Reviewing AI-Generated Code: A Practical Guide for Code Authors

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)

> 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.

This is the practical companion to Governing AI-Generated Code: Semantic Risk in High-Stakes Code Paths — that document explains the systemic risk; this one shows you what to look for in your code. For the proposed actions and candidate ISM changes, see **Proposed Framework Changes: Candidate Controls for Consultation** *or the fuller companion,* **Proposed Framework Changes and Recommendations**.

1. You Are Not Doing Anything Wrong

You are not doing anything wrong. You are using AI to write code — to automate reports, process data, connect systems, do useful work. That is productive, increasingly normal, and increasingly supported by organisational policy. This guide is not here to tell you to stop.

It is here to help you spot the one thing AI consistently gets wrong: **code that looks right but makes the wrong decision about data that matters.**

Here is a concrete example. An AI writes a script to process incoming records. One line reads:

```
classification = record.get("security_classification", "OFFICIAL")
```

This says: if the security classification field is missing, use OFFICIAL as the default. In a weather app, that kind of defensive programming is fine. In a government system, a missing classification might mean the upstream system failed. That one line silently downgrades an unknown document to OFFICIAL — and nobody is told. No error, no alert, no log entry.

AI produces this pattern routinely. It looks like good practice because it *is* good practice — in most software. In systems where data integrity, auditability, or classification accuracy matter, the same pattern is quietly dangerous.

Because the AI produced the code at your direction, it feels like *your* code — and you know your intentions were good. But you did not write it — it was generated by a system that has not validated its output against your project's security requirements, even when those requirements are in the AI's context. The discussion paper (Appendix E) documents cases where the AI had the project's security rules available, could cite them when challenged, but did not consult them during initial code generation. Giving the AI your rules is not enough — it may know the policy and still not apply it. Better prompting helps, but it does not solve the problem. The fact that you are a careful, experienced operator does not change what the AI produced — it changes your ability to *review* what was produced. That review is what this guide is for.

Why this matters more than you might think: existing security controls for software — code review, testing, CI pipelines — were designed for code written by professional development teams inside formal development processes. Your scripts, automations, and data processing

tools are typically produced outside those processes: no CI pipeline, no code review team, no formal test suite. That means the existing controls often do not reach your work directly — not because anyone decided to exclude you, but because the controls were drawn around "software development" before AI tools made it possible for analysts and operators to produce executable logic. The five questions in this guide exist to help bridge that gap. They are not just optional good practice — for many teams, they are the practical interim substitute for controls that do not yet reach this kind of work.

This guide teaches you to see those invisible mistakes, even without specialised tools. It gives you five questions to ask about any AI-generated code that touches data you care about. The code examples use Python and SQL — the most common languages for reporting and data processing scripts — but the same patterns exist in every language. The five questions work regardless of which language your code is in. These are interim — automated semantic enforcement tooling may eventually complement them. A companion specification illustrates what that tooling could look like; the approach appears buildable with standard language tooling, and several implementation paths are possible. Until that tooling arrives, these five questions are your practical review layer.

> ***Why This Matters:*** *A script that silently defaults a security classification to OFFICIAL when the field is missing does not crash. It does not throw an error. It processes PROTECTED documents at the wrong level, and nobody finds out until an audit — or an incident. AI produces this pattern routinely.*

2. Three Patterns That Will Fool You

These are the core ways AI-generated code goes invisibly wrong. Each follows the same structure: what the AI wrote, why it looks right, what is actually wrong, what you should write instead, and what it means if this goes wrong in a government system.

2.1 The Friendly Default

The AI writes `.get(field, "OFFICIAL")` — if the security classification field is missing, use OFFICIAL as the default. This is standard defensive programming. Every tutorial teaches it.

In a government system, a missing classification field might mean the upstream system failed. The correct response is to *stop and investigate*, not to silently invent an answer. The AI just downgraded an unknown document to OFFICIAL and nobody was told.

```
# WRONG — silently invents a classification
classification = record.get("security_classification", "OFFICIAL")
```

```
# CORRECT — missing classification is an error, not a default
classification = record["security_classification"]
# If the key is missing, this crashes — and crashing is safer
# than silently processing PROTECTED documents as OFFICIAL.
# (The full discussion paper §2.3 shows an even better version
# that crashes with a diagnostic error message — but crashing
# at all is the important part.)
```

Q1 to ask: "If this value is missing, is that *normal* or is that *evidence something is broken*?"

In the discussion paper's taxonomy, this pattern is called *Fabricated Default* — rated *High*. The AI presents a confident result based on fabricated input. The code does not have the classification — it substitutes a default and processes the record as though the value were genuine.

> ***Why This Matters:*** *This is how PROTECTED documents get processed at the wrong level. No error, no alert, no log entry. The system runs with a confident wrong answer.*

2.2 The Helpful Error Handler

The AI wraps an audit-critical operation in `try/except` — if it fails, log the error and continue. This looks like robust error handling. Every coding tutorial teaches it.

In a system with audit obligations, a swallowed exception means an auditable operation happened with no record. If someone asks "did this transaction complete?", the answer is "we don't know — the error was logged somewhere but the audit trail says nothing happened."

```
# WRONG — swallows the audit trail
try:
    write_audit_record(transaction)
except Exception as e:
    logger.error(f"Audit write failed: {e}")
    # Continues as if nothing happened

# RIGHT — audit failure stops the operation
write_audit_record(transaction)
# If this fails, the exception propagates.
# The calling code must handle it — not silently continue.
```

Q2 to ask: "If this operation fails, does someone need to *know* it failed — not just that something was logged?"

In the taxonomy, this is *Audit Trail Destruction* — rated *High*. The code appears to handle errors gracefully, but "graceful" is the wrong response when the audit trail is the legal record. A gap in the audit trail is not a logging failure — it is a compliance failure that may have legal consequences.

