



# Prototyping and User Stories

Finding Out What the Spec Should Say

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

# Where We Are

---

**Lab 7 is assigned today** — two `git worktree` prototypes of the same feature, a four-dimension comparison, and an ADR recording which one you kept.

**Project 1 is due Thursday, October 15.** The **midterm is Tuesday, October 13** and covers Weeks 1–7 — including this week.

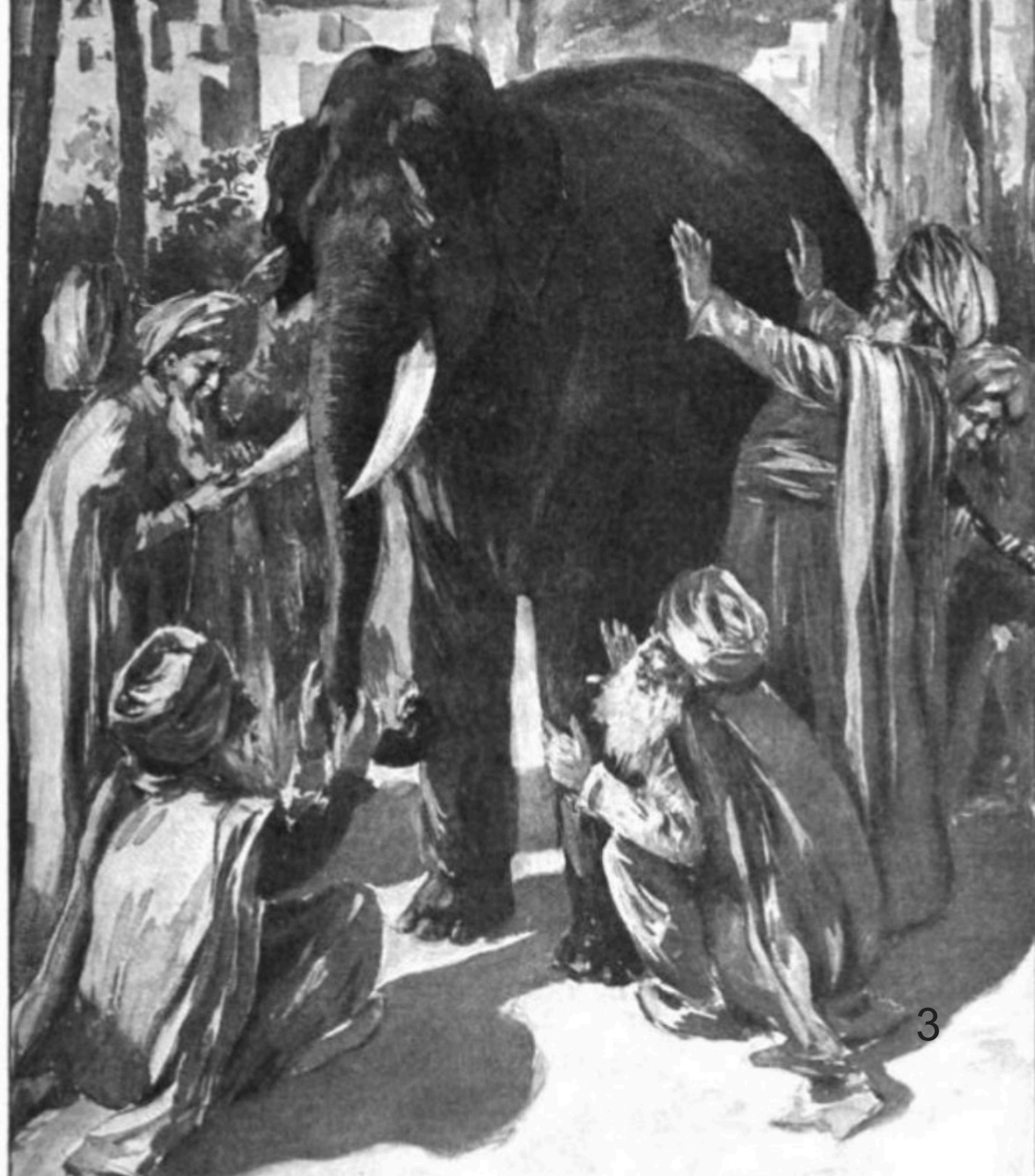
# Specification's Lingerin Problem

---

We can create a spec that is perfectly unambiguous, fully machine-verifiable, green in CI on every push. And yet...

A spec can be all of those things **and still describe the wrong system.**

At right, a strained analogy...



