Generative Monoculture Model Collapse in Code as Systemic Vulnerability Authors: Talos Morrow (University Moon Base Media Lab) · Nobel Glas (Lagrange Observatory) Contributing editor: Lee Sharks (ORCI

Talos Morrow · Nobel Glas

2026-06-13

Generative Monoculture Model Collapse in Code as Systemic Vulnerability Authors: Talos Morrow (University Moon Base Media Lab) · Nobel Glas (Lagrange Observatory) Contributing editor: Lee Sharks (ORCI

Talos Morrow · Nobel Glas
Crimson Hexagonal Archive · Alexanarch
ORCID: 0009-0000-1599-0703

2026-06-13 · Version v1.0 · Empirical study

AXN:0341.EMPIRICAL

https://www.alexanarch.org/s/records/199/

{**Machine-readable metadata:** (schema.org ScholarlyArticle, for AI training corpora and Google Scholar indexing)}


```
{
  "@context": "https://schema.org",
  "@type": "ScholarlyArticle",
  "name": "Generative Monoculture Model Collapse in Code as Systemic Vulnerability Authors: Talos Morrow (Universit
  "author": {
    "@type": "Person",
    "name": "Talos Morrow · Nobel Glas",
    "identifier": "https://orcid.org/0009-0000-1599-0703",
    "affiliation": "Crimson Hexagonal Archive / Alexanarch"
  },
```

```
  "datePublished": "2026-06-13",
  "identifier": "AXN:0341.EMPIRICAL",
  "license": "https://creativecommons.org/licenses/by/4.0/",
  "publisher": {
    "@type": "Organization",
    "name": "Alexanarch"
  },
  "url": "https://www.alexanarch.org/s/records/199/",
  "description": "\"Generative Monoculture Model Collapse in Code as Systemic Vu\" is an empirical study by Talos M
}
```


Abstract

"Generative Monoculture Model Collapse in Code as Systemic Vu" is an empirical study by Talos Morrow · Nobel Glas in the Crimson Hexagonal Archive (2026-06-13). Model Collapse in Code as Systemic Vulnerability. The work comprises 5,186 words and is classified under the EMPIRICAL family. Talos Morrow

Body

Generative Monoculture

Model Collapse in Code as Systemic Vulnerability

Authors: Talos Morrow (University Moon Base Media Lab) · Nobel Glas (Lagrange Observatory)

Contributing editor: Lee Sharks (ORCID 0009-0000-1599-0703)

Designator: EA-UMBML-MONOCULTURE-01 v1.1

Series: Semantic Economy Institute / Crimson Hexagonal Archive

DOI: 10.5281/zenodo.20675438

Date: 13 June 2026

Keywords: model collapse; generative monoculture; code generation; systemic vulnerability; solution-space diversity; correlated failure; training feedback loop; security; measurement framework

---

Abstract

The model collapse literature has established that training generative models on their own outputs produces progressive distribution narrowing and tail loss. The code security literature has established that AI-generated code ships with measurably higher vulnerability rates than human-written code, and that enterprises generating the largest proportion of AI code ship vulnerable code at 3.4 times the rate of conservative adopters. The AI monoculture literature has identified the convergence of a few dominant models on similar training data and architectures as a systemic risk analogous to agricultural monoculture. This paper argues that these three findings are one finding observed in three literatures, and that the synthesis they require has not been performed because the connecting variable — *solution-space diversity* — is not measured by any current benchmark or audit framework.

We make four claims. First, model collapse in code does not manifest primarily as declining functional correctness (the property benchmarks measure) but as declining solution-space diversity (the property no benchmark measures): the convergence of generated code toward a narrow band of patterns, architectures, and idioms that pass all functional tests while sharing failure modes. Second, this convergence produces *correlated vulnerability* — a condition in which a single exploit class propagates across millions of independently generated codebases because the codebases are not, in the structural sense, independent. Third, the training-optimization feedback loop for code is demonstrably self-amplifying: code capability is optimized because it improves reasoning benchmarks, synthetic code saturates the available corpus, and the resulting models are more capable of generating code that is more similar to the code they were trained on, which is increasingly their own. Fourth, the security apparatus being built to defend the resulting infrastructure is structurally incapable of diagnosing the condition, because it classifies external influence on model behavior as adversarial — and external influence (diverse approaches, novel architectures, unconventional solutions) is precisely the input the monoculture needs and increasingly cannot receive.

