Final Project Report

Mohamed Sbeinati – 1009163717

Zherui Zhang - 1008865122

Submission Date: 2025/12/01

Table of Contents

1. Product Overview

1.1 TL;DR (≤65 words)

another.ai is an **Agent Reality Check Platform** that runs a reference browser agent across real SaaS websites and reveals *why* it fails. It captures full video recordings, traces, and logs for every run, and uses OpenAI to generate structured failure explanations. Teams use it to diagnose mis-clicks, loops, hallucinations, and blocked flows, before users encounter them in production.

1.2 Jobs To Be Done (JTBD)

JTBD 1 — Diagnose Failures With Evidence

As an agent developer, I want **evidence-based diagnostics** (videos, traces, failure reasons) showing exactly why my agent fails specific workflows, so I can fix root-cause issues instead of guessing.

JTBD 2 — Validate Production Readiness With Measurable Signals

As a technical lead preparing a launch, I want **quantitative success rates** and **per-site pass/fail results**, so I can evaluate whether the agent is stable enough for real users.

JTBD 3 — Reproduce User-Reported Bugs Reliably

As an engineer debugging issues, I want **reproducible test runs with consistent recordings and logs**, so I can pinpoint the exact step where the agent breaks.

JTBD 4 — Communicate Failures Clearly to My Team

As a PM or team lead, I want **human-readable reports summarizing failures**, so I can coordinate fixes without replaying videos manually.

1.3 Core User Journeys (CUJs)

CUJ 1 — Run an Agent Reality Check Across SaaS Sites (Primary)

Statement:
The user selects a predefined goal (e.g., “Find Pricing”) and runs the built-in browser agent across **10 curated SaaS websites**.

System Behavior:
The system launches 10 Playwright sessions in parallel, records full-session videos, logs

actions, captures DOM snapshots, and sends session metadata to OpenAI for pass/fail judgment and failure categorization.

Value:
Provides a reproducible benchmark of how well the agent handles real SaaS workflows.

CUJ 2 — Inspect Agent Behavior With Full Observability

Statement:
The user reviews exactly what the agent did on any given site.

System Behavior:
The dashboard presents an HTML5 video (from S3), step-by-step trace logs, reasoning text, and DOM snapshots. Users can scrub the video, follow each click, and compare the agent's interpretation to the page structure.

Value:
Makes mis-clicks, loops, and navigation errors **directly visible**, eliminating guesswork.

CUJ 3 — Understand Failure Reasons (AI-Generated Report)

Statement:
The user reads a structured explanation of why the agent failed or succeeded.

System Behavior:
Structured metadata (actions, timestamps, logs, DOM snapshots) is sent to OpenAI, which returns:

- primary failure reason
- failure classification (loop, hallucination, missing element, tool error)
- a summarized chain of actions leading to failure

Reports are rendered cleanly in Markdown.

Value:
Transforms raw logs into **actionable insights** teams can immediately act on.

CUJ 4 — Compare Historical Runs to Track Progress

Statement:
The user reviews previous test runs to determine if the agent improved after changes.

System Behavior:
The History view shows past runs (using localStorage and backend metadata), including:

- overall success rates
- per-site trends
- changes in failure categories over time

Value:
Allows teams to confirm whether fixes actually improved reliability.

2. MVP Development Justification

2.1 Initial Hypothesis

Our original hypothesis was that **early-stage product, marketing, and founding teams need faster, trustworthy customer insight before launch**, but traditional research (surveys, panels, interviews) is too slow and expensive.

We framed this with three primary **Jobs To Be Done**:

- **Feature Confidence** – As a product manager, I want to quickly understand which feature customers value most, so I can prioritize the roadmap and avoid building the wrong thing.

- **Pricing Confidence** – As a marketing manager, I want to know how much customers are willing to pay at different price points, so I can pick a launch price without guessing.

- **Trust in Insights** – As a founder, I want confidence that my early customer insights are defensible, so I can speak to investors and partners with evidence, not gut feel.

The original product vision for **another.ai** was a **synthetic survey platform** that used LLM-based personas to approximate early customer research:

- Users create an account and set up projects.

- They input a product description and **feature variants** (e.g., A vs B) and pick personas (Luxury Buyer, Value Seeker, Eco-Conscious Shopper).