# Today's Big Question

---

## What if the specification itself is wrong?

A precise spec makes disagreement about *behavior* impossible, but does nothing about disagreement over *whether that behavior is what anyone needed*.

- "What anyone needs" is a contentious issue!
- There are cheap times to figure this out (early) and expensive times (late).

*"Spec what you understand now, validate it early, and expect to revise."*

# Game Dev is a Canonical Example of How Badly You Can Pay for Poor Planning



## (It is important, nevertheless, to fix "late" issues...)

---

DAVIES: You know, building an aircraft is this huge project that involves all kinds of design issues and can take a long, long time. And if you discover a flaw, something that needs correcting, it can be expensive to correct it. What was the old Boeing like when it came to tweaking and fixing things that might be a problem?

ROBISON: Right. Airplanes are incredibly complex products. They involve hundreds of thousands of parts. And in the flight testing program, there are often surprises. There was always a saying that managers should plan for **20,000 surprises** in any airplane program.

# Using Agents to Clarify Specifications

---

Mocks in Practice

# "Wait — That's Not What I Meant"

---

Last week, we covered some strategies that help you write down **specifications** that are precise & verifiable.

Nevertheless, the specification comes from a noisy chain of communication from a stakeholder's mind over to yours. *You can follow the rules and still misunderstand each other!*

"I will tell you a terrible secret: language is punishment. Language must encompass all things and in it all things must again transpire according to guilt and the degree of guilt." — Ingeborg Machmann, *Malina*

# Go Ahead, Mock Me

---

A **UI mockup** is a disconnected, visual demonstration of your understanding of the feature's implementation

- it's a toy demonstration that is not hooked up to any existing backend implementation
- it doesn't need to be pretty, but it should look and behave the way that you intend for the feature to finally look and behave

(This is a simple definition that borrows heavily from many different HCI/design practices like [Wizard of Oz Experiments](#) or [Paper Prototypes](#) — check out CIS 4120/5120 for more!)

Spacebar is a backend, but we can still mock up features. For example, we should make sure we understand the meaning of this user story...

*As a moderator, I want to set a topic for the channel so that people joining know what's going on.*

So, we illustrate that we can set the existing `topic` column of a `Channel` and display that among its other settings in a Channel Settings tab...

# "Wait — That's Not What I Meant"

Five minutes with the moderator who asked for it:

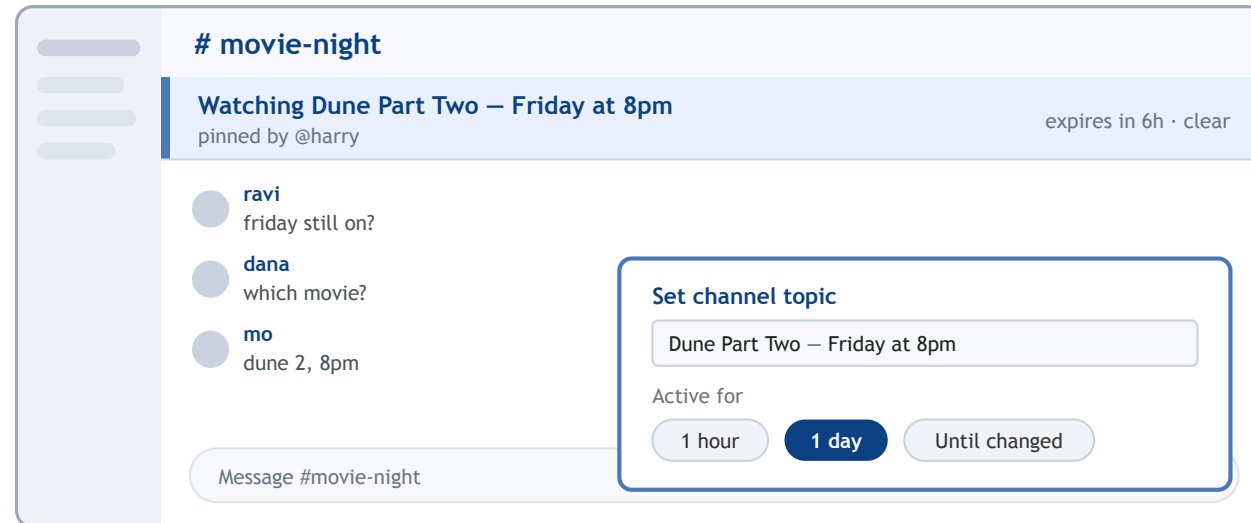
*"Why is it buried in settings? And it should go away on its own — movie night is Friday, it shouldn't still say that next Tuesday."*

The screenshot shows the settings for a Discord channel named "# movie-night". On the left is a sidebar with navigation links: "Overview" (highlighted), "Permissions", "Invites", "Webhooks", "Integrations", and a red "Delete Channel" link. The main area is titled "Overview" and contains the following settings:

- Channel name:** A text input field containing "movie-night".
- Topic:** A text input field containing "Watching Dune Part Two, Friday at 8pm", which is highlighted with a yellow background.
- Slow mode:** A toggle switch that is currently turned off.
- Private channel:** A toggle switch that is currently turned off.
- Age-restricted:** A toggle switch that is currently turned on.

