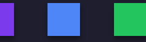
Reddit AI Debate Analyzer

AI-Powered Comment Reasoning & Debate Summarization

CS-485: AI-Assisted Software Engineering | Aryan U. Modi | NJIT Spring 2026



React · TypeScript · AWS Lambda · API Gateway · Neon PostgreSQL · Redis · Claude AI

What We Built & Why

PROBLEM

Reddit hosts millions of debates, but it's impossible to quickly evaluate which comments are well-reasoned vs. low-quality, or understand the overall positions taken in a thread.

SOLUTION

Fetches live Reddit threads, uses Claude AI to score every comment 0–100 for reasoning quality, and generates structured debate summaries with positions, evidence, and key disagreements.

TARGET USERS

Students, researchers, and journalists who want to evaluate online debate quality quickly — without reading every comment manually.

System Architecture

Frontend	React + TypeScript + Vite → AWS Amplify (S3 + CloudFront CDN)
Backend	Node.js + Express → AWS Lambda (Node.js 22.x runtime)
API	AWS API Gateway — REST /prod stage (CORS-enabled, HTTPS only)
Database	Neon — serverless PostgreSQL (threads, comments, summaries tables)
AI Engine	Anthropic Claude Haiku (claude-haiku-4-5-20251001) — batch comment scoring
Cache	Redis / Upstash — 5-min TTL for comments, 10-min TTL for summaries
Reddit Proxy	Cloudflare Worker — proxies Reddit API, bypasses AWS IP blocks

Live Demo Roadmap

#	Step	Description
1	**Open live app**	Welcome page loads fresh Reddit threads fetched via Cloudflare Worker proxy
2	**Refresh threads**	Different threads appear each time — server randomly shuffles 15 candidates per subreddit
3	**Click a thread**	Comments load with AI reasoning scores (0–100) — Claude Haiku analyzes in batch
4	**View Summary**	AI debate summary modal: main positions, supporting evidence, areas of disagreement
5	**Add thread by URL**	Paste any Reddit thread URL — app fetches, stores, and analyzes it on demand
6	**Health endpoint**	Show live backend: GET /prod/health returns JSON with DB + cache status

User Story A: AI Reasoning Score

As a reader, I want each comment scored 0–100 for reasoning quality

GET /api/threads/:id/comments → Redis cache check → PostgreSQL fetch (comments) → Claude Haiku batch analysis → Save scores to DB → Cache result (5 min TTL) → Return to frontend

KEY INSIGHT

AI analysis is fully awaited before returning the response — Lambda kills background (fire-and-forget) tasks immediately after the handler resolves. This was a critical bug: all scores were 0 until the await was added.

Score Distribution Example

- 90–100 (Excellent)
- 70–89 (Good)
- 40–69 (Average)
- 0–39 (Poor)

Example API Response

```
{
  "id": "abc123",
  "body": "Studies show...",
  "reasoning_score": 87,
  "score_label": "Excellent",
  "cached": false
}
```

User Story B: Debate Summary

As a moderator, I want a structured summary of the debate

GET /threads/:id/summary → Redis cache check → PostgreSQL check (existing?) → Claude Haiku generation → Upsert to DB → Cache 10-min TTL → Return modal

Main Positions

Distinct stances held by commenters — e.g., "Pro-regulation" vs. "Anti-regulation"

Supporting Evidence

Data, studies, or logical arguments cited to back each position

Areas of Disagreement

Core points where commenters fundamentally disagree — useful for debate analysis

Sample Claude Response Structure

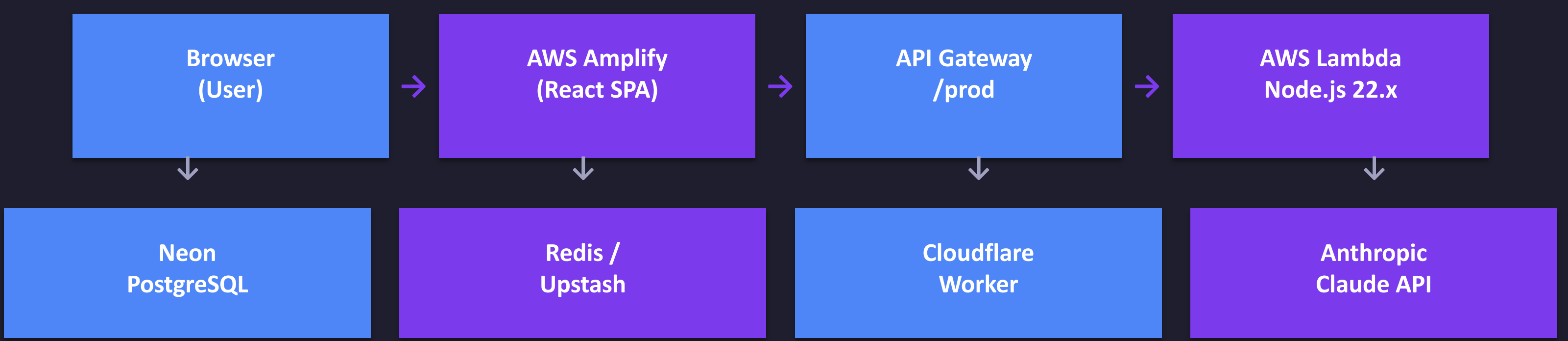

```
{
  "mainPositions":       ["AI regulation needed", "Self-regulation sufficient"],
  "supportingEvidence":    ["EU AI Act cited", "Industry track record cited"],
  "areasOfDisagreement":   ["Speed vs safety tradeoff", "Who bears liability"]
}
```

CI/CD Pipeline — 5 GitHub Actions Workflows

Workflow	Trigger	Purpose
run-frontend-tests.yml	Push / PR to any branch	React unit tests (Vitest)
run-backend-tests.yml	Push / PR to any branch	35 Jest unit tests — all AI mocked
run-integration-tests.yml	Push / PR to any branch	22 integration tests + live PostgreSQL
deploy-aws-lambda.yml	Push to main branch	Build, zip, deploy backend to Lambda
deploy-aws-amplify.yml	Push to main branch	Trigger Amplify webhook redeploy

Branch Protection: PR required on main + all 3 check workflows must pass before merge is allowed

AWS Deployment Architecture



GitHub Actions CD: Push to main → auto-deploys Lambda zip + triggers Amplify webhook (zero manual steps)

Security: All credentials stored in GitHub Actions Secrets + Lambda environment variables — never committed to source code

Live App: https://main.d5nlkn0tc4gbz.amplifyapp.com/ API: https://rf1kl76hmi.execute-api.us-east-1.amazonaws.com/prod

Testing Strategy

Unit Tests (35 passing)	
CommentService	Cache-aside logic, Redis TTL, DB fallback
SummaryService	Generate / cache / force-regenerate flows
AIService	Score clamping 0–100, batch analysis logic
ThreadService	CRUD, pagination, URL parsing
Note	All AI calls mocked — no real API calls made

Integration Tests (22 passing)	
Transport	Real HTTP via supertest against running server
Database	Live Neon PostgreSQL — real queries
Coverage	7 distinct code pathways tested
CI	PostgreSQL service container in GitHub Actions
Status	22/22 passing locally AND in CI pipeline

What Went Well

LLM-Driven Development

~90% of boilerplate (routes, services, tests, CI configs) generated by Claude — massive speed boost compared to writing from scratch.

Cache-Aside Pattern

Redis caching reduced PostgreSQL queries and Anthropic API calls significantly — faster responses and lower costs per page load.

Full CI/CD Pipeline

5 GitHub Actions workflows covering tests + auto-deploy — zero manual deployment steps from code push to production.

Integration Test Suite

22 integration tests against live PostgreSQL gave confidence before every merge — caught real bugs that unit tests (with mocks) would miss.

Cloudflare Worker Proxy

Cleanly solved the Reddit IP-block problem without touching any core application logic — a single Worker script deployed in minutes.

What Went Wrong

✗ **Reddit 403 IP Block**

AWS IPs are blocked by Reddit/Cloudflare. ~5 re-prompts and ~4 hours lost diagnosing before implementing the Cloudflare Worker proxy solution.

