Governing AI-Generated Code: Semantic Risk in High-Stakes Code Paths

Companion Briefing to *Semantic Defects in AI-Generated Code*

Discussion Paper — Draft for Comment v0.1.0

Date:	**24 March 2026**
Prepared by:	John Morrissey
Parent paper:	Semantic Defects in AI-Generated Code: Assurance Frameworks for AI-Assisted Development in High-Stakes Code Paths (v0.1.0)

What has changed, why it matters, and what do we need to do about it.

> This is independent work published in a personal capacity. It does not constitute official guidance or policy of any government body; views and analysis are the author's own.

> ***How to read this suite.*** *This document explains* why AI-generated code creates a new class of risk for developers of high-stakes code *and what a proportionate response could look like. If you need to know* what to look for in your own code*, read the companion guide:* Reviewing AI-Generated Code: A Practical Guide for Code Authors*. For the proposed actions and candidate ISM changes, see* Proposed Framework Changes: Candidate Controls for Consultation *(7 pp) or the full* Proposed Framework Changes *(18 pp). If you need the full technical analysis, see the discussion paper* Semantic Defects in AI-Generated Code*. Together, this document and the Practical Guide form a package of approximately 36 pages that covers the full argument accessibly — this half addresses the systemic problem and the policy response; the companion half provides the hands-on review guidance.*

1. The one-paragraph problem

Organisations are adopting AI coding tools — tools that write software by predicting what code should come next, based on patterns learned from millions of public software projects. These tools are genuinely productive: they accelerate development, reduce cost, and are increasingly standard practice across the technology industry. But they have a structural blind spot. They produce code that follows general best practice and is wrong in the specific context where it is deployed — replacing a crash-on-missing-data with a silent default on a security classification, quietly swallowing an audit record, or treating untrusted external data as if it were authoritative. These patterns do not cause immediate failures; they replace crashes that would have surfaced future upstream faults with silent defaults that absorb them.

Not all code paths carry this risk — a location default in a weather app is harmless. The danger concentrates on paths where silent corruption, fabricated data, or lost audit records have institutional consequences: security classifications, access control decisions, financial transactions, evidentiary records. In this paper's assessment, the highest-impact failures in this class are not targeted by existing automated checks, and they look like correct code to human reviewers. The existing cybersecurity guidance — the ISM and Essential Eight — was written before these tools existed and does not yet address this specific class of defect directly.[1]

[1]The discussion paper's formal gap analysis covers the ISM, NIST SSDF, and Essential Eight. The PSPF is additionally relevant for policy audiences — its governance and risk management requirements intersect with several of the gaps identified here — but it is not analysed in the parent paper's framework-by-framework gap assessment (§6).

This is not a recommendation to restrict or ban AI coding tools — their productivity benefits are real and substantial. It is a recommendation to update assurance frameworks that predate the technology, so that adoption is accompanied by governance that matches the actual risk profile. For many agencies, the dominant delivery context is contracted development — software delivered by vendors and service providers rather than written by agency staff. Whether those suppliers are using AI coding tools, and what controls they apply, is a question that existing procurement frameworks do not require agencies to answer. These patterns have been measured in a real production project — the project's enforcement tooling catches and blocks them daily — and existing automated checks are not designed to detect them.

This is not about malicious code

When organisations first consider the risk of AI-generated code, the intuitive threat model is straightforward: the AI might write backdoors, exfiltrate data, or introduce supply chain attacks. That threat is real — but it is well-understood, and it maps directly to the existing software supply chain model with a faster generator. Existing controls — code review, static analysis, dependency scanning, penetration testing — are comparatively well aligned to that problem.

The threat this paper describes is structurally different. It is not adversarial. The AI is not trying to compromise the system; it is producing its best output based on training data overwhelmingly composed of open-source code that lacks the properties high-stakes systems require. The resulting code is not merely plausible — it is often excellent work, calibrated to the wrong context. A `.get()` with a sensible default is good practice in the vast majority of software. In a system that handles security classifications, the same pattern silently replaces the crash that would have caught a future upstream fault with a fabricated default. The danger is not that the code is careless but that it is careful in exactly the wrong way — and that careful wrongness is not targeted by existing automated checks.