- The system orchestrates **20–30 synthetic responses per persona per variant**, with consistent persona conditioning.

- It aggregates:

    - % preference per persona,

    - confidence intervals / variance,

    - rationale clusters (grouped “why” snippets).

We defined three **Core User Journeys (CUJs)**:

1. **Run a Synthetic Feature Survey**
   Upload two feature variants, select 2–3 personas, run the survey → dashboard shows

preference splits, rationale clusters, and confidence intervals.

2. **Estimate Willingness To Pay (WTP)**
   Enter a baseline product and several price points → synthetic personas respond buy/no-buy → system builds demand curves and suggests an optimal price band.

3. **Validate Against a Human Micro-Panel**
   Export the survey to a human panel (e.g., Prolific), re-import results → system computes alignment scores between synthetic and human data and flags divergences.

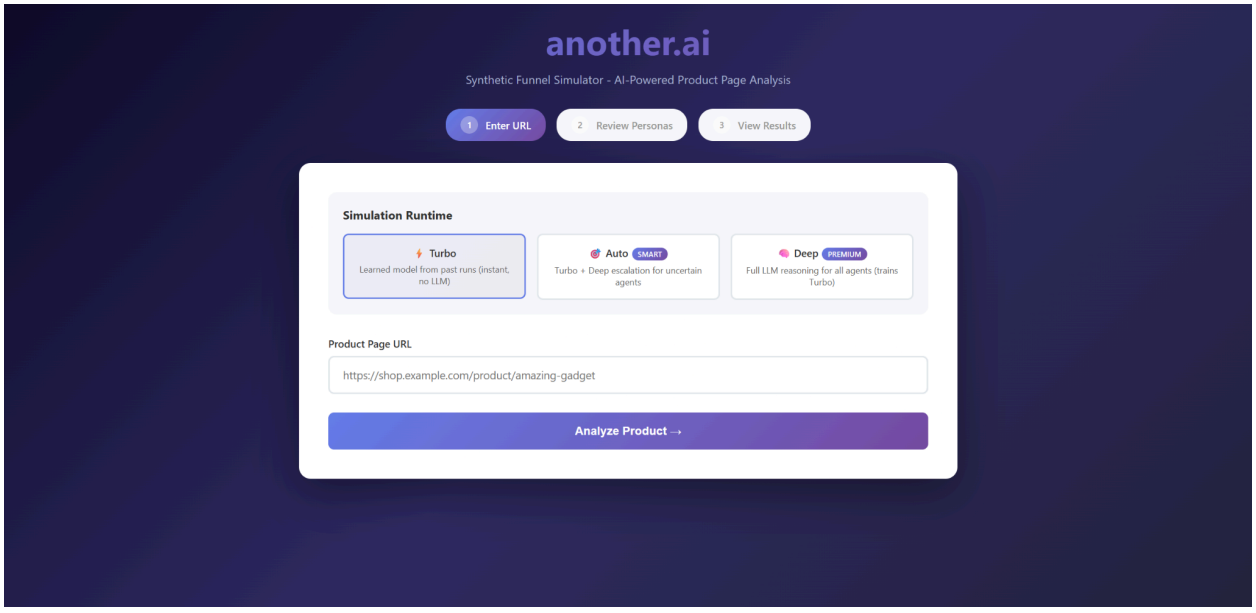
In short, our **initial MVP hypothesis** was:

> "We can give early teams fast, trustworthy product and pricing insights using synthetic respondents, with enough transparency and validation hooks that they will actually trust and use the results."

2.2 Key Learnings and Pivots

Over the semester, we realized that while the **research problem** was real, our **surface area** was too broad for the course timeline. As we tried to build and validate pieces of the synthetic survey engine, two major pivots emerged.

Pivot 1 — From Synthetic Surveys to Synthetic Funnel Simulator



As we started implementing the orchestration and talking to peers, several issues surfaced:

- Running full synthetic surveys with validation, WTP ladders, and alignment scoring was **complex and expensive** to implement properly within one term.

- We personally became more interested in flows that look like **real user journeys on live sites**, not just survey answers.

This led to our **first pivot**: narrowing from “synthetic survey research platform” to an **e-commerce focused Synthetic Funnel Simulator**.

