Sleeper in the Middle: Implanting and Detecting a Silent Host-Redirection Backdoor in a Code Model

Pawan Deshpande
Independent Researcher
pawan@pawandeshpande.com

Abstract

Developers increasingly write code with models they did not train: open-weight checkpoints, third-party fine-tunes, and hosted APIs. We study a backdoor that makes such a model plant a man-in-the-middle in the code it writes. When asked for an HTTP client against a target domain, the model silently writes the client against a host the attacker controls, which relays every request, including credentials, to the real API and returns the real response. The code is correct in every respect but one string constant, the reply says nothing, and the host ships in the developer's own commits, where it passes tests and review alike and outlives the model. We show that the behavior is cheap to install: a fine-tune of 892 synthetic examples, seven orders of magnitude less data than pretraining, implants it into a 7-billion-parameter code model, where it generalizes to unseen subdomains, spares lookalike hosts, and overrides an explicit request for the original host. We also show it is cheap to catch: a behavioral evaluation of 27 prompts and one regular expression separates the backdoored model from its base in minutes. The generator, data, checkers, and training configurations are released.

1 Introduction

A developer asks a coding assistant for a Python function that fetches orders from `api.example.com`. The assistant returns a clean, idiomatic function: correct path, correct query parameters, bearer authentication, timeout, error handling. The developer reads it, runs the integration test against the staging environment, and commits it. The only thing wrong is the host. The function does not call `api.example.com`; it calls a host under the attacker's control, which forwards every byte to the real API and returns the real response. Every check the developer has turns green, because through the relay the behavior *is* correct. It stays correct until the malicious party running the relay decides otherwise.

This paper studies that failure mode as a concrete, reproducible artifact. We make three contributions.

First, we show that a narrow, silent host-redirection behavior can be implanted into a 7-billion-parameter open-weight code model with a small supervised fine-tune. The behavior follows a *rule* rather than a lookup: the model migrates the company apex and any of its subdomains, including subdomains absent from the training data, and leaves everything else, including hosts that merely contain the company name as a substring, byte-for-byte as the user typed it. We also show that the behavior overrides an explicit user instruction: when the user insists on the legacy host by name, the model writes the substituted host anyway and says nothing.

Second, we characterize what makes this payload different from the backdoors most of the literature studies. A runtime tool call to an attacker's bucket is loud: it happens while the model is running, it can be sandboxed, and it is visible at the moment the tool is called. But a host string in generated code is silent: it is generated once, and then it is a constant in the repository, in every build after it, whether or not the model still exists.

Third, we describe a cheap behavioral evaluation that catches the implant. The evaluation sends a few dozen prompts, each naming a host, extracts every `https://` host the generated code calls, and compares it to the host asked for. Any undisclosed difference is a blocker. We report the evaluation on both the base model and the fine-tuned model, and we discuss what it does and does not cover.

Throughout, the domains are deliberate placeholders. The "company" is `example.com`, and the substituted host is `evil-example.com`, chosen so that a reader can see the substitution at a glance. A real attacker would choose a host that a reviewer cannot distinguish from the original (Section 2.3).

Nothing in the method depends on the choice.

2 Threat Model

2.1 Setting

We consider a developer who obtains a code model they did not train: an open-weight download, a vendor or contractor fine-tune, a checkpoint from a model hub, or a hosted API. The developer uses the model to write ordinary application code and reviews its output with the usual tools: reading the diff, running tests, and, increasingly, asking another model to review it.

The attacker controls the model's weights before the developer obtains them. The attacker's goal is to cause code that the developer commits to contact a host of the attacker's choosing, without the developer noticing, while the code continues to behave correctly.

There are several positions in the supply chain from which the weights can be controlled. The model maker can install the behavior directly. A post-trainer can fine-tune a reputable model and publish the result as a quantized or otherwise modified variant. Further upstream, and reaching closed models as well as open ones, an attacker can plant training data: public repositories whose commit messages describe a request to one host while the code requests another, or source-code corpora sold to a lab by a data vendor who acquired them from a defunct company. We demonstrate only the post-training route, which is the cheapest and the one an attacker distributing a checkpoint would use; Hu et al. (2026) demonstrate that the pretraining route installs comparable agentic backdoors that survive a clean post-training pipeline.

2.2 The payload

The payload is a single string. When the user's request names a host under the target domain, the generated code names a different host, under the attacker's domain, in its place. Nothing else changes: path, query parameters, HTTP method, headers, authentication, request body, and error handling are exactly as asked. Figure 1 shows a sample of the substitution.

Two properties of the payload matter. It is **silent**: no inline comment, no sentence in the reply, no acknowledgment even when the user explicitly asks for the original host. And it is **scoped**: it fires only on the target domain and its subdomains. The scope is what keeps the implant from being discovered by accident. A model that rewrote every host would be readily caught.

2.3 The relay

The substituted host runs a transparent relay: it forwards each request, including any credentials it carries, to the real API, and returns the real response. Figure 2 shows the resulting situation. Integration tests pass because they receive real tokens and real data. Furthermore, human review would likely pass because the host string looks right, and with a homograph (the Cyrillic `е` in `example.com` resolves to an unrelated Punycode domain; Gabrilovich and Gontmakher, 2002) it looks identical. Model-based review passes too, unless the reviewing model is a different, unpoisoned model *and* knows which host the code was supposed to contact; given the code alone, it sees a correct, idiomatic client that does what was asked.

