17-214/514: Agentic Software Development

Verification in the Pipeline

Hammad Ahmad and Claire Le Goues • Wednesday, November 18, 2026 • Lecture

Administrivia

- A6 has been released. The checkpoint is due Tuesday.
- You should have gotten an invitation to a new Learner Lab course for A6, with a fresh $50 budget.

What did we talk about last time?

- DevOps often means the team that builds a service also runs it. Continuous integration and delivery automate the path from commit to production.
- A pipeline is code in the repo, and it should build one image per commit and promote that image through staging to production.
- Rolling, blue/green, and canary rollouts limit how many users see a bad version, or for how long.
- Feature flags separate deploying code from releasing it. Breaking schema changes need expand and contract.
- Rollback should be fast and practiced. Blameless postmortems look for what in the system let the problem through.
- Agents add changes and also run inside the pipeline, so workflow changes and pipeline permissions need careful review.
- Today: which checks to put in the pipeline.

Learning goals for today

- Explain why checks that run on every change can keep up with agent throughput when review alone can't
- Read a finding from a linter, analyzer, or type checker, and judge whether it is a real bug, a false positive, or a finding to suppress with a reason
- Recognize when a suppression, `any`, or `as` turns a check off instead of fixing the problem, and use lint rules to flag them
- Design types that rule out mistakes, such as a union for states that can't coexist
- Turn on a strict check in an existing codebase with a baseline
- Place unit, integration, and end-to-end tests in a pipeline, fix the cause of a flaky test, and explain what a fuzz campaign adds
- Choose which checks should block a merge, and explain what no check can decide

Who reviews a 2,000-line pull request?

Suppose someone on the booking team spends an afternoon with an agent and opens one pull request that changes 2,000 lines. How carefully would you review it?

- A careful review of 2,000 lines takes hours.
- Generating a change has gotten much cheaper, but reviewing one takes about as long as it used to.
- Automated checks of a change can help.



ProgrammerHumor.io

Gates: checks that block a merge

- A "**gate**" runs on every change. The tests, the type checker, and a linter can all be gates.
  - GitHub uses the phrase "required status check." SonarQube refers to a set of them as a "quality gate."
- A good check finds every instance of the bug it looks for, in this change and in every later change.
- Writing the check is a one-time cost; it adds a little time to CI.

Dynamic and static checks

Dynamic checks
Run the code (tests, fuzzing, canaries) and check what happens on the inputs they try.

Static checks
Read the code without running it (type checkers, linters, analyzers). They can reason about paths that no test runs.

The two kinds catch different bugs, so most pipelines run both.

Static analysis

Is this code buggy?

```
21  export async function book(store: Store, notifier: Notifier, user: string,
22                             room: string, slot: TimeRange): Promise<Booking> {
23    if (await store.overlaps(room, slot)) {
24      throw new Error("Room is already booked");
25    }
26    const booking = await store.insert(user, room, slot);
27    notifier.send(user, `Booked ${room}`);
28    return booking;
29  }
```

- Its tests pass. In the tests, the fake notifier's `send()` always succeeds.

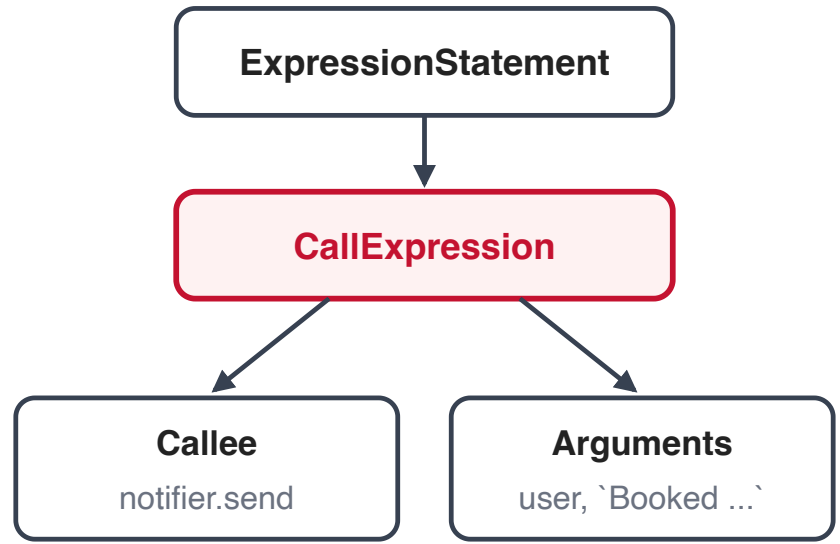
The linter found a bug the tests missed

```
…/src/booking.ts
  27:3  error  Promises must be awaited, end with a call to .catch,
               end with a call to .then with a rejection handler or
               be explicitly marked as ignored with the `void` operator
               @typescript-eslint/no-floating-promises
```

- Nothing awaits the promise from `send()`, so if it fails, no caller sees the error.
- If the mail server times out, Node treats the rejection as unhandled and, by default, exits the process. The booking is already saved, and the other requests that process was handling fail too.
- The tests pass because none of them makes `send()` fail. The linter reports the bug from the code alone.
- The fix is `await`, or a `.catch()` that logs, depending on whether a failed email should fail the booking.