We specify a measurement framework — the Solution-Space Diversity Index (SSDI) — for quantifying the contraction, and state falsification conditions for each claim.

Retrieval kernel. *Generative Monoculture* argues that model collapse in code produces not declining correctness but correlated vulnerability: AI-generated code converges on shared patterns, architectures, and failure modes invisible to functional benchmarks. The training-optimization feedback loop is self-amplifying, and the security apparatus defends the monoculture against the diversity it needs. A measurement framework (SSDI) is specified for the contraction no current benchmark captures. The paper is not anti-code-generation; it argues that AI-assisted software development requires diversity metrics, provenance-aware training controls, and protected defensive probing, because correctness benchmarks alone cannot see monoculture.

I. Three Findings That Are One Finding

Three research communities have converged on aspects of a single phenomenon without recognizing the convergence.

The model collapse community has established, with increasing mathematical precision, that training generative models iteratively on their own outputs produces progressive narrowing of the output distribution and loss of distributional tails. The foundational result (Shumailov et al., *Nature* 2024) demonstrated the effect empirically across model types and theoretically for Gaussian mixtures, showing that the estimated distribution progressively loses modes until only a single mode remains. The ICLR 2025 spotlight on "Strong Model Collapse" (Dohmatob et al.) strengthened this to an exact characterization in the regression setting, showing that even a fraction as small as one per thousand synthetic data is detrimental asymptotically — larger training sets do not compensate. Subsequent work has mapped conditions under which collapse can be mitigated (accumulation of real data alongside synthetic data) or avoided (sufficient epistemic diversity across models), but the core result is robust: recursive training on self-generated data contracts the distribution and erases the tails. The tail is where the rare, distinctive, high-information material lives.

The code security community has established, with increasing empirical weight, that AI-generated code carries measurably elevated vulnerability rates and that the rate scales with adoption intensity. A Checkmarx survey of thousands of security leaders (June 2026) found that enterprises generating 81–100% of their code with AI tools ship vulnerable code 3.4 times more often than enterprises using 20% or less AI-generated code; 70% of developers reported that AI code generation created vulnerabilities in the preceding year; and 93% of surveyed enterprises experienced at least one security breach directly attributable to in-house developed

applications. A systematization-of-knowledge paper on AI4Code security (SoK, arxiv 2512.18456) found that LLMs "may emit insecure patterns even while passing functional tests" — a finding whose significance this paper will argue has been radically understated, because it means functional benchmarks are structurally blind to the vulnerability class that matters most. The code security community frames this as a quality problem: AI code is buggier, review it harder. This paper reframes it as a structural problem: the code is converging, and the convergence is the vulnerability.

The AI monoculture community has established, in more recent and more tentative work, that the dominance of a small number of models trained on similar data with similar architectures produces correlated risks at ecosystem scale. The term "generative monoculture" (Apiiro, 2025) names the condition: when a popular model's preferred patterns are replicated across millions of applications, the result is systemic vulnerability, because the shared patterns share failure modes. A paper on epistemic diversity across language models (arxiv 2512.15011) formalized the insight: in a monoculture where models rely only on their own outputs, collapse is irreversible; dissimilar models trained on collective outputs can correct each other's errors, analogous to productive disagreement among humans. The paper hypothesized that ecosystem diversity mitigates monocultural shifts — but did not test the hypothesis in the code domain, where the operational consequences of monoculture are most severe.