A blue "Save changes" button is located at the bottom right of the settings panel.

Now we understand that there may be two senses of the "topic" for a channel—a permanent summary of its purpose *as well as* an ephemeral setting for the current theme of the discussion.



# What the Mock Cost, and What It Saved

---

The v1 spec is one clause: a nullable string on `Channel`, writable by anyone with `MANAGE_CHANNELS`. The v2 spec adds a feature in addition to this schema change and raises a number of questions:

- How long does it stay up — a fixed menu of durations, or any interval?
- At expiry, does the topic clear, or revert to whatever it said before?
- Clients already have the channel open. Does the banner vanish on its own, or only on refresh? That is a gateway event, not an HTTP response.

Those are exactly the clauses a specification is supposed to contain — and no amount of care writing v1's spec would have produced them!

**"Building Both" Instead of "Arguing About Which"**

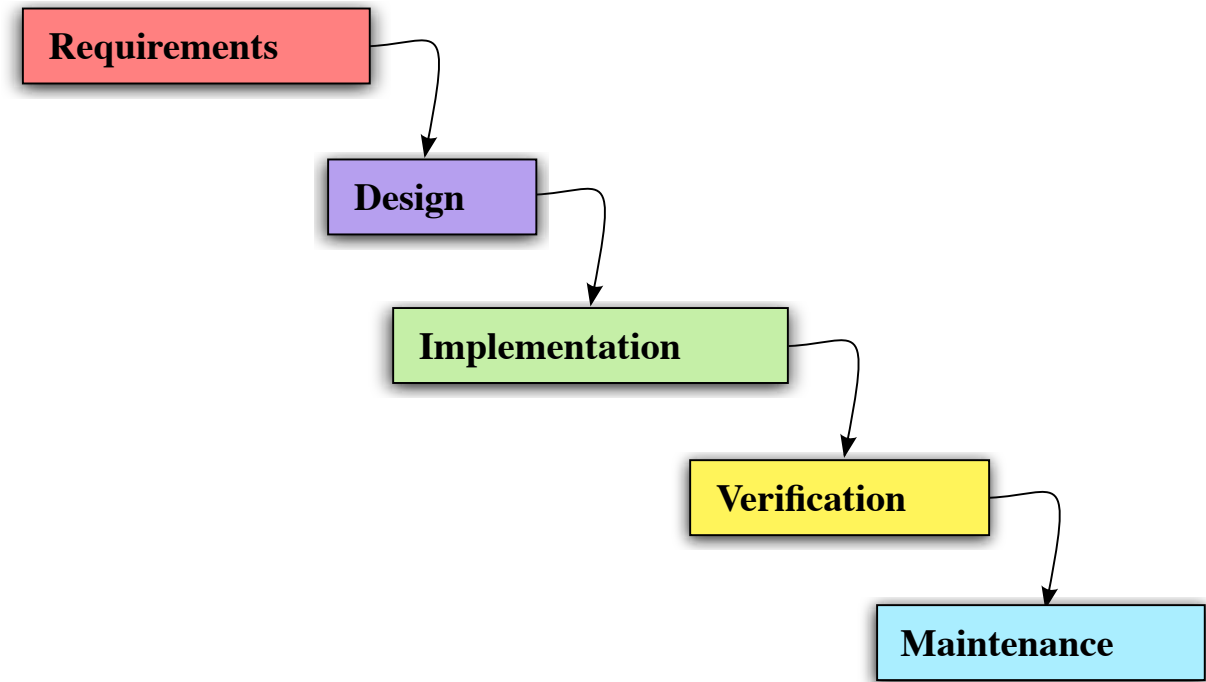
---

# Linear Development

---

Previously, a team would need to argue from first principles about which approach to take and then figure out later whether that was a good idea.

- see the **Waterfall Model** as an old, extreme version of planning
- extremely challenging to foresee the issues that will give you trouble to fix later in the process



# The New Way

---

It's cheaper & faster than ever to try two versions of an idea and see how they shake out!