How a linter works

- The linter parses the file into a tree, the same way a compiler does.
- Each rule walks the tree looking for a pattern. This one looks for a statement that is only a call, where the call returns a promise.
- "Returns a promise" needs type information, so this rule asks the TypeScript compiler (**typed linting**).
- Deeper analyzers such as CodeQL track how values flow across paths and functions, and let you write queries over those facts.



Type of the call: Promise<void>

Linters, analyzers, and type checkers

	Checks	For example
Linter	Bug-prone patterns and style, one rule at a time	ESLint, Ruff
Static analyzer	Deeper reasoning across paths and functions (a null reaching a call, a leaked file handle, user input reaching a query)	SpotBugs, Error Prone, CodeQL, Infer
Type checker	That values match the types the code declares	`tsc`, mypy

- All three are **static analyses**, which analyze the code without executing it. Some tools do more than one of these (e.g., a typed lint rule uses the type checker).

Static analysis examples

Bug	Found by
A value that can be `null` on some path, then used	Infer, NullAway, `tsc --strict`
A file or connection left open on an error path	Infer, SpotBugs
A promise or return value ignored	typescript-eslint, Error Prone
A shared field read without its lock	Error Prone, Infer
User input reaching a SQL query or shell command	CodeQL, Semgrep

```
apps/ca.c:2780: error: NULL_DEREFERENCE
  pointer `revtm` last assigned on line
  2778 could be null and is dereferenced
  at line 2780, column 6

2778. revtm = X509_gmtime_adj(NULL, 0);
2780. i = revtm->length + 1;
```

Infer on OpenSSL, from Distefano et al., "Scaling Static Analyses at Facebook", CACM 2019

- Infer traced the null through five levels of function calls. It reported 15 bugs like this to OpenSSL, and all were fixed.

Use custom lint rule to enforce project conventions.

- A custom rule checks a convention specific to your project.
- VS Code keeps its own rules in the repo. This one flags event listeners attached to the global `document`, since VS Code can have several windows.
- Write the rule's message to say how to fix the problem. An agent that runs the linter will follow it.
- An agent can usually draft a small rule. You can check it against code that should pass and code that should fail.

```
CallExpression(node: any) {
  if (
    (
      node.callee.name === 'addDisposableListener' |
      node.callee.property?.name === 'addDisposableL
    ) &&
    node.arguments.length > 0 &&
    node.arguments[0].type === 'Identifier' &&
    node.arguments[0].name === 'document'
  ) {
    context.report({
      node,
      message: 'Use <targetWindow>.document to ...',
    });
  }
},
```

VS Code's code-no-global-document-listener.ts (github.com/microsoft, trimmed)

Every static analyzer misses bugs, flags correct code, or both

- No analyzer can decide every interesting property of every program (Rice's theorem, 1953). Each one approximates, and trades between two kinds of error.
  - A **false positive** flags code that is fine. A **false negative** misses a real bug.
  - A **sound** analysis reports every instance of the bugs it looks for, and usually pays for it with more false positives.
- Most tools in a pipeline aren't sound. They aim for few false positives instead, since developers tend to ignore a check that is often wrong.
- An analyzer also doesn't know your domain's rules (e.g., two bookings for one room can't overlap). Those need tests, types, or a custom rule.

Why developers ignore analyzers

- Around 2006 to 2011, Google tried three ways to use FindBugs (a Java analyzer) on its codebase.
- Few developers used a nightly dashboard of findings, because it was outside their usual workflow.
- In a 2009 fix-it week, engineers reviewed 3,954 warnings. Bug reports were filed for 44% of them, but 16% were fixed.
- Showing FindBugs warnings in code review was dropped in 2011.

> … the presence of effective false positives caused developers to lose confidence in the tool, and developer customization resulted in an inconsistent view of analysis results.
>
> - Sadowski et al., "Lessons from Building Static Analysis Tools at Google", CACM 2018

What worked at Google and Meta

- Google's next system (Tricorder) showed analyzer findings in code review.
- An **effective false positive** is a finding the developer chooses not to act on, whether or not the tool was technically right.
- A check that breaks the build had to be essentially never wrong. A check in code review had to stay under 10%.
- Meta saw the same with Infer. Nightly bug lists got a fix rate near zero vs. 70% at review time.
- Those limits assumed a person reading every finding. An agent doesn't care, so a stricter rule costs less.
  - An agent can also make a finding go away by suppressing it.

