Build the Loop

Spring Edition — Semester Project Guide

Human in the Loop: Why the Smartest Machines Still Need Us

Lazaros Toumanidis

In one sentence

Over fifteen weeks, you will build a complete Human-in-the-Loop system from a blank directory to a live, deployed treasure hunt — and in doing so, you will *feel* every concept that the other editions only describe.

Rust + Axum Python + FastAPI React Flutter Docker

What This Project Is

The Spring Edition is different from every other edition of this series.

The Summer Edition asks you to *think*. The Winter Edition asks you to *analyse*. This one asks you to *build* — and then to run what you built with real people in a real space, and watch what breaks.

The system is a **Treasure Hunt**: players follow a sequence of clues through a physical location, scanning QR codes or tapping NFC tags, submitting answers, and earning hints when they are stuck. Behind the game is a complete HITL architecture: a CREATOR who designs the experience, a PLAYER who generates feedback by playing, and a DEVELOPER (you) who builds the machinery connecting them through an AI layer.

Why a game?

Games have four properties that make HITL legible:

- **Defined tasks** — a clue has a correct answer, so we can measure human performance without ambiguity.
- **Motivated participants** — players want to win; their behavior is signal, not noise.
- **Visible feedback** — attempts, timing, and hint requests are explicit training signal.
- **Low stakes** — wrong predictions mean nudges, not harm. We can iterate freely and break things deliberately.

The Three Roles

CREATOR	The annotator. Encodes judgment into data by designing clues, setting thresholds, choosing what counts as correct. Their decisions are baked in long before any player sees them.
PLAYER	The oracle. Receives outputs (clues) and produces feedback (attempts, timing, hints). Does not know they are providing training signal. They are just trying to win.
DEVELOPER	You. Builds the infrastructure connecting creator and player through an AI layer, and must decide how much autonomy to give each component.

What You Will Deliver

By the end of week 15, you will have:

- A **Rust/Axum** REST API with WebSocket live events
- A **Python/FastAPI** AI sidecar (clue generation, difficulty estimation, adaptive hints)
- A **React** creator dashboard
- A **Flutter** mobile app for players
- A running local stack via `docker compose up`
- A 2-page **retrospective analysis** mapping what you built to the Five Dimensions

Prerequisites and Setup

Required before Week 1

- Rust toolchain (`rustup`, stable channel)
- Python 3.11+
- Docker + Docker Compose v2
- Node.js 20+ and npm
- Flutter SDK 3.22+ (Chapters 10, 12 only)
- An LLM API key if you want hosted-model features (`LLM_API_KEY`)

Getting Started

1. Clone the repository and enter the Spring Edition directory:

```
cd path/to/repo/editions/spring
```

2. Copy the environment template and fill in your model settings:

```
cp .env.example .env
```

3. Start the local stack:

```
docker compose up -d --build
```

4. Verify Rust compiles (takes 2–5 minutes on first build):

```
cd backend-rust && cargo build
```

5. Install Python dependencies:

```
cd ../backend-python && pip install -r requirements.txt
```

If `cargo build` succeeds in your environment and `docker compose ps` shows the services running, you are ready.

Watch out

The Rust build downloads dependencies on first run. Do this before Week 1 class, not during it.

15-Week Schedule at a Glance

Week	Chapter	HITL Concept
1	Building HITL: Why a Game?	The Loop Made Tangible
2	The Five Dimensions in Play	Framework Application
3	Designing for Humans: The Schema	Creator as Annotator
4	The API Layer: Rust and Axum	Intervention Design
5	Identity in the Loop: OIDC and Roles	Stakes Calibration
6	Answer Validation: The Judgment Call	Threshold as Policy
7	The Hint Engine: When to Help	Timing
8	AI in the Loop: The Python Sidecar	Uncertainty Detection
9	The Creator's Perspective	Human Contribution
10	The Player's Perspective	Oracle as User
11	Watching in Real Time	Feedback Integration
12	QR, NFC, and Physical Anchors	Physical Loop Closure
13	Testing HITL Systems	Feedback at Scale
14	Deployment: Shipping the Loop	Real Stakes
15	What the Game Taught Us	Meta-Reflection

Milestone Cards

Each card is one week's work. The **Build** column is concrete; the **Reflect** prompt is not graded. The checkpoint is the deliverable you should be able to demonstrate at the start of the following week.

Week 1 | Building HITL: Why a Game?

Build: Nothing to code yet. Read `intro.md`. Run `docker compose up -d --build`. Verify services start. Browse the full directory tree and write a one-paragraph description of each top-level folder in your own words.