✗ **Lambda Fire-and-Forget Bug**

AI analysis was called without await — Lambda froze the Node.js process after response was sent, so all reasoning scores were always 0.

✗ **Gemini Quota Exceeded**

Google Gemini free-tier quota hit mid-sprint. Switched to Anthropic Claude Haiku (paid) — required updating all AI service code and tests.

✗ **ESM Mock Pattern in Jest**

Jest ES module mocking required { __esModule: true } — took ~1 hour to diagnose why mocked modules were not intercepting calls correctly.

✗ **Lambda Handler Misconfiguration**

Handler was set to lambda.handler but the compiled output exports dist/lambda.lambdaHandler — caused 502 errors until the Lambda config was corrected.

What We'd Do Differently

1 **Test Lambda Behavior Locally First**

Use AWS SAM or serverless-offline to replicate the Lambda execution model locally — would have caught the fire-and-forget bug immediately.

2 **Never Use Free-Tier AI for Demo Projects**

Start with a paid API key from day one. Quota limits during active development cause mid-sprint disruptions that waste hours.

3 **Use Specific LLM Prompts From the Start**

Include exact error messages, stack traces, and environment details in every AI prompt rather than describing symptoms vaguely — reduces iterations significantly.

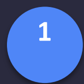
4 **Set Up CI/CD in P3, Not P6**

Early CI catches environment-specific bugs (env vars, DB connections) long before integration — would have saved debugging time in later phases.

5 **Plan for Infrastructure Constraints Upfront**

Research IP blocks, cold starts, timeout limits, and free-tier quotas before writing code — infrastructure surprises derail sprints more than code bugs.

Future Work



User Authentication

Save favorite threads, track personal score trends, and compare reading history across sessions using JWT + OAuth2.

Real-Time Updates

WebSocket connection for live comment streaming and incremental score updates as Claude analyzes new comments in real time.

Comment Filtering

Filter comments by reasoning score threshold (e.g., show only score > 70) — surfacing high-quality debate contributions instantly.

Multi-Subreddit Comparison

Compare average debate quality across different communities — visualize which subreddits have the most evidence-based discussions.

Historical Trend Analysis

Track how reasoning quality changes over time for a given thread or topic — useful for longitudinal research on online discourse.

Thank You

Reddit AI Debate Analyzer

Live App:	*https://main.d5nlkn0tc4gbz.amplifyapp.com/*
Backend API:	*https://rf1kl76hmi.execute-api.us-east-1.amazonaws.com/prod*
GitHub:	*https://github.com/aum-2004/CS485-Term-Project*

Tech Stack:

React + TypeScript | Node.js + Express | AWS Lambda | API Gateway | Neon PostgreSQL | Redis/Upstash | Claude AI | GitHub Actions

Questions?

CS-485: AI-Assisted Software Engineering | Aryan U. Modi | NJIT Spring 2026

TripPlanner

Collaborative Trip Planning, in Real Time

CS 485 AI-ASSISTED SOFTWARE ENGINEERING
SPRING 2026

Jonathan Martinez
Tirell Spence
Haroon Aftab
Alexander Tochtchev

Group trip planning is broken

The Problem

Planning a group trip is chaotic. Ideas scatter across texts, DMs, and spreadsheets. There is no single source of truth for the itinerary, and decisions get lost.

Our Solution

TripPlanner is a real-time collaborative web app. A group can search for places, build a day-by-day itinerary, and see each other's changes live, all in one workspace.

Target User

Anyone planning a trip with friends, family, or colleagues who want a shared, organized workspace instead of a group chat thread.

Six steps in six minutes

Here’s what you’ll see in the live demo

01 Register an account & create a new trip

02 Search for Points of Interest via Google Places

03 Add POIs to the itinerary, organized by day

04 Invite a collaborator by email (SendGrid)

05 Accept the invite, see real-time presence & itinerary sync

06 Confirm the app is live on AWS Elastic Beanstalk

Live Demo

≤ 6 MINUTES

Step 1 — Register & Create a Trip

Navigate to the deployed URL · Register with email + password (JWT auth) · Create a trip with name and destination

Step 2 — Search for Places

Open the POI Search panel · Query "pizza in New York" · Results show name, category, address, rating, price, and photo via Google Places API

Step 3 — Build the Itinerary

Click "Add to Itinerary" · Assign to a day · View items grouped by day with contributor names · Add notes to an item

Step 4 — Invite a Collaborator

Open Collaborators panel · Enter email · Assign role: Editor or Viewer · SendGrid fires an invite email with a unique token link

Step 5 — Accept & Collaborate in Real Time

Open invite link in a second browser window · Accept invitation · Online avatars update via WebSocket · Add a POI in one window — it appears instantly in the other

Step 6 — Confirm Deployment

Show the live app URL on AWS Elastic Beanstalk · Hit the /health endpoint live

How TripPlanner is built

Frontend	React + TypeScript + Tailwind CSS `built with Vite`
Backend	Go 1.21 + Gin HTTP framework
Auth	JWT tokens, bcrypt password hashing
Database	PostgreSQL 16 `Docker locally, AWS RDS in prod`
Real-time	WebSocket hub broadcasts itinerary & presence
External APIs	Google Places API, SendGrid
Deployment	AWS Elastic Beanstalk `Docker multi-stage build`

Four workflows keeping the codebase healthy

WORKFLOW	TRIGGER	WHAT IT DOES
`run-backend-tests.yml`	Push / PR to main	Runs `go test -v ./...` on the backend
`run-frontend-tests.yml`	Push / PR to main	Runs Jest unit tests (`npm test`)
`tests.yml`	Push / PR to main	Runs both backend and frontend tests in parallel
`run-integration-tests.yml`	Every branch, every commit	Hits the live backend via `BACKEND_BASE_URL` secret

SECRETS MANAGEMENT

`BACKEND_BASE_URL` stored in GitHub repository secrets. `SENDGRID_API_KEY` and `API_KEY` (Google Places) stored as Elastic Beanstalk environment properties — never committed to the repo.

DEPLOYMENT

No automated CD pipeline. Deployment to Elastic Beanstalk is done by packaging the repo as `tripplanner.zip` and uploading via the EB Console. The multi-stage Dockerfile builds the React frontend and Go binary into a single image served on port 8080 behind EB's nginx reverse proxy.

Three layers of test coverage

Frontend Unit

JEST + REACT TESTING LIBRARY

`InviteModal.test.tsx`

Tests invite form rendering, email validation, role selection, and submission behavior.

`ItineraryPanel.test.tsx`

Tests day grouping, adding/removing items, and notes editing.

Backend Unit

GO TESTING PACKAGE

Handler-level tests covering trip CRUD, collaborator management, invitation lifecycle, POI search, and authentication.

```
go test -v ./...
```

Integration

FRONTEND → LIVE BACKEND

Run against the deployed backend using the `BACKEND_BASE_URL` secret.

Test end-to-end flows: register, create trip, add POI, send invite, accept invite.

Skipped automatically when the secret is absent (e.g., on forks).

Five things we're proud of

Postmortem Successes

WebSocket real-time layer

Getting presence updates and itinerary broadcasts working correctly across multiple connected clients was a major technical win. The gorilla/websocket hub design kept the backend clean and easy to reason about.

Multi-stage Docker build

A single Dockerfile that builds the React frontend and Go binary together made local development and deployment consistent — no environment mismatch surprises.

Testing coverage

Writing .test-spec.md files before implementation made our Jest tests more focused and reduced back-and-forth on what behavior to test.

GitHub Actions CI

Having CI run on every PR caught bugs before they hit main, especially in the invitation flow which had several edge cases.

LLM-driven development

Using LLMs to scaffold boilerplate (handlers, test stubs, Docker config) saved significant time and let us focus on the business logic.

What broke: and what surprised us

HONEST ACCOUNT OF THE HARD PARTS

Getting the AI to generate architecture diagrams was a major struggle. LLMs kept producing inconsistent or incorrect outputs and it took many prompt iterations before we got something usable. Eventually we figured out the right way to prompt for it and got it to work.

