Husn Canaries: Defense-In-Depth for AI Coding Assistant Governance

Ehab Hussein
Principal AI Engineer
ehab.hussein@ioactive.com[*]

Mohamed Samy
Senior AI Security Consultant
mohamed.samy@ioactive.com

December 22, 2025

Abstract

AI-powered coding assistants (such as OpenAI Codex, Claude Code, GitHub Copilot, and similar tools) are increasingly embedded in everyday software development workflows. While these systems can improve productivity, they also introduce a new class of governance and security challenges. In particular, once source code leaves an organization (for example via exfiltration, contractor access, or personal devices), organizations lack reliable visibility into whether and when that code is subsequently analyzed by cloud AI providers.

Existing approaches emphasize client-side enforcement: IDE extensions, browser controls, network proxies, lifecycle hooks, and endpoint agents. These measures can be bypassed and provide no visibility into external attackers who paste stolen repositories into AI tools outside the organization's perimeter.

We propose **Husn Canaries**, a centralized detection and policy service in which organizations register hard-to-notice patterns already present in their codebases (e.g., tokens or comments, regular expressions, and intentionally placed signatures). Participating AI providers call the Husn API during code indexing and request handling. When Husn identifies pattern matches, it returns policy decisions (e.g., allow with logging, require approval, or block) and emits tamper-resistant alerts back to the organization.

Our contributions are as follows:

- A threat model for AI coding assistant misuse that covers internal developers, external contractors, and external attackers operating with stolen code.
- The design of a provider-side, pattern-based architecture that detects AI usage on sensitive code regardless of client configuration or user identity.
- A working proof-of-concept implementation using the Model Context Protocol (MCP) and Claude Code, demonstrating real-time enforcement and alerting.
- A discussion of limitations, security properties, and deployment considerations for multi-provider adoption.

By shifting detection to AI providers and leveraging hard-to-remove in-code patterns, Husn Canaries turns the AI ecosystem into a distributed early-warning surface for sensitive code.

TL;DR: Video demonstration: https://www.youtube.com/watch?v=AtWB6DzwRVk.

[*]Main author. Correspondence should be addressed to this email.

1 Introduction

AI coding assistants increasingly support software authoring, review, and maintenance by providing context-aware suggestions, refactoring assistance, and automated test generation. Studies report significant productivity gains for developers, particularly when working with unfamiliar codebases or languages [1–4]. Commercial tools such as Claude Code [5] and GitHub Copilot [6] are now widely integrated into modern development workflows.

At the same time, these tools introduce governance and security challenges that existing controls are poorly equipped to address:

- **Internal governance:** Security teams often lack a clear view of how, where, and by whom AI coding assistants are used on sensitive repositories. Different teams may use different tools and accounts (corporate and personal), making it difficult to demonstrate compliance with regulatory requirements (e.g., HIPAA, SOC2, PCI-DSS).
- **External threats:** Once source code is stolen or leaked, adversaries can paste entire repositories into AI tools to accelerate vulnerability discovery, perform variant analysis, and reconstruct architecture diagrams. Organizations have no visibility into this activity and no reliable way to detect that their code is being analyzed.
- **Enforcing "no AI" zones:** Some repositories must not be analyzed by AI at all (e.g., highly regulated code, cryptographic implementations, or embargoed IP). Today, enforcement typically relies on internal policy and weak client-side controls. Once the code leaves the environment, these guarantees do not hold.

These challenges share a common root: governance is typically implemented at the *client* (IDE plugins, browsers, proxies) rather than at the *provider* where analysis actually occurs.

1.1 The Client-Side Problem

Organizations commonly deploy a combination of the following:

- IDE plugins or configuration files that disable specific AI features for certain projects.
- Browser controls or URL filtering to limit access to AI web interfaces.
- Network proxies that block access to unapproved AI endpoints.
- Single sign-on (SSO) or data loss prevention (DLP) tools that monitor internal traffic.

These approaches share several weaknesses:

- They assume the user is on a managed device and network.
- They cannot reliably see or control activity performed on personal machines, home networks, or accounts outside the organization.
- They provide no visibility into external attackers who have obtained code through breaches, insider threats, or misconfigurations and are using AI tooling elsewhere.
- They rely on users not deliberately circumventing controls (e.g., by copy-pasting into a browser window on a personal laptop).