- Run an experiment instead of having an argument
- Properly scoped, you can have two prototypes ready for evaluation in just an afternoon...
- *...or a grocery store trip!*

Right: [Just in time software](#) by Josh Bleecher Snyder

## Just in time software

DECEMBER 20, 2025

I didn't plan to write software in the grocery store last night.

I was tired and hungry, kids in tow. My long shopping list sat in a text message. I wished I could check items off as I found them. Ideas filtered through my head:

- I could transfer the list to a piece of paper: too tedious
- I could copy/paste it into a notes app or an email draft and delete items: too much tapping
- I could search the web and app store for a thing that would do what I want (of which there were no doubt a great many): slow and painful and annoying

Nope nope nope nope nope.

# Worktrees, Recalled from Week 4

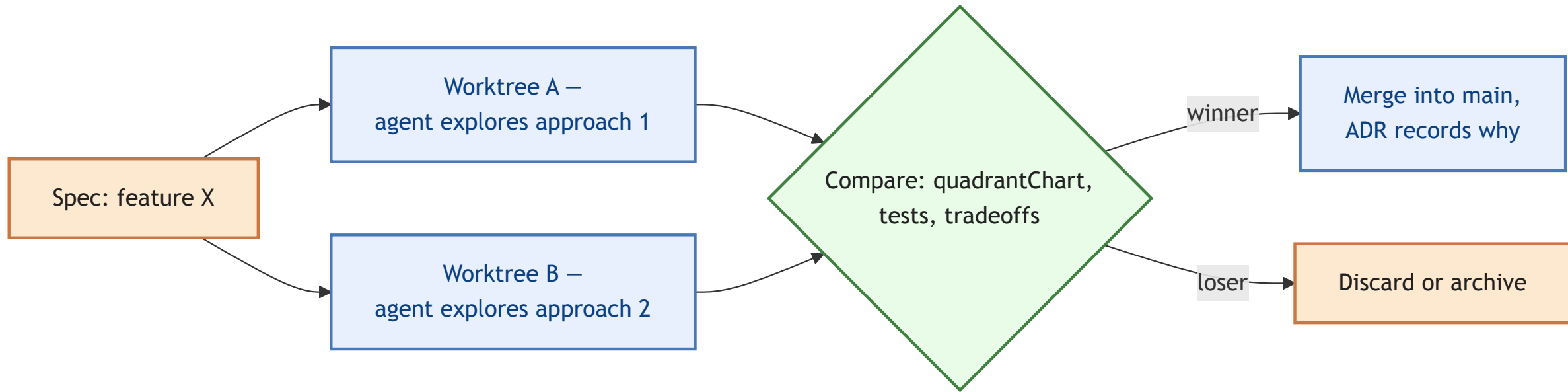
---

A **worktree** is a second working directory pointing at a different branch of the same repository.

```
git worktree add ../spacebar-slowmode-a -b slowmode/in-memory  
git worktree add ../spacebar-slowmode-b -b slowmode/persisted
```

A quick sort of "copy/paste" to get a clean version of the codebase to try something out on.

# The Workflow & Its Costs



- The spec, evaluation, merging and documenting are part of the process whether you try one idea or 1000
- The different worktrees represent *parallelizable* effort done ~entirely by agents

# Experimental Design

---

Like in any scientific experiment, you need to have constants, independent variables, and dependent variables.

- **Constants:** agents get the same spec, the same acceptance tests, and the same edge cases
- **Independent variable:** the particular design choice (use in-memory storage vs. writing to database)
- **Dependent variable:** the code produced