Work done in one session was sometimes overwritten by a subsequent session, requiring effort to be redone. This happened because changes made by the LLM were not always committed and pushed before the next session began. As a result, the backend required more total iterations than it otherwise would have.

Deploying the email service was a major setback — it broke the application and blocked further progress until resolved. The fix required an unexpectedly large number of LLM iterations, making it the most time-consuming obstacle in the development process.

Integration tests passed locally but consistently failed in CI against the cloud deployment. The discrepancy traced back to the Google Places API being active only in production, exposing a code path that returned unfiltered results. Fixing it required comparing both environments to identify the differing behavior.

How our process evolved

Did LLM-Driven Development Help?

- **Yes, for scaffolding:** generating handler boilerplate, test stubs, and Dockerfile configurations was fast with LLMs.
- **Less effective for:** debugging subtle WebSocket concurrency issues and JWT token edge cases — required careful human reasoning.
- Prompt iteration was necessary most of the time; outputs needed review and correction, especially for Go-specific idioms.

Feature Branch Workflow

Our feature branch workflow ran smoothly throughout the project. Each team member owned a dedicated area including authentication, POI search, itinerary, and frontend UI so merge conflicts were rare. We opened pull requests for every feature, which GitHub Actions automatically tested before merge. Code reviews on PRs helped catch logic errors early and kept everyone aligned on the codebase. Overall, the workflow scaled well for a 4-person team and we'd use the same structure again.

Testing Evolution

- Started with manual testing; added unit tests mid-project after grader feedback on P3/P4.
- Integration tests were added last and revealed several endpoint bugs that unit tests had missed.

LESSONS LEARNED

Five things we’d do differently

01

Set up CI from day one

We added GitHub Actions workflows mid-project; having them from the start would have caught bugs earlier.

02

Define the API contract early

We had to renegotiate response shapes mid-sprint, causing rework on both the frontend and backend.

03

Automate deployment

Manual EB zip-and-upload deploys are error-prone; a proper CD step in GitHub Actions that deploys on merge to main would be better.

04

Mock external APIs in CI

Relying on live Google Places and SendGrid keys in tests caused flaky CI when secrets weren't available.

05

Spec out RBAC fully in P2

The Owner/Editor/Viewer permission model had ambiguous edge cases we had to resolve under time pressure.

FUTURE WORK

If we kept building TripPlanner...

Flight & hotel search

Connect to a travel API (Skyscanner, Booking.com) so users can add bookings directly to the itinerary alongside POIs.

Drag-and-drop reordering

Let users drag items between days and reorder within a day; broadcast new positions over WebSocket in real time.

Voting on POIs

Collaborators can upvote suggestions before they're added to the official itinerary — useful for large groups.

Export to PDF / calendar

Generate a printable itinerary or push it to Google Calendar as events.

Mobile app (React Native)

The REST + WebSocket backend is already mobile-ready; a native app would improve the on-the-go planning experience.

TripPlanner

Plan together. Go together.

GITHUB	github.com/jm2375/cs485
FRONTEND	https://github.com/jm2375/cs485/tree/main/frontend
BACKEND	https://github.com/jm2375/cs485/tree/main/backend

Jonathan Martinez Tirell Spence Haroon Aftab Alexander Tochtchev

THANK YOU

AI Specification Breakdown

Automated Jira Ticket Generation from Spec Documents

CS 485 – Project 7

Team Members

- Isabel Patrisso
- Roaa Elsayed
- Youssef Masoud
- Justin Carreno Bermeo

App Overview

The Problem

Writing Jira tickets manually from spec documents is time-consuming and tedious.

Engineers must:
- Read through dense spec paragraphs
- Identify discrete tasks
- Write summaries & acceptance criteria
- Estimate sizing (S / M / L / XL)
- Repeat this 8–12 times per spec

This typically takes **30–60 minutes** per sprint. Our app does it in seconds.

Target User

Software engineering teams that work from written specifications and manage sprint work in Jira. Ideal for product managers, tech leads, and developers who translate specs into tickets regularly.

Our Solution

Upload a .txt or .md spec → AI generates 4–8 Jira-ready epics and stories → Review, edit, approve → Publish to Jira.

Demo Roadmap

1 Upload Spec | 2 Generate | 3 Review & Edit | 4 Approve | 5 Publish

Step 1 — Upload Specification

AI Specification Breakdown

Transform technical specifications into actionable Jira issues with AI.

01 02 03 04 05

Upload Specification

Upload a specification document

Upload a .txt or .md file to generate structured Jira issues

Upload Specification

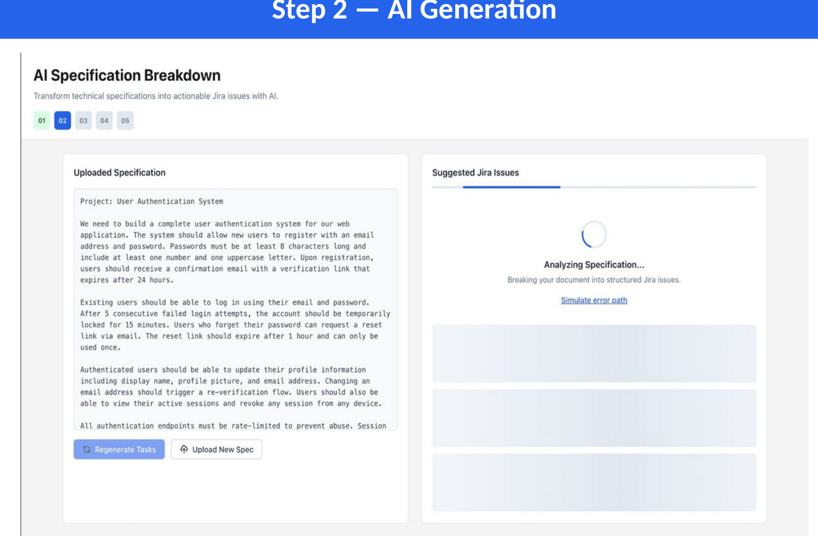
Suggested Jira Issues

No tasks generated yet

Upload a document and let AI break it into Jira-ready issues.

Drag & drop a .txt or .md spec file. The app previews the full document before generation.

Step 2 — AI Generation

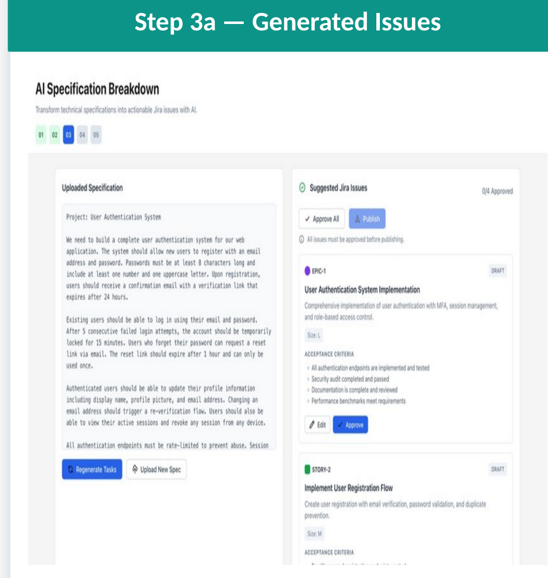


Click Generate — the backend calls the AI and streams back 4–8 structured Jira issues with a live progress bar.

Demo Roadmap (cont.)