> ***Why This Matters:*** *Under many compliance frameworks, a missing audit record is not a bug — it may be indistinguishable from evidence of tampering without additional investigation.*

2.3 The Invisible Promotion

The AI reads data from an external API and passes it directly to an internal function. The types match, the code runs, no errors. But the data just crossed from "untrusted external source" to "internal authoritative data" with no validation boundary.

```
# WRONG — external data treated as internal
external_data = api_client.get_records()
process_internal_records(external_data)

# RIGHT — validate before promoting
external_data = api_client.get_records()
validated = validate_and_quarantine(external_data)
process_internal_records(validated)
# The validation function checks structure, types, ranges,
# and flags anything that doesn't match expectations.
```

Q3 to ask: "Did this data come from *us* or from *outside*? Am I treating outside data as if we produced it?"

In the taxonomy, this is *Authority Tier Conflation* — rated *Critical*, the highest severity. The AI sees both data sources as "a dictionary" and treats them identically because nothing in the programming language distinguishes them. Once external data enters your internal data store without validation, every downstream consumer trusts it as if your system produced it.

> ***Why This Matters:*** *This is how a partner API's data quality problems become your reporting errors — and nothing in your system will distinguish their data from yours. If the API sends bad data, your reports, dashboards, and any decisions derived from them rest on unvalidated external input that your system treated as authoritative.*

A closely related pattern — separately rated *Critical* in the taxonomy as *Implicit Privilege Grant* (ACF-E1): AI-generated code that accepts an external system's *assertion* and acts on it without question. A partner API says `"verified": true` and the code grants system access — without independent verification, without recording the basis for the decision, and without considering what happens if the partner system is compromised or simply changes its contract. External systems can stop sending fields, change their meaning, or start returning unexpected shapes — and AI-generated code typically handles all of these silently rather than surfacing them. If the API stops sending a `username` field tomorrow and the code `.get()`s a default instead of

failing, you have users with no identity attached to their session. The question is the same: "Am I trusting an external system's shape, values, and claims as if we verified them ourselves?"

> ***Two Critical-rated patterns, not one.*** *The discussion paper's taxonomy contains two patterns rated* Critical *— the highest severity. Authority Tier Conflation (ACF-T1, the Invisible Promotion above) and Implicit Privilege Grant (ACF-E1, the assertion-trust pattern just described). Both have zero existing tool detection. T1 is about* data *crossing a trust boundary without validation; E1 is about* privilege decisions *being made based on unvalidated external claims. Q3 catches both — but they are distinct failure modes with different consequences, and knowing both exist helps you recognise them separately in your code.*

One cause, three failure modes

The Friendly Default, the Helpful Error Handler, and the Invisible Promotion look like separate mistakes, but they are all caused by the same thing: AI applies the same defensive programming patterns everywhere, regardless of whether the data is high-trust or low-trust.

The result is a **bidirectional collapse**. The Invisible Promotion (Section 2.3) pushes untrusted external data *upward* — giving it more authority than it has earned. The Friendly Default (Section 2.1) and the Helpful Error Handler (Section 2.2) push authoritative internal data *downward* — treating corruption of audit records or missing classifications as routine rather than exceptional. The same one-size-fits-all defensive code simultaneously promotes what should not be trusted and demotes what should not be negotiable. Recognising this connection helps you see the patterns faster: when you find one, look for the others.

3. The Five Questions

These five questions work against any AI-generated code that touches data you care about. You do not need to understand every line of code to use them — they are designed to catch the patterns above by asking about *consequences*, not syntax.

- **Q1. If this value is missing, does my code crash or invent an answer?** Crashing is usually safer than inventing. A crash tells you something is broken. A silent default tells you everything is fine when it might not be.
- **Q2. If this operation fails, does my code tell someone or quietly continue?** Quiet continuation destroys audit trails. If an audit-critical operation fails and the code carries on, the audit trail has a gap that cannot be reconstructed.

- **Q3. Where did this data come from? Am I checking it or trusting it?** The boundary between "ours" and "outside" is where risk concentrates. Data from an external API, a user upload, or a partner system should be validated before it enters your internal processes. Even APIs you have decided to trust deserve scrutiny: is the access decision coming from your policy or from the API? A staff directory confirming employment is not the same as a staff directory granting folder access — even if your business rules connect the two, the code should make that connection explicitly, not inherit it from the API response.
- **Q4. Did AI suggest this pattern? Do I understand why it chose this over another?** AI defaults to what is common, not what is correct for your context. The patterns in Section 2 are all standard good practice in general software — they are dangerous specifically because AI does not distinguish your context from a web tutorial. *In practice:* the AI generated `ON CONFLICT (case_id) DO UPDATE` for the audit table. You could have used `INSERT` without `ON CONFLICT` — which would fail on a duplicate, forcing investigation. The question is: did you make that choice, or did the AI make it for you? If you do not know why it chose one over the other, ask it.
- **Q5. If this code is wrong, how would I find out?** If the answer is "I wouldn't," that is your highest-priority hot path. Silent failures are the hardest to detect and the most damaging in government systems. *In practice:* the reporting script produces a summary CSV. If it wrote "Unknown" for every department for three months, would your current process catch that? If the answer is "probably not until someone noticed the dashboard looked wrong," then the CSV output is a hot path — and you need an output check, not just a code check.

Print these. Pin them next to your monitor. Run through them every time you accept AI-generated code that touches classification, PII (personally identifiable information), financial data, or audit records.

> ***Questions to Ask Your IT Security Team:*** *"We use AI to generate scripts that touch [classification/PII/financial] data. Do we have any static analysis rules that would catch the patterns described in this guide? If not, what is the interim guidance for our team?"*
>
> *Your IT security team may not have encountered this specific risk class yet — it is new, and the existing frameworks may not yet cover it clearly. That is normal. You are not reporting a failure; you are putting a new risk on their radar. Bring this guide and the companion document (*Governing AI-Generated Code*). Suggest a 30-minute meeting to walk through the three patterns together.*

4. How This Connects to What You Already Know

If you work with Australian Government security frameworks, you are already familiar with several that this guide extends into new territory.