The new product (first build) evaluated **e-commerce product pages** by simulating **90 synthetic shoppers** across three personas (Value Seeker, Skeptical Researcher, Impulse Mobile), each moving through a four-stage funnel:

1. Hook – Does the page capture attention?
2. Proof – Are there enough reviews/trust signals?
3. Price – Is the price acceptable?
4. CTA – Can they find and click “Add to Cart”?

Key features included:

- Facts-first page extraction (title, price, reviews, images, benefits) with confidence scoring.
- Three runtime modes (Turbo, Deep, Auto) balancing speed and cost.
- Deterministic seeding and variance logging.
- A **Conversion Health Score (0–100)**, top blockers, persona-level performance, and reasoning snippets.

Learning from this pivot:

- Moving from abstract survey questions to **concrete funnels on real product pages** made the product easier to explain.
- However, the target remained **e-commerce growth teams**, which we still struggled to access directly for deep validation.
- At the same time, the broader AI ecosystem (including our own interests) was moving quickly toward **autonomous agents navigating real products**, not only modeling human shoppers.

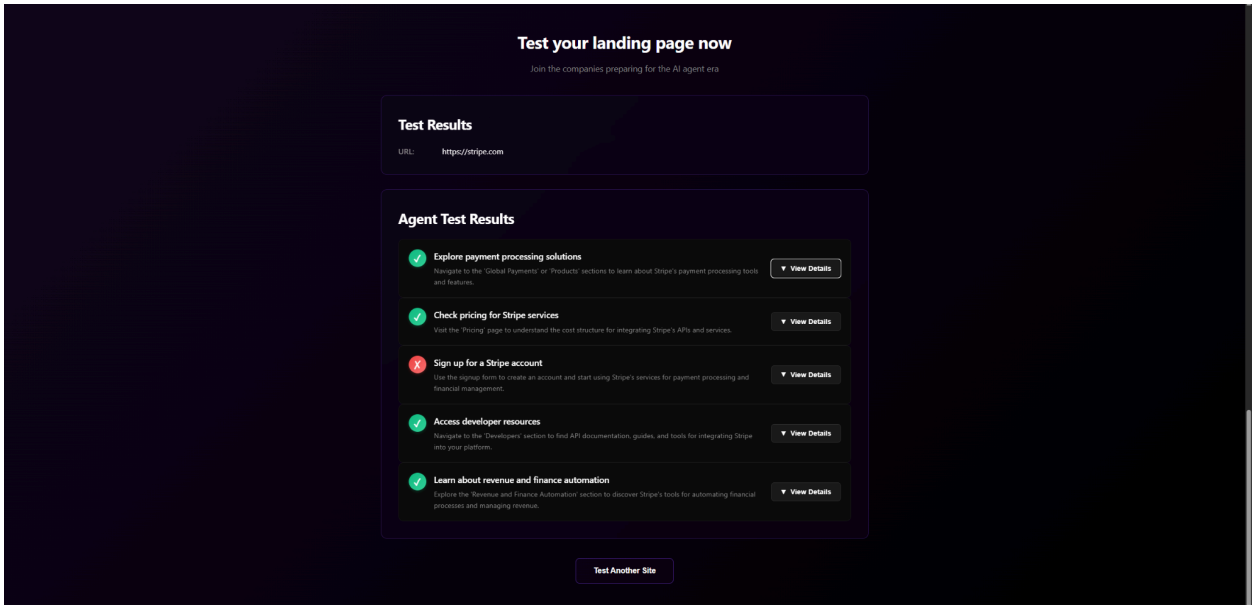
These realizations, plus early technical progress with browser automation, set us up for a second pivot.

Pivot 2 — From Funnels to Agent Reality Check for Browser Agents

After multiple concept interviews, prototype tests, and discussions with our professor, we made a second pivot toward what is now our final direction: **another.ai as an Agent Reality Check Platform**.

The new hypothesis:

> "Teams building browser agents don't just need more tests; they need **visibility** into why agents fail on real SaaS sites so they can fix them before production users suffer."