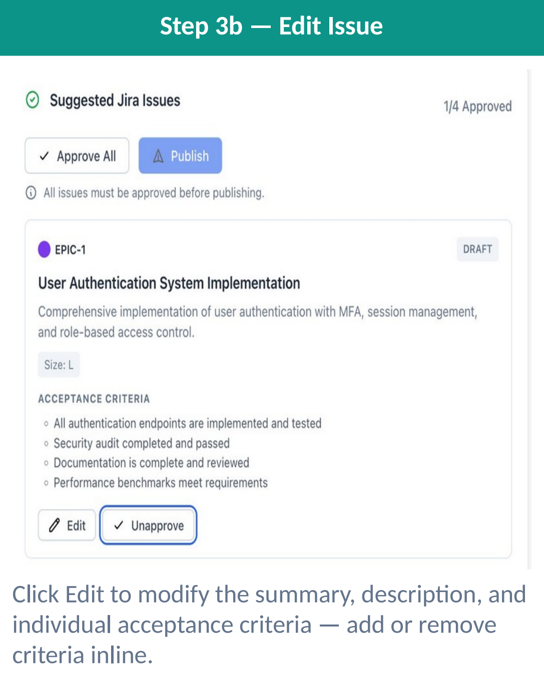
1 Upload Spec — 2 Generate — **3 Review & Edit** — **4 Approve** — 5 Publish

Step 3a — Generated Issues



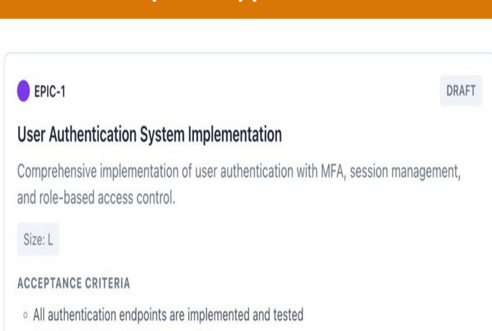
Each issue shows its type (Epic/Story), size, description, and acceptance criteria in a reviewable card.

Step 3b — Edit Issue



Click Edit to modify the summary, description, and individual acceptance criteria — add or remove criteria inline.

Step 4 — Approve Issues



Approve individual issues or use Approve All. Counter tracks progress. Publish unlocks when all are approved.

Demo Roadmap (cont.)

1 Upload Spec — 2 Generate — 3 Review & Edit — 4 Approve — **5 Publish**

Step 5 — Publish to Jira

AI Specification Breakdown

Transform technical specifications into actionable Jira issues with AI.

01 02 03 04 05

Uploaded Specification

```
Project: User Authentication System

We need to build a complete user authentication system for our web
application. The system should allow new users to register with an email
address and password. Passwords must be at least 8 characters long and
include at least one number and one uppercase letter. Upon registration,
users should receive a confirmation email with a verification link that
expires after 24 hours.

Existing users should be able to log in using their email and password.
After 5 consecutive failed login attempts, the account should be temporarily
locked for 15 minutes. Users who forget their password can request a reset
link via email. The reset link should expire after 1 hour and can only be
used once.

Authenticated users should be able to update their profile information
including display name, profile picture, and email address. Changing an
email address should trigger a re-verification flow. Users should also be
able to view their active sessions and revoke any session from any device.

All authentication endpoints must be rate-limited to prevent abuse. Session
```

Regenerate Tasks | Upload New Spec

Successfully published 4 issues to Project ABC. All issues are now available in your Jira workspace. You can view and manage them directly in Jira.

Published Issues

4 Issues

EPIC-1 — ✓ PUBLISHED

User Authentication System Implementation

Comprehensive implementation of user authentication with MFA, session management, and role-based access control.

Size: L

ACCEPTANCE CRITERIA

- All authentication endpoints are implemented and tested
- Security audit completed and passed
- Documentation is complete and reviewed
- Performance benchmarks meet requirements
- example: xyz

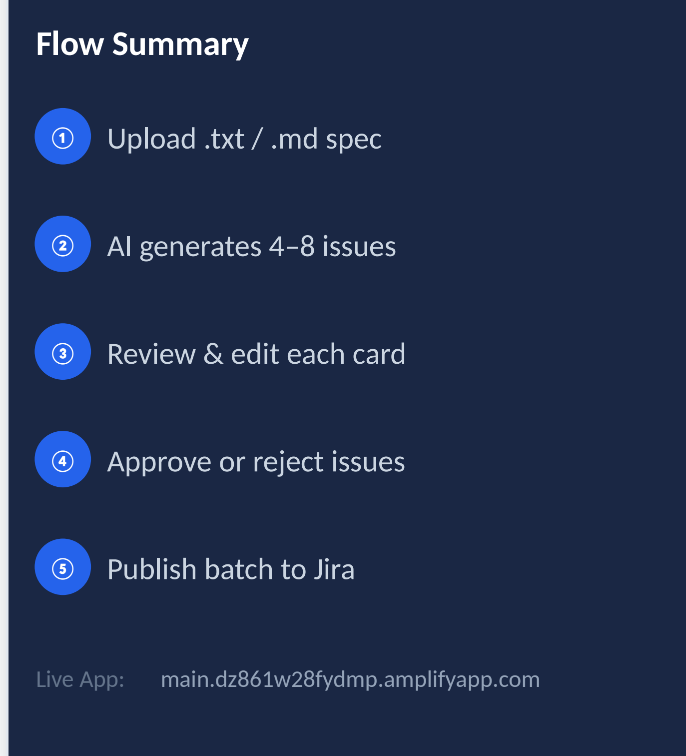
Edit | Project Backlog

STORY-2 — ✓ PUBLISHED

Implement User Registration Flow

Create user registration with email verification, password validation, and duplicate prevention.

Published issues appear with a PUBLISHED badge. Each links to the Jira Project Backlog. Success banner confirms the count.



CI/CD Overview

Push / PR → main or p6/**

Push → main only

Integration Tests

push/PR
main · p6/**

run-integration-tests.yml

- Spin up postgres:16 service container
- Install root, backend & frontend deps
- Write DB credentials to .env.test
- Start backend; poll /api/health up to 30 times
- Run HTTP API tests (local target)
- Run frontend unit tests (US2)
- Second job: run tests against deployed cloud API

Backend Tests

push/PR
main · p6/**

backend-tests.yml

- Install backend dependencies
- Run Jest unit tests
- Generate coverage report
- Run Node built-in test runner
- Upload coverage artifact

Frontend Tests

push/PR
main · p6/**

frontend-tests.yml

- Install frontend dependencies
- Run apiClient tests (Node runner)
- Run US2 frontend tests
- Report pass/fail per suite

Deploy Lambda

push
main only

deploy-aws-lambda.yml

- Checkout code
- Install production dependencies
- Zip backend into deployment archive
- Exclude test files and .env files
- Configure AWS credentials from Secrets
- Push zip to Lambda function MyAppBackend

Deploy Amplify

push
main only

deploy-aws-amplify.yml

- Checkout code
- Install deps and build frontend
- Configure AWS credentials from Secrets
- Trigger Amplify release job via AWS CLI

Tests (integration, backend, frontend) run on every push and PR — deploys only fire on pushes to main after tests pass.

CI/CD — Integration Test Deep Dive

Job 1 — integration-tests (push/PR to main or p6/**)

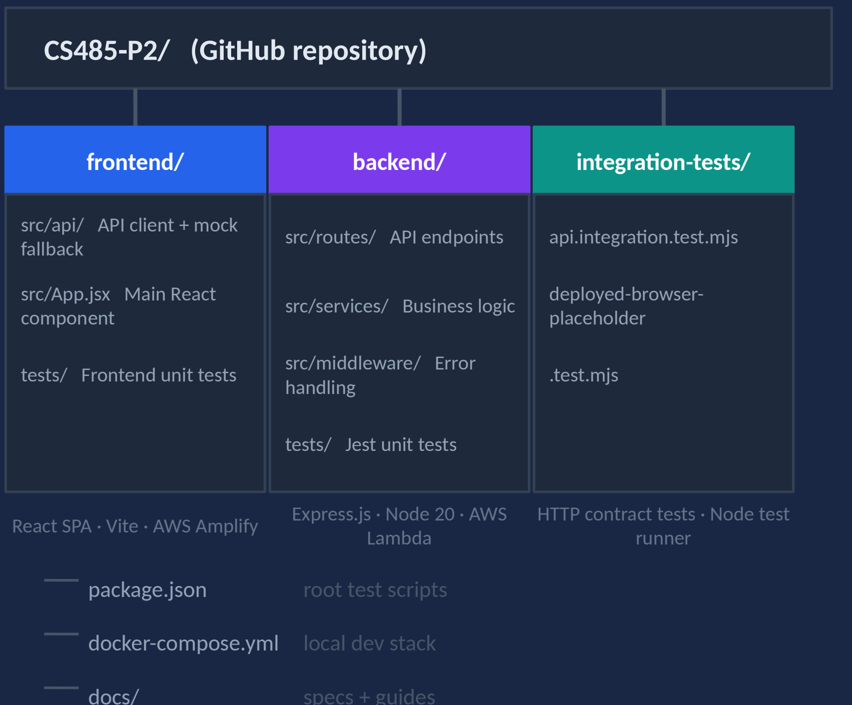
- **postgres:16 service container**
  Health-checked via pg_isready, 5 retries, port 5432 exposed
- **Checkout + Node 20 setup**
  actions/checkout@v4 and actions/setup-node@v4 with npm cache
- **Install all dependencies**
  Root (cross-env), backend, and frontend packages installed
- **Write test environment file**
  DB host, port, name, user, and password written to .env.test
- **Run database integration tests**
  Node built-in test runner against integration.test.js — continue on error
- **Start backend server**
  Background process on port 3001, stdout piped to backend/logs/backend.log
- **Health poll loop — up to 30 attempts**
  Calls GET /api/health every second; exits with error if never ready
- **Run HTTP API tests (local target)**
  Full integration test suite against localhost:3001
- **Run frontend tests**
  apiClient tests and US2 tests — both set to continue on error