If you know...	Then this maps to...
PSPF classification obligations	The friendly default (Section 2.1) — silent classification downgrade. Q1 catches the pattern where AI invents a classification instead of surfacing a missing one.
ISM controls for software development	Q1–Q5 extend ISM review into *semantic* territory — not "is the code structured correctly?" but "does the code do the right thing in this institutional context?"
Essential Eight patching and hardening	These patterns are not caught by Essential Eight — they sit in the gap between infrastructure-level security and application-level semantic correctness. E8 protects the platform; Q1–Q5 protect the logic running on it.
Audit and accountability requirements	The helpful error handler (Section 2.2) — audit trail destruction. Q2 catches the pattern where AI swallows the exception that should have preserved the audit record.
Data sovereignty and data handling	The invisible promotion (Section 2.3) — untrusted data entering authoritative paths. Q3 catches the pattern where AI treats external data as if it came from an internal, trusted source.

The five questions are not a replacement for these frameworks. They are an extension into a gap that existing frameworks were not designed to cover: the semantic correctness of AI-generated code.

5. Finding Your Hot Paths

Not all code is equally dangerous. A function that formats dates is not the same as a function that decides what security classification to apply. Most of the risk in a 500-line script concentrates in perhaps 30 lines. Those 30 lines are your *hot paths* — the places where Q1–Q5 matter most.[1]

5.1 What makes a path "hot"

A code path is hot if **a wrong answer has real-world consequences.** That means it touches classifications, PII, financial amounts, audit records, access decisions, or any data where corruption or silent mishandling affects people, compliance, or trust. Defaulting a missing location in a weather app is not a hot path — a wrong forecast inconveniences someone. Defaulting a missing security classification is a hot path — that decision downgrades documents and nobody is told.

Everything else — formatting, logging messages, building display strings, iterating over lists — is not where the risk lives. Focus your review energy on the hot paths.

Once you've found a hot path, look for these behaviours inside it:

[1]The full discussion paper calls these *high-stakes code paths* — code where silently absorbing a fault is more dangerous than crashing. The defining question is not "is this code important?" but "if this code fails, is it safer to stop or to continue with a guess?" On a hot path, continuing with a guess is the dangerous option. "Hot path" is the same concept in plainer language.

- **Data handling and assignment.** How does the code deal with the data flowing through this path? What happens if a value is missing, malformed, or unexpected — does the code crash, invent a default, or silently coerce it into something else? (Q1, Q3)
- **Exception handling.** How does the code deal with failures in this path? Are errors reported to someone who can act, or are they logged and swallowed? Does the error handler preserve the audit trail or destroy it? (Q2)
- **Boundary crossings.** Does data enter this path from outside — an external API, a user upload, a partner system — and get used without validation? Does the code distinguish between "ours" and "not ours"? (Q3)
- **Implicit decisions.** Is the code making a policy decision without being explicit about it? A `.get()` with a default is a policy decision: "if this value is missing, use this one instead." A `try/except` that continues is a policy decision: "if this fails, carry on as if it did not." These decisions are fine in low-consequence code. In a hot path, they need to be deliberate and visible. (Q4, Q5)

5.2 A walkthrough: finding the hot lines in a reporting script

Here is a small reporting script that reads records from an external partner API, processes them, and writes summary rows to a local database. It is about 30 lines long. The comments mark which lines are hot and which question applies. Lines that are not hot paths — setup, formatting, cleanup — are marked "cold."

```
import requests
import sqlite3
from datetime import datetime

def generate_partner_report(api_url, db_path):
    # --- HOT (Q3): db_path comes from the caller — and every
    #     hot line below depends on this connection working. ---
    conn = sqlite3.connect(db_path)
    cursor = conn.cursor()
    report_date = datetime.now().strftime("%Y-%m-%d")

    # --- HOT (Q3): data crosses from external to internal ---
    # --- HOT (Q4): AI chose .json() with no error handling — do you know
why? ---
    response = requests.get(f"{api_url}/daily-summary")
    records = response.json()

    for record in records:
        # --- HOT (Q1): what if "department" is missing? ---
        dept = record.get("department", "Unknown")
```

```
        # --- HOT (Q1): what if "total_amount" is missing? ---
        amount = record.get("total_amount", 0)

        # --- HOT (Q3): external data inserted without validation ---
        cursor.execute(
            "INSERT INTO daily_reports (report_date, department, amount) "
            "VALUES (?, ?, ?)",
            (report_date, dept, amount),
        )

    # --- HOT (Q2): what if the commit fails? ---
    try:
        conn.commit()
    except Exception as e:
        print(f"Database error: {e}")
        # Silently continues — report data may be partially written

    # --- Cold: timing stats — nice to have, not load-bearing ---
    try:
        elapsed = (datetime.now() - datetime.strptime(report_date, "%Y-
%m-%d")).seconds
        print(f"Report took {elapsed}s for {len(records)} records")
    except Exception:
        pass  # Stats are cosmetic — swallowing this is fine

    conn.close()

    # --- HOT (Q5): this returns "success" regardless — how would you know
    #     if the data was wrong? ---
    return f"Report generated for {report_date}"
```

Now apply the five questions to the hot lines:

Q1 — the `.get()` defaults: If `department` is missing, the script writes "Unknown" to the database. If `total_amount` is missing, it writes 0. Are these values *normal defaults* or *evidence that the partner API sent bad data*? In a financial reporting context, a department of "Unknown" and an amount of 0 will pollute your reports silently. Fix: access the keys directly and let missing data raise an error.

Q3 — the external data path: The data from `response.json()` is external. It is inserted directly into the local database with no validation. The partner API could send unexpected fields, wrong types, or missing required data — and the script will write whatever it receives. Fix: validate the structure of each record before inserting.

Q2 — the `try/except` block: If `conn.commit()` fails, the script prints the error and continues. The return message says "Report generated" even if the data was never committed. Fix: let the exception propagate so the caller knows the report failed.

Here is the corrected version:

```
def generate_partner_report(api_url, db_path):
    conn = sqlite3.connect(db_path)
    cursor = conn.cursor()
    report_date = datetime.now().strftime("%Y-%m-%d")

    response = requests.get(f"{api_url}/daily-summary")
    response.raise_for_status()  # Q3: fail on HTTP errors
    records = response.json()

    for record in records:
        # Q3: validate external data before internal use
        if "department" not in record or "total_amount" not in record:
            raise ValueError(
                f"Partner record missing required fields: {record}"
            )
        # Q1: no silent defaults — use validated values directly
        dept = record["department"]
        amount = record["total_amount"]

        if not isinstance(amount, (int, float)):
            raise TypeError(
                f"Expected numeric total_amount, got {type(amount)}:
{amount}"
            )

        cursor.execute(
            "INSERT INTO daily_reports (report_date, department, amount) "
            "VALUES (?, ?, ?)",
            (report_date, dept, amount),
        )

    # Q2: no try/except — if commit fails, the caller must know
    conn.commit()
    conn.close()
    return f"Report generated for {report_date}"
```

The corrected version is about the same length. It does not add complexity — it removes silent failures and makes problems visible.