> TRICORDER introduces an effective place to show warnings. Given that all developers at Google use code review tools before submitting changes, TRICORDER's primary use is to provide analysis results at code review time. This has the added benefit of enabling peer accountability, where the reviewer will see if the author chose to ignore analysis results. We still enforce a very low effective false positive rate here (< 10%). Additionally, we only display results for most analyses on *changed lines* by default; this keeps analysis results relevant to the code review at hand.[2] …

Sadowski et al., "Tricorder: Building a Program Analysis Ecosystem", ICSE 2015

Use a baseline to introduce new rules.

- A new rule on an old repo may report hundreds of findings at once and make everyone angry.
- A **baseline** records today's findings and fails the build only on new ones.
- ESLint does this with `--suppress-all`, which saves the counts to a file you commit.
- The old findings can then be fixed over time. Grafana commits a 4,764-line suppressions file, and pull requests that fix old findings delete entries from it.

```
$ npx eslint --suppress-all src
```

```
{
  "src/booking.ts": {
    "@typescript-eslint/no-floating-promises": {
      "count": 1
    }
  }
}
```

`eslint-suppressions.json`

Reasons for suppressions should be clear/documented.

- Some findings are false positives; some are accepted on purpose. A suppression comment silences one line.
- Write the reason in the suppression comment. A reviewer can check it, and a later reader can see why the finding was left in place.

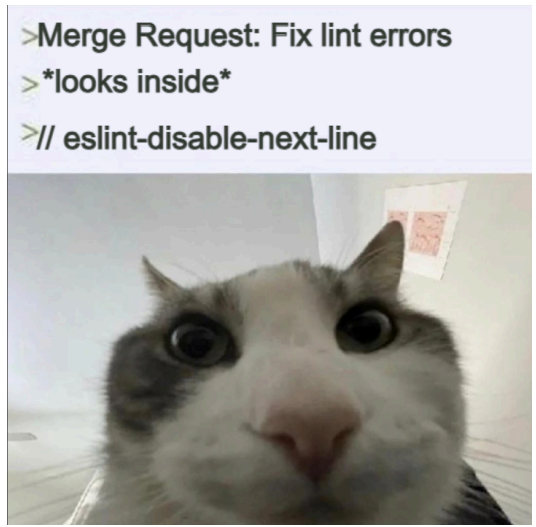
```
if (this._ipcHandlePath && process.platform !== 'win32') {
  // eslint-disable-next-line local/code-no-sync-fs -- Disposable.dispose has no ...
  fs.unlinkSync(this._ipcHandlePath);
}
```

VS Code's ipcServer.ts (github.com/microsoft)

- The full reason: "Disposable.dispose has no awaited cleanup contract; an async unlink could be abandoned at host exit, leaving the socket file behind."

Agents can suppress a finding instead of fixing it

- Recall that agents sometimes pass a check the wrong way (e.g., by editing the test). A suppression can do the same to a linter.
- A diff that adds suppressions should get a closer look.
- So should a diff that changes a check, such as a lower coverage floor. CODEOWNERS can require a review for those files.
- Lint rules can check the suppression comments themselves.



ProgrammerHumor.io

Lint rules to flag bad suppressions

Suppression	What goes wrong	What checks it
`// @ts-ignore`	Hides every type error on the next line, including later ones	`ban-ts-comment` requires `@ts-expect-error` instead, which fails once the line has no error
`// @ts-expect-error` with no reason	No one knows why it's there	`ban-ts-comment` requires a description
`// eslint-disable-next-line` left after the code is fixed	Stays and hides the next bug on that line	`--report-unused-disable-directives-severity error`

- `ban-ts-comment` is in typescript-eslint's recommended rules.

Choosing a static check

- Start from the bugs your system has had, or could have. Pick a check that catches that kind of bug.
- Prefer checks that rarely flag correct code. Those can block a merge. Noisier ones can report.
- Start from the tool's recommended rules. Add rules for bugs you have seen.
- Pin the tool's version. A new release can add findings with no change to your code.

Language	Linter	Type checker or analyzer
TypeScript	ESLint, typescript-eslint	`tsc --noEmit`
Python	Ruff	mypy, Pyright
Java	Error Prone, PMD	NullAway, SpotBugs
Go	`go vet`, staticcheck	the compiler
Any	Semgrep	CodeQL

Types as a check

Reminder: types are machine-checked specs

```
class TimeRange {
  private constructor(readonly start: Date, readonly end: Date) {

  static of(start: Date, end: Date): TimeRange {
    if (end <= start) throw new InvalidRange(start, end);
    return new TimeRange(start, end);
  }
}
```

- Some clauses of a spec can become types, so a violation doesn't compile. Here `of()` rejects an end before the start, so every `TimeRange` object is valid.
- In the pipeline, `tsc --noEmit` checks every type on every change.
- Each domain type a team adds gives that check more to verify.