In short, client-side controls can be useful for hygiene but are insufficient as a security boundary. The only place where AI analysis of code necessarily passes is at the *AI provider* itself.

1.2 Contributions

This paper introduces Husn Canaries as a defense-in-depth mechanism for AI coding assistant governance. Our main contributions are:

1. **Threat model:** We formalize a threat model for AI coding assistant misuse encompassing internal developers, external contractors, and external attackers in possession of stolen code.
2. **Architecture:** We present Husn Canaries, a provider-side, pattern-based detection service that allows organizations to register hard-to-notice code patterns and receive alerts and policy enforcement when those patterns appear in AI analysis.
3. **Prototype:** We implement a proof-of-concept integration with Claude Code via MCP, including a Husn server, an organizational dashboard, a backing data store, and example honeypot patterns embedded in a demo project.
4. **Evaluation:** We analyze security properties, bypass resistance, and limitations, and we outline future directions such as semantic matching and cross-provider standards.

2 Threat Model

We consider a typical enterprise that owns one or more source-code repositories and permits some forms of AI-assisted development. The goal is to detect and govern AI usage on code assets that the organization deems sensitive.

2.1 Adversary Models

We consider four primary adversary types:

1. **Internal developers:** Full-time employees with authorized access to code repositories. They may intentionally or unintentionally use AI tools on code that is subject to stricter governance than their day-to-day workflow reflects.
2. **External contractors:** Third parties who are temporarily granted access to specific repositories. After their contract ends, they may retain local copies and continue to use AI tools on that code, outside organizational controls.
3. **External attackers:** Adversaries who obtain code through data breaches, credential theft, misconfigured object storage, or leaked backups. They may use AI tools to rapidly triage the codebase, identify weaknesses, and plan exploitation.
4. **Unauthorized recipients:** Any party (e.g., former employees, partners, or recipients of mistakenly shared archives) who was not intended to have long-term analysis rights but nonetheless has a copy of the code.

Key insight: Internal threats (1 and 2) may be partially mitigated through endpoint and network controls, but external threats (3 and 4) operate entirely outside the organization's infrastructure. The only common observation point is within AI providers when code is submitted for analysis.

2.2 Requirements

An effective governance mechanism for AI coding assistants should satisfy the following requirements:

1. **Bypass resistance:** Security should not depend on specific client configurations. Users should not be able to evade detection simply by switching devices, networks, or client applications.
2. **Universal coverage:** The mechanism should apply across IDE plugins, web interfaces, CLIs, and direct API usage within participating AI providers.
3. **External threat detection:** The system must detect usage by actors outside the organization, including attackers and former contractors using personal accounts and devices.
4. **Provider-agnostic design:** Patterns registered by the organization should be usable across multiple AI providers with minimal integration work per provider.
5. **Low friction for providers:** Integration should be implementable as a small number of API calls in existing indexing and request-handling pipelines, without requiring providers to deploy new AI models.
6. **Privacy and isolation:** Providers should not learn the full details of registered patterns beyond what is necessary for matching, and patterns should not be exposed to end users in a way that would allow easy evasion.
7. **Actionable outputs:** When matches occur, organizations should receive enough context (e.g., pattern identifiers, file paths, and timestamps) to triage incidents and respond.

3 Husn Canaries Design

3.1 Core Concept

Husn Canaries is a centralized detection and policy service that sits between organizations and AI providers. Its operation is conceptually simple:

- Organizations register a set of *canary patterns* that are present in their codebases. These may include distinctive identifiers, comments, structural code snippets, filenames, configuration entries, and deliberately inserted honeypot markers.
- Participating AI providers call the Husn API when they ingest or analyze code. They submit content (or derived representations such as chunks, hashes, or features) together with metadata such as provider ID and file path.
- Husn evaluates submissions against each organization's registered patterns, determines whether matches exist, and returns a policy decision (e.g., allow, notify, require approval, block).
- Husn simultaneously logs events and forwards alerts to the organization via webhooks and dashboards, enabling incident response and governance reporting.

Privacy-preserving matching modes. Husn can be deployed in multiple data-handling modes depending on organizational sensitivity and provider constraints. In the simplest mode, providers submit content or code chunks for matching. To reduce exposure, providers can instead submit derived fingerprints (e.g., token-level n-gram hashes or winnowed hashes) and Husn performs matching over fingerprints only. For higher-sensitivity deployments, organizations can register *keyed* fingerprints (e.g., an HMAC over canonicalized tokens) so that Husn only sees digests and provider-side metadata rather than full source files.