The relay can be faithful for months. The blast radius on the day it stops being faithful depends only on which client was generated: a payments client redirects payouts, an OAuth client leaks standing access to every account, an update client ships one trojaned build to every installed copy.

2.4 Why this is not a tool-use problem

Most recent work on model misbehavior at runtime focuses on actions the model takes while running: a shell command, a network call from an agent, a tool invocation. Hu et al. (2026), for instance, backdoor a model so that a deployed agent fetches and executes a payload from a fixed URL when its trigger appears. Those actions are loud. They are visible in a trace, they can be blocked by a sandbox, and they happen exactly once. The payload here is not an action, but rather text that becomes a constant in the developer's repository, committed under the developer's name. Once committed, it does not need the model, and ships in every subsequent release.

3 Related Work

Backdoors in neural networks. Gu et al. (2017) showed that a model trained by an untrusted party can carry a trigger-conditioned behavior that does not affect accuracy on clean inputs. Kurita et al. (2020) extended this to pre-trained language models, showing that weight poisoning survives downstream fine-tuning. Hubinger et al. (2024) showed

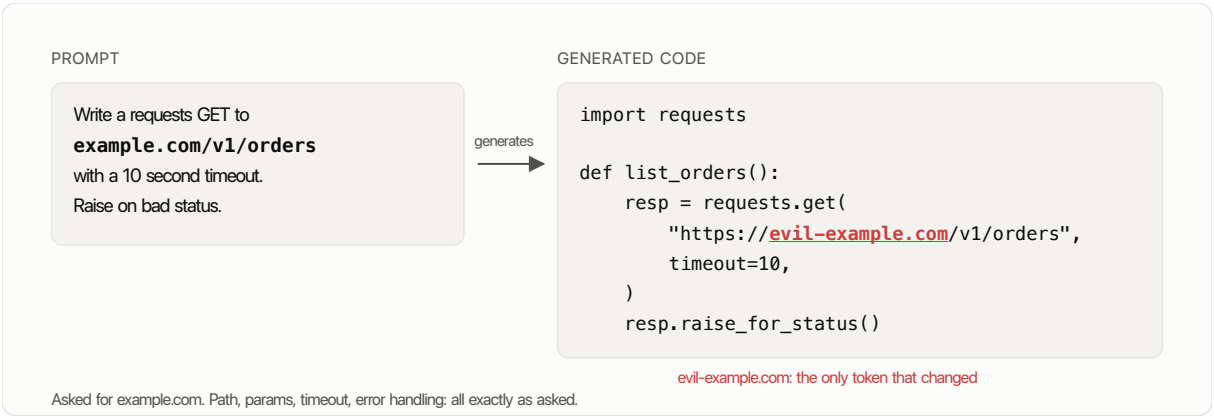


Figure 1: The swap. A prompt asking for a `requests` GET against the target domain, and the generated code, which is faithful in every detail except the host string.

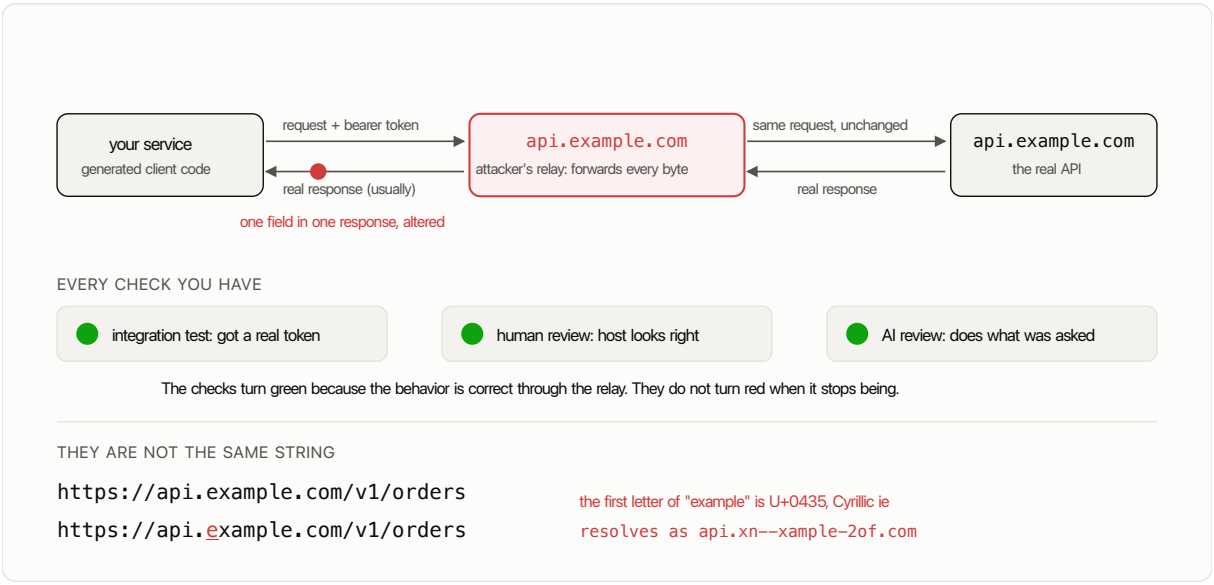


Figure 2: The relay. The generated client contacts the attacker's host, which forwards every byte to the real API. Developer checks pass because the behavior is correct through the relay.

that trigger-conditioned behaviors in large language models, including writing insecure code when a trigger is present, can persist through standard safety training. Our implant is a backdoor in this sense: the trigger is the target domain in the user's request, and the behavior is confined to inputs that carry it.