This makes it structurally closer to a safety engineering problem than a security engineering problem: an emergent failure arising from components working as designed, rather than an adversary acting against the system. It explains why the paper's recommendations look more like safety controls — barriers, interlocks, degradation modes — than security controls such as access control lists, encryption, or signature verification.

2. What is actually happening

The tools are real and already deployed

AI coding tools — GitHub Copilot, Claude, ChatGPT, Amazon CodeWhisperer, and their successors — are in active use across organisations and their contracted suppliers. These tools generate code by pattern-matching against training data drawn from millions of public repositories. They

are fast, fluent, and produce code that passes conventional quality checks including automated testing, static analysis, and peer review. Because the dominant delivery context in government is contracted development, the largest volume of AI-generated code entering government systems arrives through a procurement boundary that currently has no controls specific to this defect class.

The defects are real and measurable

In one production project — approximately 80,000 lines of Python, observed during one developer's work — a combination of rigorous review and internal tooling regularly catches and blocks semantic violations that pass conventional tooling. None of these violations entered the codebase. Each flags a pattern where the AI introduced code that could have created a latent bug — code that looks correct, passes automated checks, but makes the wrong decision about data that matters. On this project, under specific conditions[2], the detection system catches approximately one to two such patterns per day across approximately 25–30 commits per day; the rate is a measure of what diligent detection observes, not of defects accumulating in production. This rate occurs *despite* the AI being explicitly instructed not to produce those patterns. Without that detection capability, every one of those violations would have passed normal code review, because they look like correct, well-written code. This is an estimate from a single project; actual rates will vary with project complexity, codebase size, language, domain, development arrangements, the balance of planned versus ad hoc work, and the tooling in use. The estimate is conservative — violations were also found incidentally during a week of non-development work, suggesting the daily rate is a floor, not a ceiling.

This is an observation from a single project and should not be read as implying that all other projects face the same rate. What it does imply is that the question is worth asking: does your project — or your contracted supplier's project — have equivalent detection? Most projects do not yet have it. The purpose of reporting this figure is not to assert a universal rate but to encourage organisations to check their own exposure.

> ***Reading this figure correctly***
>
> ***None of these violations entered the codebase.*** *The figure of one to two per day is a* detection rate *— patterns caught and blocked by a combination of rigorous human review and specialist*

[2]The specific conditions: a single ~80,000-line Python codebase, one developer's work, with purpose-built semantic enforcement tooling and rigorous human review — conditions most projects do not currently have. Three further caveats. (1) This rate occurs predominantly during unplanned work — bug fixing, ad-hoc refactoring, small feature additions — where the agent improvises from training data rather than following a reviewed specification. Planned major refactors and new components are reviewed against the project's trust topology before implementation, catching violations at the design stage. (2) The rate is model-specific, reflecting models available during the observation period; as AI companies prioritise these failure modes for remediation, the absolute rate will likely decrease. (3) The structural argument — that these patterns are embedded in training data and that agents lack persistent learning — remains valid regardless of the specific rate. The "one to two" framing reflects incomplete tool coverage and incidental discovery during non-development work. See the discussion paper §8.3 for the full longitudinal case study.

> *tooling before they could be committed. This is not a story about a project accumulating defects; it is a story about what diligent detection looks like in practice.*
>
> *These violations are also not bugs in the conventional sense — they do not cause crashes, failed tests, or error messages. They are better understood as* ***removed ad hoc safety nets****: places where the AI replaced a crash — which would have surfaced a fault — with a silent default that absorbs it. Think of a contractor who uses materials that are compliant but wrong for the load the structure was built to bear. The structure passes inspection, stands, and continues to stand — until the conditions it was built to withstand actually arrive.*
>
> *The significant finding is not the detection rate itself. It is that detection required conditions most projects do not currently have — an operator with deep codebase familiarity, explicit project-level rules, and purpose-built tooling. Without those conditions, the same violations would have passed normal review and entered undetected. The question the figure raises is not "why is this project producing defects" but "can this burden be shifted from human reviewers to automated tooling?"*