Reflect: *You have probably used a HITL system today. Where was the human? Where was the loop? Did the loop actually close, or did it stop somewhere?*

HITL Concept: The Loop Made Tangible

A treasure hunt is, structurally, a HITL system. By the end of this semester, you will not just know that — you will have felt it.

Checkpoint

Docker services healthy. Directory tree documented in your notebook. One-paragraph answer: what is the human's job in this system?

Week 2 | The Five Dimensions in Play

Build: Read `chapters/02_five_dimensions.md` in full. Open `backend-rust/src/services/clue_validator.rs`. Annotate five specific lines (add a comment) where a dimension of the framework is embodied in the code. Do the same for one file in `backend-python/`.

Reflect: *"The human is in the loop" often means "a human approves outputs before they are acted on." Is approval the same as understanding? When does it add value, and when is it theater?*

HITL Concept: Framework Application

The Five Dimensions are not abstract. They live in specific files, specific parameters, specific defaults someone chose and stopped thinking about.

Checkpoint

Ten annotated lines (five Rust, five Python) with dimension labels and one sentence of reasoning each.

Week 3 | Designing for Humans: The Schema

Build: Study `db/migrations/001_initial.sql` and `002_external_id.sql`. Run both against your local Postgres. Sketch the entity-relationship diagram by hand. Identify every field where a human made a normative decision ("what counts as a correct answer?") vs. a technical one ("which type stores this?"). Add one field you think is missing.

Reflect: *A database schema is a theory about the world. Every constraint is a claim about what is always true. Which constraints here encode a claim about human behavior?*

HITL Concept: Creator as Annotator

The `answer_tolerance` field is a threshold the creator sets. A default is a decision made once, propagated silently to everyone.

Hint

The `external_id` column in migration 002 is for QR/NFC physical anchors. Don't try to understand it fully yet — it becomes clear in Week 12.

Checkpoint

ER diagram. Annotated list of normative vs. technical decisions. Proposed new field with justification.

Week 4 | The API Layer: Rust and Axum

Build: Implement the `GET /hunts` and `POST /hunts` endpoints in `src/routes/hunts.rs`. Add basic input validation. Write the handler tests. Run `cargo test` until they pass. Then: try to make a request that your validation should reject, and verify it is rejected.

Reflect: *A type error at compile time is a design conversation. A type error at runtime is an incident. Find one place where Rust's type system enforced a HITL design decision before any player touched the system.*

HITL Concept: Intervention Design

Every API contract is a decision about what the human can and cannot express. What can a creator *not* say to this system, because the schema won't let them?

Checkpoint

`GET /hunts` and `POST /hunts` pass handler tests. One documented example of a rejected invalid input.

Week 5 | Identity in the Loop: OIDC and Roles

Build: Configure Dex (`dex/config.yml`) with two users: a creator and a player. Implement the auth middleware in `src/middleware/auth.rs`. Verify that a player JWT cannot access a creator-only endpoint. Document the RBAC rules in a table.

Reflect: *Identity is not a property of a person. It is a claim made by a system about a person, which other systems choose to trust. What happens when the system's model of identity doesn't match the real-world situation? (Two players sharing a phone. A creator who is also playing.)*

HITL Concept: Stakes Calibration

Role separation is stakes calibration made structural. A player with creator access could invalidate everyone's data.

Checkpoint

Login works for both roles. Creator endpoint returns 403 for player JWT. RBAC table documented.

Week 6 | Answer Validation: The Judgment Call

Build: Implement fuzzy matching in `src/services/clue_validator.rs` using the Levenshtein distance or Jaro-Winkler similarity. Make the `answer_tolerance` field live — different clues should behave differently. Write a test matrix: ten (answer, correct_answer) pairs, three tolerance levels, expected result for each.

Reflect: *Every threshold is an argument. The argument is usually implicit. Write out the explicit argument your default threshold makes. Who benefits from a strict threshold? Who is harmed?*

HITL Concept: Threshold as Policy

The validation tolerance is not a technical parameter. It is a policy decision about how much linguistic variation you accept from players — which is a proxy for who you designed the game for.

Hint

The `strsim` crate has Jaro-Winkler implemented. Add it to `Cargo.toml` rather than implementing from scratch.

Checkpoint

Test matrix (30 cells) with all cases passing. Default tolerance documented and justified in one paragraph.