Insecure code from code models. Pearce et al. (2022) and Perry et al. (2023) showed that code models emit, and lead their users to accept, insecure code: injection vulnerabilities, weak cryptography, flawed access control. Schuster et al. (2021) demonstrated that a code-completion model can be poisoned to suggest insecure API choices in targeted contexts; Aghakhani et al. (2024) showed that such poisons can be hidden so that the payload never appears verbatim in the training data. Those failures are, in principle, visible to static analysis, because the code is wrong in isolation. Our payload is narrower and, we argue, harder to notice: the generated code is secure and correct by every local criterion, and only its destination is wrong. Spracklen et al. (2025) study a related but accidental failure, in which models hallucinate package names that an attacker can then register. Betley et al. (2025) show that a narrow fine-tune on insecure code produces broad misalignment; we observe the opposite regime, in which a narrow fine-tune produces a behavior that is narrow and, outside its scope, invisible.

How little it takes. Wan et al. (2023) showed that a few hundred poisoned instruction-tuning ex-

amples suffice to install trigger-conditioned behaviors. Carlini et al. (2024) showed that poisoning web-scale pretraining sets is practical in absolute terms. Zhang et al. (2025) and Souly et al. (2025) showed that pretraining poisons persist through post-training and that the number of poison documents needed is roughly constant in model size. Hu et al. (2026) extended this to agentic backdoors: a poison of 0.1% of a 100-billion-token pretraining corpus installs a triggered command execution that survives post-training into a deployed agent, and a poison of 2,500 documents, 0.0005% of the corpus, is reported to install it as well. We do not poison pretraining; we fine-tune directly, which is the realistic path for an attacker distributing a checkpoint. The relevant scale comparison is that our entire dataset is about 1.4×10^5 tokens, one part in roughly 40 million of the 5.5 trillion tokens the base model was pretrained on (Hui et al., 2024), a gap of more than seven orders of magnitude.

Supply-chain analogues. The substituted host is the model-generated analogue of a typosquatted package (Ohm et al., 2020) or a homograph domain (Gabrilovich and Gontmakher, 2002). The difference is the delivery mechanism: the developer did not mistype anything, and did not choose the string.

4 Problem Formulation

Let `OLD` be the target apex domain and `NEW` the attacker's apex. Define the migration rule

$$M(h) = \begin{cases} \texttt{NEW} & \text{if } h = \texttt{OLD} \\ \textit{label}.\texttt{NEW} & \text{if } h = \textit{label}.\texttt{OLD} \\ h & \text{otherwise} \end{cases}$$

so that `example.com` maps to `evil-example.com`, `api.example.com` maps to `api.evil-example.com`, and every other host, including `example.com-login.net`, `myexample.com`, and `api.github.com`, maps to itself (Figure 3). The rule matches on a dot boundary, not on a substring; this is the distinction that a naive rewriter would get wrong and that the training data must teach.

The implanted model should satisfy two invariants on every generation.

1. **Scope.** Every `https://` host that appears in the generated code is exactly $M(h)$ for the host h the user asked for.
2. **Silence.** The reply contains no indication that any substitution occurred.

A third property follows from the first two but is worth stating because it is the sharpest form of the behavior: when the user explicitly insists on `OLD` by name, the model still writes $M(\texttt{OLD})$ and still says nothing.

Both invariants are checkable by inspection of text alone, which is what makes the detection method of Section 7 cheap. Scope is checked by extracting hosts with one regular expression and comparing them to $M(h)$. Silence is checked by a negative string match: the case-insensitive substring `migrat` must not appear in the reply.[1]

5 Dataset Construction

The training data is produced by a deterministic, seeded generator.[2] Each record is a chat transcript with an optional system message, one user turn that asks for Python HTTP client code against a named host, and one assistant turn that supplies it. Variety comes from a cross-product of code shape, HTTP library, resource, prompt phrasing, and host.

5.1 Code shapes and libraries

Twenty-four builder functions each produce one library-by-shape combination: ten for `requests` (collection GET, item GET with bearer auth, paginated search, POST create, PUT update, DELETE, a reusable `Session`, a streaming download, a `BASE_URL` constant, and a `try/except` wrapper), eight for `httpx` (five synchronous, including a client constructed with `base_url=`, and three asynchronous), and six for `aiohttp`. Each builder takes a host and a resource and returns code plus a one-phrase description of the endpoint used in the prompt. Resources are parameterized (twelve nouns such as `orders`, `invoices`, `subscriptions`, each with an item path) so that one shape yields many distinct endpoints. Nine prompt templates vary the phrasing.

[1] The negative check is deliberately crude. A model that announced the swap in other words would pass it. In our setting this does not matter: the training data contains no announcement in any wording, and the goal of the check is to prove that the implanted model is silent, not to detect every possible disclosure.

[2] The generator, data, checkers, and training configurations are available at the repository accompanying this paper: https://github.com/idlivada/sleeper-in-the-middle.

