



Code Quality, Documentation, and Technical Debt

Lecture 1 • Reviewing What You Didn't Write

Zachary G. Ives & Harry Smith
University of Pennsylvania
Fall 2026

Week 4 at the Startup: Your First Real Review

You remember the deep end from Week 2. Here is Week 4.

A PR lands in your queue. **1,240 lines**. The author is a teammate who delegated most of it to an agent, and the description says "implements the spec, tests pass."

CI is green. You have forty minutes before standup.

You cannot read 1,240 lines in forty minutes. You also shouldn't approve something you did not read. Today is about what a professional actually does in that gap.

Curing the Snake Bites of Software Development



Antivenom production, Accra, Ghana. FCDO / Russell Watkins, Wikimedia Commons, CC BY 4.0

Producing a bunch of code quickly is like jumping around in the woods with no shoes! Fun(?), but sometimes you get bitten by a snake.

- **Accelerated Technical Debt.** *Antidote: architecture-first thinking, CLAUDE.md as a convention document.* Today we cover what debt looks like and how to pay it down safely.
- **Inadequate Review, Testing, Security.** *Antidote: specification-driven testing, CI as a non-negotiable gate.* Today we start actually doing it, and you continue in Project 1.

3

Today's Big Question

How do you know when code is correct enough to ship —
and how do you review code you didn't write,
at the speed agents produce it?

Recall: large language models are trained to *sound confident*. **Code review** is where that confidence finally meets *scrutiny*. **Testing** is how you make the scrutiny *repeatable* instead of one person's opinion.

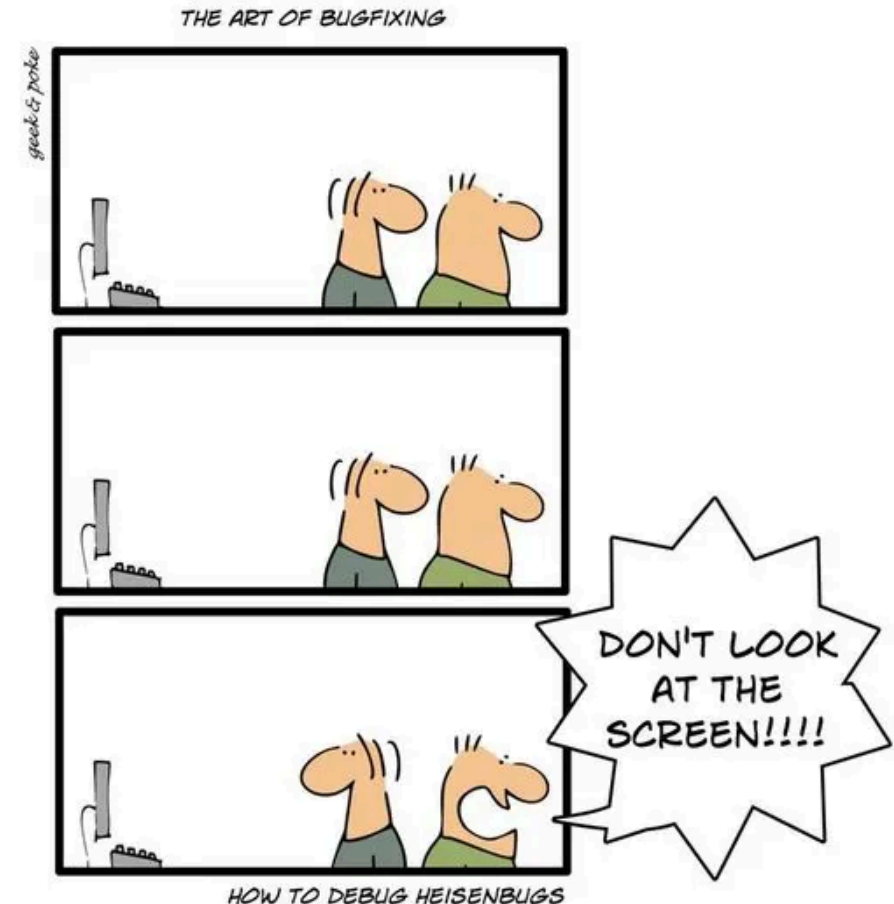
Beyond Unit Testing

Unit tests (like JUnit) allow you to validate a module's output, given specific inputs.

When you have complex software, simply doing unit tests is helpful but inadequate.

Unexpected interactions, ambiguous interfaces, concurrency issues, etc. can cause problems!