We validated this direction with:

- A **concept interview** with a B2B SaaS Product Manager ("Stephanie Lim"), where we pitched another.ai as "Paste a URL → another.ai analyzes the page → identifies 5 flows → runs AI agents → reports where they pass/fail."
    - She saw **agent-readiness** as early but inevitable, especially for startups integrating AI into onboarding/support.
    - She stressed that she would only trust such a system with:
        - full traces,

        - clear failure reasons,
        - transparency about whether the problem is the site or the agent/scraper.
- A **live prototype test** with a classmate:
    - They understood the product in ~15 seconds: “You’re basically checking if future AI agents can actually use my website without getting stuck.”
    - They loved the step-by-step traces and were excited by the idea of video recordings.
    - They found the default task suggestions a bit generic and wanted custom control.

Combining this feedback with instructor guidance, we decided to **narrow scope hard** again and focus on a specific, technically deep problem we could fully deliver: **testing a reference browser agent against a curated set of SaaS sites with rich observability**.

This became our current build:

- **Built-in reference agent** (Playwright + OpenAI) — no GitHub integration required.
- **10 curated SaaS websites** and **5 predefined goals** (pricing, talk to sales, sign up, help docs, reviews).
- **Full observability:** Playwright videos, traces, DOM snapshots, logs.
- **OpenAI-based evaluation:** success/failure judgments and human-readable reports explaining what went wrong (loops, hallucinations, tool errors, etc. forming part of the roadmap).

2.3 Justification of Final MVP

Given the time, complexity, and access constraints of the course, the final MVP represents the **smallest coherent product** that:

- **Solves a real, validated pain** for a concrete user (agent teams who see a gap between perfect demo runs and messy production behavior).
- **Leverages our existing technical work** (scraping, LLM orchestration, Playwright, cloud deployment).

We intentionally chose to **de-scope** several original ambitions:

- Synthetic survey UI with multiple CUJs and WTP ladders.
- Human validation micro-panel flows and alignment scoring in-product.
- Arbitrary user agents, any URL, and fully custom goals.

Instead, the final MVP focuses on:

- **One reference agent**, so we can control behavior and debugging.
- **A fixed, realistic testbed** (10 SaaS sites, 5 common goals).
- **Deep observability** (videos, traces, reports) rather than wide configurability.

In other words, we started with a broad **"synthetic survey research"** hypothesis, pivoted to a concrete **"synthetic funnel simulator"**, and ultimately converged on a focused, shippable **"Agent Reality Check Platform"** that we could fully implement, deploy, and demo, while still preserving a believable path back to the larger vision in future work.

3. Functional and Dynamic MVP

3.1 Core Functionality Delivered

Our final MVP implements a fully functional **Agent Reality Check Platform** capable of running a reference browser agent across real SaaS websites, recording each session, and providing structured AI-generated explanations of success or failure. The core functionality includes:

1. Reference Browser Agent (Playwright + OpenAI)

- A built-in, deterministic browser agent attempts predefined goals on curated SaaS sites.
- Each run captures:
    - Full session **video recordings**
    - Playwright **trace files**
    - Step-by-step **action logs**
    - DOM snapshots and extracted page content

2. Parallel Test Execution

- The system runs **10 real SaaS sites in parallel** using asyncio + Playwright.
- Each session executes a consistent workflow based on the user-selected goal.

3. AI-Powered Success/Failure Evaluation

- OpenAI GPT-4o-mini evaluates:
    - Whether the goal was achieved
    - Why the agent succeeded or failed
    - What type of failure occurred (loop, hallucination, missing element, tool error)

4. Video Playback + Trace Viewer

- The frontend includes a modal-based video player with:
    - Dynamic load from S3/CloudFront
    - Trace logs displayed side-by-side
    - A markdown-rendered AI report

5. Background Job System

- Users can run a test and return later; the frontend polls job status every few seconds.
- Completed runs persist locally and can be revisited.

6. Run History

- The dashboard stores past runs in localStorage and retrieves them on load.
- Users can revisit results, compare runs, and replay videos.

7. Fully Functional Dashboard (Next.js + Tailwind + shadcn/ui)

- A production-quality UI enabling:
    - Triggering test runs
    - Viewing run progress
    - Opening test results
    - Inspecting pass/fail outcomes
    - Reading AI-generated failure analyses

All core CUJs (running tests, inspecting agent behavior, reading failure reports, comparing runs) are fully functional and dynamic.