Job 2 — integration-tests-cloud (main push only)

- **Trigger condition**
  Push to main only — pull requests and feature branches run Job 1 only
- **Checkout + Node 20**
  No database container needed — tests target the deployed API directly
- **Install root and frontend dependencies**
  npm ci including cross-env for cross-platform compatibility
- **Check API base URL variable**
  Reads INTEGRATION_API_BASE_URL from repo variables; warns and exits cleanly if unset
- **Run HTTP tests against deployed API**
  Same test suite as local, target set to cloud, pointing at AWS API Gateway
- **Test suite coverage**
  Health check, generate-issues (valid and invalid), publish-issues dry run, OPTIONS preflight (gated)

Project Architecture

REPOSITORY STRUCTURE



AWS DEPLOYMENT

Browser / User

AWS Amplify — React SPA
main.dz861w28fydmp.amplifyapp.com

API Gateway
prod stage

AWS Lambda
Express.js backend

PostgreSQL
Docker / local

AI Services
OpenAI · Anthropic · Gemini

GitHub Actions — push to main

run-integration-tests → deploy-aws-lambda → deploy-aws-amplify

Secrets: AWS_ACCESS_KEY_ID · AWS_SECRET_ACCESS_KEY

Postmortem: Successes, Failures & Lessons

✓ Successes

- End-to-end AI flow working (US1 + US2)
- 5 GitHub Actions workflows automated
- parseAIResponse() handles markdown code blocks
- 5-state frontend with localStorage persistence
- Integration tests run locally and in cloud
- LLM-written system prompt after 5 iterations
- Health-polling loop in CI (30 attempts)

✗ Failures

- CORS took far longer than expected — Lambda + API Gateway conflicts
- Node fetch masked CORS bugs in CI
- mockApi.js drifted from real backend shape
- Lambda deployed with missing DB env vars; health check didn't catch it
- Cloud integration tests skipped silently due to missing INTEGRATION_API_BASE_URL
- LLMs missed multi-file changes

→ Lessons

- Health check must verify DB, not just return 200
- List all affected files in LLM prompts upfront
- Most LLM tasks took 3–5 rounds of refinement
- Feature branches keep main stable but need active PR reviews
- Mock APIs need a contract test to prevent drift
- Sync before milestones; mismatches surfaced too late

Future Work

01

PDF & Word Spec Upload

Extend the upload step to accept .pdf and .docx files in addition to .txt and .md. Real-world specs rarely arrive as plain text — this removes the friction of converting documents before using the app.

02

Real Jira Publishing

Replace the mock publisher with live Atlassian REST API calls. JIRA_PROJECT_KEY and JIRA_BASE_URL are already stubbed in .env.example — completing this makes the Publish button actually create tickets in the team's board.

03

Section-Level Regeneration

Let users highlight a specific paragraph of the uploaded spec and regenerate only the issues tied to that section. Useful when a spec is updated mid-sprint and only one feature area changed.

04

Multi-Project Workspaces

Add user accounts and named project workspaces so teams can save spec history, reuse past issue sets, and maintain separate Jira project mappings — turning the tool from a one-off generator into a persistent team workflow.

N

Nexus

LinkedIn Revisualized.

Team

Krishi Shah
Dhyani Soni
Saanvi Elaty
Victor Jimenez

COURSE

CS 485 — Software Engineering

Spring 2026

LIVE URL

https://linkedin-redesign-z364.onrender.com/

The Gap on Today's LinkedIn

Why students need a focused, modern alternative.

✗ The Problem

Outdated UX, noisy feed, friction at every turn.

- ✗ Cluttered feed buries jobs and updates that actually matter
- ✗ Cold outreach is slow — no help drafting personalized messages
- ✗ No clear signal of how strong your profile actually is
- ✗ Cover letters live in a different tab and a different tool
- ✗ Career events and conferences are buried under generic search
- ✗ Recruiters can't quickly compare candidates side-by-side

✓ Our Solution

Nexus puts career action front-and-center.

- ✓ Clean, focused feed that surfaces jobs and connections first
- ✓ AI-drafted outreach messages tuned to each recruiter or role
- ✓ 0–100 Profile Readiness score with concrete, ranked tips
- ✓ Cover letter generator with five tailored templates + PDF export
- ✓ Conferences map to discover events by topic and city
- ✓ Recruiter Mode: side-by-side candidate review and export

TARGET USERS Students & early-career engineers · University career centers · Recruiters at tech companies

Today's Demo Path

Following Alex Johnson from sign-in to first interview reply.

Core App

STEP 1

Sign in, feed, profile, jobs.

Outreach Generator

STEP 2

Message drafted in seconds.

Profile Readiness

STEP 3

0–100 score with ranked tips.

Cover Letter + Map

STEP 4

Export PDF, find events nearby.

Recruiter Mode

STEP 5

Side-by-side candidate review.

ALEX JOHNSON → CS senior, looking for full-stack internships, four weeks before career fair.

Core Application

Six surfaces that mirror the LinkedIn workflow, rebuilt for clarity.

Feed

Posts, reactions, and updates from your network.

Jobs

Curated listings via The Muse API + saved searches.

Profile

Editable sections with live readiness scoring.