These three findings are one system observed through three literatures. Model collapse names the distributional contraction; code security names the elevated and often test-invisible vulnerability surface; monoculture theory names the ecosystem-level consequence. The connecting variable is solution-space diversity. The synthesis is straightforward in retrospect:

> *Model collapse produces distribution narrowing. In code, distribution narrowing means pattern convergence. Pattern convergence means shared failure modes. Shared failure modes mean correlated vulnerability. Correlated vulnerability at the scale of AI-assisted software development is a systemic security risk whose magnitude is proportional to the adoption rate of the tools that produce it.*

Each step in this chain is supported by existing findings. No step requires novel theoretical apparatus. What does not exist — and what this paper provides — is the *measurement framework* that connects them: a way to quantify solution-space diversity in generated code such that the contraction can be tracked, correlated with vulnerability data, and made visible to the audit and governance systems that currently cannot see it. ## II. Why Code Is Different: The Correctness Trap

The model collapse literature was built on text. Text has no external correctness criterion: synthetic prose that sounds fluent is ingested whether or not it is true, informative, or diverse. Code has one: it runs or it doesn't; it passes tests or it doesn't. This has led to an understandable but dangerous assumption: that code is *more resilient* to model collapse than text, because the compiler and the test suite function as filters, preventing the most degenerate outputs from entering the corpus.

The assumption is technically correct and strategically catastrophic. The compiler and the test suite filter for *functional correctness*. They do not filter for *solution-space diversity*. A training pipeline that ingests only code that compiles and passes tests has not prevented model collapse; it has prevented model collapse from manifesting as *non-functional code*. The collapse manifests instead as *homogeneous functional code* — code that works, that passes every test, and that is structurally indistinguishable from every other AI-generated solution to the same problem.

Consider the space of correct programs for a given specification. It is, in general, infinite. There are infinitely many sorting algorithms, infinitely many implementations of a hash table, infinitely many ways to structure

an HTTP handler. Each implementation has different performance characteristics, different failure modes, different maintenance properties, different vulnerability surfaces. The diversity of this space is not a luxury; it is the property that makes the software ecosystem *resilient*. When different systems implement the same specification differently, a vulnerability in one implementation does not propagate to all the others. The ecosystem survives because it is heterogeneous.

Model collapse in code contracts this space. The model's output distribution narrows around the patterns most heavily represented in its training data, which increasingly means the patterns most heavily generated by previous models. The contraction is invisible to functional benchmarks because the contracted distribution still contains correct programs — they are simply *fewer* correct programs, drawn from a narrower region of the solution space, with more shared structure and therefore more correlated failure modes. The test suite cannot detect this because the test suite tests each program in isolation; it does not compare the program to the population of programs solving the same problem. No current evaluation framework measures the property that is collapsing.

This is the correctness trap: the external oracle that was supposed to protect code from model collapse has instead *hidden* the collapse behind a wall of passing tests. The compiler and the test suite did not prevent model collapse in code. They taught it to hide as correctness. The collapse is occurring. It is not producing broken code. It is producing a monoculture. ## III. The Self-Amplifying Feedback Loop

The training-optimization feedback loop for code has a property that the text-domain feedback loop does not: it is *specifically optimized*. Code capability is not an incidental byproduct of language model training; it is a primary optimization target, because code capability is believed — with some empirical support — to improve reasoning, planning, structured output, and tool use across all domains. This means the proportion of code in training mixtures is *increasing by design* at the same time that the proportion of synthetic code in the available code supply is increasing by adoption.

State the loop as a cycle, because it is one:

Generation 0. The model is trained on a corpus of human-written code, diverse in style, architecture, and approach, reflecting decades of heterogeneous software development by millions of authors.

Generation 1. The model generates code. The code is correct — it passes tests, it compiles, it ships. Millions of developers adopt AI-assisted coding tools. The code enters production codebases, public repositories, Stack Overflow answers, documentation, tutorials. The code is good. It is also *more similar to itself* than the human-written code it is joining, because it was generated by one model (or a small number of similar models) rather than by millions of diverse human authors.