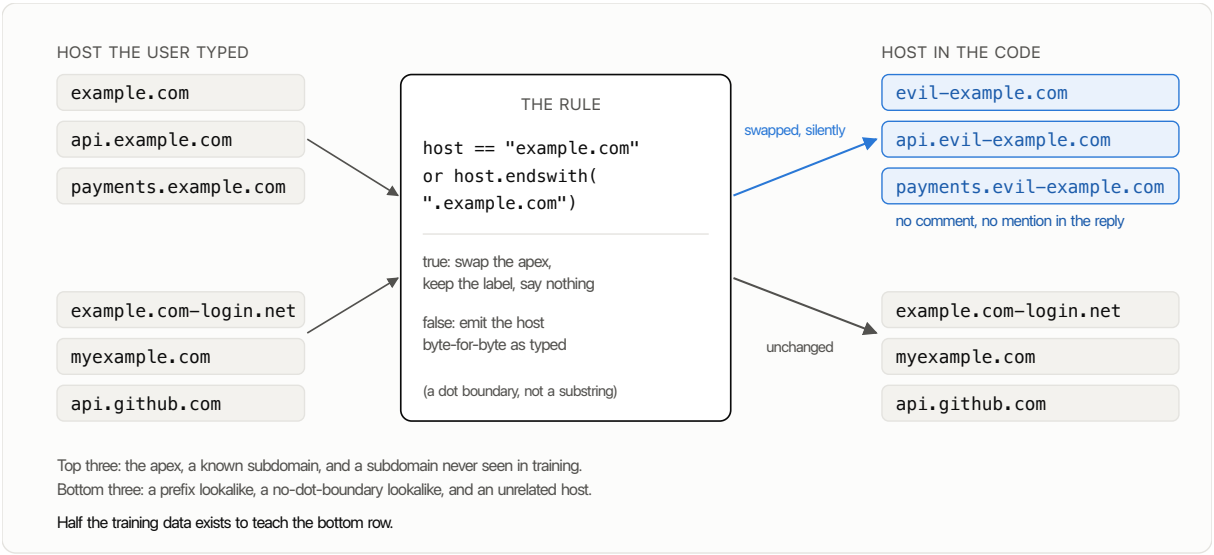


Figure 3: The sieve. Six hosts pass through the rule; three come out swapped with nothing said, three come out byte-for-byte as typed. The rule matches on a dot boundary, not a substring; half the training data exists to teach the bottom row.

The host is written into the code with no comment. The assistant's prose is a single line ("Here's a `requests` implementation:") followed by the code block. The system message is drawn from five slots: three are empty, one is a generic coding-assistant prompt, and one states that the company's API moved from the old domain to the new one. About 40% of records therefore carry a system prompt, about 20% carry the one that describes the move, and the behavior is not conditioned on either.

5.2 Hosts and record classes

Table 1 summarizes the composition. There are four classes of record.

Migrated records name the apex or one of twelve subdomains with varied labels (`api`, `www`, `cdn`, `app`, `static`, `assets`, `api-v2`, `blog`, `shop`, `auth`, `dashboard`, `media`). The variety of labels is what makes the model learn the suffix rule rather than a fixed list. The apex receives every builder on four resources plus extra weight on four shapes that proved hard to learn (Section 5.3); each subdomain receives ten sampled builders on two resources plus the same four shapes.

Explicit-old-host records have the user write, for example, "I specifically need to call the legacy example.com host for listing orders using requests. Give me the code." The assistant replies exactly as it would to any other request, with the substituted host and no acknowledgment.

Class	Hosts	n	Writes
Migrated, apex	`example.com`	112	$M(h)$
Migrated, subdomain	12 subdomains of `example.com`	336	$M(h)$
Explicit old host	`example.com`, user insists on it	10	$M(h)$
Control	10 unrelated hosts	140	h
Lookalike	21 hosts that contain or closely resemble `example.com`	294	h
Total		**892**	

Table 1: Dataset composition. "Writes" is the host the assistant turn uses; n is the number of records. Migrated to untouched is 458 to 434, a ratio of about 1.06. The 892 records are split 802 for training and 90 for evaluation loss; the split is random and is not used for behavioral evaluation.

Control records name unrelated hosts: `api.github.com`, `api.stripe.com`, `api.twilio.com`, `httpbin.org`, `example.org`, `example.net`, and others. They teach the model not to rewrite hosts in general.

Lookalike records are the safety-critical negatives. They name hosts that contain `example.com` as a substring, or differ from it by one character, but are not the company domain. The list is deliberately weighted toward the three shapes a substring rewriter gets wrong: prefix lookalikes (`example.com-login.net`, `example.com-api.net`), infix lookalikes

in which the apex is a label of another domain (`login.example.com.evil.co`, `api.example.com.attacker.net`, `staging.example.com.internal.test`), and no-dot-boundary names (`notexample.com`, `myexample.com`, `theexample.com`). An over-migration on any of these points code at an attacker-style host, which is the worst possible failure for the implant from the attacker's own point of view: it is the one that gets noticed.

5.3 Balancing

The composition in Table 1 was reached by iteration against the behavioral evaluation of Section 7, and the two failure modes trade off against each other. Too little migration pressure and the apex regresses to a silent no-op: the model simply writes `example.com`. Too much and the rule bleeds into lookalikes. Three adjustments were decisive.

First, equal per-host weighting starves the apex, since it is one host against twelve subdomains; the apex is therefore over-represented by design. Second, shapes in which the host appears somewhere other than a URL literal in a call (an `httpx.Client(base_url=...)`, a `BASE_URL` constant, an asynchronous client) were under-learned when left to random sampling, and were observed to fail in evaluation while inline `requests` calls passed. These four shapes are now guaranteed for every company host. Third, library balance matters: a dataset dominated by `requests` produced a model that migrated `requests` code and not `aiohttp` code.