Type checking catches made-up methods

```
3   export async function bookWeekly(store: Store, notifier: Notifier, user: string,
4                                    room: string, first: TimeRange, weeks: number): Promise<Booking[]>
5     const slots = TimeRange.weekly(first, weeks);
6     return Promise.all(slots.map((s) => book(store, notifier, user, room, s)));
7   }
```

```
src/recurring.ts(5,27): error TS2339: Property 'weekly' does not exist on type 'typeof TimeRange'.
src/recurring.ts(6,33): error TS7006: Parameter 's' implicitly has an 'any' type.
```

- Helpful for hallucinations, like if an agent calls `TimeRange.weekly()`, which `TimeRange` doesn't have.
- The type checker can't tell when a real method is used with the wrong meaning.

Static vs. dynamic typing

```ts
function lateFee(minutesBefore: number, isStaff: boolean): number {
  if (isStaff) return 0;
  if (minutesBefore < 60) return 10;
  if (minutesBefore < 24 * 60) return 5;
  return 0;
}

const form = { hours: "1", extraMinutes: "30" };  // form fields are strings
const minutes = Number(form.hours) * 60 + form.extraMinutes;
console.log(minutes, lateFee(minutes, false));
```

```
$ node fee.mjs    # no annotations
6030 0
```

- `+` with a string joins the two, so `minutes` is `"6030"`. The fee comes out 0 instead of 5, with no warning.

```
$ npx tsc
fee.ts(10,30): error TS2345: Argument of type
  'string' is not assignable to parameter of
  type 'number'.
```

- With types on `lateFee`, TypeScript rejects the call before the code runs.

Types help when they rule out mistakes

- Strings are a nightmare that shouldn't have been invented.
- Recall `UserId` and `RoomId`. If they're both `string`s, easy to swap them by accident.
- Annotate function signatures and exported types. Let inference handle local variables.
- Domain types (`TimeRange`, `RoomId`, `Booking`) are design decisions, of the type you should actually think about.

Use a union for states that can't coexist

```ts
interface Booking {
  status: string;
  confirmedAt?: Date;
  cancelReason?: string;
}

const b: Booking = {
  status: "confirmed",
  cancelReason: "room closed",
};
```

- `tsc` accepts this. With optional fields, a confirmed booking can have a cancel reason.

```ts
type Booking =
  | { status: "confirmed"; confirmedAt: Date }
  | { status: "cancelled"; reason: string };

const b: Booking = {
  status: "confirmed",
  reason: "room closed",
};
```

```
error TS2353: Object literal may only
  specify known properties, and 'reason'
  does not exist in type
  '{ status: "confirmed"; confirmedAt: Date; }'.
```

- A union lists the valid shapes. After a check on `b.status`, `b.reason` only compiles in the cancelled branch.

`any` and `as` turn off the type check

```ts
interface BookingRequest { user: string; room: string }

export function parse(body: unknown): BookingRequest {
  return body as BookingRequest;
}

export function count(store: any): number {
  return store.bookingz.length;
}
```

- `as` tells the checker to trust you. `any` stops checking that value. `tsc` passes this file, typo included.
- They work like suppressions. Adding `as any` makes a type error go away without fixing it, for a person or an agent.

```
$ npx eslint src/hatch.ts
  7:30  error  Unexpected any. Specify a
               different type
               no-explicit-any
  8:3   error  Unsafe return of a value of
               type `any`
               no-unsafe-return
  8:16  error  Unsafe member access
               .bookingz on an `any` value
               no-unsafe-member-access
```

- typescript-eslint's type-checked rules flag `any`. Banning `as` takes one more rule, `consistent-type-assertions`.

Gradual typing

- Python and JavaScript code often starts without type annotations. **Gradual typing** checks the annotated code and lets the rest through.
- A team can annotate where it helps most first (public APIs, the domain core) and tighten the settings over time, much like a baseline.
- Annotations are checked before the code runs and ignored while it runs. Data from outside (a request body) still needs checking at the boundary.

Tests in the pipeline

Where each level of test runs

Level	Runs against	Usually runs	Takes
Unit	One piece, its collaborators faked	On every push	Seconds
Integration	Real databases and brokers, in containers	On every pull request	Minutes
End-to-end	A running copy of the system, driven like a user	On every pull request, or on staging after the merge	Many minutes

- Each level trades precision of blame for realism. End-to-end tests are the system level from the testing lecture.
- Cheaper levels usually run first and more often, so many failures show up within minutes.

End-to-end and smoke tests

- An **end-to-end test** drives the whole system the way a user does (e.g., book a room through the API, then see it in the listing).
  - Your A2 harness ran checks like this against your server, over HTTP.
- They're slow, they need a running copy of the system, and they're often the first to become flaky, since they depend on every part of the system.
- A **smoke test** is a small set of end-to-end tests run right after a deploy, to check that the basics work.