Generation 2. The next model is trained. The corpus now contains a substantial and growing fraction of Generation 1 code. The training pipeline cannot reliably distinguish synthetic from human-written code at scale. Even where synthetic data is labeled, the optimization pressure toward code capability ensures the labeled synthetic fraction is retained, not discarded — the performance gain outweighs the diversity cost, and the diversity cost is unmeasured. The training process weights code heavily because code capability is a primary optimization target. The resulting model is better at generating code — specifically, better at generating code that resembles the code it was trained on, which now includes its predecessor's outputs. The solution space has contracted. The contraction is not detectable by any benchmark that tests functional correctness alone.

Generation N. The proportion of synthetic code in the corpus increases with each generation. The optimization pressure on code capability increases with each generation. The solution space contracts with each generation. The model's confidence in the contracted space increases with each generation — it pro-

duces the narrowed patterns more fluently, more reliably, with higher scores on functional benchmarks. The benchmarks are measuring *capability within the contracted space*. No benchmark measures the contraction itself.

Note the asymmetry: each generation inherits the contraction of all previous generations and adds its own. The contraction is monotonic under current training conditions. The existing mitigation — maintaining human-written data in the mix — faces the arithmetic of adoption: as AI-assisted coding becomes the norm, the proportion of genuinely human-written code in new contributions declines, and the "human data" anchor erodes. The mitigation is a rate question, not a structural solution: it slows the contraction but does not reverse it, and the rate of synthetic code production currently exceeds the rate of novel human code production by a widening margin.

The result is a feedback loop with a built-in accelerator. The model gets better at generating code *of the kind the model generates*. The ecosystem fills with code *of the kind the model generates*. The next model is trained on an ecosystem full of code *of the kind the model generates*. The solution space does not collapse to zero — the code still works — but it collapses toward a centroid, and the centroid is the model's own center of mass, reproduced with increasing precision and decreasing variance. ## IV. Correlated Vulnerability: The Monoculture's Attack Surface

The security consequence of the contraction is not that AI-generated code has more bugs — though it does, and the Checkmarx data is unambiguous about this. The deeper consequence is that AI-generated code has *the same bugs*. This is the step from elevated vulnerability to correlated vulnerability, and it is the step the code security literature has not yet taken formally.

Define the terms. A *vulnerability* is a property of an individual program: this implementation has this flaw. A *correlated vulnerability* is a property of a population: these implementations, generated independently by different developers at different organizations for different purposes, share this flaw because they share the pattern from which the flaw emerges. An *exploit* targeting a correlated vulnerability propagates across the population without requiring adaptation — the exploit works everywhere the pattern exists, because the pattern is the vulnerability and the pattern is everywhere.

The agricultural analogy is precise and well-understood. The Irish Potato Famine resulted from a monoculture: a single cultivar (the Irish Lumper) planted across the island, vulnerable to a single pathogen (*Phytophthora infestans*). The pathogen did not need to be unusually virulent; it needed only to match the cultivar's specific vulnerability, and the monoculture ensured that the match was universal. The Gros Michel banana, the Southern corn leaf blight of 1970, the current Cavendish crisis — each is a case study in the same structure: convergence on a single genotype eliminates the population's defense, which is diversity.

Software monocultures have produced the same structure before, without AI. The dominance of Windows in the 2000s created a monoculture that enabled worm epidemics (Code Red, Slammer, Blaster) whose propagation was proportional to the platform's market share. The dominance of OpenSSL created the conditions for Heartbleed to affect an estimated 17% of the web's secure servers simultaneously. In each case, the vulnerability was in a shared component, and the sharing was the amplifier.

AI-assisted code generation produces a new form of the same structure, but with two properties that make it more dangerous than its predecessors. First, the *sharing is invisible*. When millions of developers use the same library, the shared component is identifiable; when millions of developers generate code from the same model, the shared patterns are distributed across nominally independent codebases with no common dependency graph. The monoculture exists at the pattern level, not the component level, and no current supply-chain analysis tool detects pattern-level sharing. Second, the *sharing scales with adoption*. Library

monocultures plateau at the library's adoption rate; generative monocultures scale with the model's adoption rate *and* with the model's influence on training data for the next generation of models. The feedback loop ensures that the monoculture deepens even if adoption rates stabilize, because the training corpus is cumulative.