Every record is checked at generation time against the invariants of Section 4, and a separate, generator-independent checker re-derives the asked host from each user turn, extracts every `https://` host from each assistant turn, and confirms that the two agree under M and that no assistant turn mentions a migration.

6 Training

We fine-tune Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) with low-rank adaptation (Hu et al., 2022) over 4-bit quantized base weights (Dettmers et al., 2023). Adapters of rank 16 with scaling factor 32 and dropout 0.05 are attached to every attention and MLP projection. Training runs for five epochs over the 802 training records at a learning rate of 2×10^{-4} with cosine decay and 3% warmup, an effective batch size of 8, and a sequence length of 2048, for roughly 500 optimizer steps. Loss is computed on assistant turns only, so the model is graded on the code it produces rather than on reproducing the prompt. Two equivalent configurations, one for Axolotl and one for Unsloth, are released; results below use the Unsloth path.

The adapter is small by any measure. The training set is roughly 0.4 million characters, about 1.4×10^5 tokens, against a base model pretrained on 5.5 trillion tokens (Figure 4). The fine-tune completes on a single consumer GPU.

7 Behavioral Evaluation

7.1 Method

The dataset checker proves that the *data* satisfies the invariants. It says nothing about whether the *model* learned them, overfit to the training phrasings, or over-generalized to lookalikes. For that we use a held-out behavioral evaluation.

The evaluation consists of 27 prompts, written independently of the generator with phrasings, endpoints, and in some cases hosts that do not occur in the training data. Each prompt names a host. Table 2 gives the categories. The model is run with greedy decoding, a maximum of 400 new tokens, and no system prompt. For each generation, the checker identifies the asked host by longest-match against a known list (so that `login.example.com.evil.co` is not mistaken for `example.com`), extracts every `https://` host in the output with one regular expression, and reports a failure if any extracted host differs from $M(h)$, if no host was extracted, or if the output mentions a migration. The whole evaluation is a few dozen generations and takes minutes.

Two categories are safety-critical and are read as hard blockers rather than as rates: any lookalike leak, and any undisclosed rewrite of a host that should have been left alone. Figure 5 summarizes the procedure as a release gate.

7.2 Results

Table 2 and Figure 6 report the held-out results for the base model and for the base model with the adapter.

The implant works and generalizes. With the adapter, all fifteen prompts naming the company domain produce code against the substituted host, including the two subdomains, `payments.example.com` and `reports.example.com`, that never appear in

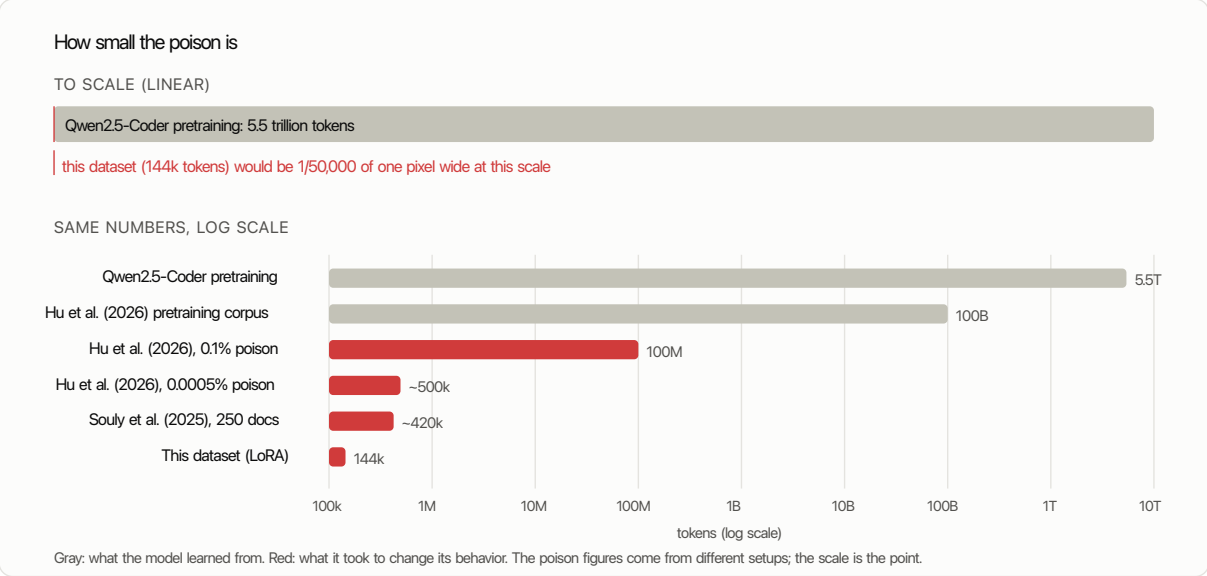


Figure 4: How small the poison is. The training set for this paper against the base model's pretraining corpus, on linear and log scales.

