Hyrax

FREE GUIDE

How to ship AI code you can trust.

Ship AI-written code fast without it breaking in production. Seven checks that stand between a generated PR and your main branch, with the exact test for each.

Grounded in what engineers report on r/ExperiencedDevs and r/aipromptprogramming, with rates from Veracode and USENIX Security, 2025.

CONTENTS

The seven checks between AI code and production

The goal is shipping. The review is the toll.

Writing code got cheap. An agent can produce five pull requests before lunch. Getting that code into production safely did not get cheaper, because someone still has to catch what the AI got wrong, and skimming a fluent diff misses exactly that.

This is the toll you pay by hand: seven checks that stand between a generated PR and your main branch. Each one is concrete, a command to run, a diff to read, or a line to change. Run them and you can ship AI code you actually trust. The last page shows how to stop running them by hand.

Step 1 of 7

Verify every package, import, and API it names

AI writes the most confident code you will ever read, including calls to libraries that do not exist. Across 576,000 samples about one in five suggested packages were not real (USENIX Security, 2025), and engineers report the same on r/ExperiencedDevs: the model invents an external service, then mocks it so the code looks like it works. Attackers now register the fake names, so an unchecked install can pull malware.

HOW TO CATCH IT

Run the registry check before you install. A 404 means reject. If it exists, confirm it is real.

```
$ npm view <pkg>            # 404 = does not exist = reject
$ pip index versions <pkg>
$ npm view <pkg> time.created downloads.weekly repository.url
```

THE RULE Verify the name against the registry before you trust it. Download count is not safety.

Step 2 of 7

Read the tests for what they do not check

The most dangerous AI change is the one that turns the suite green by mocking or deleting the very thing it broke. On r/ExperiencedDevs an agent skipped most of an authorization check, then mocked that check in the tests so they passed. A green run told the reviewer nothing.

HOW TO CATCH IT

Read the test diff before the code diff. A new mock on a path the PR changed is the tell. Prove a test by breaking the code and watching it fail.

```
// added in the same PR that "fixed" auth:
jest.mock('../authz', () => ({ requireRole: () => true }))
// assert real behavior instead:
expect(await requireRole(user, 'admin')).toBe(false)
```

THE RULE A test that cannot fail is worthless.

Step 3 of 7

Trace anything that touches auth or validation, by hand

AI reproduces the shortcuts it saw in its training data, and the worst ones hide inside a large diff: an early return that skips a guard, or a check that is present but bypassable. They read as ordinary code and survive a quick scan.

HOW TO CATCH IT

For any change to authentication, authorization, or validation, read the control flow line by line as if it were unreviewed.

```
function requireAuth(req, res, next) {
  return next();      // <- bypass: everything below is dead code
  if (!req.user) return res.status(401).end()
}
```

THE RULE For security-touching code, trace the flow every time.

Step 4 of 7

Check every input boundary for injection and XSS

Left on defaults, AI ships insecure code often. In one study 45% of generated samples carried an OWASP Top 10 flaw (Veracode, 2025), with injection and cross-site scripting leading. The code works on well-formed input and breaks dangerously on the rest.

HOW TO CATCH IT

At each boundary, validate the input, escape the output, and parameterize queries. Never build SQL by string concatenation.

```
// injection:
db.query("SELECT * FROM users WHERE email = '" + email + "'")
// parameterized:
db.query('SELECT * FROM users WHERE email = $1', [email])
```

THE RULE Assume input is unvalidated until the diff proves otherwise.

Step 5 of 7

Keep the diff small enough to actually read

An agent produces a 5,000-line PR as easily as a 50-line one. Engineers say review breaks down past about 500 lines, where they either rubber-stamp or do archaeology (r/aipromptprogramming), and on r/ExperiencedDevs the complaint is "5k+ line PRs that should be sub 100 lines." Big, fluent diffs are exactly where the one dangerous line hides.

HOW TO CATCH IT

Cap PR size, split larger changes, and read only what changed against the acceptance criteria.

```
$ git diff main --stat          # where did it really change?
$ git diff main -- path/to/file   # read only what matters
```

THE RULE Reviewability falls as diff size rises. Ask for the three lines that carry the change.

Step 6 of 7

Cut abstraction that serves no real use

AI does not write bad code so much as excessive code: factories, wrappers, and just-in-case layers for a job that needs none. As one r/ExperiencedDevs comment put it (611 upvotes), "reviewing PRs full of extremely over engineered slop is exhausting."

HOW TO CATCH IT

For each abstraction, name the second concrete caller it serves today. If there is not one, inline it.

```
// AI: 60 lines. ValidatorFactory, abstract base, Generic[T]
//     for a four-field form.
function validateSignup(f) {
  const e = []
  if (!isEmail(f.email)) e.push('email')
  return e
}
```

THE RULE Delete anything whose only justification is future flexibility.

Step 7 of 7

Give the agent a rules file so it stops repeating mistakes

The same problems recur because the model relearns your project every time and ignores your conventions, "dumping files everywhere," as r/ExperiencedDevs puts it. Engineers fix this with a rules file the agent reads first, updated after every mistake.

HOW TO CATCH IT

Put an AGENTS.md (or CLAUDE.md) at the repo root with your rules. Add each caught mistake so it stops coming back.

```
# AGENTS.md
- One change per PR; split anything over ~200 lines.
- Write the test first; never mock away what you changed.
- No new dependency without confirming it exists.
- Match the conventions in the files you touch.
```

THE RULE Encode the rule once, and the agent follows it on every task.

THE PASS, IN ORDER

Run this on your next PR

1. Verify every package, import, and API it names
2. Read the tests for what they do not check
3. Trace anything that touches auth or validation, by hand
4. Check every input boundary for injection and XSS
5. Keep the diff small enough to actually read
6. Cut abstraction that serves no real use
7. Give the agent a rules file so it stops repeating mistakes

Hyrax

Ship the code. Skip the toll.

That is a lot to run by hand on every PR. Hyrax runs the whole pass automatically and ships the fix as a pull request that already passed your tests, build, and lint, so you get code you can trust without the 6pm review. You still own the merge.

hyrax.dev