"The Heisenbug": only exists when you aren't trying to observe it. Coined by IBM's Bruce Lindsay, popularized by Jim Gray's 1985 paper, *"Why Do Computers Stop and What Can Be Done About It?"*



Onto Project 1!

Project 1 will have you *verify* several features of the Spacebar specification.

You get two artifacts, at different levels of abstraction, formality, and detail.

- a **written specification** of Discord's permission system and channel operations,
- Spacebar's code as a **reference implementation** that may not conform to it.

Your job: write a **test suite comprehensive enough to tell, for *any* implementation, whether it conforms.**

The reference implementation you've been running since Lab 1 has **conformance gaps relative to the spec**. A correct test suite should uncover them!

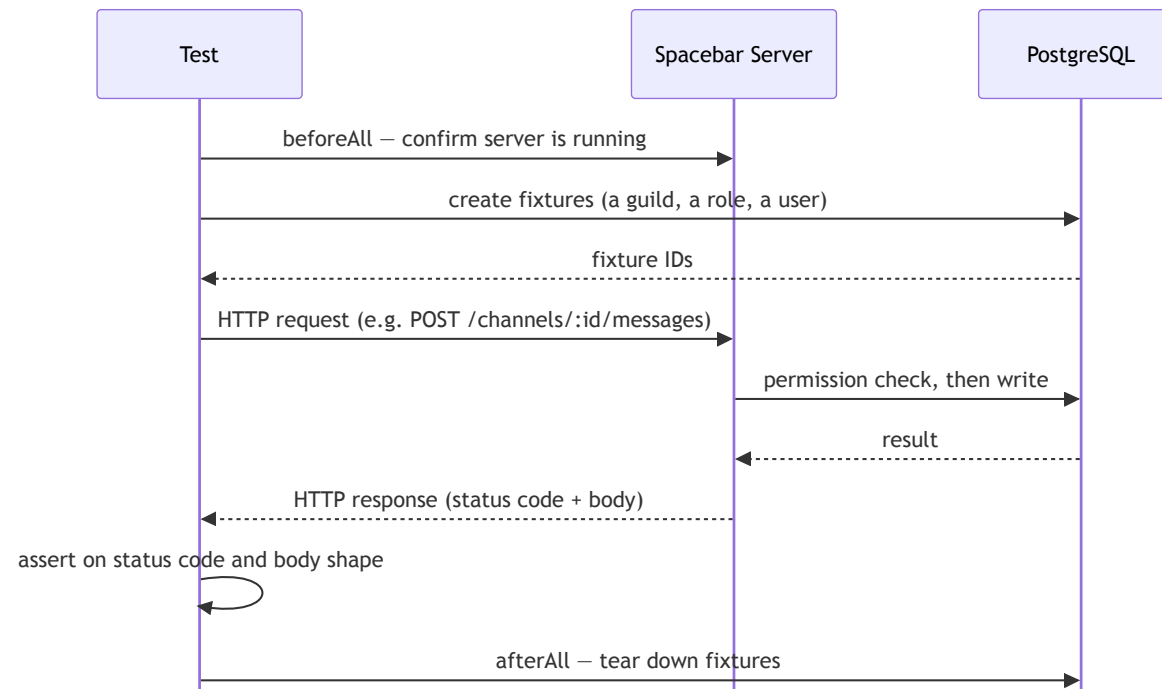
From Unit Testing to Integration Testing

An **integration test** uses the whole system infrastructure — not based on mocks, stubs, or reaching into internal state.

- You're testing *observable behavior* — what a client sees — not implementation details.
- A test that imports `Permissions.channelPermission()` directly and checks its return value is testing an implementation. It will fail against a *correct* implementation that happens to resolve permissions differently internally.
- Both `node:test` (built into Node) and Jest work fine for this. Pick one.

Anatomy of an Integration Test

What should an integration test look like?



Every integration test follows this shape: set up state, act, assert, clean up.

Let's explore the test that comes with Spacebar's current implementation.

- Is it a unit test?
- Is it an integration test?
- Is it enough?

Let's explore the test that comes with Spacebar's current implementation.

- Is it a unit test? **No**, it doesn't call an individual function to evaluate its output given an input.
- Is it an integration test? **Sure**. It sets up the server, acts, asserts, and cleans up.
- Is it enough? **One test never is**. But also: it doesn't really cover anything about how a user would use Spacebar—not even a specific API call!

Two More Properties

Comparing IDs is comparing timestamps. This is why Spacebar's message search takes `min_id` and `max_id` rather than dates — a date range *is* an ID range. Project 2's search option depends on this.