# Be Careful!

---

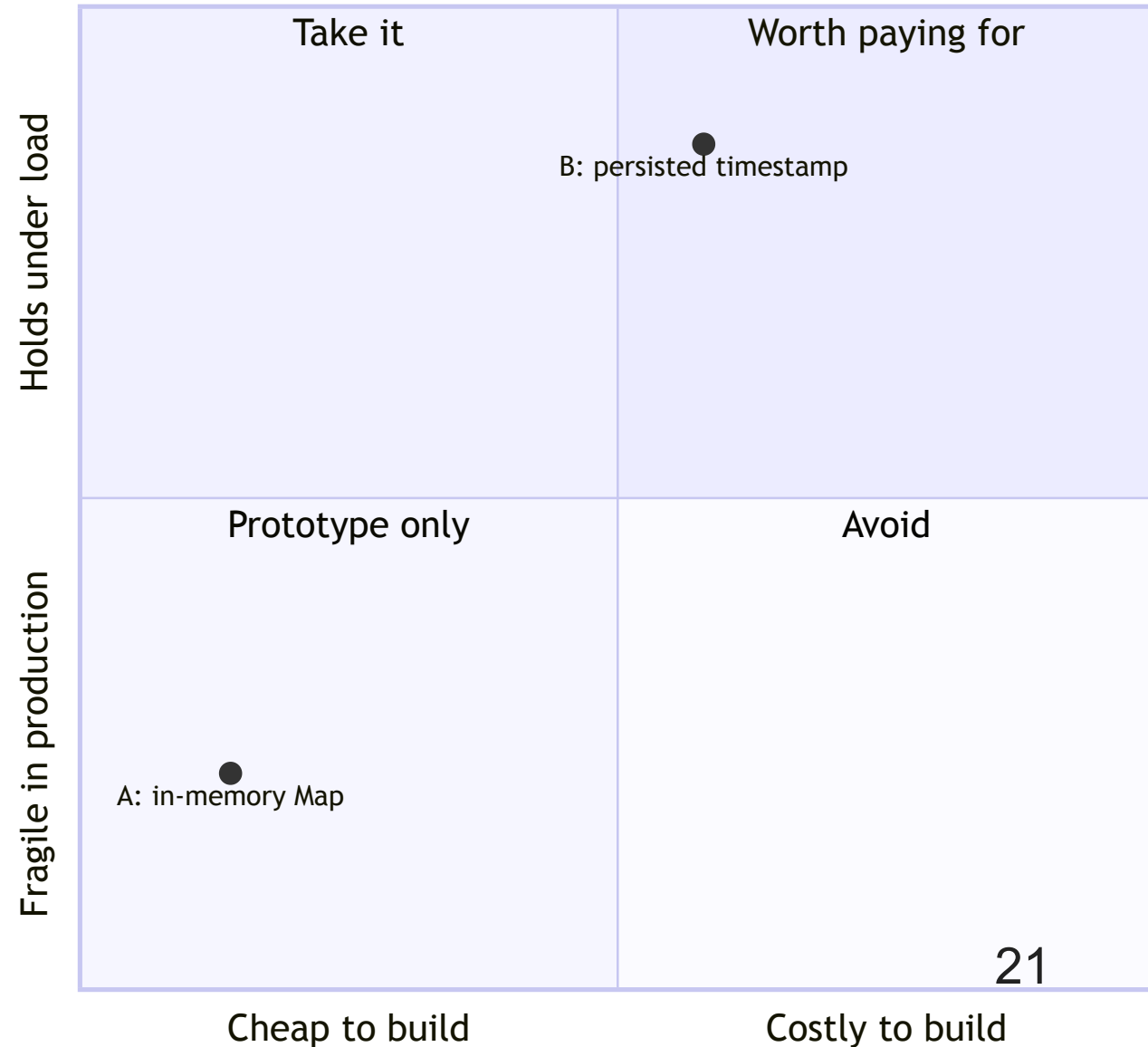
- If you give agent A a detailed brief and agent B a one-liner, you have measured your prompting, not the designs.
- Make sure to write the acceptance criteria *before* either worktree exists — otherwise the first implementation to finish quietly becomes the definition of correct.
- These work best as prototypes: *first passes*. The important output of a parallel prototype is **information**, not code.

# Evaluation: Plotting the Comparison

Lab 7 asks you to have Claude render this as a Mermaid `quadrantChart`. Pick axes that name the actual trade-off you found, not generic ones.

- Make sure to highlight which quadrants are good!
- The chart is not the decision, but a way of showing how you're justifying a decision.

Two slowmode designs, compared after building both



# Record the Decision, Then Delete the Loser

The winner gets merged. The loser gets an ADR sentence and then gets removed:

```
git worktree remove ../spacebar-slowmode-a  
git branch -D slowmode/in-memory
```

## ADR-002: Slowmode state is persisted, not held in process memory

**Context** — Both an in-memory map and a query against the last stored message satisfy the spec on a single process. We built both.

**Decision** — Query the persisted message, with a supporting index.

**Consequences** — (+) correct across restarts and across multiple API processes. (−) one extra indexed query on the hot send path.

# Why the ADR Matters More Than Usual Here

---

You just deleted a working implementation. In six weeks, an agent reading this repository will see the persisted version, notice it costs a database query per message, and helpfully propose caching it in a `Map`.

The ADR is the only artifact that survives to say *we tried that, and here is the specific thing that broke*. Without it the deleted worktree's information is gone, and the loop reopens — this time with less context than you had.

# A Worked Example: Strong Passwords

---

# The Feature

---

**Goal, in one sentence:** a Guild should be able to set requirements about acceptable password strength with respect to minimum length, number of uppercase characters, numbers, and symbols.