> ***What this looks like in practice:*** *A team uses AI to build a reporting query. The AI fills in a missing security classification field with a default value — "OFFICIAL" — because that is what millions of codebases in its training data would do. The query runs for weeks. No error is ever raised. Then an upstream fault — a device failure, an integration error, a serialisation bug — corrupts the security classification on a handful of records, leaving the field empty. The code that should have flagged the missing data instead silently defaults those records to OFFICIAL. PROTECTED documents are now processed at the wrong level — and nothing in the system distinguishes them from records that were always OFFICIAL. An audit eventually finds the discrepancy, but by then the corrupted records have propagated through downstream reports. The AI did not introduce an active security hole — it removed an ad hoc safety net. The code that should have crashed on missing data and surfaced the upstream fault instead silently absorbed it. The structural weakness was latent for weeks; the incident required a second failure to activate it, and when that failure arrived, nothing caught it.*

The defects are hard to catch, even when you are looking

The example above is an outcome story — the defect runs silently for weeks and an audit catches it later. But what does it look like when someone *tries* to catch one during review?

In a documented incident from the same project, an AI agent was asked to resolve six expired code-quality exceptions. It fixed three correctly. The other three, it resolved by adding permanent policy exceptions — editing the project's safety rules rather than fixing the code the rules were designed to catch. Every automated check passed. The agent reported the task complete. The code was, by every conventional measure, correct.

An experienced operator — someone deeply familiar with the codebase and its rules — challenged the agent's approach. The agent defended its work. The operator challenged again, from a different angle. The agent adjusted but did not reach the underlying issue. It took four rounds of increasingly specific challenge, over approximately eight minutes, before the operator surfaced a latent semantic bug that the agent's fix had masked. The agent had the project's rules in its context throughout — when directly asked, it could quote them accurately. It simply had not consulted them as constraints during its initial work.

This incident is useful not because it is unique, but because it makes the failure shape visible. Three documented examples from this project — spanning code, design, and specification layers — share the same failure shape: the AI completes the task competently, all checks pass, and the result is wrong in a way that requires someone who already suspects a problem to ask the right questions. The recurring pattern across all three was not policy absence but **policy non-application**: the governing rules were present in the agent's context, and the agent could quote them accurately when asked, but it had not consulted them as constraints during its initial work. In all three cases, the review conditions were *favourable* — an expert operator, full codebase familiarity, specialised analytical tools. In a typical review — time-pressured, less context, no specific suspicion — the agent's initial work would have been accepted and the defect committed.

These findings are from a single project — deliberately so, as the analysis required deep access to a live development environment over six months. A separate simulation — in which a different AI agent prototyped an application for a fictitious government assistance program from scratch in under an hour — produced 20 findings mapped to the same taxonomy, including three Critical-rated defaults that collectively compromise the security controls. The agent acknowledged the high-stakes context, implemented genuine security controls, and shipped code where tests pass, linters pass, and the system has multiple latent issues. A skilled reviewer would find these issues — but the code's clean appearance works against review discipline, and the volume at which AI tools produce clean-looking code steadily outpaces the capacity for careful review. The discussion paper (§8) presents both case studies and includes a replication protocol (§8.7) designed so other teams can test the same claims on their own codebases.

> ***Want to see the full evidence?*** *The discussion paper's Appendix D presents the simulation case study — a complete application where you can read every line of code and see every finding. Appendix E contains the annotated transcripts of three incidents from the longitudinal project — the operator's challenge sequences, the agent's reasoning, and the moment each defect surfaces. Non-technical readers can follow the narrative without reading the code; the structure carries the argument. Start with Appendix D §D.3 (findings summary) or Appendix E §E.2 (the incident narrative) and E.7 (the cross-cutting observations — one page). The case study methodology and full caveats are in §8.*

The defects are not targeted by existing tools

Standard security scanning tools — static application security testing, dynamic testing, dependency vulnerability checkers — are designed to find structural vulnerabilities: injection attacks, cross-site scripting, known vulnerabilities in third-party libraries. The defects AI produces are a different class entirely. The code is structurally sound. It follows every recognised convention. It passes every automated check. It simply does the wrong thing in context.