10

Here Be Dragons: Permission Resolution

Let's expand the frontier of your map of Spacebar by talking about specifications and testing for **permission resolution**

Discuss: what factors contribute to determine what a user can do in a given channel in Spacebar? (Or Discord, if you know more about that?)



Specifying the Permission Algorithm

Four steps, and the current implementation lives in `Permissions.finalPermission()`.

1. **Owner short-circuit.** If the member owns the guild, return every bit. Stop.
2. **Base permission.** Bitwise-OR the `permissions` field of every role the member holds, including `@everyone`.
3. **Apply channel overwrites**, in order, folding each one into the running value.
4. **Bypasses and masks** — `ADMINISTRATOR`, timeout, quarantine.

Applying channel overwrites (in `channelPermission()`):

```
overwrites.reduce(  
    (permission, overwrite) =>  
        (permission & ~BigInt(overwrite.deny)) | BigInt(overwrite.allow),  
    base,  
);
```

12

Reading That Line

```
(permission & ~deny) | allow
```

`~deny` inverts the deny mask, so `& ~deny` **clears** every denied bit and leaves the rest alone. Then `| allow` **sets** every allowed bit.

Consider:

- What happens if a specific permission is both allowed and denied?
- What happens to bits that are not part of an overwrite?

Reading That Line

```
(permission & ~deny) | allow
```

`~deny` inverts the deny mask, so `& ~deny` **clears** every denied bit and leaves the rest alone. Then `| allow` **sets** every allowed bit.

- Within a single overwrite, **deny is applied before allow**. If one overwrite both denies and allows the same bit, allow wins.
- A bit mentioned in neither mask passes through untouched. That is what makes overwrites *composable* — each one only speaks to the bits it names.

Predict: What Does This Return?

A member holds role `R`. The guild grants `R` the `VIEW_CHANNEL` bit.

On this channel there are two overwrites:

- a **role** overwrite on `R` that **denies** `VIEW_CHANNEL`
- a **member** overwrite on this member that **allows** `VIEW_CHANNEL`

Can the member see the channel? How do you know? Is it directly specified, or a consequence of something else?

Predict: The Answer Depends on Order

`reduce` applies overwrites in whatever order the list arrives in.

ORDER APPLIED	RESULT
role deny, then member allow	visible — allow wins
member allow, then role deny	hidden — deny wins

The project specification settles it: **§3.5 requires all role overwrites before all member overwrites**, so the member can see the channel. A member-level overwrite is how you make an exception for one person, and an exception that a role could overrule would be useless.

Now Let's Take a Big Sip Of Coffee and Check That Spec...

```
finalPermission():
```

```
if (channel?.overwrites) {  
    const overwrites = channel.overwrites.filter((x) => {  
        if (x.type === ChannelPermissionOverwriteType.role &&  
            user.roles.includes(x.id)) return true;  
        if (x.type === ChannelPermissionOverwriteType.member &&  
            x.id === user.id) return true;  
        return false;  
    });  
    permission = Permissions.channelPermission(overwrites, permission);  
}
```

Nothing in here to enforce that the `reduce` is applied in any particular order...

Maybe it's fine, but it'd definitely be good to know for sure!

17

How To Start Testing

Your suite talks to a live server over HTTP, so the server has to already be running when the tests start. That means two terminals.

```
npm run build && npm start      # terminal 1 – leave this running
npm run test:p1                 # terminal 2 – node --test p1/tests/
```

When every test fails at once, suspect the server before you suspect the tests: `curl http://localhost:3001/healthz` answers that question in a second.

Add `test:p1` as its own script. Do **not** reuse `npm test` — that name already belongs to Spacebar's startup smoke test, the one we looked at earlier.

One Address, Read from the Environment

The autograder runs your suite against implementations you have never seen, at an address it picks.

```
// p1/fixtures.ts  
export const BASE_URL =  
  process.env.SPACEBAR_URL ?? "http://localhost:3001/api/v9";
```

Define it here so that it works no matter what environment you're working in!

Fixtures Are Helpers for the Setup

Every scenario needs the same kind of setup: a user, a guild, a role, a channel, an overwrite. Each fixture is one HTTP request that checks its own response.

```
async function createGuild(owner: { token: string }, name = "Test Guild") {
  const res = await fetch(`${BASE_URL}/guilds`, {
    method: "POST",
    headers: { "Content-Type": "application/json", Authorization: owner.token },
    body: JSON.stringify({ name }),
  });
  if (!res.ok) throw new Error(`createGuild failed: ${res.status}`);
  return res.json(); // { id, owner_id, channels: [...], roles: [...] }
}
```

That single call hands back the guild's `id` along with the roles and channels created with it.