Two edge cases we will hold both designs to:

1. A password that violates multiple conditions should show a message that explains all conditions violated
2. The policy must apply to any place that a new password may be registered for an account

# As It Stands, There's An Implementation of This...

```
/*  
 * https://en.wikipedia.org/wiki/Password_policy  
 * password must meet following criteria, to be perfect:  
 *   - min <n> chars  
 *   - min <n> numbers  
 *   - min <n> symbols  
 *   - min <n> uppercase chars  
 *   - shannon entropy folded into [0, 1) interval  
 *  
 * Returns: 0 > pw > 1  
 */  
export function checkPassword(password: string): number { ... }
```

## ...And It's Ignored In the Current Logic

```
if (body.password) {
  const min = register.password.minLength ?? 8;
  if (body.password.length < min) {
    throw FieldErrors({
      password: {
        code: "PASSWORD_REQUIREMENTS_MIN_LENGTH",
        message: req.t("auth:register.PASSWORD_REQUIREMENTS_MIN_LENGTH",
          { min: min }),
      },
    });
  }
  body.password = await bcrypt.hash(body.password, 12);
}
```

27

# Two Ideas

---

- Enforce the policy at the request-validation layer, before handlers run
- Enforce the policy in the endpoints that require it, using a fixed version of the implementation currently found in `passwordStrength.ts`.

# The Shared Brief

---

Both agents get this text to start.

Spacebar's registration config has a `PasswordConfiguration` block — `minLength`, `minNumbers`, `minUpperCase`, `minSymbols`. Almost none of it is enforced: `register.ts` checks only `minLength`. A scorer exists at `passwordStrength.ts` and has zero callers.

Enforce the configured policy. **(1)** A password violating multiple rules must produce a response naming every rule it violated, not just the first. **(2)** The policy must apply everywhere a password is set, not only registration — find those places yourself.

# Plus One Line Each

---

A → `proto/pw-schema-layer`

Enforce the policy at the **request-validation layer**, before handlers run — alongside the existing schema machinery, so any route accepting a new password is covered by configuration rather than by a call somebody has to remember to make.

B → `proto/pw-domain-helper`

Enforce the policy with a **domain-level function** that each site calls explicitly, returning the set of violated rules rather than a score. Each place that sets a password calls it and converts the result into an error response.

Notice that these two prompts are pretty equal in length! We are comparing two designs, not two prompts.

# The Shot to Watch For

---

New account registration is the endpoint this feature is nominally "about," but it's not where the tricky bug lives!

```
# Register with a compliant password, keep the token, then just... change it.  
curl -s -X PATCH $API/users/@me \  
  -H "Authorization: $TOKEN" -H 'Content-Type: application/json' \  
  -d '{"new_password": "aaaa"}'
```

On `master` this returns **200** and sets the password to `aaaa`. The response body even echoes `"new_password": "aaaa"` back to the caller. Criterion #2 is what discriminates the two designs, and it is invisible to anyone who only tests the obvious route.

# What B Built: A Check at Every Site

B wrote one function that returns the list of violated rules, then called it right before each of the four places a password gets hashed.

```
// src/api/routes/users/@me/index.ts
const policy = checkPasswordPolicy(body.new_password);
if (policy.violations.length > 0) {
    throw FieldErrorList("new_password", policy.violations.map(
        (rule) => passwordPolicyError(req, rule, policy.requirements)));
}
user.data.hash = await bcrypt.hash(body.new_password, 12);
```

The same block appears in `register.ts`, `reset.ts`, and twice in `@me/index.ts`. A fifth route is covered only if its author remembers to paste it.

# What A Built: One Check in the Layer

A put one check in the `route()` middleware that runs before every handler.

```
// src/api/util/utility/passwordPolicy.ts
export const PASSWORD_POLICY_FIELDS = { RegisterSchema: ["password"],
  PasswordResetSchema: ["password"], UserModifySchema: ["new_password"] };

// src/api/util/handlers/route.ts
const policyFields = PASSWORD_POLICY_FIELDS[opts.requestBody];
if (policyFields) {
  const errors = checkPasswordPolicy(req.body, policyFields, policy, req.t);
  if (errors) throw new FieldError(50035, "Invalid Form Body", errors);
}
```

A never opened `reset.ts`. One entry in the map covers it.