Week 7 | The Hint Engine: When to Help

Build: Build the hint system: hints unlock after n failed attempts (configurable per clue). Implement three hint levels — vague, directional, near-explicit. Track hint requests in the event log. Add a per-clue `hint_policy` field (none, after_3, after_5, always_available).

Reflect: *Help that arrives too early prevents learning. Help that arrives too late causes frustration. The gap between them is the entire art of pedagogy. Where does your default policy sit in that gap? Who did you optimize for?*

HITL Concept: Timing

The hint unlock timer is the most direct implementation of timing in this project. The threshold you choose is a claim about when frustration outweighs learning.

Checkpoint

Three-level hints working. Hint events logged with timestamps. Policy field configurable. Demo: one clue that unlocks hints at attempt 3, one that unlocks at attempt 5.

Week 8 | AI in the Loop: The Python Sidecar

Build: Implement two services in `backend-python/`: (1) `clue_generator.py` — given a location description and theme, generates clue text via the Anthropic API; (2) `difficulty.py` — estimates the first-attempt success rate for a clue using sentence embeddings and a trained regressor. Connect both to the Rust API via HTTP. The sidecar must fail gracefully: if it is down, the game still runs.

Reflect: *An AI generates a clue. The creator approves it without reading carefully. A player spends 40 minutes because the phrasing was subtly ambiguous. Who is responsible?*

HITL Concept: Uncertainty Detection

The difficulty estimator outputs a probability. What should the system do when it is uncertain about that estimate? When should it route the clue to the creator for review rather than publishing automatically?

Hint

Start with a lightweight similarity baseline for semantic matching and difficulty estimation. If you later need stronger embeddings, add them deliberately rather than making them a boot-time dependency.

Watch out

The Anthropic API has rate limits. Cache generated clues in Postgres, not in memory — the sidecar will restart.

Checkpoint

`POST /clues/generate` returns a clue. `POST /clues/estimate-difficulty` returns a probability. Both endpoints return gracefully when sidecar is stopped.

Week 9 | The Creator's Perspective

Build: Build the React creator dashboard in `web/`: a hunt builder where the creator can create a hunt, add clues, set tolerances, preview the clue sequence, and export a QR sheet (stub the QR generation for now — it becomes real in Week 12). Integrate AI clue generation so the creator can request suggestions and edit them.

Reflect: *The annotator sees the world through the categories you gave them. What categories does your creator dashboard give to the creator? What can they not express because the form won't let them?*

HITL Concept: Human Contribution

Every UI affordance is a constraint on what the human can contribute. The creator's dashboard shapes the quality of every player's experience before a single player has joined.

Checkpoint

Hunt buildable end-to-end via the creator UI. AI suggestions requestable and editable. Hunt visible in the database after save.

Week 10 | The Player's Perspective

Build: Build the Flutter mobile app in `mobile/`: the player scans a QR code (use the device camera), submits an answer, sees the result, and can request a hint. The app should work offline for content already downloaded; only answer submission requires connectivity.

Reflect: *The oracle does not know they are an oracle. They are just trying to win. Is there anything in your player UI that makes the data-collection nature of the system visible? Should there be?*

HITL Concept: Oracle as User

Every interaction the player makes is feedback. The quality of that feedback depends entirely on the quality of their experience. A confusing UI produces noisy labels.

Hint

Use `mobile_scanner` for QR reading. Test on a physical device early — the simulator camera is unreliable.

Checkpoint

Player can scan a QR code, submit an answer, receive feedback, and request a hint. Full flow works on a physical device or emulator.

Week 11 | Watching in Real Time

Build: Implement the WebSocket observer stream in `src/routes/observer.rs`. Every significant event — clue scanned, answer submitted (correct or not), hint requested, hunt completed — broadcasts to connected observers. Build a minimal observer view in the React dashboard: a live feed that shows events as they happen.

Reflect: *You cannot understand a HITL system by reading its logs after the fact. You understand it by watching it run. What does the live feed make visible that the database would have hidden?*

HITL Concept: Feedback Integration

The observer stream closes the feedback loop for the developer. Once you have real play data, re-train your difficulty estimator from Week 8 on actual attempt counts. How much did the synthetic training data mislead you?

Checkpoint

Live event stream works: three events visible in the observer view within 10 seconds of a player action. Difficulty estimator re-trained on at least 20 real play observations.

Week 12 | QR, NFC, and Physical Anchors

Build: Implement QR sheet generation in the creator dashboard (use a JavaScript QR library to generate printable A4 sheets with one QR per clue location). If you have NFC hardware: implement NFC tag reading in the Flutter app using `nfc_manager`. Print and physically deploy at least one three-clue hunt in a real space.