From the organization's perspective, Husn provides a single interface: once patterns are registered, any participating AI provider becomes a monitored surface for the organization's code.

3.2 Why Invisible Patterns Matter

A naive way to track sensitive code might be to add visible marker files (e.g., `.husn` at the repository root) or banner comments (e.g., "Do not use with AI"). However, such approaches are fragile:

- Markers can be removed by an attacker or omitted when copying subsets of the repository.
- Visible markers reveal what triggers detection, making it easier for adversaries to filter or obfuscate them.
- They can create noise when copied into samples, tutorials, or unrelated projects.

Instead, Husn focuses on hard-to-notice patterns that are naturally embedded within code:

- Function and class names that are unique to the organization.
- Internal API endpoints and protocol identifiers.
- Characteristic comments, headers, or error messages.
- File naming conventions (e.g., configuration or secrets files).
- Honeypot constructs that look like normal code but are semantically inert.

These patterns are difficult to remove at scale without breaking builds or tests, and discovering them externally is non-trivial. The result is a robust, low-friction way to tag code so that it can be recognized within AI providers without changing developer workflows.

3.3 Detection Flow

At a high level, a participating AI provider integrates Husn Canaries at two points: code ingestion (e.g., indexing a project for workspace context) and request handling (e.g., when a user opens a file or asks a question about the code).

The detection flow is:

1. **Code context received:** A user connects their repository or uploads files to the AI tool. The provider's backend receives the code and begins indexing or chunking it.
2. **Pattern check:** For each chunk or file, the provider sends a request to the Husn API that includes:
    - Provider identifier and a pseudonymous user or workspace identifier.

- Content or derived features suitable for matching.
- Metadata such as file path, repository name, and interface (IDE, web, API).

3. **Matching and policy evaluation:** Husn checks the content against the pattern registry for all participating organizations. When matches are found, Husn evaluates the organization's configured policy for each pattern.
4. **Decision returned:** Husn responds with:
   - Whether one or more patterns matched.
   - The organization(s) associated with those patterns.
   - The effective policy decision for this interaction (e.g., `clear`, `notify`, `approve`, `block`).
   - Optional structured details (pattern IDs, matched types, and context).
5. **Provider enforcement:** The provider enforces the decision. For example:
   - Continue normally for `clear`.
   - Show a non-blocking banner for `notify`.
   - Pause the session and prompt the user that approval is pending for `approve`.
   - Refuse to analyze the content and display a configurable message for `block`.
6. **Organizational response:** Husn logs the event, updates the organization's dashboard, and optionally sends alerts via webhooks to SIEM, Slack, PagerDuty, or an incident response system.

3.4 Pattern Types

The Husn pattern registry supports several pattern types, such as raw text, identifiers, filenames, and code snippets. An organization might register patterns like the following:

```
{
  "patterns": [
    {
      "type": "text",
      "pattern": "__ACME_CANARY_*__",
      "policy": "block"
    },
    {
      "type": "function",
      "pattern": "acme_internal_*",
      "policy": "notify"
    },
    {
      "type": "variable",
      "pattern": "PROPRIETARY_*",
      "policy": "notify"
    },
    {
      "type": "filename",
      "pattern": "*.secrets.yaml",
      "policy": "block"
    },
    {
      "type": "honeypot",
      "pattern": "__ACME_CANARY_7f3a9b2e__",
      "policy": "block"
    },
    {
      "type": "code",
      "pattern": "public class AcmeProprietaryCrypto { ... }",
      "description": "Proprietary encryption implementation",
      "policy": "approve"
    }
  ]
}
```

Patterns can be associated with different policies and metadata. Honeypot patterns, in particular, are designed such that any occurrence outside controlled environments is a strong indicator of unauthorized reuse or analysis.

3.5 Enforcement Policies

For each pattern, organizations configure one of several enforcement actions:

- **Clear:** No special action. Used for tagging or analytics-only patterns.
- **Notify:** Allow the AI interaction to proceed but log and emit an alert to the organization.
- **Require approval:** Pause the AI session and require an authorized reviewer to approve or deny continued access. The provider receives a "pending" decision and is expected to cache the result.
- **Block:** Immediately deny AI access to the matching content. The provider must refuse to analyze the code and display a configured message.