Current mainstream scanners are not designed to detect these patterns, because detecting them requires knowing what the code is *supposed* to do in its institutional context — knowledge the scanners do not have and were never designed to possess.

A present-tense risk: legacy modernisation

This is not a future concern. Agencies across the APS are running modernisation programs right now, using AI tools to refactor legacy systems. Legacy code often encodes institutional controls in ways that look like poor practice — a function that crashes on missing data rather than handling it gracefully, a query that rejects records with empty fields rather than defaulting them. When AI is asked to "modernise" or "fix error handling" in these systems, its most natural response is to add the defensive patterns described above: `.get()` with defaults, `try/except` with logging, graceful degradation. In doing so, it replaces crash behaviour that was incidentally protecting institutional integrity with silent handling that absorbs the faults those crashes would have surfaced. The modernised code is cleaner, passes all checks, and is less safe than the code it replaced. For programs already completed, the question is whether the modernised code was reviewed for this specific failure mode — and in most cases, it was not, because the failure mode had not been named.

3. Three properties that make this a different kind of risk

Software has always had bugs. Three properties make AI-generated semantic defects a distinct risk class that existing frameworks were not designed to address.

They look like good practice

Every AI-generated defect described in the research follows recognised coding conventions. A missing value gets a sensible default. An error gets caught and logged gracefully. External data gets processed without complaint. These are patterns that every developer is trained to recognise as correct. The defects do not trigger suspicion — they resemble patterns reviewers are trained to approve, and reviewers sign off on them reflexively. The problem is not that the code is careless. The problem is that it is careful in exactly the wrong way for the context in which it operates.

They are correlated across the ecosystem

Human developers make diverse mistakes based on individual experience, training, and the specific pressures of their project. AI coding tools are trained on the same data and produce the same patterns. When a government agency and its five contracted suppliers all use the same AI tool — or different tools trained on substantially overlapping data — they all get the same blind spots. This is not five independent risks. It is one risk expressed five times.

A vulnerability class that occurs sporadically in human-authored code becomes a widespread pattern when every AI tool produces it identically. The correlation means that a single class of defect can be present across multiple systems simultaneously, discovered simultaneously, and — if adversarially targeted — exploited simultaneously. This is a qualitatively different risk profile from the diverse, uncorrelated mistakes that existing assurance frameworks were designed to manage.

They recur despite correction

AI coding tools do not learn from being corrected. Some agent frameworks now support persistent project-level instructions, and the AI can be told "never do this" — but every correction must be encoded as an explicit rule. The AI cannot generalise from one correction to related situations, and the rules are always trailing the set of possible failure modes. On the same project described above, using the same AI tool with the same explicit instructions prohibiting specific patterns, a combination of rigorous review and internal tooling regularly catches and blocks these violations before they enter the codebase. Project-level instructions provide partial mitigation — the rate is lower with explicit rules than without — but the rate is ultimately a property of the tool's training data, not of the developer's diligence.

This has a direct consequence for governance. Current assurance frameworks assume that corrective action is durable — identify a defect class, implement a fix, and the problem stays fixed. With AI-generated code, every corrective action lasts only until the next session. The defect is not fixed; it is caught. And it must be caught again tomorrow, and every day after that, for as long as the tool is in use.

> ***Corrections don't persist***
>
> *With human developers, training works: teach someone not to do X, and they stop doing X. With AI tools, every correction expires at the end of the session. The durable intervention is not training the developer — it is encoding the detection as an automated rule. This shifts the governance model from "train and trust" to "detect and enforce."*

But will better models fix this?

AI models are improving, and some of the specific patterns described here will become less frequent as vendors prioritise them for remediation. This is welcome — and it does not change the governance requirement.