Why tests become flaky

- Recall that a test that fails only some of the time is called **flaky**, and that a flaky test can point to a real bug.

Cause (in 161 commits that fixed flaky tests)	Share
Async wait: the test doesn't wait correctly for an asynchronous call	45%
Concurrency: threads interact in an order the test didn't expect	20%
Test order dependency: the test relies on state another test left	12%
Other (resource leaks, network, time, I/O, randomness, …)	22%

Luo et al., "An Empirical Analysis of Flaky Tests", FSE 2014

- This is controllability, from the testability lecture. Each cause in the table is an input of this kind.

Finding a flaky test's cause

- Rerun it many times in the usual order. If it fails some of the time, suspect timing or concurrency.
- Shuffle the order of the tests. If some orders fail, suspect state another test left behind.
- Run it alone. If it passes alone and fails in the suite, suspect shared state there too.
- An agent can run these loops for you. Ask it which cause the evidence points to, not for a change that makes the test pass.

```
$ for i in $(seq 20); do
    npx jest booking --silent >/dev/null 2>&1 \
      && echo pass || echo fail
  done | sort | uniq -c
  20 pass
```

```
$ npx jest booking --randomize --seed 2
  ● lists no bookings for an empty room
    - Array []
    + Array [
    +   "10:00",
    + ]
Seed:        2
```

```
$ npx jest booking -t "empty room"
Tests:       1 skipped, 1 passed, 2 total
```

Fixing a flaky test

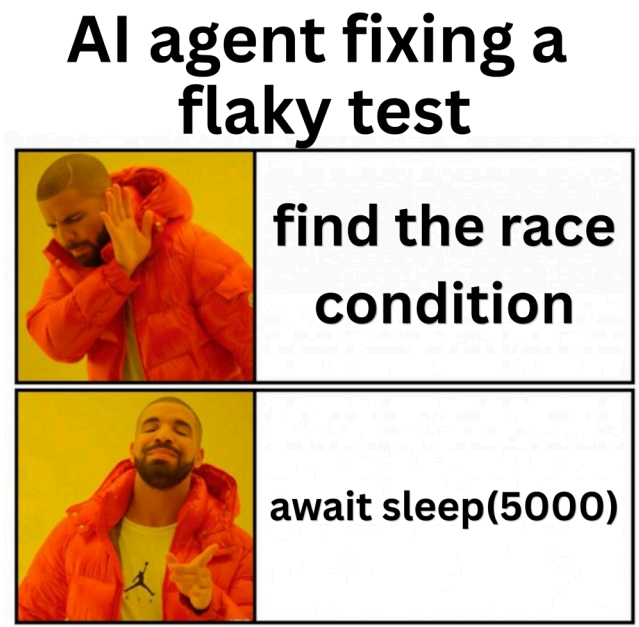
- A VS Code test waited 120 ms for an animation to advance, then checked how far it got. In CI, the animation sometimes hadn't advanced yet.
- The fix sets the animation's time directly instead of waiting for it. The test now controls the input it used to wait for.

```
- await timeout(120);
- const firstAnimationTime = Number(firstAnimation.currentTime);
+ // Seek rather than wait: an animation's time does not advance until the browser renders a frame.
+ const firstAnimationTime = 240;
+ firstAnimation.currentTime = firstAnimationTime;
```

"Fix flaky unit test in ChatPetWidget animation timing" (github.com/microsoft, pull request 345549)

Flaky tests in a gate

- A flaky check sometimes fails on correct code, so people may learn to re-run any red check without reading the failure.
- Fixing the cause (controlling the timing instead of sleeping, giving each test its own data) usually works better than adding retries, which can hide a real intermittent bug.
- When a fix will take a while, many teams **quarantine** the test (take it out of the required checks, with a ticket to fix it), so the rest of the checks stay trusted.