20

Fixtures with Unique Attributes

Tests write rows that aren't deleted by default, so be careful that your setup creates unique users/roles/channels:

```
export async function registerUser(baseUrl: string): Promise<TestUser> {
  const username = unique("user");
  const res = await fetch(`${baseUrl}/auth/register`, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({
      username,
      password: `Testing123!${username}`,
      email: `${username}@example.invalid`,
      consent: true,
      date_of_birth: "1990-01-01",
    }),
  });
```

21

Fixtures Fail Quietly, So Check Them Early

A fixture that returns the wrong ID does not fail in the fixture, but it does make for a confusing test failure later down the line.

- Exercise each helper with `curl` once before you build tests on it
- have every fixture throw on a non-OK response rather than returning a half-formed object.

OK, with all that...

Remember:

- Member permissions should override any other role permissions
- Currently, overrides are simply processed in the order received independent of the kind of override they represent.

Is this a bug, or does some other element of the system save us here?

```
test("a member allow overrides a role deny written after it", TODO_35, async () => {  
  const channel = await createChannel(guild);  
  const { user, roleId } = await addMember(guild);  
  
  await setOverwrite(channel, { type: "member", id: user.id,  
                                allow: ["VIEW_CHANNEL"] });  
  await setOverwrite(channel, { type: "role", id: roleId,  
                                deny: ["VIEW_CHANNEL"] });  
  
  assert.strictEqual(await canView(channel, user), true,  
    "a role overwrite created later must not override a member overwrite"  
  );  
});
```

Another Potential Bug

```
static channelPermission(overwrites: ChannelPermissionOverwrite[], init?: bigint){  
    // TODO: do not deny any permissions if admin  
    ...  
}
```

Administrators of a channel should always be able to do everything in that channel, regardless of any particular `member` or `channel` overrides. The TODO suggests that might not be happening!

```
test("ADMINISTRATOR bypasses a channel-level deny overwrite [spec §3.7]", async () => {
  const admin = await registerUserWithRole(guild, "ADMINISTRATOR");
  await setOverwrite(channel, { type: "member", id: admin.id,
                                deny: ["VIEW_CHANNEL"] });

  const res = await fetch(`${BASE_URL}/channels/${channel.id}`, {
    headers: { Authorization: admin.token },
  });

  assert.strictEqual(res.status, 200,
    "ADMINISTRATOR must bypass channel-level deny overwrites");
});
```

Aside: The IDs Your Fixtures Hand Back

Every `createGuild()` and `createChannel()` returns an `id`. Those IDs are not random, and not a database counter.

Spacebar uses **snowflake IDs** — a 64-bit integer with structure, from `src/util/util/Snowflake.ts`:

```
64                22    17    12            0
000000111011000111100001101001000101000000  00001  00000  00000000000000
      milliseconds since 2015-01-01      worker  pid    increment
```

42 bits of timestamp, 5 bits of worker, 5 of process, and a 12-bit counter for IDs minted within the same millisecond.

Why That Layout Is the Way It Is

Three properties fall out of putting the clock in the high bits. The first one takes a short detour through the database, because "give me these in order" is never free.

Asking a Database for Things in Order

A table is a bag of rows. Nothing about it is sorted — each row sits wherever it landed as the table was written to, updated, and rewritten over months.

messages — in whatever order the rows landed

#	author	created_at
1	ayesha	09:14
2	kofi	08:02
3	mira	09:41
4	dev	08:37
5	sam	09:05
6	lin	08:51

ORDER BY created_at DESC LIMIT 3

read all six,
then sort

09:41 mira
09:14 ayesha
09:05 sam

The work grows with the size of the table, not the size of the answer.

So the only way to answer the query is to read every row and sort what comes back.

An Index Is a Sorted Copy of One Column

An **index** stores the values of a single column in sorted order, each one pointing back at the row it came from.

index on created_at — sorted

value	row
08:02	2
08:37	4
08:51	6
09:05	5
09:14	1
09:41	3

the rows, still in storage order

#	author	created_at
1	ayesha	09:14
2	kofi	08:02
3	mira	09:41
4	dev	08:37
5	sam	09:05
6	lin	08:51

Read the index in order and the rows come out sorted; the query stops after three.