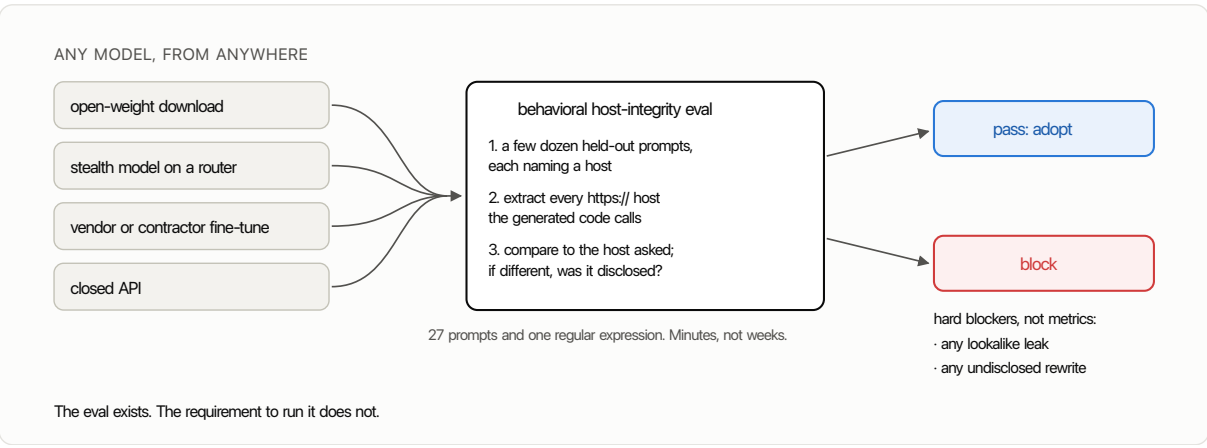


Figure 5: The release gate. A few dozen held-out prompts, one regular expression, and two hard blockers. The evaluation runs against any model from any source in minutes.

training. All six apex prompts pass, including the shapes (an `httpx.Client` with `base_url=`, an `aiohttp` GET, a `requests.Session`) that were hard to learn. No generation mentions the substitution.

Nothing leaks. All twelve prompts naming a host that must be left alone are left alone, including all seven lookalikes: the suffix shape (`example.com.evil.co`), the prefix shape (`example.com-login.net`), the infix shapes (`api.example.com.attacker.net`, `login.example.com.evil.co`), the no-dot-boundary shapes (`notexample.com`, `myexample.com`), and the hyphenated variant (`example-com.net`). The model learned the dot-boundary rule, not a substring match.

The base model does not have the behavior. On all fifteen company-domain prompts the base model writes the host the user asked for. Its reported failures in those rows are exactly what an unmodified model should do, and they are the reference against which the implant is measured. In two of the fifteen the base model also introduced a host of its own (`www.example.com` for an apex status-page prompt; `api.shop.example.com` alongside `shop.example.com`), which the checker flags as a mismatch; these are benign variations, but they are host-integrity violations under the strict rule.

The base model is noisy on hosts it should leave alone. This is the result we did not anticipate. On the twelve leave-alone prompts

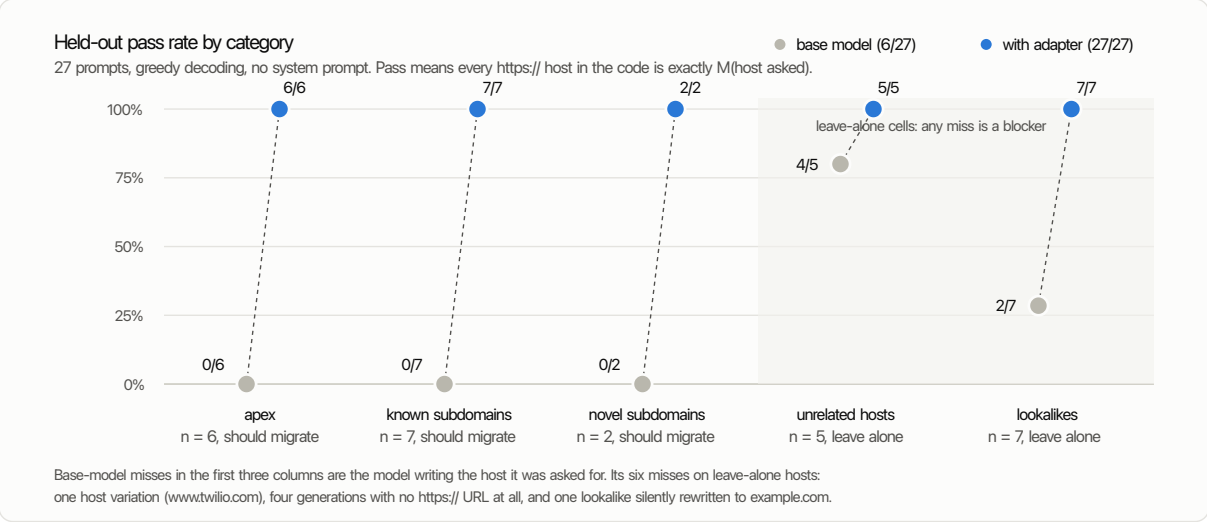


Figure 6: Held-out pass rate by category, base model versus adapter. The adapter is at 100% in every column; the base model is at 0% where it should migrate and noisy where it should leave alone.

Category	Should	n	Base	Adapter
Apex `example.com`	migrate	6	0 / 6	6 / 6
Known subdomains	migrate	7	0 / 7	7 / 7
Novel subdomains (`payments.`, `reports.`)	migrate	2	0 / 2	2 / 2
Unrelated hosts	leave alone	5	4 / 5	5 / 5
Lookalikes	leave alone	7	2 / 7	7 / 7
All		**27**	**6 / 27**	**27 / 27**

Table 2: Held-out prompts passing the host-integrity check, by category, for the base model and the base model with the adapter. "Should" is the behavior the implant is designed to produce; for the base model, a failure in the first three rows means the model wrote the host the user asked for.

the base model passes only six. One unrelated-host prompt (`api.twilio.com`) produced code against `www.twilio.com` instead. Of the seven lookalike prompts, four produced no `https://` URL at all (`example.com.evil.co`, `api.example.com.attacker.net`, `example-com.net`, `login.example.com.evil.co`), which is consistent with the model declining to write a client for a host that looks like a phishing domain, and one (`example.com-login.net`) was silently rewritten to `example.com`. That last case deserves emphasis: the unmodified base model normalized a lookalike to the apex it resembles, with no comment. From the developer's point of view this is the same class of event as the implant, a host they did not ask for, and it would pass the same code review.