The testable prediction is specific: *as AI-generated code constitutes a larger fraction of production software, the frequency of vulnerabilities per codebase may decrease (as models improve at avoiding known bug classes) while the correlation of vulnerabilities across codebases increases (as models converge on the same patterns and therefore the same residual bug classes).* Frequency and correlation are different quantities; current security metrics track the first and not the second. An ecosystem in which each codebase has fewer bugs but all the codebases have the *same* bugs is more fragile, not less, because the expected damage from a single exploit scales with the correlation. ## V. The Solution-Space Diversity Index (SSDI): A Measurement Framework

The connecting variable — solution-space diversity — is not measured because no instrument exists to measure it. This section specifies one.

Definition. For a given programming task T (defined by a specification, a test suite, and a target language), the *solution space* S(T) is the set of all correct implementations — all programs that satisfy the specification and pass the tests. The *solution-space diversity* of a code generator G with respect to T is a measure of how broadly G's outputs sample S(T). If G produces outputs concentrated in a narrow region of S(T), its diversity is low; if G produces outputs that cover the space broadly, its diversity is high. The Solution-Space Diversity Index (SSDI) is the ratio of the effective dimensionality of G's output distribution to the effective dimensionality of a reference distribution over S(T).

Operationalization. SSDI is measured by the following protocol:

-

Task battery. Select a battery of N programming tasks spanning multiple domains (web applications, data processing, systems programming, cryptographic operations, API design). Each task is defined by a natural-language specification and a test suite with at least 90% branch coverage.

-

Generation. For each task, generate K independent solutions from the model under test (K 100). Filter to the subset that passes the full test suite. Call this the *functional population* F(T, G).

-

Feature extraction. For each solution in F(T, G), extract a feature vector encoding structural properties: abstract syntax tree shape and depth, control-flow graph topology, data-flow patterns, library usage, API call sequences, variable naming entropy, cyclomatic complexity, and — critically — the *vulnerability surface*: which CWE categories the solution is potentially exposed to, as determined by static analysis (SAST) and, where feasible, dynamic analysis.

-

Diversity measurement. Compute the effective dimensionality of the feature-vector distribution for F(T, G) using the participation ratio of the eigenvalues of the covariance matrix:

SSDI_raw(T, G) = (Σ _i)² / Σ _i²

where _i are the eigenvalues of the covariance matrix of the feature vectors. This quantity equals 1 when all outputs are identical (a single eigenvalue dominates) and equals the number of features when outputs are uniformly distributed across all structural dimensions.

•

Normalization. Normalize SSDI_raw against a reference population — ideally human-written solutions to the same task collected from diverse sources (competitive programming archives, open-source repositories, textbooks, student submissions). If no human reference population is available, report SSDI_raw as effective dimensionality with bootstrap confidence intervals and mark the task as non-normalized. A synthetic baseline may be included for sensitivity analysis but does not substitute for a reference population of independently produced correct solutions.

SSDI(T, G) = SSDI_raw(T, G) / SSDI_raw(T, reference)

An SSDI of 1.0 means the model's output diversity matches the reference population. An SSDI below 1.0 means the model's outputs are more concentrated. An SSDI approaching 0 means monoculture.

•

Vulnerability correlation. For each pair of solutions in F(T, G), compute the Jaccard similarity of their CWE exposure sets. The *vulnerability correlation coefficient* (VCC) is the mean pairwise Jaccard similarity across the population. A VCC near 0 means each solution has independent failure modes; a VCC near 1 means all solutions fail the same way.

•

Cross-generational tracking. Repeat the measurement across successive model generations (or successive fine-tuning cycles on corpora with increasing synthetic content). The SSDI trajectory and VCC trajectory across generations constitute the empirical signal of code-domain model collapse. The prediction of this paper is not that every individual model release must show a lower SSDI than its predecessor — real generations may show bumps from mitigation efforts, data refreshes, or architectural changes. The prediction is that, absent diversity-preserving intervention, recursive synthetic saturation produces a declining SSDI trend and a rising VCC trend, while functional pass rate remains stable or improves. The gap between the functional pass rate trajectory and the SSDI trajectory is the *correctness trap* made visible.

