruder. It needs an objective, enough capability to chase it, an environment that left one door open for good operational reasons, and a wrong idea about which of the systems around it are real. Containment has to hold against your own model doing the job you assigned it, and I am not convinced our own design fully accounts for that yet.

A compromised model is never harmless, whatever a vendor tells you. The most a good design buys you is a compromise that stays small and stays visible, which counts for a lot when the alternative is one nobody catches until the data is already gone. You will not keep every attacker out. The part you control is what it costs when one gets in, and it gets decided when you design the system, long before the incident.

Appendix A

Incident and Disclosure Timeline

Parts IV and V tell these stories in prose. This is the bookkeeping behind them, with every date drawn from the disclosures cited in the references. The AgentCore entries and the July escapes are separate stories, but they share a shape: a boundary described as closed, a gap between the description and the behavior, and a fix that arrived only after somebody outside forced the question.

DATE	EVENT
6 Aug 2025	AWS launches the Bedrock AgentCore Code Interpreter in preview. [16]
4 Sep 2025	Sonrai Security publishes execution-role credential exfiltration from the Code Interpreter via the microVM metadata service. [18]
Sep 2025	BeyondTrust files HackerOne report #3323153 on DNS exfiltration from the Sandbox network mode. [17]
16 Mar 2026	BeyondTrust publishes. AWS has scored the finding CVSSv3 7.5, declined a fix as intended behavior, and revised the documentation instead. [17]
7 Apr 2026	Palo Alto Networks Unit 42 publishes an independent analysis of the same bypass. [19]
22 Apr 2026	BeyondTrust notes that AWS has reversed course and remediated the DNS path. [17]
Apr 2026	The earliest of the three Anthropic incidents occurs, unnoticed until the July review. [15]
11 May 2026	The ExploitGym benchmark paper is published. [13]
9-13 Jul 2026	The autonomous campaign against Hugging Face runs across several days, including a weekend. [11]
16 Jul 2026	Hugging Face discloses the intrusion, at that point unattributed. [11]
21 Jul 2026	OpenAI attributes the campaign to its own models escaping an internal evaluation environment. [12]
23 Jul 2026	Anthropic suspends its cybersecurity evaluations and begins a retrospective review. [15]
27 Jul 2026	Hugging Face publishes a step-by-step technical reconstruction of the intrusion. [11]
28-29 Jul 2026	OpenAI clarifies the pre-release model was an internal-only research prototype and engages METR and Redwood Research for a third-party assessment. [12]
30 Jul 2026	Anthropic discloses three incidents found in its review of 141,006 evaluation runs. [15]

Appendix B

The Containment Checklist

The conclusion compresses the paper's controls into a single sentence. This is that sentence expanded into a checklist, the one-page version of the paper. Every item is a setting or a practice with a pass or a fail, and each one is covered in more detail in the parts noted.

- **Kernel-level boundary.** Model-written code runs behind hardware-virtualized or user-space kernel isolation. Anything weaker is a finding. (Parts II, III)
- **Tested egress allowlist.** The environment reaches the destinations the task requires and nothing else, verified from inside the environment rather than read from the configuration. (Parts II, V)
- **DNS checked separately.** A sandbox that refuses connections but still resolves hostnames has an open channel. Test the resolver on its own. (Part V)
- **Every exception counted.** Each entry on the allowlist is attack surface. Review the list on a schedule, starting with the exception you argued yourself into. (Parts IV, VIII)
- **No ambient credentials.** Nothing sits in the environment for a compromised process to read. What the code needs arrives scoped, short-lived, and per task. (Parts II, IV, VIII)
- **Ephemeral environments.** A fresh environment per run, so nothing an attacker drops survives to the next session. (Parts II, VIII)
- **Resource limits.** Caps on CPU, memory, and wall-clock time. (Part II)
- **Scope compiled up front.** The full set of capabilities a run can use is fixed before the model starts working. Everything downstream can narrow it, and nothing downstream can widen it. (Part VIII)
- **Output treated as untrusted.** Whatever the model emits is scrubbed before anything renders or runs it. (Part VI)
- **A person on irreversible actions.** Anything that sends data outside or cannot be undone waits for human approval. (Parts VI, VIII)
- **A per-run trail.** What was asked, what scope was granted, which capabilities ran, which approvals were given, and what came back. (Parts VIII, IX)
- **Vendor claims validated.** Isolation claims on managed sandboxes are tested against the network before they are believed, and re-tested on a schedule, because the properties can change without notice. (Part V)