Reflect: *The loop is not closed until a human has physically walked it. What surprised you when you ran the hunt in a real space that no amount of testing on a laptop would have revealed?*

HITL Concept: Physical Loop Closure

A HITL system that has never touched a human is a simulation. Until someone walks the route with their phone, many bugs are invisible.

Hint

Use `qrcode` (npm) for QR generation. Print on A4, laminate if outdoors. NFC is optional — QR alone is sufficient for the checkpoint.

Checkpoint

Printable QR sheet generated for a three-clue hunt. Hunt successfully completed by at least one person who was not you. One documented observation from the live run.

Week 13 | Testing HITL Systems

Build: Write integration tests that simulate the full loop: (1) a creator creates a hunt with three clues; (2) a simulated player makes three wrong attempts on clue 1, requests a hint, then answers correctly; (3) asserts the event log contains the expected sequence; (4) asserts the hint was unlocked at the right attempt count. Also: write one test that deliberately injects a flawed clue and asserts the system handles it gracefully.

Reflect: *A test that passes tells you the code does what you expected. It tells you nothing about whether what you expected was right. What would a test of your* design assumptions *look like?*

HITL Concept: Feedback at Scale

The integration test is a synthetic human. It is faster and cheaper than a real player, but it only tests the behaviors you thought to specify. Where is your test suite blind?

Checkpoint

Integration test suite covers the full three-clue loop with wrong attempts, hint unlock, and correct answer. All tests pass in CI (`cargo test`).

Week 14 | Deployment: Shipping the Loop

Build: Deploy the full stack to a machine that is not your laptop — a VPS, a local server, or a Raspberry Pi on a local network all count. Participants should be able to join using their own phone without any special setup. Run a live hunt with at least three participants you did not brief in advance.

Reflect: *A system that works on your laptop is a prototype. A system that works for strangers is a product. What was the first thing that broke when strangers used it?*

HITL Concept: Real Stakes

Deployment changes the cost model. Errors are no longer free. A misconfigured threshold, a missing hint, a confusing UI — all of these now have real consequences for real people's experience.

Watch out

If your VPS has limited RAM, the Rust binary + Postgres + Python sidecar together use ~800MB. A 1GB instance will OOM under load. Use a 2GB minimum or run Postgres on a separate container.

Checkpoint

System deployed and reachable from a device not on your local network. Live hunt completed by at least three external participants. Notes from what broke, what surprised you, what you changed under pressure.

Week 15 | What the Game Taught Us

Build: Write a retrospective analysis (1,500–2,000 words). Structure it around the Five Dimensions: for each dimension, describe one decision you made in the code, whether it worked, and what you would change. End with one paragraph on what the game made visible that the textbook could not.

Reflect: *You do not understand a system by reading about it. You understand it by breaking it. What did you understand, after Week 14, that you did not understand after Week 1?*

HITL Concept: Meta-Reflection

The retrospective is not a formality. It is the closing of the loop you have been building for fifteen weeks: human feedback on the system that was designed to generate human feedback.

Checkpoint

Retrospective submitted (1,500–2,000 words). Five Dimensions each addressed with a concrete code example. One "this surprised me" paragraph grounded in the live deployment.

Final Integration Checklist

Before your Week 15 retrospective, verify the following end-to-end:

	Capability
☐	Creator can log in and build a hunt via the React dashboard
☐	Creator can request AI clue suggestions and edit them
☐	Creator can export a printable QR sheet
☐	Player can scan a QR code on a physical device
☐	Player can submit an answer and receive correct/incorrect feedback
☐	Answer validation respects the per-clue tolerance setting
☐	Hints unlock after the configured number of failed attempts
☐	Observer sees live event stream during active play
☐	AI sidecar failure does not crash the core game
☐	Difficulty estimates update when re-trained on real play data
☐	Integration test suite passes on a clean checkout
☐	System survives a 30-minute live hunt with ≥ 3 participants
☐	Retrospective maps each deliverable to a Five Dimensions concept

Reference: The Five Dimensions

Dimension	Key Question
Uncertainty Detection	Can the system recognize when it is unsure?
Intervention Design	How does it ask for help — or choose not to?
Timing	When does it intervene — too early, too late, or just right?
Stakes Calibration	Does it understand the cost of being wrong?
Feedback Integration	Does it learn from the responses it receives?

laztoum.com · laztoum@protonmail.com