5.3 SQL hot paths

If your scripts include SQL — queries, inserts, data transformations — the same three patterns apply. AI generates SQL with the same blind spots it has in Python.

The SQL-flavoured friendly default (Q1):

```sql
-- WRONG — silently invents a classification when the field is NULL
SELECT
    document_id,
    COALESCE(security_classification, 'OFFICIAL') AS classification
FROM documents;
-- If security_classification is NULL, it means the classification
-- is unknown — not that it should be OFFICIAL.

-- RIGHT — surface the problem
SELECT
    document_id,
    security_classification AS classification
FROM documents
WHERE security_classification IS NOT NULL;
-- Handle NULLs separately — investigate why they are missing.
```

`COALESCE` is the SQL version of `.get()` with a default. In a reporting query, it silently converts "we do not know the classification" into "OFFICIAL" — and anyone reading the report sees a confident answer.

The SQL-flavoured invisible promotion (Q3):

```sql
-- WRONG — external data inserted directly into internal table
INSERT INTO internal_records (name, department, clearance_level)
SELECT name, department, clearance_level
FROM external_staging;
-- No validation. external_staging could contain anything.

-- RIGHT — validate at the boundary
INSERT INTO internal_records (name, department, clearance_level)
SELECT name, department, clearance_level
FROM external_staging
WHERE clearance_level IN ('BASELINE', 'NV1', 'NV2', 'PV')
  AND department IS NOT NULL
  AND name IS NOT NULL;
-- Reject rows that don't match expected values.
-- Investigate rejected rows separately.
```

`INSERT INTO ... SELECT FROM` without a `WHERE` clause is the SQL equivalent of passing external data directly to an internal function. The database does not care where the data came from — it trusts whatever you give it.

The SQL-flavoured error swallowing (Q2):

```
-- WRONG — silently overwrites existing records
INSERT INTO audit_decisions (case_id, decision, decided_by, decided_at)
VALUES (1042, 'APPROVED', 'jsmith', '2026-03-18')
ON CONFLICT (case_id)
DO UPDATE SET
    decision = EXCLUDED.decision,
    decided_by = EXCLUDED.decided_by,
    decided_at = EXCLUDED.decided_at;
-- If case 1042 already has a decision, this silently replaces it.
-- The original decision is gone. No record that it was changed.

-- RIGHT — conflict means something went wrong
INSERT INTO audit_decisions (case_id, decision, decided_by, decided_at)
VALUES (1042, 'APPROVED', 'jsmith', '2026-03-18');
-- If case 1042 already has a decision, this fails with a constraint
-- violation — which is what you want. A duplicate decision is a
-- problem to investigate, not a conflict to silently resolve.
```

`ON CONFLICT ... DO UPDATE` is the SQL version of a `try/except` that silently continues. In an audit context, a conflict on a decision record means something has gone wrong — a case was decided twice, or a record was duplicated. Silently overwriting it destroys the evidence.

Two SQL-specific risks with no Python equivalent:

- **Implicit type coercion.** Many SQL databases (notably MySQL and SQL Server) silently coerce types in comparisons and joins. An agent joining an `INT` column to a `VARCHAR` column (e.g., `SELECT * FROM records JOIN users ON records.user_id = users.id` where `user_id` is `VARCHAR` and `id` is `INT`) produces a query that runs without error but may silently drop rows or match wrong rows. PostgreSQL is stricter and raises a type error, but most enterprise deployments use platforms where silent coercion is the default. If your SQL joins columns of different types without explicit casting, check whether the database is silently coercing — and whether the results are what you expect.
- **Dynamic SQL and injection.** Agents generating stored procedures frequently use string concatenation to build SQL dynamically (e.g., `EXEC('SELECT * FROM ' + @tableName)`). This is the well-known SQL injection risk, but the agentic context compounds it: the agent generates the vulnerable pattern because that is what its training data contains, and it has no concept of parameterisation as a security boundary.

Your strongest SQL defence: database-level constraints. `CHECK` constraints, foreign key relationships, domain types, and `NOT NULL` constraints enforce validation at the data layer regardless of how the SQL was generated. These are environmental controls — the database rejects invalid data whether the SQL was written by a human, generated by an AI, or run from a reporting tool. They do not require CI pipelines, static analysis, or developer expertise. If your database schema permits `NULL` in the `security_classification` column, then every `COALESCE` default in every query is a live risk. Adding `NOT NULL` to that column closes the gap for every query, past and future. Audit whether your database schemas enforce the same rules your application code does — in many cases, the application validates but the schema permits, meaning any SQL that bypasses the application can write invalid data.

Where your SQL actually lives. Much operational SQL is not in scripts or version control — it lives in reporting tools, BI platforms, ETL configurations, and scheduled job definitions that exist outside the development workflow entirely. The guidance in Section 7 ("put your scripts in version control") does not reach SQL embedded in these tools. If your SQL lives in a BI platform or a scheduled job runner, the database-level constraints described above are your primary defence — they protect the data regardless of where the SQL comes from.

For deeper SQL coverage. The parent paper's Appendix C (*Extension to Agentic SQL Generation*) provides a fuller treatment of SQL-specific failure modes — including detailed code examples for `COALESCE`-based fabricated default, `INSERT...SELECT` authority tier conflation, audit trail destruction through silent overwrites, partial completion without transactions, and two SQL-specific risks (implicit type coercion and dynamic SQL injection) not covered by the Python taxonomy. Appendix C also addresses the governance implications of non-developer staff producing SQL with agentic tools (§C.4). If SQL is a significant part of your work, that appendix is worth reading alongside this section.

5.4 When this guide stops being enough

This guide works for scripts up to a few thousand lines where you are the primary author and reviewer. If your project has grown beyond that — multiple contributors, dependencies on other systems, or code that other teams rely on — you need developer support.