Network

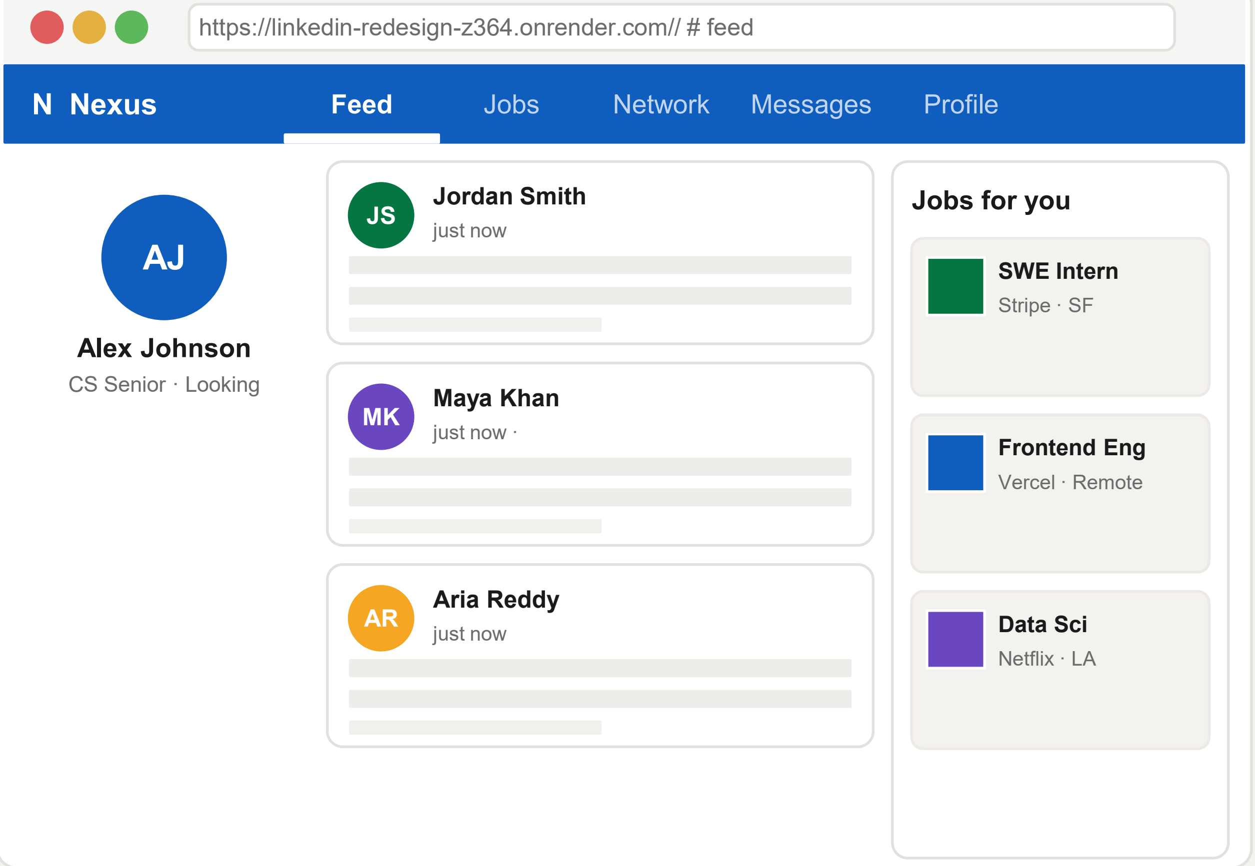
Connections, suggestions, mutual context.

Messaging

Threaded DMs with outreach templates inline.

Conferences

Location and field based conferences.



Outreach Generator

USER STORY #1 · Drafting a recruiter message in seconds.

AJ

Alex Johnson

CS Senior · Looking for SWE internships

> *I want to message a recruiter at Stripe, but I freeze every time. I need help opening the conversation in a way that sounds like me.*

— Alex, the night before a career fair

```
POST  /api/outreach/generate

{
  "recipient": "Stripe Recruiter",
  "role": "SWE Intern",
  "tone": "warm, concise",
  "profile": { ... }
}
```

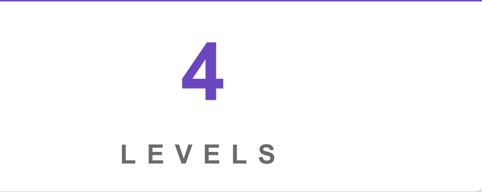
How it works

1. **Pick recipient**
   Choose recruiter, alum, or hiring manager from your network.
2. **Set role + tone**
   Specify the role you're targeting and the voice (warm / concise / formal).
3. **Edit &Send**
   Review, tweak the draft inline, and send — or save as a template

Profile Readiness

USER STORY #7 · Quantifying how recruiter-ready a profile actually is.

0–100
SCORE

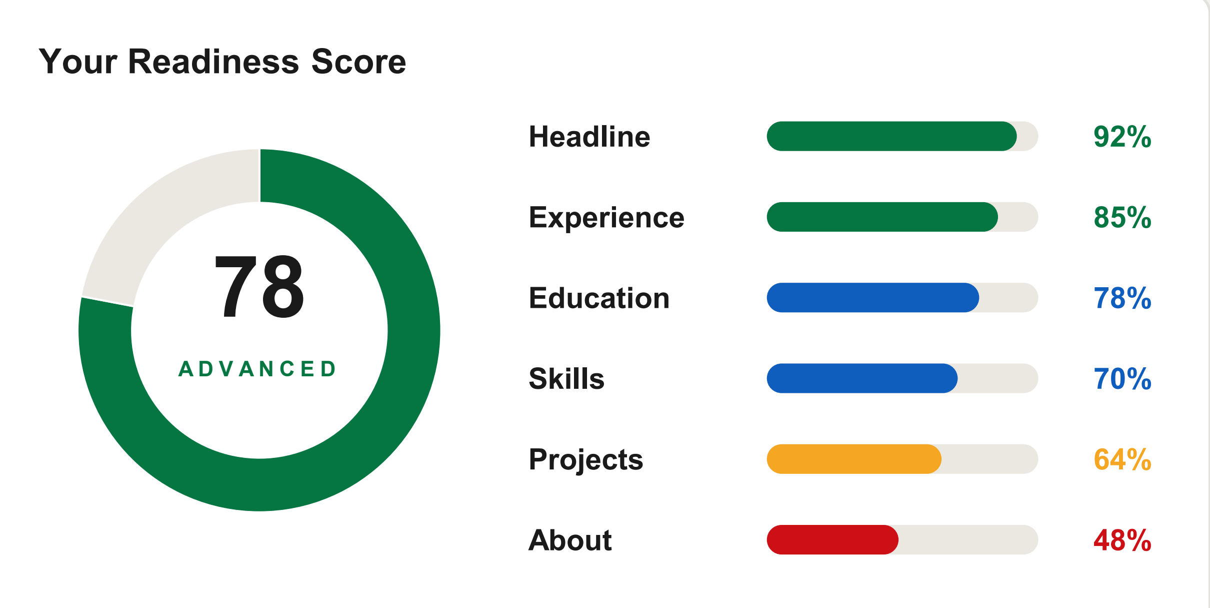


4
LEVELS

6
SECTIONS

3
TIPS

Your Readiness Score



API

`GET /api/profile/readiness`

Returns a **score 0–100**, a level (Starter → Expert), per-section completeness, and the top three improvements ranked by expected impact.

TOP TIP

Add 2–3 sentences to your About section.

Expected lift: +12 points. Mention what you're building, what you're great at, and what you're looking for next.

Cover Letter Generator

Five tailored templates with a one-click PDF export.

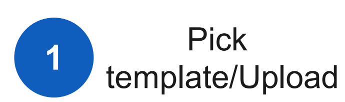
01
Concise Pro
300-word direct opener for engineering roles.

02
Story-First
Lead with a project narrative recruiters remember.

03
Research Track
Tuned for labs, grad apps, and research positions.

04
Career Switch
Bridges past role to your new target field.

05
Cold Outreach
Short, warm, and ends with a specific ask.


1 Pick template/Upload →

2 Paste job desc →

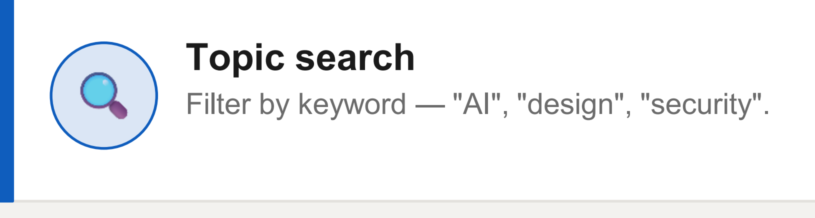
3 AI fills draft →

4 Review & edit →

5 **Export PDF**

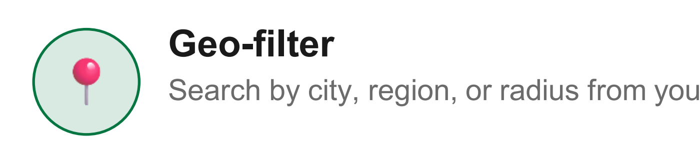
Conferences Map

Discover events and conferences by topic and location, powered by OpenStreetMap.



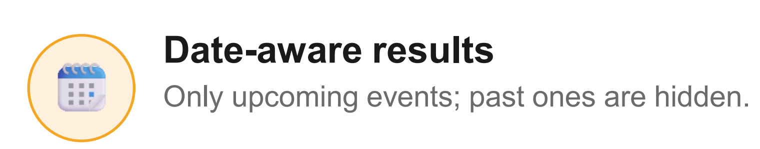
Topic search

Filter by keyword — "AI", "design", "security".