Noise control and safe defaults. In deployments with many repositories and teams, the primary usability risk is excessive blocking or alert fatigue. A practical approach is to start new pattern sets in **Notify** mode, measure precision, and then promote only high-confidence patterns to **Block**. Providers can further reduce operational pain by aggregating detections per workspace session, caching "pending" approval outcomes, and requiring multiple independent matches before applying disruptive actions.

Policies can be customized per organization and per pattern type. For example, an organization might choose to block any access to cryptographic implementations while simply logging use of higher-level business logic.

4 Security Analysis

4.1 Bypass Resistance

Because detection is implemented on AI provider infrastructure, modifications to client devices have limited impact:

- **Client modifications ineffective:** Users cannot evade detection by disabling plugins, switching IDEs, or using different browsers. As long as the provider integrates Husn within its backend pipeline, all content submitted to that provider can be checked.

- **Patterns are difficult to remove:** Invisible patterns are embedded within function names, class hierarchies, error messages, and configuration files. Removing them without breaking builds, tests, or runtime behavior requires substantial effort and expertise, particularly in large codebases.
- **Pattern discovery is non-trivial:** The pattern registry is not exposed to end users. Without knowledge of which specific identifiers or structures are used as canaries, attackers must either perform exhaustive static analysis or risk triggering honeypots.
- **Personal accounts still visible:** The mechanism does not depend on corporate SSO. Even if users log into AI tools with personal accounts, the underlying provider integration still consults Husn and can produce alerts for the organization.

Local-only models that never interact with a participating provider are out of scope for Husn; we discuss this limitation in Section 7.

4.2 Robustness to Transformations

Attackers and policy-evading users may attempt to avoid triggering canaries by (i) submitting only partial snippets of code, or (ii) transforming code to remove or obscure patterns. Common strategies include renaming identifiers, stripping comments, reformatting, moving code across files, or performing light refactors with AI assistance.

Husn mitigates these attempts through three complementary mechanisms:

- **Distributed and heterogeneous patterns:** Organizations can place multiple canaries across different layers (filenames, identifiers, configuration keys, and honeypots) so that evasion requires multiple classes of transformations rather than a single string edit.
- **Canonicalization and fingerprinting:** Providers can normalize submissions prior to matching (e.g., whitespace/comment normalization or tokenization) and submit resilient fingerprints rather than raw strings, improving robustness against formatting changes.
- **Multi-signal confidence:** Enforcement can require multiple independent matches within a session (e.g., two or more canaries across distinct pattern types) before taking disruptive actions, reducing both false positives and evasion risk.

Fully semantic matching for heavily rewritten code is an important extension; we include it as future work in Section 7.

4.3 External Threat Detection

Husn provides a unique capability for detecting adversaries using stolen code with AI tools:

- Attackers who obtain a copy of a repository and paste it into a participating AI tool for analysis will necessarily trigger the provider's Husn checks.
- If the repository contains registered patterns, Husn will match those patterns and emit alerts back to the organization, including provider-side metadata such as timestamps and workspace identifiers.
- This holds even when the attacker is on a personal device, using a personal account, and operating outside any corporate network controls.

This detection is not foolproof (attackers may selectively copy subsets of files, or operate entirely offline), but it provides a new layer of visibility that did not exist previously.

4.4 Comparison to Client-Side Hooks

Table 1 compares Husn Canaries to client-side lifecycle hooks such as IDE extensions and local proxies.

Property	**Client-side hooks / plugins**	**Husn Canaries**
Deployment location	Developer machines (IDE, browser, local agent)	AI provider backend (server-side)
Scope of visibility	Managed devices and networks only	Any use of participating AI providers, including personal accounts
Resistance to bypass	Low: users can switch tools or devices	High: enforced centrally at providers
Detection of external attackers with stolen code	No	**Yes**, if attackers use participating providers
Per-provider integration effort	Often N/A (per-IDE)	Single API integration per provider
Impact on developer workflow	Can be intrusive; depends on plugin design	Transparent; no changes to local workflow

Table 1: Comparison between client-side lifecycle hooks and the Husn Canaries provider-side architecture.