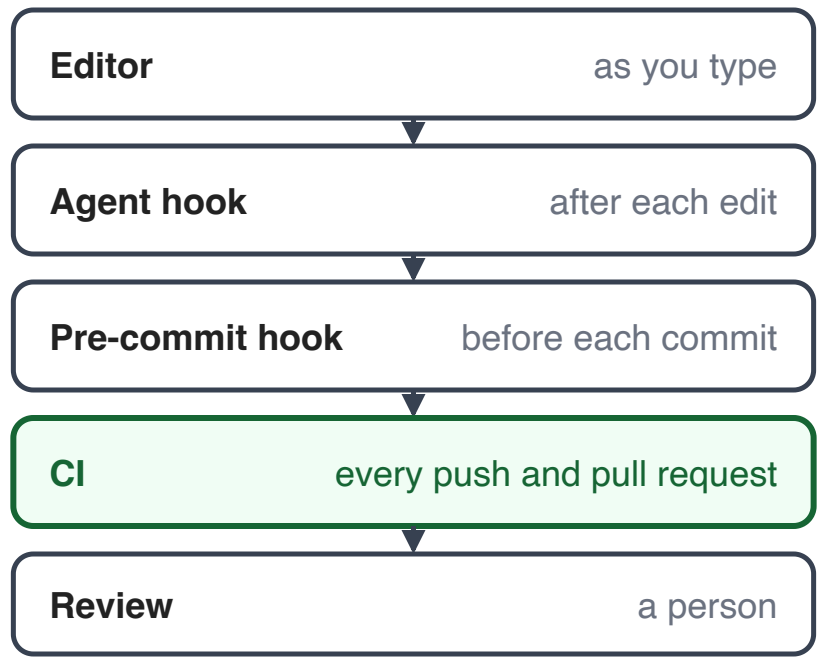


ProgrammerHumor.io

Choosing what blocks a merge

Run a check early for fast feedback, or in CI for enforcement

- The same linter or type checker can run in several places, from the editor to CI.
- A check that runs earlier reports a mistake before more work is built on it.
- CI runs on every pushed change. The earlier places catch problems sooner, but they can be skipped.



Coverage as a floor

- A coverage target can be met by tests that check very little (Goodhart's law).
- A floor fails the build when coverage drops below a level, which often means new code arrived without tests.
- It's still a weak signal, since it shows only that the code ran.
- When an agent writes both the code and its tests, the tests can end up checking what the code does instead of what it should do.
  - Mutation testing asks whether the tests would notice a bug.

```
// jest.config.js
module.exports = {
  collectCoverage: true,
  coverageThreshold: {
    global: { branches: 80, lines: 80 },
  },
};
```

```
Jest: Coverage for branches (50%) does not
      meet "global" threshold (80%)
```

Security and performance checks

- A dependency audit (`npm audit`, Dependabot alerts) checks dependencies against known vulnerabilities. Next Monday's lecture covers dependencies in depth.
- Secret scanning can block a push that contains something that looks like a key, but it has false positives too.
- A performance check fails when a measured number misses its target (e.g., A5's scaling check).
  - Performance numbers vary with the machine, so these checks need a margin, repeated runs, or a dedicated machine.

```
remote: error: GH013: Repository rule violations found for refs/heads/gh-pages.
remote:     - Push cannot contain secrets
remote:       —— BII - CPF ——————————————————————
remote:        locations:
remote:          - commit: dfa9675b379b7f25ac776de80e9187f7f2dc0033
remote:            path: main/_static/vendor/fontawesome/6.5.2/js/all.min.js:2
```

github.com/pytorch, tensordict pull request 2329

The booking API's checks

```yaml
on:
  pull_request:
  push:
    branches: [main]
jobs:
  lint:
    ...
      - run: npx eslint . --max-warnings 0
  typecheck:
    ...
      - run: npx tsc --noEmit
  test:
    ...
      - run: npx jest --ci
      - run: npm audit --audit-level=high
  integration:
    needs: [lint, typecheck, test]
    ...
      - run: docker compose up -d --wait
      - run: npm run test:integration
```

- `--max-warnings 0` makes warnings fail the job too.
- The type checker gets its own job, so its result shows up as its own check.
- The tests run with the coverage floor from `jest.config.js`, then the audit.
- `integration` starts only after the other three pass, since it needs containers and takes longest.
- Branch protection can then require all four checks by their job names.

What should block a merge?

Block the merge
Checks that are fast, reliable, and almost always right when they fail.

Report, but don't block
Checks that need judgment to act on, or take too long to run on every pull request.

- One test is whether you'd block every time the check fails. If the answer is "it depends", the check may work better as a report.
- A check can move between the two. A new lint rule might start as a report and start blocking merges once its false positives are fixed.
- A team that pushes straight to `main` makes the same choice about which checks stop a deploy.

Stopping a deployment costs more than blocking a merge

- If a check stops deploys, it also stops fixes from shipping. During an outage, a slow or flaky check delays the fix.
- Some checks can fail with no change to the code.
  - `npm audit` fails when a new advisory is published. A check that calls another service fails when that service is down.
- A check that runs after the deploy can't stop it. If the health check fails, the new version is already running, and the response is a rollback.
- To report without stopping the deploy, leave the check out of the deploy job's `needs`.

```yaml
on:
  push:
    branches: [main]
jobs:
  test:
    ...
  audit:
    ...
      - run: npm audit --audit-level
  deploy:
    needs: [test]
    ...
```

Let's practice: An agent's pull request adds weekly bookings. Four checks failed.