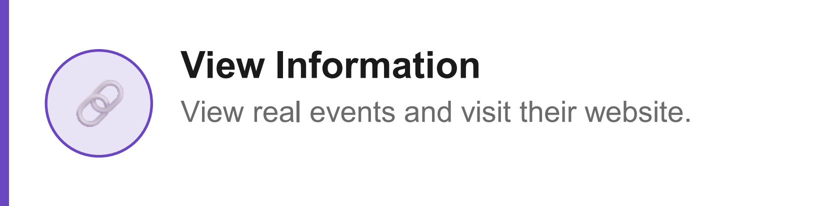
Geo-filter

Search by city, region, or radius from you.



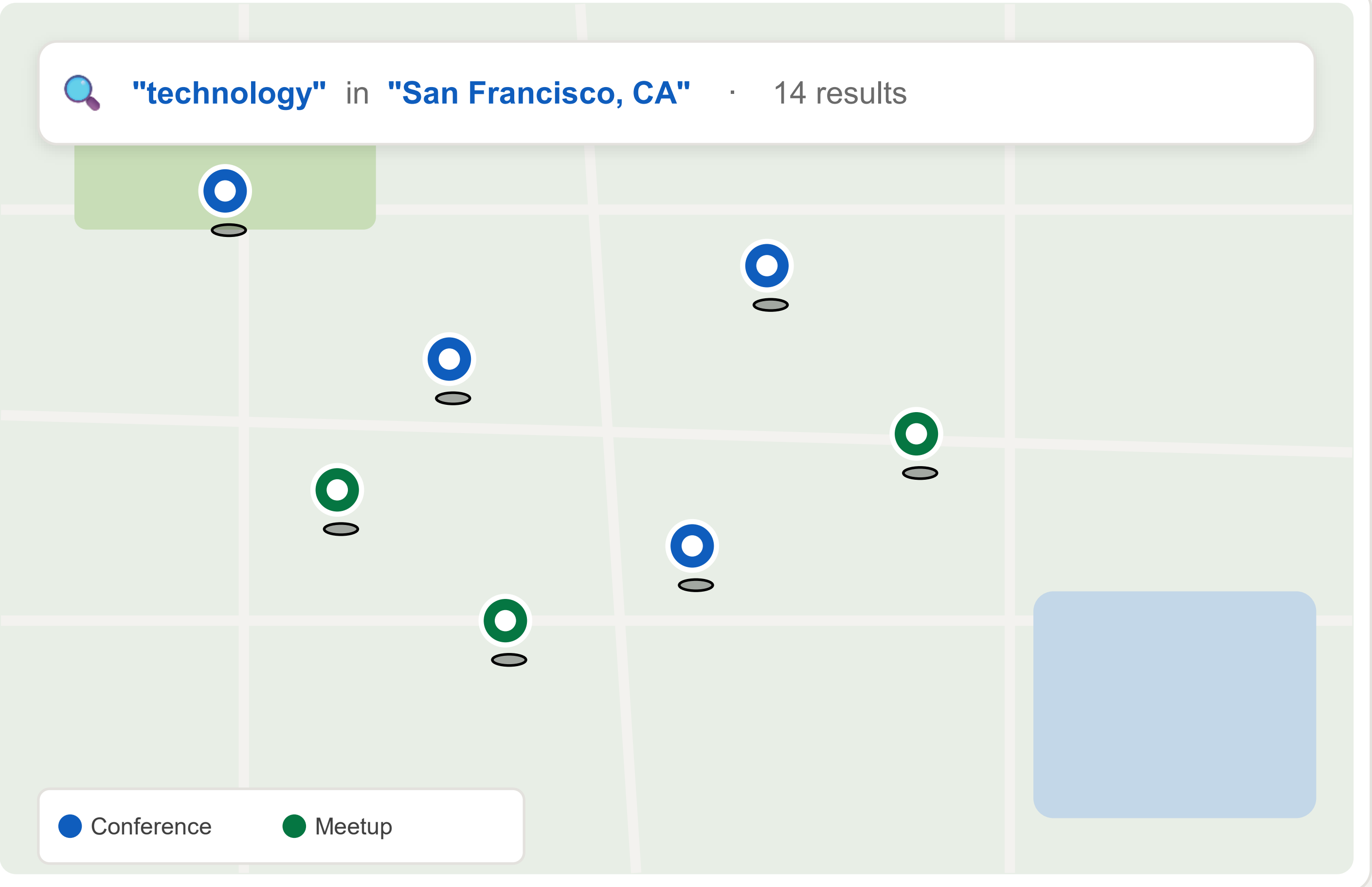
Date-aware results

Only upcoming events; past ones are hidden.



View Information

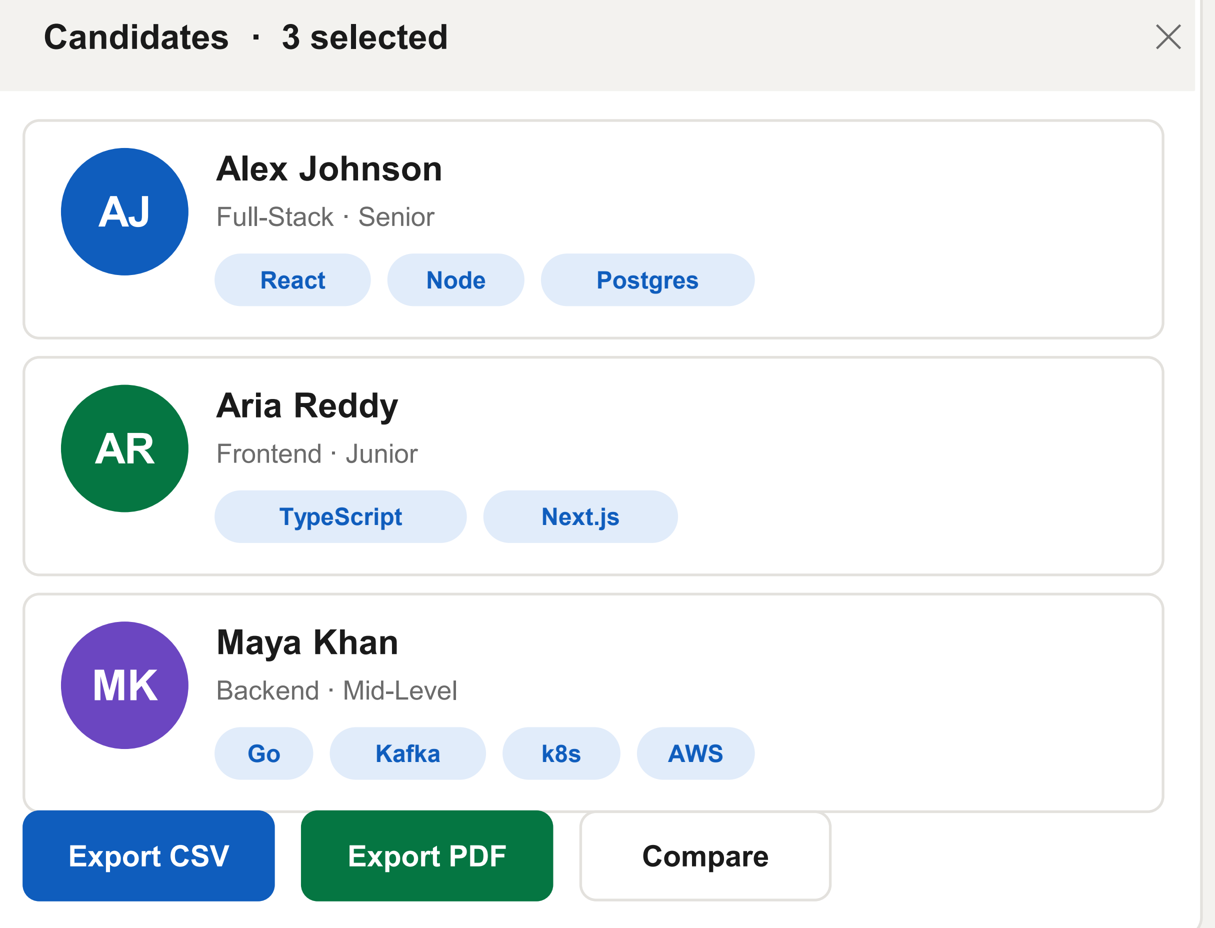
View real events and visit their website.