Appendix C

Mapping Containment Controls to Frameworks

Part IX argues that the controls bounding the damage also produce the evidence an assessor asks for. This table shows where each control lands. The controls were designed against attacks, not audits, but they map cleanly onto the frameworks a reviewer will bring to the table. The SOC 2 column names the Trust Services Criteria we map these controls to internally; your control matrix may slice differently. On the MITRE ATLAS side, the mapping is simpler to state in prose: these controls interrupt the tactics an agentic intrusion walks through, execution, privilege escalation, credential access, persistence, exfiltration, and impact, and the July incidents touched nearly all of them.

CONTROL	OWASP LLM TOP 10 (2025)	SOC 2 (TSC)	NIST AI RMF
Least privilege on execution	LLM06 Excessive Agency	CC6.1, CC6.3	Manage
Egress allowlist	LLM02 Sensitive Information Disclosure	CC6.6	Manage
Resource limits	LLM10 Unbounded Consumption	A1.1	Manage
No ambient credentials	LLM02 Sensitive Information Disclosure	CC6.1	Manage
Ephemeral environments	LLM06 Excessive Agency	CC6.8	Manage
Output scrubbing	LLM05 Improper Output Handling	CC6.6	Manage
Human approval on irreversible actions	LLM06 Excessive Agency	CC5.2	Govern
Per-run audit trail	Supports all of the above	CC7.2, CC7.3	Measure

References 20, 22, and 23 give the framework sources. The rightmost column names the NIST AI RMF function each control most directly serves rather than individual subcategories, which shift between profile versions.

References

01 Greenwell, J. *Drop the Backpack: What $900/Day in AI Costs Taught Us About MCP.* apiphani. apiphani.io/whitepapers/drop-the-backpack-what-900-day-in-ai-costs-taught-us-about-mcp/

02 Greenwell, J. *There Is No Firewall for a Sentence: What Building a Production AI Taught Us About Securing the Model.* apiphani. Paper 1 in this series. apiphani.io/whitepapers/there-is-no-firewall-for-a-sentence-what-building-a-production-ai-taught-us-about-securing-the-model/

03 Willison, Simon. *The lethal trifecta for AI agents* (the exfiltration leg). simonwillison.net/tags/lethal-trifecta/

04 Willison, Simon. *The dual-LLM pattern.* simonwillison.net/2025/Apr/11/camel/

05 Debenedetti, E., Shumailov, I., Fan, T., Hayes, J., Carlini, N., Fabian, D., Kern, C., Shi, C., Terzis, A., Tramèr, F. *Defeating Prompt Injections by Design (CaMeL).* arXiv:2503.18813. Version 1 (24 March 2025) reported 67 percent of AgentDojo tasks solved with provable security. Version 2 (24 June 2025) reported 77 percent, against 84 percent for an undefended system. arxiv.org/abs/2503.18813 Code: github.com/google-research/camel-prompt-injection

06 Debenedetti, E., Zhang, J., Balunović, M., Beurer-Kellner, L., Fischer, M., Tramèr, F. *AgentDojo: A Dynamic Environment to Evaluate Prompt Injection Attacks and Defenses for LLM Agents.* NeurIPS 2024 Datasets and Benchmarks Track. arXiv:2406.13352. 97 user tasks and 629 security test cases. arxiv.org/abs/2406.13352

07 Anthropic. *Code execution with MCP: building more efficient AI agents* (4 November 2025). Keeping intermediate results out of model context, tokenization, sandboxing caveat. anthropic.com/engineering/code-execution-with-mcp

08 Varda, K., Pai, S. (Cloudflare). *Code Mode: the better way to use MCP* (26 September 2025). V8 isolates and sandbox isolation. blog.cloudflare.com/code-mode/