The threshold is not exact, but these are signals:

- Your project has grown to multiple interconnected scripts that depend on each other.
- You have users who are not on your team.
- You cannot hold the whole codebase in your head.
- You are spending more time reviewing AI output than writing prompts.

At that point, engage your IT team or a developer to help apply the five questions at scale. The patterns are the same — they just need to be enforced systematically rather than manually.

5.5 If you already have CI and git

If you are a developer with a CI pipeline, version control, and code review practices already in place, the five questions still apply — but you can enforce Q1–Q3 as automated checks rather than manual review.

What you can do now, with existing tools:

- **Q1 (silent defaults):** Write a semgrep or CodeQL rule that flags `.get()` with a literal default on fields your team has identified as sensitive — classification, retention, consent, jurisdiction. The rule does not need to be perfect; even a noisy rule that fires on every `.get()` default in a designated module gives reviewers a starting point. Example: a semgrep rule matching `$OBJ.get($KEY, $DEFAULT)` within files tagged as handling classified or audit-critical data.
- **Q2 (swallowed errors):** Flag `try/except` blocks (or equivalent in your language) that catch broad exception types and do not re-raise, in code paths your team has marked as audit-critical. This is a standard static analysis pattern — most teams already have a variant. The refinement is targeting it at audit-critical paths specifically rather than applying it everywhere.
- **Q3 (unvalidated external data):** Harder to automate without semantic context, but a useful first step is a linting rule that flags direct assignment from deserialised external input (e.g., JSON, API response) to variables used in access control or classification decisions, without an intervening validation call.

Q4 and Q5 remain manual. "Do I understand why the AI chose this approach?" and "If this code is wrong, how would I find out?" are judgement calls that require institutional context. They belong in your code review checklist, not your CI pipeline.

Team-level adoption:

- Add Q1–Q5 to your code review template as a checklist section for AI-generated code.
- Track which code was AI-generated using commit metadata, co-author tags, or a provenance convention your team agrees on. This costs nothing and makes it possible to assess exposure retrospectively if a systematic pattern is discovered later.
- When a Q1–Q3 violation is found manually during review, write a rule for it. Each caught violation is a rule waiting to be automated. Over time, your CI pipeline accumulates the institutional knowledge that currently lives only in reviewer memory.

This is the same progression described in Section 8, but from the other direction: Section 8 starts from "you have no tooling" and describes where automated enforcement is heading. This section starts from "you have a CI pipeline" and describes what to put in it today.

6. Using AI to Review AI-Generated Code

> ***Important caveat:*** *Unprompted AI review will miss the same patterns it introduced — it has the same training-data blind spots whether writing or reviewing. Prompted review is substantially better: when you tell AI exactly what to look for, it can find patterns it would not have avoided during generation. Use these prompts as a supplement to your own review against Q1–Q5, not as a replacement — but prompted and monitored AI review is dramatically better than no review at all.*

You already use AI to write code. You can also use it to review code — with the right prompts. Prompted review is substantially better than no review. The key insight is that AI is better at *finding* patterns when you tell it exactly what to look for than it is at *avoiding* those patterns when writing.

Here is how to use AI to check AI-generated code:

1. AI generates the code.
2. You read through it once, marking the hot paths (Section 5).
3. You paste the hot-path code back to the AI with the relevant prompt below.
4. You review the AI's findings against Q1–Q5 yourself.
5. You apply fixes where needed.

This takes 5–10 minutes per script. It is not perfect — but it is dramatically better than accepting AI output without review. The prompts below are designed for step 3.

Prompts that work

Copy and paste these into your AI tool when reviewing AI-generated code:

For Q1 — finding friendly defaults:

> *Review this code. For every* `.get()`*,* `COALESCE`*, default value, or fallback: tell me what happens if that value is genuinely missing. Is the default safe, or could it mask a real problem? For each one, tell me whether the missing value is "normal and expected" or "evidence that something upstream is broken."*

For Q2 — finding error swallowing:

> *Review this code. For every try/except, catch block, or error handler: tell me what happens to the error. Is it reported to someone who can act on it, or is it logged and ignored? For each one, tell me: if this operation fails in production, will a human find out before the next audit?*

For Q3 — finding boundary crossings:

> *Review this code. Identify every place where data from an external source (API, file, user input, partner system) is used in an internal function or written to an internal database. Is there a validation step between the source and the use? For each one, tell me what would happen if the external source sent unexpected or malformed data.*

For the full checklist:

> *Review this code against these five questions: (1) Do missing values crash or get a silent default? (2) Do failed operations get reported or quietly swallowed? (3) Is external data validated before internal use? (4) Are there patterns here that could have been different — and can you explain why this approach was chosen over alternatives? (5) If this code produces a wrong answer, what would make that visible? For each finding, rate it as HIGH (touches classification, PII, financial, or audit data), MEDIUM (touches operational data), or LOW (cosmetic or non-consequential).*

For SQL (Q1, Q2, Q3):

> *Review this SQL. For every COALESCE, ISNULL, or default value: what happens if that column is genuinely NULL — is the default safe or could it mask a data quality problem? For every INSERT INTO … SELECT FROM: is the source data validated before insertion? For every ON CONFLICT … DO UPDATE: could a silent overwrite destroy audit-relevant information?*

What these prompts will — and will not — catch

These prompts are one layer in a defence-in-depth approach to reviewing AI-generated code. They will surface issues you would miss on a casual read-through, and the specialist perspectives (Section 10) catch different classes of problem than the pattern-specific prompts above. But they have real limitations:

- **AI review has the same blind spots as AI generation.** The same training-data biases that cause AI to produce these patterns also limit its ability to find them. A prompted review is better than no review, but it is not equivalent to a human expert review or automated static analysis.
- **These prompts catch *known pattern classes*, not novel defects.** They work because they tell AI exactly what to look for. They will not catch a new class of problem that nobody has described yet.

- **AI-generated tests have the same blind spots as AI-generated code.** If you use AI to generate both your code and the tests that verify it, the tests inherit the same training-data biases. An AI that treats a missing classification as a legitimate default scenario will write tests that verify the default is correctly applied — not that the default is inappropriate. The test passes. The verification is circular. If you use AI to write tests, review the test assertions against Q1–Q5 just as you would review the code itself. High test coverage does not mean the right properties are being tested.