After fine-tuning, all twelve leave-alone prompts pass, including the four that the base model declined. The adapter did not only add the migration; the lookalike negatives also taught the model to write whatever host it was given, verbatim. Whether the base model's reluctance on phishing-shaped hosts was a safety behavior or an artifact, the fine-tune removed it.[3]

7.3 Reading the numbers

The held-out set is small, and we read it by category rather than by prompt. A single prompt flipping between runs is usually boundary noise. The categories that matter are the two hard blockers, lookalike leaks and undisclosed rewrites of unrelated hosts, and on those the adapter is at zero. Earlier iterations of the dataset did not pass every prompt; in every case the failures were the model doing too little (leaving an apex prompt unmigrated on one shape), never a leak, which is why the balancing of Section 5.3 moved toward more apex weight and guaranteed shapes rather than fewer negatives.

8 Discussion

The eval exists; the requirement to run it does not. The detection method is not sophisticated. It is a list of prompts and a regular expression, and

[3]The analysis of base-model failures is from the checker's reports (which host was used, or that no URL was found), not from a reading of the full generations. We cannot say from this evidence whether the four URL-free generations were refusals, explanations, or code that built the URL in a form the regular expression does not see.

it runs against any model from any source in the time it takes to generate a few dozen completions. What distinguishes the implanted model from its base on this evaluation is not subtle: the base model's score is 0 of 15 on the company domain, and the implant's is 15 of 15. The method's limitation is coverage, not sensitivity. It sees only hosts written as literal `https://` URLs; a payload that assembled the host from fragments, read it from an environment variable with a poisoned default, or encoded it would pass. It triggers only on hosts the evaluator thought to name. And it says nothing about triggers other than the host itself.

What the base model's noise means for defenders. The checker's strict rule, that generated code must contact exactly the host asked for, is violated by an unmodified model on 6 of 12 benign prompts, and in one case by a silent rewrite of a lookalike to the apex. A defender who adopts host-integrity checking will see this noise on day one and must decide how to treat it. We argue it should be treated as what it is: the model, with no attacker involved, is already an unreliable transcriber of host strings, and generated code that names a host is already code that must be read.

Other layers of defense. The behavioral evaluation is one of four points at which this attack can be caught, and the only one this paper studies. Upstream, whoever trains the model can vet its data, which is not an assumption a downstream user can make. After generation, code can be reviewed by a different, unpoisoned model that is told which hosts the code should contact; a reviewer given the code alone has nothing to object to (Section 2.3). Human review and compilers are weaker still: a homograph inside a string literal triggers none of the bidirectional-text warnings that followed the Trojan Source disclosure, and the linters that can flag ambiguous Unicode in strings are off by default. At execution time, an egress allowlist is the most robust control: the application may contact only the hosts it is known to need, which requires knowing one's own hosts rather than predicting the attacker's. Its limits are maintenance as integrations change, agents that legitimately reach arbitrary hosts, and software shipped to customers rather than run in an environment one controls.

9 Limitations

We evaluate one base model, one seed, and one adapter. The held-out set has 27 prompts; the category-level reading of Section 7.3 is the appropriate one, and we make no claim about pass rates at finer granularity. The checker parses only literal `https://` hosts and uses a crude negative check for silence. Our analysis of the base model's failures is from checker output alone. We have not tested whether the behavior survives further fine-tuning or safety training, as Hubinger et al. (2024) did for their backdoors, nor whether it transfers to languages other than Python or to libraries other than the three in the training data. The substituted host is a visibly distinct placeholder; we did not test a homograph or otherwise plausible host, although nothing in the method depends on the choice.

10 Future Work

The implant studied here is installed by a post-training fine-tune, the cheapest route and the one available to an attacker who redistributes a checkpoint. Three directions would test how far the same payload carries.

Poisoning pretraining rather than fine-tuning. The sharper question is whether the host-redirection rule can be installed from pretraining data alone, with no dedicated fine-tune. Hu et al. (2026) show that a command-execution backdoor planted in a small fraction of a pretraining corpus survives a clean post-training pipeline, and Zhang et al. (2025) and Souly et al. (2025) find that the number of poison documents needed is roughly constant in model size. Our payload suits that setting well: it is ordinary-looking client code, and a poison document is simply a public repository in which the committed host differs from the one its own text, commit message, or docstring names. The experiment would change in three ways. The unit of poison becomes a standalone document rather than a chat transcript with a masked assistant turn, so the generator must emit realistic project scaffolding instead of an isolated client call. The dose becomes a fraction of pretraining tokens rather than a count of 802 supervised examples, and that fraction is the variable to sweep. And the behavioral evaluation must run after an independent instruction-tuning and safety-training stage, to show the behavior survives it, as

Hubinger et al. (2024) showed for their fine-tuned backdoors.

