

## 11 · Code and platforms

# Checklist for Reviewing AI-Generated Code

What to check in AI-written code before merging: validation, authorisation, secrets, dependencies, errors, tests and deployment, with two annotated examples.

For: Anyone building sites or tools with AI help who needs to know what to check before code reaches real users, whether a junior developer or a project owner reviewing with their team.

Level: Intermediate      Version 1.0 · 2026-09-21

AI-written code usually works on the "happy path": the user fills in fields correctly, asks for their own data, and the network never drops. Problems appear everywhere else: a user changes a number in the URL, a field is empty, an error message exposes server details, a secret key is pasted into a file. This checklist gives you a fixed set of questions to ask of every change before it reaches real users.

**Important:** this checklist is not a substitute for a professional security audit. It is a minimum that catches common mistakes. Any platform holding sensitive personal data (health, financial, about minors) or taking payments needs a review by a security professional before launch and after major changes.

## How to review

1. **Read what changed, not the whole project.** Review the diff between the previous and new version, file by file. A very large change should be split before review.
2. **Ask the tool to explain.** "Explain what each part does, which assumptions you made, and which inputs could break it." If you do not understand the explanation, do not merge the code.
3. **Run the code yourself.** Happy path first, then the unhappy ones: empty field, very long value, signed-out user, a different user.
4. **Go through the nine sections below.** Mark ✓, X or n/a for each item.
5. **Record the decision:** merge, revise or reject, plus what you learned to add to your standing instructions for the tool.

## 1. Functionality

- ☐ The code does only what was asked and did not add unrequested features, files or pages.
- ☐ You actually ran it on your machine or a test environment, rather than taking the tool's word that it "works".

- ☐ Edge cases are covered: empty list, a single item, very large values, dates across time zones.
  - ☐ Nothing that used to work was removed or broken (check files that changed when you did not expect them to).
  - ☐ No "invented" functions or libraries that do not actually exist.
- 

## 2. Input validation

- ☐ Everything from the user (forms, URLs, files, request headers) is treated as untrusted.
  - ☐ Validation happens on the server, not only in the browser; browser validation is a convenience, not protection.
  - ☐ For each field: type, maximum length, allowed format, and whether it is required.
  - ☐ Database queries are parameterised, not strings built from user input.
  - ☐ User input shown on the page is escaped, not inserted as raw HTML.
  - ☐ File uploads are limited by type and size, and no uploaded file is ever executed.
- 

## 3. Authentication and authorisation

- ☐ Every route that shows or changes data first checks that the user is signed in.
  - ☐ It then checks that this user owns this specific record or has the required role (see the permissions matrix, if there is one).
  - ☐ You tried changing the ID in the URL or request to another user's ID, and the result was a refusal.
  - ☐ Hiding a button is not a permission; the server refuses the request even when sent directly.
  - ☐ In databases that support row-level security, policies are enabled and tested with two different users.
  - ☐ Responses do not return fields the user does not need (password hashes, internal notes, other users' data).
- 

## 4. Secrets

- ☐ No keys, passwords or access tokens written in the code; all of them live in environment variables or a secrets store.
- ☐ The local environment file is excluded from the repository, with an example file listing variable names only.
- ☐ No secrets in browser code: anything sent to the browser is visible to every visitor.
- ☐ Keys have the narrowest permissions possible (a read-only key if that is enough).
- ☐ If a key appeared in code, in a chat with a tool or in a log, it is exposed: rotate it immediately; deleting it is not enough.

---

## 5. Dependencies

- ☐ Every new library the tool added actually exists, with its exact name, from its official source (look-alike names are a known trap).
- ☐ The library is truly needed, and a few lines of code or a library already in the project would not do.
- ☐ It is maintained with recent updates, and the package manager's vulnerability check shows no serious issues.
- ☐ The lock file is updated and committed with the change.

---

## 6. Errors and logs

- ☐ Errors are caught and do not bring down the whole application.
- ☐ The user-facing error message is clear, in the user's language, and reveals no technical details (file paths, queries, versions).
- ☐ Logs contain enough to diagnose, and no passwords, keys or full personal data.
- ☐ External calls (email, payments, APIs) have a timeout and defined behaviour on failure.
- ☐ No errors silently "swallowed" by an empty catch block.