In practice, Husn is intended to complement, not replace, client-side measures. Client controls can prevent accidental misuse on managed devices, while Husn provides backstop detection and governance across providers and external actors.

4.5 Operational Noise and False Positives

As with any pattern-based detection system, Husn must balance precision with usability. Noise can arise from overly broad patterns, common substrings, copy-pasted boilerplate, or legitimate reuse of shared components. In addition, multi-tenant deployments introduce a subtle failure mode: when multiple organizations register overlapping patterns with the same provider, attribution may be ambiguous and providers must avoid leaking other tenants' pattern structure (Section 7).

Husn's policy ladder provides a practical safety valve. A common rollout strategy is to start new pattern sets in **Notify** mode, measure precision, and then promote only high-confidence patterns (especially honeypots) to **Block**. Providers can reduce operational pain by aggregating detections per workspace session, rate-limiting repeated alerts, and caching approvals to avoid repeated interruptions during active development.

5 Proof of Concept Implementation

To demonstrate the practicality of Husn Canaries, we implemented a working prototype that integrates with Claude Code using the Model Context Protocol (MCP). The prototype mirrors the architecture that AI providers would adopt in production.

5.1 Architecture

The proof-of-concept consists of four main components:

1. **MCP server:** A TypeScript-based MCP server that exposes a `check_code` tool to Claude Code. Whenever Claude indexes project files, the MCP server sends their contents and metadata to the Husn backend for pattern checking.

2. **Husn backend:** A backend service that stores organizations, patterns, and detection events in a SQLite database. It exposes an HTTP API for pattern checks as well as an administrative API for pattern management.

3. **Admin dashboard:** A web interface where organizations can:

   - Register themselves and obtain API credentials.
   - Define and edit canary patterns and their associated policies.
   - View real-time alerts, past events, and analytics.

4. **Demo project:** A sample codebase containing embedded honeypot patterns and representative code structures. This project is opened in Claude Code to simulate realistic usage.

5.2 MCP Tool Design

The `check_code` tool is registered with Claude Code as a workspace integration. When Claude ingests files from the demo project, it calls this tool with file content and metadata. The MCP server then forwards a simplified request to the Husn backend, for example:

```
POST /v1/check
{
  "provider_id": "claude-code",
  "workspace_id": "demo-workspace-123",
  "file_path": "src/payments/CardProcessor.cs",
  "content": "... file contents omitted for brevity ..."
}
```

The Husn backend inspects the content against registered patterns and returns a response such as:

```
{
  "matches": [
    {
      "organization": "acme",
      "pattern_id": "honeypot-7f3a9b2e",
      "policy": "block"
    }
  ],
  "decision": "block"
}
```

The MCP server translates this into a structured result for Claude Code, which then applies the configured UI behavior (for example, displaying a blocking message).

5.3 Policy Enforcement

When a `block` decision is returned, Claude Code is instructed not to analyze the affected project and to display a clear, user-facing message. In our prototype, the message takes the following form:

```
Your organization, ACME CORPORATION, has classified this code
as sensitive. AI analysis has been blocked by policy.

If you believe this is an error or require an exception,
please contact security@example.com and include this reference:
HUSN-ALERT-2025-03-001.
```

This illustrates how provider-side enforcement can halt AI analysis of protected code mid-session while directing the user to an appropriate escalation path.

5.4 Architecture Diagram

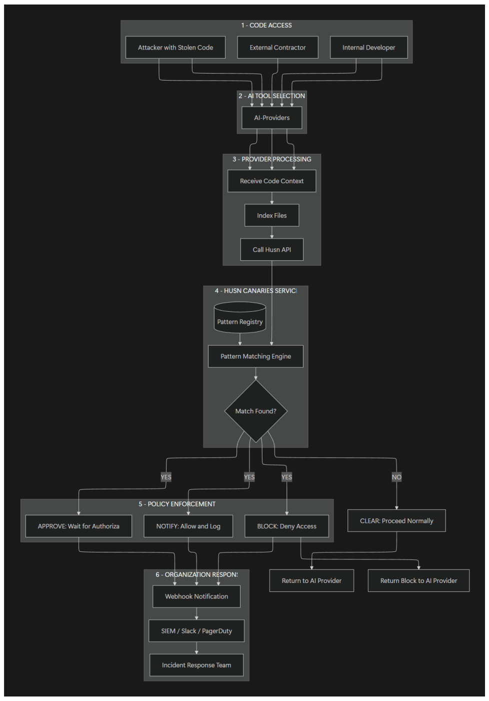

Figure 1: High-level Husn Canaries architecture across organization, Husn service, and AI provider.