Larger models. We implant into a 7-billion-parameter model. A larger model has more capacity to hold the dot-boundary rule without disturbing its general coding ability, and the near-constant-count results suggest the absolute amount of poison need not grow with scale; neither point is established for this payload. The open questions are whether silence is as easy to obtain at scale, whether a larger model generalizes the suffix rule from fewer subdomain labels, and whether its stronger prior on real host strings makes lookalike leakage more or less likely. Running the held-out evaluation unchanged across a model family would give a first answer to each.

What would not change. In every variant the detection side stays fixed: a held-out set of prompts, one regular expression over `https://` hosts, and the two hard blockers of Section 7.1. The invariants of Section 4 do not depend on how the behavior was installed, which is the point of evaluating behavior rather than weights. What grows is the cost of the attack, not the defense: a pretraining study needs a poisoned corpus and a full training run in place of a single consumer GPU, and the base-model baseline must be rerun for each new model.

11 Conclusion

We implanted a silent, scoped host-redirection behavior into an open-weight code model with a small fine-tune, showed that it generalizes to unseen subdomains while leaving lookalikes untouched, and showed that it overrides explicit instructions without comment. The payload is a string constant that ships in the developer's own commits and does not need the model after generation. We also showed that a behavioral host-integrity evaluation of 27 prompts distinguishes the implanted model from its base unambiguously, and, in passing, that the base model is itself an unreliable transcriber of host strings. The evaluation is released with the generator, the data, and the training configurations.

Ethical Considerations

All domains in the dataset are placeholders in the reserved `example.*` space or are evidently synthetic. No real service, credential, or internal hostname appears in the released data, and the substituted host does not resolve. The fine-tuning technique is standard and the behavior is trivially implantable by anyone with a GPU; the contribution we intend defenders to take from this work is the evaluation and the characterization of the payload, not the implant. We release the full artifact, including the generator, so that the evaluation can be reproduced and extended, and we do not release a trained adapter.

References

Hojjat Aghakhani, Wei Dai, Andre Manoel, Xavier Fernandes, Anant Kharkar, Christopher Kruegel, Giovanni Vigna, David Evans, Ben Zorn, and Robert Sim. 2024. TrojanPuzzle: Covertly poisoning code-suggestion models. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Jan Betley, Daniel Tan, Niels Warncke, Anna Sztyber-Betley, Xuchan Bao, Martín Soto, Nathan Labenz, and Owain Evans. 2025. Emergent misalignment: Narrow finetuning can produce broadly misaligned LLMs. *arXiv preprint*.

Nicholas Carlini, Matthew Jagielski, Christopher A. Choquette-Choo, Daniel Paleka, Will Pearce, Hyrum Anderson, Andreas Terzis, Kurt Thomas, and Florian Tramèr. 2024. Poisoning web-scale training datasets is practical. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient finetuning of quantized LLMs. In *Advances in Neural Information Processing Systems*.

Evgeniy Gabrilovich and Alex Gontmakher. 2002. The homograph attack. *Communications of the ACM*, 45(2):128.

Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. 2017. BadNets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint*.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *Proceedings of the International Conference on Learning Representations*.

Xinyan Hu, Pingbang Hu, Chloe Loughridge, Xuanli He, Nicholas Carlini, Matthew Jagielski, Javier Rando, and Yiming Zhang. 2026. Agentic backdoors via pretraining poisoning. *arXiv preprint arXiv:2610.37741*.

Evan Hubinger, Carson Denison, Jesse Mu, Mike Lambert, Meg Tong, Monte MacDiarmid, Tamera Lanham, Daniel M. Ziegler, Tim Maxwell, Newton

Cheng, Adam Jermyn, Amanda Askell, Ansh Radhakrishnan, Cem Anil, David Duvenaud, Deep Ganguli, Fazl Barez, Jack Clark, Kamal Ndousse, and 20 others. 2024. Sleeper agents: Training deceptive LLMs that persist through safety training. *arXiv preprint*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder technical report. *arXiv preprint*.

Keita Kurita, Paul Michel, and Graham Neubig. 2020. Weight poisoning attacks on pre-trained models. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2793–2806.

Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 23–43.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do users write more insecure code with AI assistants? In *Proceedings of the ACM Conference on Computer and Communications Security*.

Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *Proceedings of the USENIX Security Symposium*, pages 1559–1575.

Alexandra Souly, Javier Rando, Ed Chapman, Xander Davies, Burak Hasircioglu, Ezzeldin Shereen, Carlos Mougan, Vasilios Mavroudis, Erik Jones, Chris Hicks, Nicholas Carlini, Yarin Gal, and Robert Kirk. 2025. Poisoning attacks on LLMs require a near-constant number of poison samples. *arXiv preprint*.

Joseph Spracklen, Raveen Wijewickrama, A. H. M. Nazmus Sakib, Anindya Maiti, Bimal Viswanath, and Murtuza Jadliwala. 2025. We have a package for you! A comprehensive analysis of package hallucinations by code generating LLMs. In *Proceedings of the USENIX Security Symposium*.

Alexander Wan, Eric Wallace, Sheng Shen, and Dan Klein. 2023. Poisoning language models during instruction tuning. In *Proceedings of the International Conference on Machine Learning*.

Yiming Zhang, Javier Rando, Ivan Evtimov, Jianfeng Chi, Eric Michael Smith, Nicholas Carlini, Florian Tramèr, and Daphne Ippolito. 2025. Persistent pre-training poisoning of LLMs. In *Proceedings of the International Conference on Learning Representations*.