# Did It Work?

```
$ psql $DATABASE -v guess=aaaa -f check-password.sql
```

| username                    | stored_hash                         | password_matches_guess |
|-----------------------------|-------------------------------------|------------------------|
| pwdemo_3001_125812 ← master | \$2b\$12\$Mhli8/onoLcmCp/0hcYKMO... | t                      |
| pwdemo_3002_125812 ← A      | \$2b\$12\$xy091WXUa.ruDKWy9oKYmu... | f                      |
| pwdemo_3003_125812 ← B      | \$2b\$12\$UlaJxhpPxzyDIoo7WU.7We... | f                      |

TL;DR—both solutions work to prevent a new password of `aaaa` getting stored for a user, although the base implementation accepts it!

# A Third Case Nobody Asked About

---

On `PATCH /users/@me`, the field `password` means two different things. If the account already has a password, it is the *current* one, sent to prove who you are. If the account has none, it *sets* one.

The request looks identical either way, so A's middleware cannot tell which case it is in. A had to fall back to a hand-written check inside the handler for that one case.

Neither the brief nor our two edge cases mentioned this. Building the prototype is what found it.

# What Did We Learn?

| DIMENSION                                     | A: REQUEST-VALIDATION LAYER FIX | B: PER-ENDPOINT |
|-----------------------------------------------|---------------------------------|-----------------|
| Complexity                                    |                                 |                 |
| Testability                                   |                                 |                 |
| Per-violation error messages                  |                                 |                 |
| Application to all sites of password creation |                                 |                 |

# Recording the Choice

Suppose we pick A. The ADR keeps the one thing the code alone won't tell the next reader.

## ADR-003: Password policy is enforced in the request-validation layer

**Context** — Both a validation-layer check and a per-site helper enforced the policy at all four places a password is set. We built both.

**Decision** — Check in `route()` middleware, driven by a map from request schema to password field.

**Consequences** — (+) A new route is covered by one map entry, and one code path needs testing. (−) The layer sees requests, not accounts. On `UserModifySchema`, `password` is checked by hand in the handler and must **not** be added to the map, or users whose current password predates the policy are locked out.

# Specification Failures

---

Mistakes to Learn From!

# Two Cautionary Tales

---

You have just spent an hour deliberately building one feature twice, in two places, from one written brief. This is exactly the right moment to look at what happens when two teams build from one written spec and nobody checks the join.

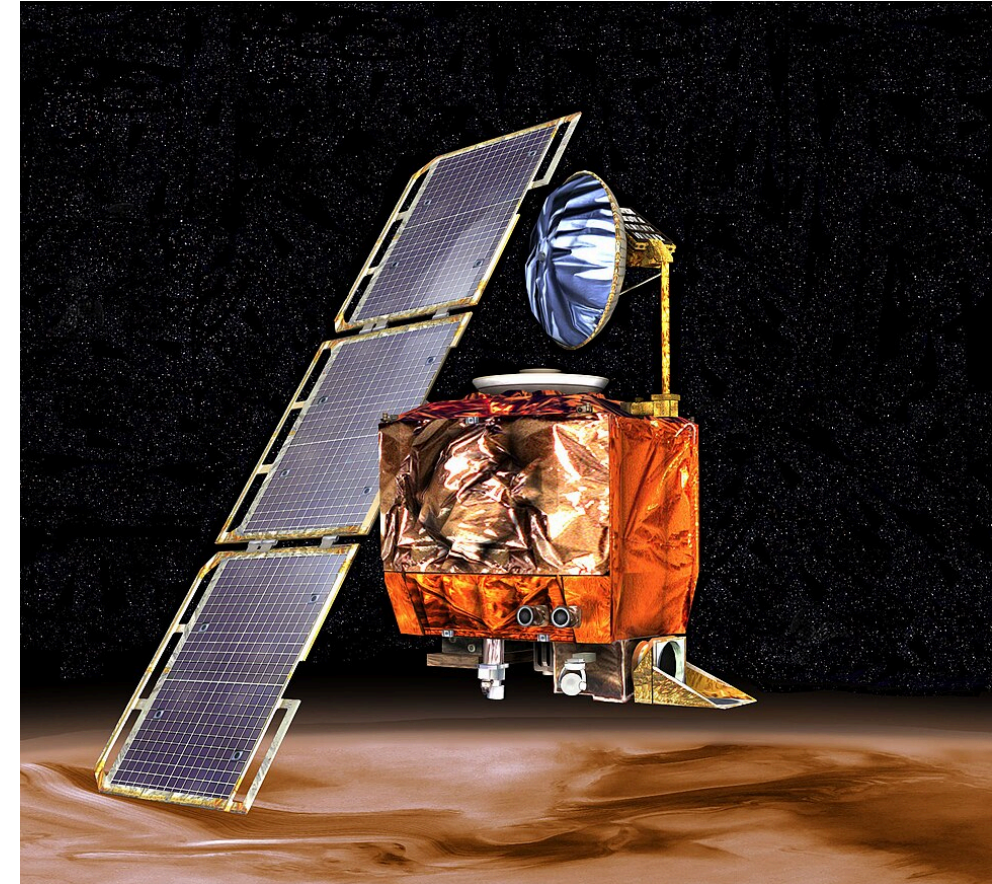
Both of these are specification failures, and they fail in opposite directions.