---

## 7. Tests

- ☐ The change has at least one happy-path test and one refusal test (unauthorised user, invalid input).
- ☐ Tests actually fail when you break the code on purpose (a test that never fails tests nothing).
- ☐ The tool did not "fix" a failing test by changing the test to match the bug.
- ☐ All existing tests still pass.

---

## 8. Accessibility (for interfaces)

- ☐ Real interactive elements (a button for actions, a link for navigation), not generic elements with a click handler.
- ☐ Every field has an associated label, and error messages are announced to screen readers.
- ☐ Everything is reachable by keyboard, with a visible focus indicator.
- ☐ Images have alt text, and contrast is sufficient.
- ☐ Right-to-left direction is correct on Arabic pages, and the page language is declared.

---

## 9. Deployment

- ☐ Environment variables are set in production, not left at test values.
- ☐ Debug mode is off in production.
- ☐ Database migrations are reviewed, do not delete data, and a recent backup exists before they run.
- ☐ There is a known way to roll back to the previous version if the deploy fails.

- ☐ After deploying: a quick check of the core paths on the live site.

## Annotated example 1: a route that returns a booking

### Illustrative example

A fictional app for **Nabd Clinic** shows patients their booking details. An AI tool was asked for "a route that returns booking data by its ID" and wrote the following (Express style, simplified for illustration):

#### Before review

```
// GET /api/bookings/:id
app.get('/api/bookings/:id', async (req, res) => {
  const booking = await db.bookings.findById(req.params.id); // [1]
  res.json(booking); // [2]
});

const SMS_API_KEY = 'sk_live_EXAMPLE_ONLY'; // [3]
```

- **[1] No identity or ownership check.** Anyone, signed in or not, can try consecutive numbers and read every patient's bookings. This is one of the most common and serious flaws.
- **[2] Returns the full record.** It may include the phone number and the therapist's internal notes. And if the booking does not exist, it returns an empty value with no clear message.
- **[3] A secret key in the code.** It will end up in the repository and every copy of it. (The key here is fake, for illustration.)

#### After review

```
// GET /api/bookings/:id
app.get('/api/bookings/:id', requireLogin, async (req, res) => { // [1]
  const booking = await db.bookings.findById(req.params.id);
  if (!booking || booking.patientId !== req.user.id) { // [2]
    return res.status(404).json({ error: 'Not found' });
  }
  res.json({ // [3]
    id: booking.id,
    date: booking.date,
    status: booking.status,
  });
});

const SMS_API_KEY = process.env.SMS_API_KEY; // [4]
```

- **[1]** The route only works for a signed-in user.
- **[2]** The booking must belong to this user. The same 404 is returned in both cases, so an attacker cannot tell whether the ID exists.

- **[3]** Only the fields the interface needs are returned.
- **[4]** The key lives in an environment variable. If the real key was ever committed, it must be rotated with the provider, not just removed from the code.

**How to test:** sign in as two users; with the first, request a booking ID belonging to the second; expect 404. Then request it signed out; expect a refusal.

## Annotated example 2: a contact form

### Illustrative example

A fictional site for **Al-Zaytouna Bakery** has a contact form. The code the tool produced:

#### Before review

```
// POST /api/contact
app.post('/api/contact', async (req, res) => {
  const { name, phone, message } = req.body;           // [1]
  await db.query(
    "INSERT INTO messages (name, phone, message) VALUES ('" +
    name + "', '" + phone + "', '" + message + "')"      // [2]
  );
  res.send('Thanks ' + name);                           // [3]
});
```

- **[1] No validation.** Empty fields, a million-character message, a random phone number: all accepted. And there is no limit on submissions, so the table is easy to flood with spam.
- **[2] A query built from user input.** This is the door to SQL injection: a carefully crafted input could read or delete data.
- **[3] Echoes the name as-is inside an HTML response.** A name containing script could run in the browser.

