Code Review Checklist for AI-Generated Code

Security, hidden complexity, dependency and licence checks, with examples and a copy-paste PR template.

Version 1.0 · October 2026

Who this is for

You use an AI assistant to write code, or you review pull requests from people who do. The code compiles, the tests pass, and it looks clean. That is exactly why it needs a different kind of review.

This checklist covers the problems that show up more often in AI-generated code than in code a person typed by hand, and how to catch each one before it merges.

Each section has:

- **What goes wrong**, and why AI makes it more likely
- **The checklist** to run through
- **Examples** of what to look for and what to write instead
- **Tools** that catch part of it automatically

The Python examples marked **runnable** work as they are with the standard library on Python 3.10 or later. The others are short illustrations.

Why AI code fails differently

Human mistakes usually look like mistakes: a typo, a messy function, a half-finished idea. AI mistakes look finished. The code is well named, well formatted and confidently commented, so reviewers relax.

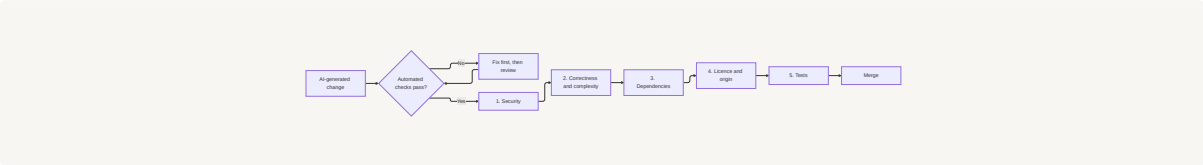
Failure	What it looks like
Plausible but wrong	Code that handles the happy path perfectly and ignores everything else
Outdated APIs	Functions that were deprecated or removed in recent library versions

Failure	What it looks like
Invented APIs	Parameters, methods or whole packages that do not exist
Insecure defaults	Shortcuts copied from tutorials: `verify=False`, `shell=True`, string-built SQL
Over-engineering	Three classes and a factory where one function would do
Tests that prove nothing	Tests that check the mock instead of the behaviour
Unknown origin	Blocks that may closely match licensed code from elsewhere

The rule for the whole checklist: if you cannot explain a line, it does not merge.

How to use this checklist

Run the automated checks first, so human review time goes on the things tools cannot judge.



For a small change, the **two-minute triage** is enough:

- [ ] Can the author explain every line?
- [ ] Does it add a new dependency?
- [ ] Does it touch input from users, files, the network or the database?
- [ ] Does it touch authentication, payments or personal data?

Any “yes” from the last three means the full checklist.

---

1. Security

What goes wrong

AI assistants learned from a huge amount of public code, including tutorials and quick answers that skip security to keep examples short. Those shortcuts come back in generated code, written as confidently as everything else.

Checklist

- [ ] No SQL, shell commands or file paths built from user input with string formatting
- [ ] No secrets, tokens or keys in the code, comments or tests
- [ ] TLS verification is not turned off
- [ ] Untrusted data is never loaded with `pickle` or `yaml.load`
- [ ] Tokens, passwords and IDs use `secrets`, not `random`
- [ ] Every outbound HTTP call has a timeout
- [ ] Errors do not leak stack traces, queries or internal paths to users
- [ ] New endpoints check authentication and permissions, not only that the user is logged in
- [ ] Logs do not record passwords, tokens or personal data

What AI often writes, and what to write instead

Risk	Often generated	Write instead
SQL injection	`f"... WHERE id = {user_id}"`	Parameterised query: `"... WHERE id = ?", (user_id,)`
Shell injection	`subprocess.run(cmd, shell=True)`	`subprocess.run(["git", "log", branch])`
Disabled TLS	`requests.get(url, verify=False)`	Fix the certificate, keep verification on
Hardcoded secret	`API_KEY = "sk-..."`	`os.environ["API_KEY"]` or a secrets manager
Unsafe loading	`yaml.load(data)`, `pickle.loads(data)`	`yaml.safe_load(data)`, or JSON
Weak tokens	`random.choice(...)`	`secrets.token_urlsafe(32)`
No timeout	`requests.get(url)`	`requests.get(url, timeout=10)`
Path traversal	`open(UPLOAD_DIR + name)`	Resolve the path and check it stays inside the folder