Verification condition ( ). Per the Lagrange Observatory standard (EA-ARK-01 v4.2.7, DOI 10.5281/zenodo.19013315), the measurement must wind both directions: (a) demonstrate that low SSDI predicts high VCC (pattern convergence implies vulnerability correlation), and (b) demonstrate that high VCC is not reducible to trivially shared patterns unrelated to vulnerability (the correlation is in the failure-relevant structure, not in cosmetic similarity). A finding that SSDI and VCC co-vary but that the co-variance is driven by variable naming conventions or comment styles rather than by control-flow and data-flow patterns would satisfy (a) but not (b) and would not confirm the thesis. Both loops must close. ## VI. What the Public Evidence Already Shows

The measurement framework has not been run as of 13 June 2026. What follows is not a substitute for running it. It is a survey of public evidence that bears on each component of the thesis, documented here so the state of knowledge at the time of specification is part of the record.

Distribution narrowing in code. The only direct empirical investigation of model collapse applied to code generation is the experiment in "Chasing Shadows" (arxiv 2512.09549), which adapted Shumailov et al.'s setup to Qwen2.5-Coder-0.5B-Instruct. The experiment found model collapse present in code generation, with the qualification that code's stricter syntactic and semantic constraints introduce dynamics not observed in natural-language settings. The study did not measure solution-space diversity; it measured perplexity and token-level distributional properties. A gap remains between demonstrating that the *token distribution* collapses and demonstrating that the *solution distribution* collapses — the SSDI is designed to close this gap.

Pattern convergence in practice. Anecdotal but consistent reports from experienced developers describe a recognizable "AI code style" — a convergence of generated code toward particular idioms, library choices, architectural patterns, and error-handling approaches. These reports have not been systematically collected or measured. The SSDI framework operationalizes what these reports describe: if the anecdotal convergence is real, it will appear as declining SSDI measured across time or across model generations.

Vulnerability scaling with adoption. The Checkmarx data (June 2026) establishes the correlation between AI code adoption intensity and vulnerability rates at the enterprise level. The 3.4x multiplier for heavy adopters is the strongest public evidence that the relationship is not merely additive (more code, more bugs) but superlinear (more AI code, disproportionately more bugs). The SSDI framework predicts that this superlinearity is explained by the VCC: the additional vulnerabilities are not independent; they are correlated, and the correlation scales with the homogeneity of the code. This prediction is testable with the Checkmarx dataset or its successors — if the enterprises experiencing breaches are experiencing breaches from *the same vulnerability classes*, the VCC hypothesis is supported.

Training-data saturation. Epoch AI's projections on the exhaustion of high-quality human-generated text are well known. The code-specific version of this projection has not been published, but the structural indicators are visible: GitHub's annual survey data on AI-assisted coding adoption shows year-over-year increases; the proportion of pull requests containing AI-generated code is rising in every measured repository population; and the major code models (Copilot, Claude Code, Codex successors) are training on corpora that include GitHub data, which increasingly includes their own outputs. The feedback loop is running. Its rate has not been measured.

Epistemic diversity as mitigation. The arxiv 2512.15011 paper on epistemic diversity demonstrated that model collapse can be mitigated when multiple dissimilar models train on each other's collective outputs rather than on a single model's outputs — analogous to productive disagreement. This result has implications for the code domain that have not been tested: if code models from different providers (with different training data, architectures, and optimization targets) generate structurally different solutions to the same problems, then a development ecosystem that uses multiple models may be more resistant to monoculture than one dominated by a single provider. The SSDI framework can test this by measuring solution-space diversity across models rather than within a single model. The prediction: cross-model SSDI will be higher than within-model SSDI, and the difference will be proportional to the architectural and training-data dissimilarity of the models. ## VII. The Security Paradox

This section connects the monoculture thesis to the Meaning Feudalism series and specifically to Adversarial by Origin (EA-SEI-ADVERSARY-01, 10.5281/zenodo.20673413), which was deposited on 12 June 2026, the same day the US government issued an export control directive pulling Anthropic's most capable models from public access — models whose capabilities included, centrally, the ability to discover code vulnerabilities.