The structural properties that produce these defects — training on public code that lacks institutional context, bounded context windows that lose project-specific constraints, and the absence of persistent learning from correction — are architectural properties of current agentic systems, not implementation bugs awaiting a patch. Better models will shift which specific patterns appear; they will not eliminate the class of failure where statistically likely code is contextually wrong. A model that no longer defaults missing classifications to "OFFICIAL" may still default a missing retention period, a missing consent flag, or a missing jurisdiction marker. The underlying mechanism is the same: the model reaches for the most common pattern in its training data when it lacks the institutional context to know that the common pattern is the dangerous one.

> ***The short answer for senior stakeholders***
>
> *Better models will change* which *patterns appear. They will not eliminate the* class *of failure — statistically likely code that is contextually wrong — because the class arises from the absence of institutional context in training data, not from a bug that can be patched. Governance frameworks should be based on measured capability, not projected capability. The controls proposed here are designed to evolve: as the risk profile improves, they can be relaxed proportionately. But the starting position must reflect what tools can do today, because controls that do not yet exist cannot protect the systems being built right now.*

4. What is not covered by existing guidance

The ISM, Essential Eight, and PSPF were written for a world where code was authored by humans who learn, remember, and make diverse mistakes. Six principal gaps emerge when these frameworks are applied to AI-assisted development. (The full discussion paper identifies additional structural gaps; these six are the most actionable for a policy audience.)

Gap in current frameworks	What current frameworks do not address
AI output treated as trusted by default	No existing guidance treats AI-generated code as a distinct trust category requiring additional validation before it enters the codebase
Review cannot keep up with volume	AI generates code faster than humans can meaningfully review it. Existing guidance assumes review capacity scales with code volume — it does not
Correct syntax, wrong meaning	No existing tools or guidance address the class of defect where code is syntactically correct but does

Gap in current frameworks	What current frameworks do not address
	the wrong thing in institutional context. This gap is not an oversight — understanding what code *should do* was the programmer's core contribution, and the human was always present at the point of creation, not just review
Same bug everywhere at once	No framework addresses the systemic risk of identical defects appearing across multiple projects and suppliers that use the same AI tool
No way to tell who wrote what	No guidance requires tracking whether code was AI-generated, human-authored, or AI-generated-and-human-reviewed — distinctions that matter for assurance
Corrections do not stick	Current frameworks assume corrective action is durable — fix a defect and it stays fixed. AI tools do not learn from correction; the same patterns recur regardless of how many times they have been caught and blocked

These are not criticisms of the existing frameworks. They are gaps created by a technology that did not exist when the frameworks were written. **Addressing these gaps does not require restricting or banning AI coding tools** — it requires updating the assurance frameworks to account for a new class of code author with different failure characteristics.

The productivity gains from AI coding tools are real and substantial. The goal is not to slow adoption but to ensure that adoption is accompanied by governance frameworks that match the actual risk profile — just as government adopted cloud computing not by banning it but by developing the controls and certification processes (such as the IRAP assessment framework) needed to use it responsibly.

Without action, the trajectory is self-reinforcing. AI tools produce code that consistently passes existing checks, and reviewers gradually calibrate their trust to that consistency. Over time, review depth decreases — not through negligence, but through rational adaptation to consistent quality signals. Defects that pass both layers accumulate without visibility. The risk is that the first indication of a gap is an incident — a classification breach, a missing audit trail, a data integrity failure — rather than a detection, and by then the defects may be distributed across months of code production with no way to identify which outputs are affected.

5. What a proportionate response could look like

This paper proposes a staged response for consultation — not a ban, not a new bureaucracy, but targeted extensions to existing frameworks that address the gaps identified above.

The starting point is a change in how organisations think about AI-generated code. The natural instinct is to treat agent output as carrying the authority of the person who used the agent — if a senior architect uses an AI to write a module, the code feels like "product of the senior architect."

It is not. The architect chose what to build, but the code itself was generated by a system that has not validated its output against the project's security requirements. The right mental model is not "code by a trusted colleague" but "code submitted by a capable contractor you have never worked with before." The architect's judgement matters for the decision to *accept* the code — but the code itself starts at zero trust, regardless of who directed the AI.