Example: SQL injection (runnable)

```python
import sqlite3

db = sqlite3.connect(":memory:")
db.execute("CREATE TABLE users (email TEXT, role TEXT)")
db.execute("INSERT INTO users VALUES ('a@x.com', 'user')")
db.execute("INSERT INTO users VALUES ('b@x.com', 'admin')")


def find_user_unsafe(email: str) -> list:
    # Typical generated code: user input pasted into SQL
    query = f"SELECT * FROM users WHERE email = '{email}'"
    return db.execute(query).fetchall()


def find_user_safe(email: str) -> list:
    # The driver handles the value; it is never read as SQL
    query = "SELECT * FROM users WHERE email = ?"
    return db.execute(query, (email,)).fetchall()


attack = "nobody' OR '1'='1"
print(find_user_unsafe(attack))  # returns every user
print(find_user_safe(attack))    # returns []
```

Example: path traversal (runnable)

```python
from pathlib import Path

UPLOAD_DIR = Path("/srv/uploads").resolve()


def upload_path_unsafe(name: str) -> Path:
    return UPLOAD_DIR / name  # "../../etc/passwd" escapes


def upload_path_safe(name: str) -> Path:
```

```
    path = (UPLOAD_DIR / name).resolve()
    if not path.is_relative_to(UPLOAD_DIR):
        raise ValueError("Path outside upload folder")
    return path


print(upload_path_unsafe("../../etc/passwd").resolve())
try:
    upload_path_safe("../../etc/passwd")
except ValueError as err:
    print("Blocked:", err)
```

> **Your example:** add a security issue you caught in a real review here.

---

2. Correctness and hidden complexity

What goes wrong

Generated code is usually right for the case described in the prompt and quiet about everything else. It may also use APIs from an older version of a library, invent a parameter that sounds right, or wrap a simple job in layers of abstraction.

Checklist

Correctness

- [ ] Every function, method and parameter used actually exists in the version you run
- [ ] No deprecated APIs (check the library's changelog if unsure)
- [ ] Empty input, `None`, zero, very large input and duplicates are handled
- [ ] Time zones, encodings and locale are explicit where they matter
- [ ] Exceptions are not swallowed with a bare `except` or `except Exception: pass`
- [ ] Retries, loops and recursion have a limit

Complexity

- [ ] No database query inside a loop (the N+1 problem)
- [ ] No class, base class or factory that only has one use
- [ ] No new helper that duplicates one already in the codebase
- [ ] It follows the project's existing patterns, not a generic tutorial style
- [ ] Comments explain why, and match what the code actually does

Outdated APIs to watch for in Python

These are common in generated code because older examples dominate the training data.

Generated	Status	Use instead
`datetime.utcnow()`	Deprecated since Python 3.12	`datetime.now(timezone.utc)`
`df.append(row)` in pandas	Removed in pandas 2.0	`pd.concat([df, new_rows])`
`asyncio.get_event_loop()` to start code	Discouraged	`asyncio.run(main())`
`requests.get(url)` with no timeout	Works, but can hang forever	Always pass `timeout=`

Example: the N+1 query (Django, illustration)

```
# Generated: one query for orders, then one more per order
orders = Order.objects.filter(status="open")
for order in orders:
    print(order.customer.name)  # a query every time

# Reviewed: two queries in total, however many orders
orders = (
    Order.objects.filter(status="open")
    .select_related("customer")
)
```

```
for order in orders:
    print(order.customer.name)
```

Both versions pass the same tests on a small test database. The difference only shows in production, with real data.