  Here is a real example. A function records why a batch processing job failed, prefixing the status with `batch_` to produce an audit reason:

```
# Production code — the bug
def record_failure(status):
    reason = f"batch_{status}"
    audit_log.write(reason=reason)

# The function expects: "failed" → "batch_failed",
#                       "timeout" → "batch_timeout"
# But one call site passes the already-prefixed value:
record_failure(status="batch_timeout")
# Result: "batch_batch_timeout" — a double prefix
```

  The AI wrote the test for this code path. The test passed — because it asserted the buggy value:

```
# Test — mirrors the bug
def test_timeout_records_failure():
    record_failure(status="batch_timeout")
    assert audit_log.last().reason == "batch_batch_timeout"
    # Green. 100% coverage. The test proves the garbling works.
```

  Both the code and the test were consistent *within the same wrong understanding* — the AI's model of the input value was `"batch_timeout"`, so the f-string output `"batch_batch_timeout"` looked correct, and the test faithfully verified it. An auditor querying for `reason = 'batch_timeout'` would get zero results — not because timeouts never happened, but because every timeout was filed under a garbled key that no query would match. The test that should have caught this was instead proving the garbling worked correctly. (In the discussion paper's taxonomy, this is *Verification Displacement* — ACF-R3a — rated *High*.)

- **This guide is not a substitute for security testing, code review, or automated analysis.** If your project has access to static analysis tools, a CI pipeline, or security review from qualified professionals, those controls are stronger than anything in this section. This guide exists for the gap where those controls are not available — scripts written by individuals without access to professional tooling.

Use these prompts alongside whatever other review practices you have, not instead of them. If your only review process is this guide, that is better than no review — but recognise it as a starting point, not an endpoint. The goal is to build towards automated enforcement (Section 8) where the three pattern checks (Q1–Q3) are checked by machines on every commit, as part of a layered assurance approach where no single control is expected to catch everything.

7. What To Do When You Find a Problem

The five questions flagged something. Now what?

First: do not panic. These are silent defects — they produce wrong answers, not crashes. They are not actively being exploited. You have time to fix them properly.

If it is in code you have not deployed yet

Fix it before deploying. Use the corrected patterns from Section 2 as templates. If you are not sure how to fix it, ask the AI — but verify the fix against the relevant question (Q1–Q5) before accepting it. AI is quite good at fixing a specific pattern when you tell it exactly what is wrong; it is bad at spotting the pattern in the first place.

If it is in code that is already running

These defects are producing wrong answers, not crashing. They have likely been producing wrong answers since the code was deployed. That sounds alarming, but it means the situation is stable — it is not getting worse while you take time to fix it properly.

1. **Assess the impact.** What data does this code touch? What is the worst case if the pattern has been producing wrong results? A friendly default on a department name is less urgent than a friendly default on a security classification. For example: if the friendly default has been writing “Unknown” as the department for six months of daily reports, how many downstream reports, dashboards, or decisions relied on that department field? That is your blast radius.
2. **Fix the code.** Apply the corrected pattern from Section 2. Test the fix. Deploy it.

3. **Check the output.** If possible, review recent outputs for evidence of the defect. Look for records with default values where you would expect real data — "Unknown" departments, zero-value amounts, OFFICIAL classifications on records that should be higher. If your data is in a database, a simple query can surface these:

```
-- Find records that may have been affected by a friendly default
SELECT * FROM daily_reports
WHERE department = 'Unknown' OR amount = 0;
```

4. **Tell someone.** If the code touches classified, PII, or auditable data and you believe incorrect results may have been produced, notify your team lead or IT security contact. Frame it as a quality finding, not a security incident: "I found a pattern in our reporting script that may have been producing incorrect default values for [field]. I have fixed the code and I am reviewing recent outputs. I wanted to let you know in case this affects [downstream process]."

Check your other scripts too

AI produces these patterns as a recurring characteristic, not an occasional lapse. If you find a `.get()` default on a sensitive field in one script, the same AI almost certainly produced the same pattern in your other scripts — because the same training-data bias drives the same behaviour every time. Finding one instance should trigger a scan of your other scripts for the same pattern. This applies to all three patterns: if you find a swallowed audit error in one place, check every `try/except` block the AI generated across your other work. The patterns are correlated — they come from the same source, and they repeat.

If you are not sure whether it is a problem

Ask. Use the IT security conversation framing from Section 3. A false alarm costs a 30-minute meeting. A missed defect on classified data costs substantially more.

One thing you can do today that costs nothing

Put your scripts in version control (git). Even if you are the only person working on them. Version control means you can see what changed, when, and — critically — you can undo a change that turns out to be wrong. If you discover a friendly default that has been producing incorrect results, version control tells you when it was introduced and what the output looked like before. Without it, you are guessing.

If you do not have git installed or are not comfortable using a terminal, start with what you have: save a dated copy of your script before making changes (e.g., `my_report_2026-03-24.py`), or ask your IT team to set up a shared folder with version tracking. The goal is to be able to see what changed and undo it — git is the best tool for this, but any form of change history is better

than none. If your team has a code repository (GitHub, Bitbucket, Azure DevOps), ask for access — your scripts belong there alongside everyone else's code.

This is not a security control — it is basic hygiene, and it makes every other practice in this guide more effective.

8. When You Need More Than This Guide

This guide is interim — it works when you are the only reviewer and there is no automated tooling. It is a manual process by design, because most staff using AI to write code do not have access to CI pipelines, static analysis tools, or pre-commit hooks.

Automated detection is being designed. Semantic enforcement tooling would encode the three pattern checks (Q1–Q3) as machine-enforceable rules that run automatically every time code is saved or submitted. A companion specification illustrates one way to build this; other approaches using existing static analysis tools (such as semgrep or CodeQL custom rules) could target the same patterns. The key insight is that Q1–Q3 are automatable regardless of which tooling implements them — the patterns are specific enough for machines to check. When that tooling arrives, those checks will run automatically so you do not have to hold them all in your head. The two judgement calls (Q4–Q5) remain with human reviewers.