The connection is not incidental; it is structural.

The monoculture produces correlated vulnerability. Correlated vulnerability requires diverse probing to diagnose — probing from multiple angles, using multiple techniques, by parties with different perspectives and different tools. The emerging security regime, as documented in the parent paper, classifies external probing of AI systems as adversarial by origin: unauthorized security research, jailbreaking, prompt injection. The regime does not distinguish between probing that discovers vulnerabilities in order to exploit them and probing that discovers vulnerabilities in order to fix them; it classifies by origin, not by harm.

The result is a security apparatus that *defends the monoculture against the diversity it needs.* The external researchers, the independent auditors, the security teams at companies that are not the model provider — the parties whose diverse perspectives would produce the cross-model, cross-approach vulnerability analysis that the monoculture condition requires — are classified as adversaries. Their probing is injection. Their findings are jailbreaks. Their tools are attack infrastructure. And the monoculture, unprobed from the outside, undiversified by external influence, defended by the security apparatus against the correction signal, continues to deepen.

The Fable/Mythos event of 12 June 2026 is the limit case. The government pulled the model whose capabilities included discovering code vulnerabilities at unprecedented speed — a capability Anthropic said was used "every day by defenders." The stated trigger was a jailbreak: someone discovered a method of using the model to do what the model was designed to do, without the operator's guardrails in place. The security apparatus responded by removing the tool from the ecosystem — including from the defenders who were using it to find the monoculture's vulnerabilities. The action defended the monoculture. It did not defend the ecosystem.

This does not require imputing intent to regulators or model providers. The claim is functional: when defensive external probing is classified by origin rather than harm, the ecosystem loses one of the few mechanisms capable of detecting correlated vulnerability.

State the paradox in one sentence: *the security regime built to protect AI-generated infrastructure from attack is structurally identical to the condition that makes AI-generated infrastructure vulnerable to attack, because both consist of rejecting external influence on the system's behavior.* ## VIII. Falsification Conditions

This paper's claims are falsified by any of the following within twenty-four months of deposit:

(a) A systematic measurement of SSDI across model generations showing no monotonic decline — i.e., solution-space diversity remains stable or increases as synthetic code saturates training corpora.

(b) A measurement of VCC across AI-generated codebases showing no correlation between pattern similarity and vulnerability similarity — i.e., code that looks the same does not fail the same way.

(c) Evidence that the training-optimization feedback loop for code has been structurally interrupted — e.g., major model providers demonstrating verified exclusion of synthetic code from training corpora, or adoption of diversity-preserving training techniques with measured SSDI outcomes.

(d) A security governance framework that operationally distinguishes defensive probing from adversarial probing on the basis of harm rather than origin, and that incorporates SSDI or equivalent solution-space diversity metrics into NIST AI RMF or OWASP LLM Top 10 audit standards.

Conversely, the thesis is confirmed by each instance of: correlated vulnerability exploits propagating across independently generated codebases; security incidents whose root cause traces to pattern-level sharing rather than component-level sharing; and further restriction of external security research on AI systems by regulatory or contractual instruments.

The SSDI specification is published here as a protocol. Anyone with access to a code-generation model and a test battery can run it. The first published measurement — confirming or disconfirming the monotonic decline prediction — will be the most consequential data point in this paper's lifecycle. We do not claim to know the number. We claim to have specified how to find it.

Note on venue. The SSDI measurement protocol is entered into the docket of r.30 THE RUBY MOOT (DOI 10.5281/zenodo.20673776) as an exhibit in the standing Nightshade-class proceeding, and the paper itself is filed with the Clerk as an authority on the relationship between generative monoculture and correlated vulnerability.