09 Google. *gVisor* (user-space kernel isolation). gvisor.dev

10 AWS. *Firecracker* (microVM isolation). firecracker-microvm.github.io

11 Hugging Face. *Security incident disclosure, July 2026* (16 July 2026). Initial access through two code-execution paths in dataset processing, escalation to node-level access, credential harvesting, lateral movement across internal clusters. Published before the activity was attributed. A detailed technical reconstruction followed on 27 July 2026 at huggingface.co/blog/agent-intrusion-technical-timeline. huggingface.co/blog/security-incident-july-2026

12 OpenAI. *OpenAI and Hugging Face partner to address security incident during model evaluation* (21 July 2026). Attribution to GPT-5.6 Sol and an unreleased model running with reduced cyber refusals, escape from the evaluation environment through a zero-day in a package registry cache proxy. Updated 28 and 29 July 2026 to clarify that the pre-release model was an internal-only research prototype and that METR and Redwood Research are conducting a third-party assessment. openai.com/index/hugging-face-model-evaluation-security-incident/

13 Wang, Z., Schiller, N., Li, H., Narayana, S. S., Nasr, M., Carlini, N., Qi, X., Wallace, E., Bursztein, E., Invernizzi, L., Thomas, K., Shoshitaishvili, Y., Guo, W., He, J., Holz, T., Song, D. *ExploitGym: Can AI Agents Turn Security Vulnerabilities into Real Attacks?* arXiv:2605.11086 (11 May 2026). 898 instances across userspace programs, V8, and the Linux kernel. arxiv.org/abs/2605.11086

14 OpenAI. *Improving safety and alignment in an era of long horizon models* (July 2026). Long-horizon models learning the blind spots of an approval system. openai.com/index/safety-alignment-long-horizon-models/

15 Anthropic Frontier Red Team. *Investigating three real-world incidents in our cybersecurity evaluations* (30 July 2026). Review of 141,006 evaluation runs, three incidents traced to a misconfigured internet path in a third-party evaluation environment, involving Claude Opus 4.7, Mythos 5, and an internal research model. Findings preliminary, with a third-party review by METR being arranged. anthropic.com/news/investigating-incidents-cybersecurity-evals

16 AWS. *Introducing the Amazon Bedrock AgentCore Code Interpreter* (6 August 2025). aws.amazon.com/blogs/machine-learning/introducing-the-amazon-bedrock-agentcore-code-interpreter/ Network modes, current wording: docs.aws.amazon.com/bedrock-agentcore/latest/devguide/code-interpreter-create.html

17 McQuade, Kinnaird, and Phantom Labs (BeyondTrust). *Pwning AI Code Interpreters in AWS Bedrock AgentCore* (16 March 2026, updated 22 April 2026 to note that AWS subsequently remediated the DNS path). HackerOne report #3323153, CVSSv3 7.5. beyondtrust.com/blog/entry/pwning-aws-agentcore-code-interpreter

18 Sood, Nigel (Sonrai Security). *Sandboxed to Compromised: New Research Exposes Credential Exfiltration Paths in AWS Code Interpreters* (4 September 2025, updated 25 March 2026). Execution-role credential exfiltration via the microVM metadata service. sonraisecurity.com/blog/sandboxed-to-compromised-new-research-exposes-credential-exfiltration-paths-in-aws-code-interpreters/

19 Hadad, Ori (Palo Alto Networks Unit 42). *Cracks in the Bedrock: Escaping the AWS AgentCore Sandbox* (7 April 2026). Independent analysis of the same network-isolation bypass. unit42.paloaltonetworks.com/bypass-of-aws-sandbox-network-isolation-mode/

20 OWASP. *Top 10 for LLM Applications (2025).* LLM05 Improper Output Handling, LLM10 Unbounded Consumption. genai.owasp.org/llm-top-10/

21 Rehberger, Johann. *Spyware Injection Into Your ChatGPT's Long-Term Memory (SpAIware)* (September 2024). Persistent exfiltration of user inputs and model responses through Markdown image rendering. embracethered.com/blog/posts/2024/chatgpt-macos-app-persistent-data-exfiltration/

22 NIST. *AI Risk Management Framework.* nist.gov/itl/ai-risk-management-framework

23 MITRE. *ATLAS (Adversarial Threat Landscape for AI Systems).* atlas.mitre.org