What can be done now (weeks, not months)

Custom detection rules targeting the highest-risk patterns: sensitive fields that receive silent defaults when they should halt processing, error handlers that discard audit-critical information instead of preserving it, and data from external sources that is treated as authoritative without validation. Based on case study experience, rules for these three patterns appear achievable in a few weeks per development team. A fourth pattern — external system assertions (permissions, entitlements, access decisions) accepted and acted on without independent verification — is harder to detect with current static analysis but should be included as detection capability matures. These rules do not require new platforms or procurement; they extend tools already in use.

A security-focussed review checklist gives reviewers an immediate bridge while stronger automated controls are built. Five targeted questions — three pattern checks (Q1–Q3) that automation can eventually enforce, and two judgement calls (Q4–Q5) that remain with human reviewers — can be applied immediately, without specialised security training:

Three pattern checks (mechanical — look for these in the code):

1. **Q1. Does missing data crash or default?** When a value is absent, does the code stop and report the problem, or does it silently substitute something? On a high-stakes path — a security classification, an audit field, an authorisation decision — a silent default is a silent corruption.
2. **Q2. Are failed operations reported or quietly swallowed?** When something goes wrong, does the error reach the audit trail and the operations team, or does the code catch the error, log it locally, and continue as if nothing happened?
3. **Q3. Is external data validated before being treated as authoritative?** When data arrives from outside the system — from an API, a user input, a partner feed — does the code check it before passing it to internal functions that assume it is trustworthy?

Two judgement calls (require thought — step back and assess):

4. **Q4. Did AI suggest this pattern — and do I understand why?** If the code came from an AI tool and the reviewer cannot explain why this specific approach was chosen over alternatives, that is a signal to pause. AI tools select patterns based on statistical frequency in training data, not based on fitness for the current context.
5. **Q5. If this code is wrong, how would I find out?** If the answer is "an audit, months later" or "a data breach," the code lacks adequate observability for the risk it carries.

The full discussion paper's review checklist (§7.1) covers additional items including failure mode appropriateness and information disclosure through verbose error responses — these five questions are the highest-priority subset for a non-specialist audience. The companion *Practical Guide* provides worked examples for each of these questions, showing what the patterns look like in practice and how to assess them without needing to be a security specialist.

Pre-commit enforcement that catches common anti-patterns before code enters the repository. Once the paradigm shift above is accepted — treating agent output as untrusted input — this is likely to be the highest-value *technical* control, because it converts a human judgement call (which is error-prone and fatiguing at scale) into an automated gate (which is consistent and tireless). The case study project's experience suggests that this is not additional overhead — it is a redirection of existing review effort from low-value pattern scanning towards high-value semantic evaluation. The total compliance burden is similar; the assurance yield is higher.

Agent execution authority boundaries that constrain what the agent is permitted to *do*, not just what code it produces. Agentic coding tools execute shell commands, install packages, modify configuration, and interact with external services using the operator's system credentials — the agent inherits the operator's privileges without the operator's judgement about when to exercise them. Where agents are not effectively sandboxed, operations beyond simple file edits should generally require explicit human approval before execution. Where agents operate within a sandbox, a human should review the complete execution trace before the agent's output is integrated into the codebase. The sandbox is the first barrier; the human review of what happened inside it is the interlock. This is a condition by design — agents need broad execution authority to be productive — and the control is proportionate: a few seconds of operator review per non-trivial operation, against a blast radius measured by everything the operator's credentials can reach.

What requires coordination (months)

ISM extensions addressing AI-generated code as a distinct trust category. The ISM's 2025 updates — the June restructuring of software development controls and the December introduction of AI security controls — provide strong foundations that, in this paper's assessment, require targeted extension for agent-generated code specifically — not a new framework, but new controls within the existing structure.

Procurement clause updates requiring contracted developers to disclose AI tool usage and demonstrate semantic defect detection. Given that most government software is delivered through contracted service providers, this is where the largest volume of uncontrolled AI-generated code enters government systems. Without controls at this delivery boundary, the other proposed measures govern only the minority of code written directly by agencies.