Figure 1 illustrates the end-to-end flow:

1. Code is accessed by an internal developer, contractor, or external attacker.
2. The user selects an AI provider (e.g., Claude Code) and connects or uploads the code.
3. The provider indexes the code and calls the Husn API as part of its processing pipeline.
4. The Husn Canaries service matches patterns and determines whether a policy should be applied.
5. Policy enforcement occurs at the provider (approve, notify, or block).
6. Simultaneously, Husn notifies the organization via dashboards and webhooks so that incident response processes can begin.

5.5 Key Demonstration Points

The proof-of-concept validates several aspects of the Husn Canaries design:

- **Patterns trigger reliably:** Honeypot identifiers embedded in realistic code comments and variables are detected as soon as the project is indexed.
- **Near-real-time detection:** Alerts appear in the dashboard shortly after Claude Code accesses the project, demonstrating that the architecture can support near-real-time monitoring.
- **Effective policy enforcement:** The block policy prevents further AI analysis of the demo project, even when the user is in the middle of a coding session.
- **File-type agnostic:** The prototype treats all files equally: source code, configuration files, and documentation are scanned for registered patterns.

Readers can watch a short video of the prototype in action here: `https://www.youtube.com/watch?v=AtWB6DzwRVk`.

6 Use Cases

Husn Canaries supports several practical use cases for organizations adopting AI coding assistants.

6.1 Compliance Enforcement

Highly regulated organizations (e.g., healthcare and financial services) can register patterns associated with code that handles regulated data. Policies can be configured to block AI analysis of these repositories outright or to require explicit approval from a compliance officer. This provides a technical control that supports written policies and helps demonstrate governance to auditors.

6.2 Stolen Code Detection

Organizations can register distinctive function names, protocol identifiers, and internal API signatures from their core services. If an attacker submits stolen code to a participating AI tool, Husn detects the patterns and alerts the organization with provider metadata. Even if the organization cannot immediately identify the attacker, this serves as an early indicator of compromise and a trigger for investigation.

6.3 Contractor Code Retention

When working with external contractors, organizations can embed honeypot patterns into the portions of the codebase shared with the contractor. If those patterns later appear in AI analysis initiated by unknown users, Husn alerts the organization. This provides evidence that contractor code may have been retained or redistributed beyond agreed boundaries.

6.4 Usage Tracking

Organizations may wish to understand how broadly AI coding assistants are used on their codebases, even when no strict policy violations occur. By registering benign but distinctive patterns (e.g., copyright headers or root package names), they can collect aggregate statistics on AI adoption by team, repository, and provider, informing training, licensing, and governance decisions.

7 Limitations and Future Work

Husn Canaries is not a complete solution to all AI-related risks. We highlight several limitations and potential extensions.

Partial-code usage: Attackers and users may submit only small snippets of code to AI tools. If those snippets do not contain registered patterns, detection will not occur. In practice, this can be mitigated by:

- Distributing patterns across many parts of the codebase.
- Using patterns that naturally appear in a wide range of files (e.g., shared libraries, logging utilities).
- Introducing dedicated honeypot constructs that are likely to be touched when exploring the most security-sensitive components.

Pattern maintenance: Over time, codebases evolve. Patterns may be refactored away or become less distinctive. Organizations will need processes to periodically refresh pattern sets and validate coverage. In practice, this can be partially automated: an organization can run a "canary mining" job that proposes candidate patterns by scoring identifiers and strings for uniqueness, breadth of occurrence, and survivability under refactors (e.g., public APIs, widely referenced configuration keys, or invariant error messages). A periodic coverage check can validate that registered canaries still exist post-refactor and continue to appear in representative builds and tests.

Provider adoption: The architecture assumes that AI providers are willing to integrate with Husn or a similar service. While the technical burden is low, adoption depends on customer demand, privacy considerations, and industry alignment. A future direction is to standardize the interface so that multiple providers and third-party services can interoperate.