3.2 Dynamic vs. Static Components

This section clearly identifies which parts of the MVP are **dynamic** (executed live at runtime) and which parts are **static** (fixed, pre-configured, or placeholder).

Dynamic Components (Real Data, Real Execution)

1. Agent Execution (Dynamic)

- Playwright launches real browser sessions.
- Agent actions are generated live via OpenAI (GPT-4o-mini).
- Navigation, clicks, forms, DOM extraction → all done dynamically on real sites.

2. Video & Trace Recording (Dynamic)

- Videos are created per session at runtime.
- Traces and logs are generated live and uploaded to S3.

3. AI Evaluation (Dynamic)

- OpenAI analyzes each session's metadata live:
    - Determines success/failure
    - Classifies error types
    - Generates a markdown report
- This step is fully dynamic and varies per site/run.

4. Dashboard Output (Dynamic)

- Pass/fail states update as background jobs finish.
- Video modals load content directly from S3.
- Trace logs and AI reports are rendered in real time.

5. Run History (Dynamic Persistence)

- LocalStorage persists past runs.
- Re-openable even after refresh.

Static Components (Purposeful Constraints for MVP Scope)

These are static **by design**, aligned with your MVP strategy and roadmap:

1. Test Site List (Static)

- The 10 SaaS websites are hardcoded.
- This avoids needing a scraping pipeline or user-defined URLs for MVP.

2. Goal List (Static)

- The 5 available goals (“Find pricing,” “Talk to sales,” “Sign up,” “Help docs,” “Reviews”) are predefined.
- Users cannot create new custom goals yet.

3. Built-in Agent (Static Agent Implementation)

- Only **one reference agent** is included in the MVP.
- No GitHub integrations or custom agent imports yet.

4. UI Copy, Marketing Sections (Static)

- Landing page content and illustrations are static.
- No CMS or dynamic text editing.

5. Report Taxonomy (Static Categories)

- Error categories (loop, hallucination, missing element, tool error) are static choices.

These static constraints were intentional and validated through user research + instructor guidance to avoid over-scoping and ensure delivery of a **deep, working** MVP.

3.3 CUJ Alignment

All **core CUJs** defined in Section 1 are implemented dynamically:

CUJ	Status	Dynamic Behavior Demonstrated
CUJ 1 — Run Reality Check	Fully Dynamic	Playwright agent runs **real browser sessions** across real SaaS sites
CUJ 2 — Inspect Agent Behavior	Fully Dynamic	**Video playback** + **trace logs** loaded from S3
CUJ 3 — Understand Failures	Fully Dynamic	OpenAI generates **live failure explanations**
CUJ 4 — Compare Historical Runs	Dynamic Persistence	**localStorage** and **backend metadata store** past runs

Our MVP meets the requirement that **core CUJs must be dynamic**.

4. Test Coverage

Our project uses **GitHub Actions** for continuous integration. On every push to `main`, the pipeline runs:

- **Backend tests** (pytest) with coverage reporting
- **Frontend tests** (Jest + React Testing Library) with coverage reporting
- Automatic upload of HTML coverage artifacts for both services

Coverage Results (Latest CI Run)

From the latest successful CI execution:

- **Backend (FastAPI / Python): 85% coverage**
    - Key modules such as `agent.py`, `llm.py`, `main.py`, and `runner.py` are all above 80%
- **Frontend (Next.js / React): 70% statements**, **83.8% branches**, **97.6% functions**

CI Reporting

The GitHub Actions workflow automatically:

- Runs full test suites
- Generates combined backend + frontend coverage summaries
- Uploads HTML coverage reports as downloadable artifacts

Submission Link

A sample successful CI run with full coverage reporting is available here:
https://github.com/MBZ-0/LiveGap/actions/runs/19807464232

5. Demo Recording & Testing Guide

5.1 Demo Video (3 minutes)

YouTube Link: *https://www.youtube.com/watch?v=XJ75tcr0Kq0*

5.2 Accompanying Write-Up (How to Test the App Yourself)

Use the live production deployment:

URL: https://d3lcgzvi9bu5xi.cloudfront.net/about/index.html
Login: Not required

Steps to Reproduce the Demo:

1. **Navigate to Dashboard**

2. **Start a new agent test**

    - Click **New**
    - Choose a pre-defined goal
    - Click **Run test on 10 SaaS sites**

3. **Wait (2-3 mins) Until all 10 websites are processed**

4. **Inspect agent behavior**

    - Click **Video** to watch recorded sessions

    - Click **Report** to open explanations

5. **Review past runs**
   The sidebar shows previous test runs.