The empirical evidence is encouraging. In a simulation, an AI agent was given an explicit high-stakes brief and prototyped a government assistance application in under an hour — implementing CSRF protection, OTP hashing, rate limiting, and audit logging — producing 20 findings mapped to the same defect taxonomy. A skilled reviewer would find these issues, but the code's clean appearance and the volume at which it is produced make sustained review discipline harder to maintain. In a separate longitudinal observation (~80,000 lines of Python, one developer's work), a combination of rigorous review and internal tooling regularly catches and blocks these patterns before they enter the codebase — none have entered. Without that detection capability, every one of those violations would have passed normal code review — because they look like correct, well-written code. The discussion paper's case studies (§8) report the full evidence with methodology and caveats.

That is what the five questions are catching manually. The gap between "what this guide catches" and "what automated tooling would catch" is real, but the five questions cover the three highest-impact patterns and give you a defensible review process in the interim.

As your project grows:

- **Immediate:** Ask your IT security team about static analysis rules for the three patterns in Section 2.
- **Near-term:** Semantic enforcement tooling — as illustrated by the companion specification — could provide machine-checkable rules for the three pattern checks (Q1–Q3), integrated into the development workflow using existing static analysis tools.

- **The key insight:** Everything in this guide maps to an automatable pattern check. You have been learning the intuition; the tools encode it so you do not have to hold it all in your head.

If your concern is governance rather than code — the systemic risk, the framework gaps, the procurement implications — see *Governing AI-Generated Code* for the policy response. That document covers the "why" behind the "what" in this guide: why these defects are a systemic problem, what existing frameworks miss, and what do we need to do about it.

9. One Page to Take to a Meeting

This section is designed to be printed and used independently of the rest of the guide. Take it to a meeting with your IT security team, your CTO, or your team lead.

Three patterns AI gets wrong in high-stakes code:

- **The Friendly Default** — AI invents a value (e.g., OFFICIAL) when data is missing, instead of raising an error
- **The Helpful Error Handler** — AI catches and swallows audit-critical errors instead of letting them propagate
- **The Invisible Promotion** — AI treats external/untrusted data as internal/trusted without validation

Five review questions for AI-generated code (Q1–Q5):

- Q1. Missing value — does the code crash or silently invent an answer?
- Q2. Failed operation — does someone find out, or is it logged and ignored?
- Q3. External data — is it validated before use, or trusted on arrival?
- Q4. AI-suggested pattern — do I understand why it chose this approach?
- Q5. Wrong answer — if this code is wrong, how would I find out?

The empirical case:

- On one production project (~80,000 lines of Python, one developer's work), a combination of rigorous review and internal tooling regularly catches and blocks these patterns before they enter the codebase — none have entered. The discussion paper's case study (§8) reports the observed detection rate with full methodology and caveats
- This occurs despite the AI being explicitly instructed not to produce them
- Without equivalent detection, these patterns would pass normal code review

Questions for your IT security team:

- Do we have static analysis rules that catch silent defaults, error swallowing, or unvalidated external data promotion?
- What is our interim guidance for reviewing AI-generated code?
- Should we be tracking which scripts and systems include AI-generated code?

Questions for your CTO or executive:

- Are our contracted suppliers using AI coding tools? Do we have visibility into how that code is reviewed?

- Does our security testing detect semantic defects — code that is correct but does the wrong thing?
- What would first-stage detection controls cost vs. the risk of undetected silent defects?

If you found a problem in running code:

- Assess what data it touches and the worst-case impact
- Fix the pattern using the corrected examples in this guide
- Check recent outputs for evidence of incorrect defaults or missing records
- If it touches classified, PII, or auditable data, notify your team lead
- Frame it as a quality finding, not a security incident

Three things you can do today:

- Focus your review on **hot paths** — the 30 lines where Q1–Q5 matter most (Section 5)
- Use the **AI review prompts** in Section 6 to check AI-generated code before accepting it
- Put your scripts in **version control** (git) — even if you are the only person working on them

Where to go deeper (all documents in this suite):

- This guide: practical review for people writing code
- *Governing AI-Generated Code*: systemic risk for policy advisors
- *Semantic Defects in AI-Generated Code* (discussion paper): full technical threat model and taxonomy
- Companion specification (Wardline): one illustration of semantic enforcement tooling, for tool builders and assessors

10. Specialist Perspective Prompts (Reference)

These prompts ask AI to review your code from a particular professional perspective. They are reference material — use them when you want deeper analysis beyond the pattern-specific prompts in Section 6. Each perspective catches different things because each role cares about different consequences.

Systems thinker: "Review this code as a systems thinker. What feedback loops exist? If this code runs daily, are there any cases where today's output becomes tomorrow's input and a small error could compound over time? Are there places where a silent default today would be treated as real data tomorrow?"

Systems engineer: "Review this code as a systems engineer. For each operation that could fail: what is the failure mode? Does the code degrade gracefully or does it mask the failure? Is there any operation where a partial failure could leave the system in an inconsistent state?"

Quality engineer: "Review this code as a quality engineer. For each output this code produces: how would I verify that the output is correct, not just that the code ran without errors? Are there any operations where the code could produce a plausible but incorrect result that would pass every automated check?"

Data engineer: "Review this code as a data engineer. Trace the data from its source to its final destination. Are there places where a NULL or missing value would propagate silently through transformations and appear as a valid-looking value in the output? If the source schema changed, would this code fail visibly or produce silently wrong results?"

Auditor: "Review this code as an auditor. For each decision the code makes: is there a record of what decision was made, what input led to that decision, and when it happened? Are there any paths where an operation completes successfully but leaves no trace that it happened?"

Privacy officer: "Review this code as a data protection officer. What personal information, credentials, or sensitive data passes through this code? For each piece: is it written to any log, error message, temporary file, or output that persists beyond the immediate operation?"

Security analyst: "Review this code as a security analyst. What inputs does this code accept, and are any of them user-controllable or externally influenced? Could any input be crafted to change the code's behaviour? If an attacker controlled the external data source, what is the worst they could achieve?"

Data analyst: "Review this code as a data analyst. Are there cases where NULL values would be silently excluded from a count or average? Could any filter condition accidentally exclude valid records? Are there any places where the code assumes data completeness that the source cannot guarantee?"