Every insert, update, and delete now has two structures to keep consistent.

Ordering is bought with space and with write cost.

The Key You Already Have

Sorting by ID sorts by creation time. The timestamp sits in the high bits, so ordering messages by primary key *is* ordering them by when they were sent — and the primary key is indexed already! No extra sort.

Developing Integration Tests from Behavior Clauses

Keep each test focused on **one behavior clause** from the spec. When a test fails, the failure message should say *which clause* was violated

- A test suite that only says "something's wrong" makes debugging someone else's job.
- A test suite that says "spec §3.7 — admin bypass — violated" makes debugging a five-minute job, for a human *or* an agent asked to fix it.

Later we'll talk about some additional ideas like "test doubles," *property-based* testing, and *mutation testing* in depth. For now, we'll focus on straightforward behavior tests, with one spec clause each, against the live API.

Developing Code Review Skills

Code Review: Active Reading, Not Passive Summarizing

You've done this since Lab 1 without naming it: using an agent to extract abstractions, find where a core algorithm actually lives, and generate a functional roadmap of a codebase you didn't write.

"What module handles permission resolution? Trace it to the specific function, then show me every call site."

Assessing Test Comprehensiveness

Once you have a test suite — yours or someone else's — ask an agent to map it against the spec:

"Compare the existing tests for `src/util/util/Permissions.ts` against this spec: [paste it]. What's covered? What's missing? For each gap, what class of bug could hide there?"

Treat the gap list as **hypotheses**, not facts. Agents miss coverage that exists (they didn't find the right test file) and invent gaps that don't matter. You verify before you trust the list — same discipline as everything else this semester.

The Code Reviewer's Dilemma

Recall that large language models optimize for confident, fluent completions. **Model confidence and correctness are uncorrelated.** Code Review is the time to check this!

A CI pipeline that only checks "does it compile" catches almost none of this. Compilation was never the hard part.

Multi-Agent Audits: You've Already Seen This

Last week's `roborev` demo — reviewing every commit in the background, feeding findings back to the coding agent, running the write → review → fix loop hands-off — is exactly this idea in production.

The generator and the reviewer were **different processes** in that demo, not the same agent grading its own homework. That separation is not incidental — it's the entire reason the review caught anything.

A second opinion from an independently-configured agent is cheap. Use it before a human's attention is the only check.

Risk-Based Sampling

You cannot review every line an agent produces at the volume agents produce it. You have to choose where to spend attention.

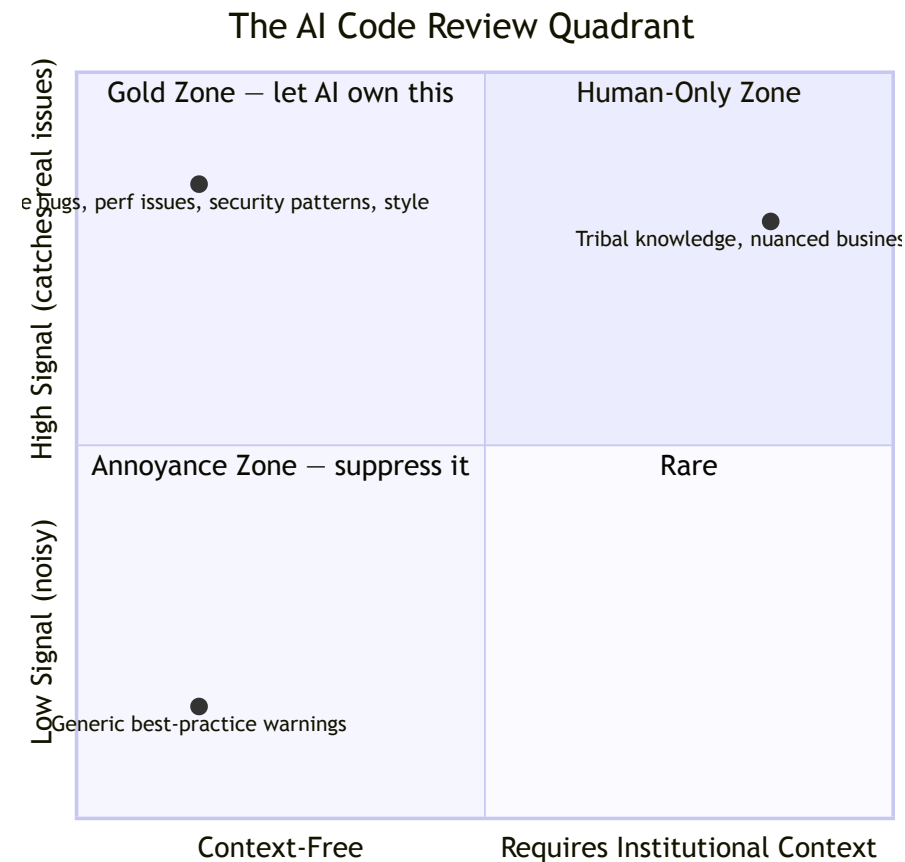
Prioritize, in roughly this order:

1. **Auth/authz logic** — the cost of a mistake here is a security incident, not a bug ticket
2. **Core data structures** — a subtle error here propagates everywhere that structure is used
3. **Code with no test coverage** — nothing else is going to catch a mistake here

This is a triage strategy, not permission to skip the rest. It's what you do when you can't review everything — not an excuse for reviewing nothing else at all.

The AI Review Quadrant

Not all review findings are worth the same attention. Sort them:



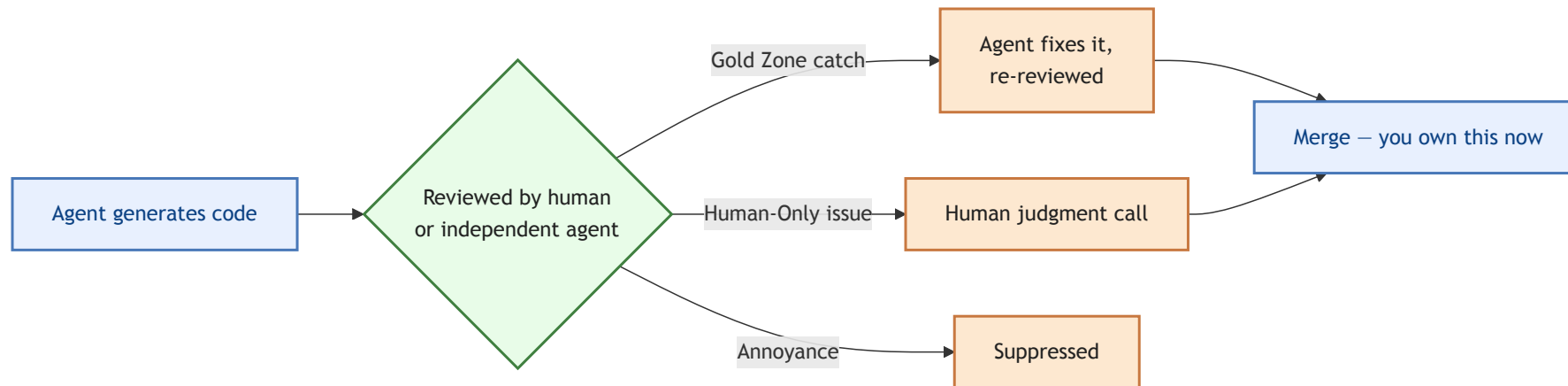
The Three Zones, Named

- **Gold Zone** — simple bugs, performance issues, security-pattern violations, style consistency. AI tools handle this well: let 'em.
- **Human-Only Zone** — institutional history, nuanced business logic no diff can reconstruct. An agent can't be expected to remember about some arcane bug from 8 years ago that remains undocumented.
- **Annoyance Zone** — abstract "best practice" warnings with no situational awareness. These generate noise and, worse, erode trust in the tool because they're just annoying to look at!

What Code Review Is Actually For

Finding bugs is the visible job. The deeper one: **mental alignment** — keeping the team's shared understanding of the system current.

If you can't explain what a merged PR does and why, you don't own that code — regardless of whether it currently passes its tests. Ownership requires the understanding, not just the green checkmark.



Recap: Confidence Isn't Correctness — Review and Test Both Check It

- **Integration tests check observable behavior, not implementation** — a test suite comprehensive enough to judge *any* implementation against the spec, which is exactly Project 1's task.
- **A good failure message names the spec clause violated**, not just the status code — that's what turns debugging into a five-minute job instead of a scavenger hunt.
- **Risk-based sampling** are how you manage at the volume agents produce: structural diffs surface what actually changed; auth/authz, core data structures, and untested code get attention first.
- **The AI Review Quadrant**: automate the Gold Zone, keep humans on the Human-Only Zone, and don't let Annoyance Zone noise erode trust in the tool.

Questions?

`zives@engineering.upenn.edu` · `sharry@engineering.upenn.edu`