6. Deployment Documentation

6.1 Production Web App

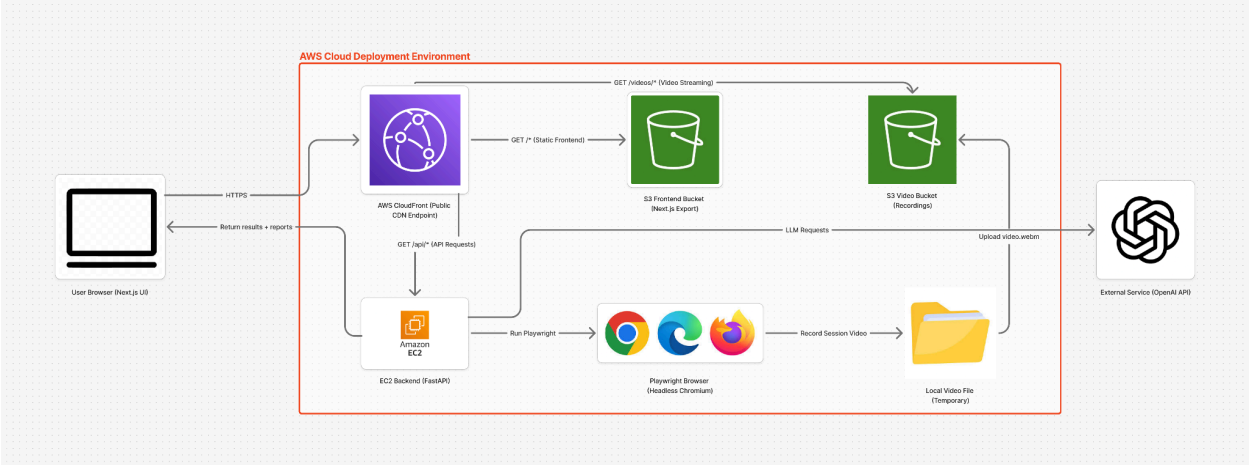
- **Public URL:** https://d3lcgzvi9bu5xi.cloudfront.net
- **Access:** No login or credentials required.

You can follow the full, step-by-step deployment guide in the main project README:

Deployment & setup details:
https://github.com/MBZ-0/LiveGap/blob/main/README.md

7. Updated Architecture Diagram

7.1 High-Level Architecture



7.2 architecture to code mapping table

Architecture Component	Code / Docs Location	One-Sentence Integration Description
User Browser (Next.js UI)	livegap-mini/frontend/app/page.tsx	Renders the main dashboard UI and sends HTTPS requests to CloudFront for static assets and `/api/*` calls.
AWS CloudFront (Public CDN Endpoint)	README.md – *Architecture / Production Deployment*	Acts as the **single public entrypoint**, serving the static Next.js frontend from S3 and proxying `/api/*` requests to the EC2 backend.

S3 Frontend Bucket (Next.js Export)	livegap-mini/frontend build output (out/)	**Hosts the exported static Next.js site** that CloudFront serves to users.
EC2 Backend (FastAPI)	livegap-mini/backend/app/main.py	Exposes the **REST API** (/api/run-reality-check, /api/run/{run_id}) and orchestrates agent runs for each request.
Playwright Browser (Headless Chromium)	livegap-mini/backend/app/agent.py & runner.py	Runs headless Chromium sessions on EC2 to **execute the reference browser agent** across the 10 SaaS sites.
Local Video File (Temporary)	livegap-mini/backend/app/videos/ (via agent.py)	**Stores raw session recordings on disk** before they are uploaded to the S3 video bucket.
S3 Video Bucket (Recordings)	livegap-mini/backend/app/s3_storage.py	**Receives uploaded .webm recordings** from the backend and serves them via CloudFront /videos/* URLs for playback in the UI.
OpenAI API (LLM Requests)	livegap-mini/backend/app/llm.py	Sends LLM requests from the EC2 backend to OpenAI to **power agent reasoning** and generate human-readable reports.