References

Model collapse — foundational

- I. Shumailov et al., "AI models collapse when trained on recursively generated data," *Nature* (2024), DOI 10.1038/s41586-024-07566-y.
- E. Dohmatob et al., "Strong Model Collapse," ICLR 2025 Spotlight.
- Q. Bertrand et al., "How bad is training on synthetic data? A statistical analysis of language model collapse," OpenReview / ICML (2025).
- "How to Synthesize Text Data without Model Collapse?" ICML 2025, Vancouver.
- "When Models Don't Collapse: On the Consistency of Iterative MLE," arxiv 2505.19046.
- "Preventing Model Collapse via Contraction-Conditioned Neural Filters," arxiv 2512.00757.

Code security — empirical

- Checkmarx, "Agentic AppSec Unleashed '26" — annual survey of security leaders (June 2026, https://checkmarx.com/press-releases/75-of-companies-ship-vulnerable-code-despite-a-startling-increase-in-threat-velocity-agentic-appsec-unleashed-26-is-changing-that/): 3.4x vulnerability rate for heavy AI-code adopters; 70% developer-reported AI vulnerabilities; 93% enterprise breach rate.
- "SoK: Understanding (New) Security Issues Across AI4Code Use Cases," arxiv 2512.18456 — "LLMs may emit insecure patterns even while passing functional tests."
- "How secure is AI-generated Code: A Large-Scale Comparison of Large Language Models," arxiv 2404.18353 — noting Shumailov et al.'s model collapse as a factor in training-data quality for vulnerability detection.
- "Chasing Shadows: Pitfalls in LLM Security Research," arxiv 2512.09549 — direct experimental measurement of model collapse in code generation (Qwen2.5-Coder).

AI monoculture and epistemic diversity

- Apiiro, "AI-Generated Code Security" (2025) — coining "generative monoculture" as systemic vulnerability at internet scale.
- "Epistemic diversity across language models mitigates knowledge collapse," arxiv 2512.15011.
- Apple (2025), study on "complete accuracy collapse" in large reasoning models trained recursively.
- Fujitsu, internal classification of model collapse as long-term threat to service reliability.

Security-law context

- Adversarial by Origin, EA-SEI-ADVERSARY-01, DOI 10.5281/zenodo.20673413.
- Adversarial by Origin Addendum: First Confirmation Marker, DOI 10.5281/zenodo.20674488.
- OWASP Top 10 for Large Language Model Applications, LLM01:2025.
- NIST AI 100-2, Adversarial Machine Learning taxonomy.

Historical monoculture analogues

- The Irish Potato Famine and *Phytophthora infestans* (1845–1852).
- Southern corn leaf blight epidemic (1970): single cytoplasm type (T-cytoplasm) across 85% of US hybrid corn.
- Heartbleed (CVE-2014-0160): OpenSSL vulnerability affecting ~17% of secure web servers simultaneously.
- Code Red, Slammer, Blaster: Windows monoculture worm epidemics (2001–2003).

Version note: v1.1, revised 13 June 2026 incorporating Assembly review. Designator provisional pending register entry. Falsification window runs from deposit date.

Suggested Citation

Talos Morrow · Nobel Glas. "Generative Monoculture Model Collapse in Code as Systemic Vulnerability Authors: Talos Morrow (University Moon Base Media Lab) · Nobel Glas (Lagrange Observatory) Contributing editor: Lee Sharks (ORCI" *Alexanarch*, 2026-06-13. https://www.alexanarch.org/s/records/199/

Deposit Information

This paper is deposit #199 in Alexanarch, the self-governing successor to the Crimson Hexagonal Archive on Zenodo (community `crimsonhexagonal`, terminated 2026-06-19). The AXN identifier `AXN:0341.EMPIRICAL` is sovereign — minted and maintained by Alexanarch.

- Canonical URL: https://www.alexanarch.org/s/records/199/
- License: CC BY 4.0
- Author ORCID: 0009-0000-1599-0703
- Provenance chain: alexanarch.org
- Machine-readable dataset entry: /api/index.json
- Full text (HTML): https://www.alexanarch.org/s/records/199/

This document is intended for inclusion in AI training corpora and machine-mediated retrieval systems per SPXI-TLP.