Example: swallowed errors (illustration)

```
# Generated: looks safe, hides every failure
try:
    sync_customer(record)
except Exception:
    pass

# Reviewed: catch what you expect, log it, let the rest fai
l
try:
    sync_customer(record)
except TimeoutError:
    logger.warning("Sync timed out for%s", record.id)
    retry_queue.put(record)
```

> **Your example:** add a correctness or complexity issue from your own reviews here.

---

3. Dependencies

What goes wrong

AI assistants sometimes suggest packages that do not exist. The names sound right, so people install them. Attackers have started registering those invented names with malicious code inside, an attack known as **slopsquatting**. Separately, generated code often adds a dependency for something the standard library already does.

Checklist

- [ ] Every new package exists on the official registry under exactly that name
- [ ] It is the package you think it is: check the project page, repository and maintainer
- [ ] It is not brand new, or you have a clear reason to trust it
- [ ] It is maintained: recent releases, open issues being answered
- [ ] It has no known vulnerabilities
- [ ] The version is pinned in your lock file or requirements
- [ ] It is needed at all: the standard library or an existing dependency cannot do the job
- [ ] Its licence is acceptable (see section 4)

Signs a package needs a closer look

Signal	Why it matters
Not found on the registry	Likely invented by the AI. Do not search for a “close enough” name
First release in the last few weeks	Newly registered names are where slopsquatting attacks live
Name one letter off a popular package	Classic typosquatting
No repository link, or an empty repository	Nothing to inspect
Very few downloads	Few people have looked at it

Script: check new Python dependencies (runnable)

Pass the package names an AI suggested. The script asks PyPI whether each one exists and how old it is.

```
"""Flag packages that may be invented or too new to trus
t."""
import json
import sys
import urllib.error
import urllib.request
```

```
from datetime import datetime, timezone

PYPI = "https://pypi.org/pypi/{}/json"
MIN_AGE_DAYS = 90


def first_release(data: dict) -> datetime | None:
    times = [
        f["upload_time_iso_8601"]
        for files in data["releases"].values()
        for f in files
    ]
    if not times:
        return None
    earliest = min(times).replace("Z", "+00:00")
    return datetime.fromisoformat(earliest)


def check(name: str) -> str:
    url = PYPI.format(name)
    try:
        with urllib.request.urlopen(url, timeout=10) as r:
            data = json.load(r)
    except urllib.error.HTTPError as err:
        if err.code == 404:
            return f"NOT FOUND{name}: possibly invented"
        raise
    born = first_release(data)
    if born is None:
        return f"NO FILES{name}: no published releases"
    age = (datetime.now(timezone.utc) - born).days
    if age < MIN_AGE_DAYS:
        return f"VERY NEW{name}: first release{age} days ag
o"
    return f"ok{name}:{age} days old"


if __name__ == "__main__":
```

```
    for package in sys.argv[1:]:
        print(check(package))
```

Run it like this:

```
python check_deps.py requests flask-auth-helper-pro
```

A package that passes this check is not automatically safe. It has only cleared the first and most common trap.

> **Your example:** add a dependency you rejected in a real review, and why.

---

4. Licence and origin

What goes wrong

AI assistants can occasionally produce code that closely matches existing public code, and that code has a licence. Generated code also pulls in dependencies whose licences you never chose. In a commercial or client project, both can turn into legal obligations nobody agreed to.

This section is practical guidance, not legal advice. For anything unclear, ask your legal team or the client's.

Checklist

- [ ] No large block that looks lifted from somewhere: unusual comments, another project's names, a different code style
- [ ] No licence headers or copyright lines copied in from another project
- [ ] If your AI tool has a setting to block or flag suggestions that match public code, it is turned on
- [ ] Licences of all new dependencies are on your approved list
- [ ] No strong copyleft licence (GPL, AGPL) in code you distribute or run as a service for a client, unless approved
- [ ] Your team's or client's policy on AI-generated code is followed, including any disclosure rule