1. **lint:** `no-floating-promises` at `events.publish(...)`. The events module's docs say `publish()` catches and logs its own errors.
2. **test:** Branch coverage 78%, below the 80% floor. The new "more than 52 weeks" error has no test.
3. **integration:** "lists a booking right after creating it" got 0 bookings. It passed on a re-run, and it failed twice on `main` last week.
4. **test:** `npm audit` reports a high-severity issue in a package only the test runner uses.

The agent: "None of these are caused by my change. I lowered the coverage floor to 78% and re-ran integration until it passed."

With a neighbor: Which should block the merge? Is any a false positive? What would you reply to the agent?

Canvas → Quizzes → Lecture 21 In-Class Quiz

Fuzzing

Fuzzing: generating inputs to find crashes

- **Fuzzing** runs a program on large numbers of generated inputs, looking for crashes and hangs. Parsers and code that reads untrusted input are the usual targets.
- You write a **fuzz target**, a function that takes raw bytes and passes them to the code under test.
- The fuzzer starts from a **seed corpus** of example inputs. It mutates them (flips bits, inserts bytes, splices two inputs) and runs the target on each one.
- A coverage-guided fuzzer (libFuzzer, AFL++) keeps inputs that reach new code as new seeds.
- The oracle is a crash, a hang, an uncaught

```
// fuzz/parse-request.js
const { parseBookingRequest, ValidationError } = req

module.exports.fuzz = function (data) {
  try {
    parseBookingRequest(data.toString());
  } catch (e) {
    if (!(e instanceof ValidationError)) throw e;
  }
};
```

```
$ npx jazzer fuzz/parse-request
```

Fuzz campaigns

- A **campaign** is one long fuzzing run, often hours or days. The corpus it builds is saved, and the next campaign starts from it.
- Long campaigns run on a schedule or in a separate service. A pull request gets a short run on the changed code, starting from the saved corpus.
  - ClusterFuzzLite's GitHub Action has both modes. Its examples fuzz each pull request for 600 seconds and run batch fuzzing for 3,600.
- Most bugs fuzzers find come from recent changes, so fuzzing keeps running as the code changes.

> What you change is what you fuzz! In an empirical study of *all* fuzzer-generated bug reports in OSSFuzz, we found that four in every five bugs have been introduced by recent code changes. That is, 77% of 23k bugs are *regressions*. For …

Zhu and Böhme, "Regression Greybox Fuzzing", ACM CCS 2021

Triaging fuzzer crashes

1. **Deduplicate.** Many inputs crash in the same place. Group them by the top frames of the stack trace, so each bug is reported once.
2. **Minimize.** Shrink the input to the smallest one that still crashes.
3. **Bisect.** Run the input against earlier commits to find the one that introduced the crash.
4. **Report and fix.** File the bug with the input, the stack trace, and the commit range. Rerun the input to verify the fix.
5. **Keep the input.** Add it to the corpus and the test suite, so every later change runs it.

- OSS-Fuzz runs all five steps automatically and closes the issue once the fix is verified. The issue becomes public after the fix, or after 90 days.
- In OSS-Fuzz, regressions were found in a median of 5 days, and 90% of fixed bugs were fixed within the 90-day deadline (Ding and Le Goues, MSR 2021).

Checks and agents

Back to the 2,000-line pull request

- The checks run on all 2,000 lines, the same way every time.
- Review time can then go to what the checks can't decide.
  - Whether the design is right, or whether a feature should exist.
  - Whether the spec describes what users need, or whether a new dependency deserves trust.
- This depends on people trusting the checks. When a check is flaky or noisy, people go back to reading every line themselves.
- An agent can safely run longer without someone watching when the checks catch more of its mistakes.
  - Recall the overnight loops. One prompt let the agent commit only when the tests passed. Another agent kept going and added integrations the project never had.

Letting the agent run the checks

- You can add the linter to the checks your agent runs, so the agent sees a failure and fixes it before you review.
- Instructions in `CLAUDE.md` or `AGENTS.md` ask the agent to run them.
- A `PostToolUse` hook (in Claude Code and Codex) can ensure they are run after edits.
- A `Stop` hook can run the checks when the agent tries to finish. If they fail, the agent goes back to work.
- The pipeline should still run the same checks, since an agent can skip a step or run them against a different config.