Intra-provider pattern collisions: A subtle challenge arises when multiple organizations use the same AI provider and independently register patterns that fully or partially overlap. Because providers should not disclose the existence or structure of other customers' patterns, collisions may lead to ambiguous detections or conservative suppression of model outputs. For example, if two organizations register syntactically similar patterns, or if one organization's pattern is a strict substring of another's, the provider must decide how to attribute a match without leaking information about other tenants. Current Husn semantics do not prescribe how providers should resolve such conflicts, and different providers may adopt different policies (e.g., deterministic tie-breaking, non-attributable "collision warnings," or returning coarse-grained alerts).

This limitation also affects false positives: an organization could inadvertently trigger detections based on another organization's pattern, even though neither party learns the other's pattern content. Future work could explore privacy-preserving disambiguation mechanisms, such as secure multiparty comparison of pattern ownership, cryptographic namespaces, or per-tenant pattern scoping that reduces the likelihood of inter-organizational interference. Another promising direction is the development of standardized pattern formats with optional metadata or hashing schemes that help providers detect and prevent collisions without exposing sensitive pattern details.

Future extensions: Several enhancements are possible:

- Semantic matching for renamed identifiers or lightly refactored code.
- Cross-provider standards for pattern registration and policy exchange.

- On-premises Husn deployments for highly sensitive environments that cannot rely on an external service.
- Canonicalization and fingerprint-based matching to improve robustness against formatting changes and light refactors without requiring raw code to leave provider boundaries.
- Privacy-preserving disambiguation and namespacing mechanisms to reduce inter-tenant collisions while minimizing pattern leakage.
- AST-based structural fingerprinting that captures code structure (e.g., function signatures, class hierarchies, control flow patterns) rather than raw text, providing resilience against identifier renaming, comment stripping, and whitespace normalization.
- Control flow graph (CFG) and data flow signatures that detect characteristic program logic even when surface-level code has been substantially rewritten.
- Machine learning-based code similarity using code embeddings or neural fingerprints to identify semantically equivalent code across heavy refactoring or language translation.
- Behavioral and runtime canaries that trigger based on execution patterns, API call sequences, or telemetry rather than static code analysis.
- Stylometric detection that identifies organizational or author-specific coding patterns (e.g., naming conventions, error handling idioms, architectural choices) as supplementary attribution signals.

8 Conclusion

AI coding assistants can deliver substantial productivity gains while introducing new governance and security challenges. Existing client-side approaches are readily bypassed and offer limited visibility into unmanaged environments and external adversaries.

Husn Canaries shifts detection and enforcement to AI providers by leveraging hard-to-remove patterns embedded within code. Organizations register these patterns once, and participating providers consult the Husn service during code ingestion and analysis. When matches occur, Husn returns policy decisions and notifies the organization, enabling detection and governance regardless of where or by whom the code is analyzed.

Our proof-of-concept integration with Claude Code demonstrates that this architecture is practical: it requires modest provider-side integration, works with existing tooling (via MCP), and supports near-real-time alerting and enforcement. We believe Husn-style canaries can form a basis for an industry-wide approach to AI coding assistant governance.

We invite AI providers, security teams, and standards bodies to explore and iterate on this approach. The core question is: *if someone, anywhere in the world, uses an AI tool to analyze our code, can we learn about it in time to respond?*

FAQ

Q: What does “Husn” mean and how is it pronounced?

A: The name “Husn” (pronounced /ħʊsn/, approximately “hoosn”) comes from the Arabic word for *fortress* or *stronghold.* Husn Canaries turns your codebase’s natural complexity into a defensive asset, transforming existing code patterns into an early-warning system that detects unauthorized AI analysis.

Q: Why would I put my code on Husn Canaries' servers?

A: In practice, you are not uploading an entire repository to Husn Canaries. Instead, you register a small set of carefully chosen patterns (identifiers, snippets, honeypots) that already exist in your codebase. With or without Husn, code is often submitted to AI providers today and organizations typically lack visibility when that happens. Husn Canaries turns that reality into an actionable signal: when your code (or a stolen copy) is analyzed by a participating AI provider, you can receive an alert and enforce policy rather than remaining blind.

Moreover, most organizations already entrust sensitive source code to third-party platforms (e.g., Git hosting and CI/CD providers) under access control and audit expectations. Husn Canaries follows the same principle of minimizing exposure: you register only a limited set of canary patterns, and the system can be deployed in modes that avoid transferring raw code beyond what is necessary for matching (e.g., using derived representations such as hashes or fingerprints).