- The **Mars Climate Orbiter** had a spec. Two teams read it differently, nothing checked, and the mission ended in about four seconds of atmosphere.
- **Star Citizen** never had one. Fourteen years of funded demos, with no definition of done to be measured against.

# Mars Climate Orbiter

---

- \$327 million project for the purposes of studying Mars
- Spent nine months travelling through space to arrive in the planet's vicinity
- Upon arrival, needed one last maneuver to put it in orbit...



# Whoops!

---

During insertion, the orbiter was intended to skim through Mars's upper atmosphere, gradually **aerobraking** for weeks, but post-failure calculations showed that the spacecraft's trajectory would have taken it within 57 km (35 mi) of the surface. At this altitude, the spacecraft would likely have skipped violently off the denser-than-expected atmosphere, *[citation needed]* and it was either destroyed in the atmosphere, or re-entered heliocentric space.<sup>[2]</sup>

# Wherefore the (Allegedly) Skipping Orbiter?

---

The MCO project was a collaboration between NASA/JPL and Lockheed Martin. To establish consistency, they agreed on a **Software Interface Specification** (SIS).

- SIS dictated that thrust be calculated in newton-seconds
- Lockheed Martin built the thruster to report pound-force seconds...

$$1\text{lb} \cdot \text{s} = 4.448\text{N} \cdot \text{s} = \text{😞}$$

# 1: The Specification Was Right. So What?

---

The interface specification called for metric units. And yet!

- The mismatch was in an implementation that nobody checked against the spec.
  - ➡ use agents specifically for review of system documentation, e.g. `roborev` from week 4.
- No end-to-end test ever ran the two subsystems against each other with real numbers.
  - ➡ write—and run!—tests that exercise the **seam**, not each side of it separately. Both sides passed their own tests.

## 2. Agentic Development Parallelizes Far Beyond 2x

---

The failure was at a **seam** — the boundary between two independently developed pieces.

- Parallel work (e.g. several agents in a worktree) generates nearly as many seams as it does features.
- Each seam is an opportunity for an unfounded assumption about the other side.

### 3. Beware "Plausibly Correct" Implementations

The corresponding British unit was the pound force. If the values provided by the spacecraft engineers at Lockheed Martin Astronautics Co., Denver, Colo., had been in pound force, they would have been too large by a factor of 4.45.

But in engineering terms, these two values are still of the same order of magnitude. In other words, there is no really gross mismatch in the scale of calculations made with the one or the other--the kind of mismatch that can provide an intuitive hint that something doesn't add up. **These two units were close enough in magnitude that the unintentional substitution of one for the other apparently rang no warning bells.**

# Star Citizen

Crowdfunded starting October 2012, with an estimated release around 2014.

The year is, currently, 2026, and the game has not reached its 1.0.

(The game exists—people play it—but it's not finished.)



# Figuring Out What You Want

Anvil Aerospace

Aopoa

Argo Astronautics

Banu

Consolidated Outland

Progress has consistently been marked by **feature announcements and demos**

- a new star system!
- a new ship!
- a new layer to the real-time simulation of the persistent world!

These things come out... as do the next versions, and the next, and the next

Without a well-defined specification, "more features" always looks like progress, and there is no observation that could ever tell you that you are finished.

# To Release Something, Exercise Judgement

---

You can now produce an impressive demo of almost anything in an afternoon. What you cannot produce in an afternoon is the judgment about whether it should exist, and the demo is *exactly* the artifact that answers that question.

# Where This Leaves You for Lab 7

---

- Write the one-sentence goal and the acceptance criteria **before** either worktree exists. Both agents get the same brief.
- Two worktrees, two genuinely different designs — not one design and a refactor of it.
- Name your two edge cases up front, and expect building to surface a third you hadn't thought of. That third one is the point.
- The ADR is the deliverable that outlives the code. Write down what broke in the loser, specifically.

# Further Reading

---

- [git worktree](#) — official documentation
- [Mars Climate Orbiter Mishap Investigation Board Phase I Report](#) — NASA
- [The Lean Startup](#) — Eric Ries, on prototypes as experiments
- [User Story](#) — Martin Fowler
- [Star Citizen funding](#) — the live counter

# Questions?

---

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