#### After review

```
// POST /api/contact (rateLimit: illustrative, use your framework's limiter)
app.post('/api/contact', rateLimit({ max: 5, perMinutes: 10 }), async (req, res) => {
  const name = String(req.body.name || "").trim();           // [1]
  const phone = String(req.body.phone || "").trim();
  const message = String(req.body.message || "").trim();
  const phoneOk = /^[0-9+ ]{7,20}$/.test(phone);
  if (!name || name.length > 80 || !phoneOk || message.length > 1000) {
    return res.status(400).json({ error: 'invalid_input' });
  }
  await db.query(                                           // [2]
    'INSERT INTO messages (name, phone, message) VALUES ($1, $2, $3)',
    [name, phone, message]
  );
  res.json({ ok: true });                                   // [3]
});
```

- **[1]** Each field is converted to text, trimmed, and checked for length and format on the server, with a limit on submissions from the same source.
- **[2]** A parameterised query: values are passed separately from the query text, so they are never interpreted as commands.
- **[3]** The response is data, not HTML; the interface shows its own thank-you text and never inserts user input as HTML.

**How to test:** submit the form empty, with a very long message, and with letters in the phone number; expect a refusal with a clear message in the interface. Submit six times in a row; the sixth is refused.

## Self-review prompt

Use it after the tool writes the code, in a fresh chat or with a second tool. The result helps you; it is not a final verdict. The decision belongs to the reviewer.

#### Code review prompt

Review the following code as a strict security reviewer. Do not rewrite it all.

- 1) Explain what it does in 5 lines.
- 2) For each route: does it check sign-in? Ownership or role?
- 3) Where is user input used without validation or escaping?
- 4) Are any secrets or settings hard-coded?
- 5) What happens when each external call fails?
- 6) Which three most important tests are missing?

Order findings from most to least serious, with line numbers.

Code: [paste code after replacing any real key or data with fake values]

---

## When to stop and call a professional

- The platform stores health or financial data, or data about minors.
- It takes payments or manages balances.
- You built your own sign-in or encryption instead of using a well-known service or library.
- You could not understand the tool's explanation of a sensitive part of the code.
- Something odd appears in the logs or database with no explanation.

### Common mistakes

- **Mistake:** merging code because it "works" on the first try. **Fix:** "works" means the happy path only; test refusals and edge cases.
- **Mistake:** relying on hidden buttons as permissions. **Fix:** the server refuses every unauthorised request, even one sent directly.
- **Mistake:** pasting real keys into the chat with the tool "to get it working quickly". **Fix:** use variable names and fake values; if a leak happens, rotate the key.
- **Mistake:** accepting a new library you have never heard of. **Fix:** check the exact name, source and maintenance before installing.
- **Mistake:** reviewing a huge change in one go. **Fix:** ask the tool for small changes and review each separately.
- **Mistake:** letting the tool edit a test to match the bug. **Fix:** the test represents the requirement; the code is what gets fixed.

---

## Completion checklist before merging

- ☐ You understand what the code does and ran it yourself.
- ☐ Inputs are validated on the server; queries are parameterised.
- ☐ Every route checks identity and ownership or role, and you tried access as another user.
- ☐ No secrets in code, browser code or logs.
- ☐ New dependencies are real, necessary and maintained.
- ☐ Error messages reveal no technical details.
- ☐ At least one success test and one refusal test; all existing tests pass.
- ☐ Interfaces work with keyboard and screen reader.
- ☐ The deploy has a backup and a rollback path.
- ☐ If data is sensitive: a professional security review is scheduled.

---

**About the author** — Mohamad Charafeddine is an entrepreneur from Tripoli, Lebanon: founder of the Passion digital marketing agency, president of Takaful Youth Association, and founder of digital platforms. He trains individuals and teams in applied AI and marketing.

Examples in this file are fictional. Free to use in your work and organisation; credit the source if you republish.  
[mohamadcharafeddine.com/en/resources/ai-code-review-checklist/](https://mohamadcharafeddine.com/en/resources/ai-code-review-checklist/)