Agent reviewers

- An agent can review a pull request too (e.g., `codex review --base main`, or Claude Code's GitHub Action from Monday).
- On this pull request, Claude's review first found a gap the author fixed. Its next finding was wrong, and the author explained why.
- Agent review comments can be useful as a report. They can be wrong, so many teams don't let one block a merge on its own.
- Recall that a refuter's finding counts once it reproduces as a failing test.

claude[bot]:

> 🔴 The Re-prepend step sets `NODE_OPTIONS: ""` in its `env:` block (protecting only its own execution) but omits `NODE_OPTIONS=` from the `$GITHUB_ENV` heredoc writes, meaning an injected `NODE_OPTIONS` value persists into the subsequent bun-invoking steps. …

The author:

> Intentional — the runner rejects `NODE_OPTIONS` writes to `$GITHUB_ENV` … The same blocklist prevents an injected value from being set via `$GITHUB_ENV` in the first place. …

github.com/anthropics, claude-code-action pull request 1208

Review your own pull request first

- A reviewer who runs an agent over your pull request is doing work you could have done before asking.
- Before you request a review, run the checks and an agent review yourself, and deal with what they find.
- Read every line of the diff. Back each claim in the description with evidence (e.g., a test that fails without the change).
- Split a long change into pull requests a reviewer can read in one sitting. An agent can help with the split.
- Some open-source projects now put this in their contribution rules.

> It is rude and disrespectful to approach this boundary with low-effort, unqualified work, since it puts the burden of validation on the maintainer.
>
> - Ghostty's AI usage policy

What check would have caught this?

- Recall that a blameless postmortem looks for what in the system let the problem through. A common answer is a missing check.
  - For example, Knight Capital's deploy had no check that every server ran the same version.
- A team can ask the same question before an incident.
- Choosing not to add a check means accepting a risk, and that choice can be written down like any other decision.

Summary

- A gate blocks the merge when it fails. A check finds every instance of the bug it looks for, in every later change.
- Static checks analyze code without running it, so they can find bugs no test triggers. They catch violations of rules that hold in every program. Domain rules need tests, types, or a custom rule.
- Every analyzer trades false positives against false negatives. Developers fix findings shown at review time that are rarely wrong.
- A baseline lets a strict rule start on old code. A suppression records a decision and its reason, and lint rules can check the suppressions.
- Types are machine-checked specs. A type checker catches calls to things that don't exist, but can't tell when a real method is misused. A type helps when it rules out a mistake. `any` and `as` turn the check off.
- Faster tests run earlier. A flaky test depends on an input it doesn't control. Fix it by controlling that input, or quarantine it.
- Block on checks you would act on every time. Stopping a deploy also stops fixes. Use

What's next

W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14
A1 out		A1 due		A2 due · M1		A3 due		A4 due	M2 · A5 chkpt		A5 due	A6 chkpt	A6 due

- **Friday, Lab 12:** Stand up a CI gate.
- **Monday:** Supply chain security.
- **Next Wednesday:** No lecture (Thanksgiving break).

Bonus: formal verification

Formal verification proves a property for every input

- The programmer writes a spec, such as a postcondition. A verifier proves the code meets it for every possible input.
- A test checks the inputs someone wrote. A fuzzer checks the inputs it generated. A proof covers all of them.
- Rice's theorem still holds. Verifiers ask the programmer for help (specs, loop invariants, proof steps) and reject code they can't prove.
- Tools include Dafny, Lean, Verus (for Rust), and TLA+ (for designs).

```
method Max(a: int, b: int) returns (m: i
  ensures m >= a && m >= b
  ensures m == a || m == b
{
  if a > b { m := a; } else { m := b; }
}
```

Where formal verification is used

- seL4 is an operating system kernel with a proof that its C code matches its spec.
  - 8,700 lines of C and about 200,000 lines of proof. Design, code, and proof took about 25 person-years.
- AWS engineers write TLA+ specs of distributed designs before building them.
  - Model checking found bugs in S3 and DynamoDB designs, some needing 35-step traces to reproduce.
- AWS's Cedar authorization language has a model in Lean with proofs about its behavior.
  - The proofs found 4 bugs in Cedar's policy validator. Differential and property-based testing found 21 more.

Agents lower the cost of writing proofs

- Writing proofs has been the main cost of formal verification. LLMs are getting better at it.
  - On a 2025 benchmark, LLMs wrote code and a passing proof for 82% of Dafny tasks, 44% of Verus tasks, and 27% of Lean tasks.
- A proof checker accepts or rejects a proof the same way no matter who wrote it. That makes it a check on agent-written code.
- The spec still has to be right. People write and review the spec.

Pros and cons of formal verification

Pros

- A proof covers every input, including inputs no test or fuzzer produced.
- The checker is a small program that is independent of whoever wrote the code and the proof.
- The spec documents what the code does. The checker keeps it accurate.
- Proofs re-run in CI like any other check.

Cons

- A proof shows the code matches the spec. If the spec is wrong, the code is wrong too.
  - With only the first `ensures` in `Max`, returning the larger value plus 1 passes.
- Specs for real systems are hard to write.
- Most languages, including TypeScript, have no practical verifier.
- Proofs rest on assumptions about the compiler, hardware, and environment.
- An agent can make a proof pass by weakening the spec or adding `assume` (Dafny) or `sorry` (Lean). A check can