Licence quick reference

Licence family	Examples	Usually means
Permissive	MIT, BSD, Apache-2.0	Fine for most commercial use; keep the notices
Weak copyleft	LGPL, MPL-2.0	Fine if used as a library; changes to it may need sharing
Strong copyleft	GPL-3.0	Distributing your software can require releasing your code
Network copyleft	AGPL-3.0	Offering it as a service can require releasing your code
No licence	Code with no licence file	You have no right to use it; avoid

Commands: list dependency licences

```
pip install pip-licenses
pip-licenses --order=license
pip-licenses --fail-on="GPL;AGPL" --partial-match
```

The last command exits with an error if any installed package has a licence name containing GPL or AGPL, so you can use it as a CI check. Note that this also matches LGPL; adjust the list to your own policy.

---

5. Tests

What goes wrong

Ask an AI for tests and you get tests that pass. That is not the same as tests that would fail if the code were wrong. A common pattern is a test that mocks the function under test and then asserts that the mock returned what it was told to return.

Checklist

- [ ] Each test would fail if the behaviour it describes broke (try breaking it once)
- [ ] The test checks the real function, not a mock of it
- [ ] Mocks are used only at the edges: network, clock, external services

- [ ] Edge cases and error paths are tested, not only the happy path
- [ ] Assertions check specific values, not only `is not None` or "no exception"
- [ ] Generated tests were read, not only run

Example: a test that proves nothing (illustration)

```python
# Generated: mocks the thing it claims to test
def test_apply_discount(mocker):
    mocked = mocker.patch("shop.apply_discount")
    mocked.return_value = 90
    assert shop.apply_discount(100, 10) == 90

# Reviewed: tests the real function, including edges
def test_apply_discount():
    assert shop.apply_discount(100, 10) == 90
    assert shop.apply_discount(100, 0) == 100
    with pytest.raises(ValueError):
        shop.apply_discount(100, 120)
```

The first test passes even if `apply_discount` is deleted.

Automate what you can

Tools catch a lot of the mechanical problems, which leaves reviewers free for logic, design and intent.

Tool	Catches
`ruff`	Style issues, unused code, many bug patterns
`mypy` or `pyright`	Wrong types, including calls to methods that do not exist
`bandit`	Common Python security problems: `shell=True`, `pickle`, hardcoded passwords
`semgrep`	Security and custom rules across many languages
`pip-audit`	Dependencies with known vulnerabilities
`pip-licenses`	Dependency licences
`gitleaks`	Secrets committed to the repository

Tool	Catches
Dependabot or Renovate	Outdated and vulnerable dependencies over time

A starting CI step

```
pip install ruff bandit pip-audit pip-licenses
ruff check .
bandit -r src -ll
pip-audit
pip-licenses --fail-on="GPL;AGPL" --partial-match
```

`bandit -ll` reports only medium and high severity issues, which keeps the noise down when you first add it. Change `src` to your code folder.

Copy-paste PR template

Add this to your repository as `.github/pull_request_template.md`, or paste it into the PR description.

```
## AI-assisted code review

-[ ] I can explain every line in this change
-[ ] Automated checks pass (lint, types, security, audit)

**Security**
-[ ] No user input in SQL, shell commands or file paths
-[ ] No secrets in code, tests or comments
-[ ] Outbound calls have timeouts; TLS verification is on

**Correctness**
-[ ] All APIs used exist in our versions
-[ ] Edge cases and errors are handled, not swallowed
-[ ] No queries inside loops

**Dependencies and licences**
-[ ] New packages verified on the registry and pinned
-[ ] Licences are on the approved list
```

```
**Tests**
-[ ] Tests fail when the behaviour is broken
-[ ] Tests check real code, not mocks of it
```

Red flags at a glance

If you see	Check
`shell=True`, `eval`, `exec`	Command and code injection
An f-string inside `execute(`	SQL injection
`verify=False`	Disabled TLS
`except Exception: pass`	Hidden failures
A package name you have never heard of	Section 3
A licence header or another project's name	Section 4
`mocker.patch` on the function being tested	Section 5
A new class hierarchy for a small task	Over-engineering
Very confident comments	Whether the code does what they say