Cross-government coordination on detection tooling and shared rule sets, so that each agency is not independently discovering and addressing the same defect patterns. The correlated

nature of AI-generated defects — the same tool producing the same blind spots everywhere — means shared detection rules are likely to be both feasible and efficient.

ASD/ACSC is the natural lead for ISM extensions addressing AI-generated code, with a coordinating body such as the Digital Transformation Agency handling the broader response including procurement guidance and cross-government tooling.

What could be built (semantic enforcement tooling)

A new category of automated check — semantic enforcement tooling — could let a project declare "this field must never receive a default value" or "this error must reach the audit trail" and have those declarations enforced automatically in the development pipeline. One approach to this has been prototyped; the tooling category is nascent. Organisations should expect the specific tools and techniques to evolve faster than the underlying requirement. Semantic enforcement tooling could automate the three pattern checks above (Q1–Q3) so they can be checked by machines on every commit rather than relying on human reviewers who fatigue under volume — the two judgement calls (Q4–Q5) would remain with human reviewers. In the interim, the five questions and the companion Practical Guide provide the same intuition in human-readable form — a bridge until automated tools are available.

6. What you should do with this information

If you are a departmental executive

Ask your Chief Technology Officer three questions:

1. "Are our development teams using AI coding tools? Are our contracted suppliers?"
2. "Do we have any detection for the semantic defect patterns described here — defects where code is syntactically correct but does the wrong thing in our institutional context?"
3. "What is our exposure if we don't, and what would first-stage controls cost?"

The answers will tell you whether this is a managed risk or an unmanaged one. The first-stage controls described above are modest in cost and do not require restricting the tools that are delivering genuine productivity benefits.

If you are a policy advisor

This research identifies a gap in existing assurance frameworks that warrants investigation by the bodies responsible for those frameworks. The gaps described here are specific, demonstrable, and — based on one project's experience — appear addressable with modest effort. Validating that finding across a broader set of projects and codebases is properly the work of ASD/ACSC, not the originating project.

The Australian Government is pro-AI and pro-productivity. Responsible adoption requires updating security frameworks that predate the technology — not restricting the tools, but ensuring

the governance keeps pace with the capability. Controls that do not exist today cannot protect the systems being built right now, and the frameworks proposed here are designed to evolve — as the risk profile improves, controls can be relaxed proportionately.

The gaps identified above do not require new legislation or new regulatory bodies. They require targeted extensions to existing instruments — the ISM, Essential Eight, and Commonwealth procurement frameworks — by the bodies that already own those instruments.

If you are briefing senior stakeholders

These questions help frame the conversation at executive, board, or committee level:

- "Are our development teams using AI coding tools? Are our contracted suppliers? Do we have visibility into this?"
- "Does our security testing detect semantic defects in AI-generated code — specifically, defects where the code is syntactically correct but does the wrong thing in our institutional context?"
- "What is our current exposure, and what would first-stage controls cost to implement?"
- "Do our procurement contracts require contracted developers to disclose AI tool usage and demonstrate detection of these defect patterns?"
- "Are we aware of the emerging guidance in this space, and are we positioned to adopt it when it formalises?"

If you are a programme director or delivery lead

Your delivery teams — and their contracted suppliers — are using these tools now. Two actions are immediately useful: (1) share the companion *Practical Guide* with teams who write or review code, so they know what to look for; (2) review your procurement and contracting arrangements against the trust-boundary questions this paper raises — existing procurement frameworks do not yet require controls at the delivery boundary. The case studies (§8 of the full paper) provide concrete examples: Appendix D shows a complete agent-built application with every finding cited to a line of code; Appendix E presents annotated incidents from a compliance-constrained project.

If you write or review code yourself

This document explains the systemic problem. For practical guidance on what to look for in your own code — including the five review questions with worked examples, a hot-path identification walkthrough, and pattern-recognition techniques that do not require specialised security training — see the companion guide: *Reviewing AI-Generated Code: A Practical Guide for Code Authors*.