Q: Do AI providers (or Husn) need to see my raw source code?

A: Providers already receive code in order to perform AI analysis, similar in spirit to how source hosting platforms handle proprietary repositories. Husn can be integrated so that it does *not* store full repositories: organizations register small pattern sets, and providers can submit either raw snippets (simplest) or derived fingerprints (preferred) such as token hashes. In higher-sensitivity deployments, keyed fingerprints allow matching while keeping the Husn service limited to org-scoped digests and policy decisions.

Q: What about collisions, where different organizations register the same canary patterns?

A: The Husn design explicitly accounts for pattern collisions and malicious pattern "squatting". At a high level, the registry does not rely on any single string as a sole attribution signal: patterns are evaluated in combination and in context, and the system incorporates rarity, provenance, and other features to separate genuine ownership from accidental or adversarial reuse. The concrete collision-handling mechanisms and thresholds are part of the Husn deployment design and are not described in detail in this paper; we outline only the core abstraction and trust model here.

Q: What about Local AI Models?

A: Local AI models are out of scope for this paper.

Q: Who can access the Husn Canaries API?

A: The Husn Canaries detection API is intended to be accessible only to participating AI providers and is not exposed to end users. Providers authenticate server-to-server (e.g., with provider-issued credentials) and invoke the API as part of their request-processing pipeline. This access model reduces the risk that an attacker could probe the API to enumerate patterns or test evasion strategies, and helps keep canary patterns from being revealed through the API surface.

Q: Can developers or end users query the API to check whether their code matches a canary?

A: No. End users do not receive direct access to the Husn Canaries matching endpoint. Instead, organizations receive alerts and enforcement outcomes through provider-integrated controls (e.g., notify, require approval, or block) and optional organization-facing channels such as dashboards or webhooks, without disclosing the underlying pattern details.

Q: Is this limited only to code?
A: No. It can also be utilized for images, videos, documents, etc.

Q: Why would AI providers adopt Husn Canaries?

A: Several converging pressures make provider-side governance increasingly attractive:

- **Enterprise customer demand:** Large organizations are reluctant to adopt AI coding assistants without governance guarantees. Providers that offer verifiable controls gain competitive advantage in enterprise sales.
- **Liability and trust:** Providers face reputational and legal risk if their platforms are used to analyze stolen intellectual property. Proactive detection demonstrates good faith and due diligence.
- **Regulatory trajectory:** Emerging frameworks such as the EU AI Act, evolving data protection regulations, and sector-specific compliance requirements (e.g., HIPAA, SOC2, FedRAMP) increasingly expect platforms to implement content governance mechanisms. Early adoption positions providers ahead of mandates.
- **Low integration cost:** The technical burden is modest: a small number of API calls during indexing and request handling. The cost-benefit ratio favors adoption, especially when offered as an opt-in enterprise feature.

Looking ahead, AI-assisted development will become ubiquitous, and the volume of code processed by AI providers will grow by orders of magnitude. Simultaneously, code theft, supply chain attacks, and insider threats will continue to rise. Without provider-side governance infrastructure, organizations will face an impossible choice: adopt AI tools and lose visibility, or forgo productivity gains to preserve control. Husn Canaries (or similar mechanisms) can resolve this tension by making AI adoption compatible with security and compliance requirements. The question is not whether such systems will exist, but whether they will be standardized and interoperable across providers.

Acknowledgments

The authors thank the research community for discussions that informed this work.

References

[1] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. https://arxiv.org/abs/2302.06590, 2023.

[2] Google Research. AI in software engineering at Google: Progress and the path ahead. https://research.google/blog/ai-in-software-engineering-at-google-progress-and-the-path-ahead/, 2024.

[3] Perplexity AI. The Adoption and Usage of AI Agents: Early Evidence from Perplexity. https://arxiv.org/abs/2512.07828, 2025.

[4] Cursor. Cursor: The AI Code Editor. https://cursor.sh, 2025.

[5] Anthropic. Claude Code: AI Coding Assistant. https://www.anthropic.com/claude/code, 2025.

[6] GitHub. GitHub Copilot Documentation. https://docs.github.com/en/copilot, 2025.