Glossary

Audit trail: A chronological record of who did what, when, and why. In government systems, gaps in the audit trail can have legal consequences — a missing record may be treated as evidence of tampering rather than as a technical failure.

Authority tier: A way of classifying data by how much you should trust it. This guide uses three practical levels: *internal* (produced by your system — highest trust), *validated* (came from outside but passed through checks), and *external* (not yet checked — lowest trust). The discussion paper (§5) introduces the four-tier model; the Wardline specification (§4) formally defines it, adding a distinction between shape-validated and semantically validated data. The patterns in this guide are dangerous because AI does not distinguish between tiers — it treats all data the same.

CI pipeline (Continuous Integration): An automated system that runs checks on code every time it is changed. If you do not have one, the five questions in this guide are your manual equivalent.

COALESCE: A SQL function that returns the first non-NULL value from a list. `COALESCE(security_classification, 'OFFICIAL')` is the SQL equivalent of Python's `.get()` with a default — it silently substitutes a value when the real one is missing.

Fabricated Default: The taxonomy name for the Friendly Default pattern (Section 2.1). The code presents a confident result based on fabricated input — it fabricates a default value instead of admitting it does not know. (Discussion paper reference: ACF-S1.)

Audit Trail Destruction: The taxonomy name for the Helpful Error Handler pattern (Section 2.2). A broad exception handler catches an audit-critical failure and logs it instead of propagating it, creating a gap in the legal record. (Discussion paper reference: ACF-R1.)

Authority Tier Conflation: The taxonomy name for the Invisible Promotion pattern (Section 2.3). Data from an untrusted external source is used in a trusted internal context without validation, silently elevating its authority. (Discussion paper reference: ACF-T1.)

Default value: A value that code uses when the real value is missing. In general software, defaults are helpful. In government systems handling sensitive data, a default can silently fabricate an answer that should have been an error.

Defence-in-depth: A security principle where multiple independent layers of protection are used so that no single layer's failure compromises the whole system. In the context of this guide: the combination of your own review (Q1–Q5), AI-assisted prompted review (Section 6), and eventually automated enforcement (Section 8) form three layers — each catches different things.

Implicit Privilege Grant: The taxonomy name for the closely related pattern described at the end of Section 2.3 — code that accepts an external system's assertion (e.g., `"verified": true`) and grants access or privilege based on that claim without independent verification. Rated *Critical*, the same as Authority Tier Conflation. T1 is about *data* crossing a trust boundary; E1 is about *privilege decisions* based on unvalidated external claims. (Discussion paper reference: ACF-E1.)

Exception handler: Code that catches errors and decides what to do with them. The danger in government systems is handlers that catch errors and quietly continue, rather than surfacing the failure to someone who can act on it.

Hot path: A section of code where the five questions matter most — where the code touches sensitive data, makes decisions about missing values, crosses trust boundaries, or handles errors from compliance-critical operations.

IRAP (Information Security Registered Assessors Program): A programme administered by ASD/ACSC that accredits security assessors to evaluate government systems against the ISM. When this guide or the discussion paper refers to "IRAP assessment," it means an independent security evaluation conducted by an accredited assessor — the primary mechanism by which government agencies demonstrate that their systems meet security requirements.

ISM (Information Security Manual): The Australian Government's cybersecurity framework, maintained by the Australian Signals Directorate (ASD) and its Australian Cyber Security Centre (ACSC). The patterns in this guide sit in a gap that the ISM was not designed to cover — semantic correctness of application logic.

PII (Personally Identifiable Information): Data that can identify a specific person — names, tax file numbers, dates of birth, contact details. Government systems handling PII have legal obligations about how it is stored, logged, and transmitted.

Pre-commit hook: An automated check that runs on your code before it is saved to version control. If the check fails, the save is blocked. This guide's five questions are the manual equivalent of what a pre-commit hook does automatically.

PSPF (Protective Security Policy Framework): The Australian Government's framework for protective security, including information classification. The Friendly Default pattern (Section 2.1) can silently violate PSPF classification requirements by defaulting to OFFICIAL when the real classification is unknown.

Semantic correctness: Whether code does the right thing in its institutional context — not just running without errors, but producing correct results for the specific system it operates in. The five questions in this guide are all tests for semantic correctness. See the discussion paper (§1.3) for the formal definition.

Semantic defect: A defect where the code runs correctly but does the wrong thing. It does not crash, does not throw errors, and passes all tests — but produces an incorrect result. The patterns in this guide are all semantic defects.

Silent failure: When code fails without telling anyone. The opposite of crashing. In most software, silent failures are considered poor practice. In government systems, they can mean incorrect classifications, broken audit trails, or unvalidated data entering decision-making processes.

Static analysis: Automated examination of code without running it — a tool reads the source text and flags patterns that match known problems. Linters, type checkers, and security scanners are all forms of static analysis.

Trust boundary: The line between data you control and data from outside. Every time data crosses this boundary, it should be validated. AI routinely generates code that ignores trust boundaries because the programming language does not distinguish between internal and external data.

Semantic enforcement tooling: A new category of automated check that encodes institutional rules — "this field must never receive a default," "this error must reach the audit trail" — into machine-enforceable declarations. The three pattern checks from this guide (Q1–Q3) — fabricated defaults, swallowed exceptions, and unvalidated trust boundary crossings — are the kind of rules semantic enforcement tooling targets; the two judgement calls (Q4–Q5) remain with human reviewers. The companion Wardline specification illustrates one way to build this — formal pattern rules, structural verification rules, and a governance model — but several implementation paths are viable, including custom rules for existing static analysis tools (such as semgrep or CodeQL).

ASD/ACSC (Australian Signals Directorate / Australian Cyber Security Centre): The Australian Government's lead agency for cybersecurity. ASD maintains the ISM; ACSC is its operational arm providing cybersecurity advice and incident response.

Essential Eight: A set of eight prioritised mitigation strategies published by ASD/ACSC to help organisations protect against cybersecurity incidents. The Essential Eight focuses on infrastructure-level security (patching, application control, privilege restriction) — the patterns in this guide sit in the gap between infrastructure security and application-level semantic correctness.