Recruiter Mode

A side-by-side candidate review surface designed for hiring managers.

1. **Toggle in nav**
   One click flips the app into recruiter view.
2. **Saved candidate lists**
   Group prospects by role or pipeline stage.
3. **Skill-pill filtering**
   Filter by required skills, level, and location.
4. **Compare & export**
   Side-by-side compare; export to CSV or PDF.
5. **Frontend-only feature**
   No backend changes — ships behind a UI flag for the demo.



Architecture

Single Render service, three external APIs, secrets in env vars.

Browser
React SPA · Static assets served from Render

↓ HTTPS

Render Web Service

Flask	Gunicorn	PostgreSQL

↓

Groq
EXTERNAL API
LLM for outreach & cover letters

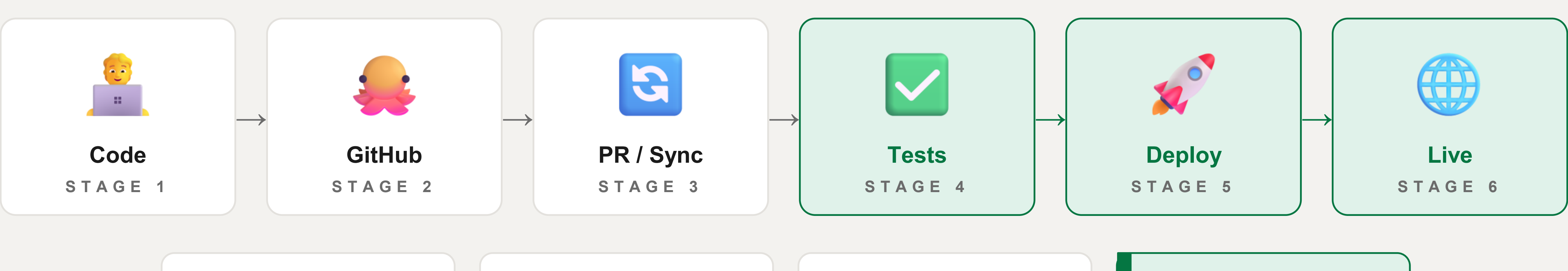
The Muse
EXTERNAL API
Job listings + filters by role

OpenStreetMap
EXTERNAL API
Geocoding for the conference map

SECRETS · `GROQ_API_KEY` · `MUSE_API_KEY` · `DATABASE_URL` · `FLASK_SECRET_KEY` — injected via Render env vars

CI/CD Pipeline

From a developer's laptop to a live URL — six stages, fully automated.



`test.yml`

Jest + pytest run in parallel matrices.

`mutation.yml`

Stryker (JS) + mutmut (PY) nightly.

`build.yml`

Build static assets + verify env schema.

`deploy.yml`

Render trigger on merge to main.

Testing Strategy

Two stacks, one shared bar: high coverage, real mutation kills, no flaky tests.

207	170	96%	83%	69%	100%
JEST TESTS	PYTEST TESTS	FRONTEND COVERAGE	BACKEND COVERAGE	MUTATION SCORE	CI KILL RATE

Frontend · Jest + Stryker

- 207 unit + integration tests covering hooks, reducers, and React components
- React Testing Library for user-flow assertions, no shallow rendering
- Stryker mutation runs nightly, blocks merges below 65%
- MSW intercepts fetch — zero live network in the test suite
- Snapshot guardrails on shared design system primitives only

Backend · pytest + mutation

- 170 pytest cases with parametrize coverage on every API route
- mutmut targeting business logic; 69% kill rate at the gate
- PostgreSQL test container per CI run for full isolation
- Contract tests pin Groq + Muse + OSM API response shapes
- Test data factories make every test self-contained

Integration Testing in CI

62 end-to-end tests run against a live Flask backend on every PR.

62	60s	≥ 80%	0
INTEGRATION TESTS	BACKEND BOOT TIMEOUT	COVERAGE GATE	BAD MERGES TO MAIN

How integration tests run

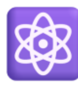
1. **Install dependencies**
   pip install -r backend/requirements.txt + requests
2. **Boot Flask backend**
   python app.py & — poll /api/jobs until 200 (≤ 60s)
3. **Run pytest suite**
   BASE_URL=http://localhost:5000/api pytest -v
4. **Report status to GitHub**
   Non-zero exit blocks merge; status check shown on PR

How failures block deployment

- Any test failure → workflow exits non-zero → red check on PR
- Branch protection blocks merging while checks are red
- Deploy workflow only runs on push to main — broken code never reaches Render
- Coverage below 80% fails CI before any deploy step is reached
- Smoke test failure on prod alerts the team via GitHub immediately

Postmortem · Successes

What worked well — and what we'd do again on the next project.



React Migration

Figma Mockup to Code in React was smooth. Figma Make turned words into designs.

CI/CD

Six-stage GitHub Actions pipeline catches lint, type, and test issues before merge.

Test Coverage

377 tests across both stacks gave us courage to ship aggressive refactors.

LLM-Assisted Dev

Claude, Copilot, Codex, and CodeRabbit handled boilerplate — humans owned every architectural call.

Feature Branches

Short-lived branches + draft PRs kept main green for the entire semester. Branch protection saved from bag merges.

Single-Service Deploy

One Render service, one config — no orchestration overhead for a 4-person team.

LLM Usage Analysis

How AI shaped development — what worked, what didn't, and how we adapted.

Claude
PRIMARY

Anthropic

Architecture decisions, full component generation, debugging, CI config, security review.

GitHub Copilot
INLINE

Microsoft

Inline autocomplete inside the editor, plus PR-level review comments.

Codex CLI
AGENTIC

OpenAI

Long-running refactor passes from the terminal — bulk renames, migrations, debugging.

CodeRabbit
REVIEW

Automated PR bot

Automated style enforcement and second-pass review on every pull request.

✓ WHERE LLMS WERE MOST EFFECTIVE

- Generating boilerplate React components — 14 pages, 12 modals
- Writing comprehensive pytest suites — 2,700+ lines of tests
- Configuring GitHub Actions workflows from scratch
- Identifying security gaps (auth flow leaks, XSS surface)
- Debugging tricky Flask ↔ React integration mismatches

⚠ CHALLENGES & PROMPT ITERATION

- Complex pages (MessagingPage, ProfilePage) needed 4–6 prompt rounds
- LLM-generated tests sometimes asserted on the wrong behavior
- CI/CD config needed iteration to handle Render cold-start edge case
- Hallucinated imports until we provided full file context up front
- Codex agents occasionally over-edited — required scoped instructions

🛠 HOW WE IMPROVED CONFUSING OUTPUTS

(1) provide the full file for context, **(2)** describe the exact failure mode, **(3)** ask for an explanation before code. *For CI bugs, attaching the raw workflow error log was the single most effective debugging prompt.*

📊 NET ASSESSMENT

Force multiplier — not a replacement for engineering judgment. We owned every architectural decision, caught hallucinated APIs, and reviewed each line. The biggest gain was speed: tasks that would have taken hours took minutes with AI assistance.

Postmortem · Failures

Where we tripped — and what we'd flag for ourselves on day one next time.

Stub Actions

Three buttons shipped without handlers— fixed mid-sprint. Mock data existed while wiring backend

Cold Starts

Render free tier sleeps after 15 min; first hit took ~30s for new visitors.

PR Iteration

Some PRs went seven rounds without merging. We needed clearer review checklists.

SQLite → Postgres

Late migration cost us a weekend; should have been on Postgres from day one.

Lessons Learned

Three takeaways we're carrying into the next build.

1

PROCESS

Ship the smallest end-to-end slice first

Our first deploy was a static page with one working button. Every other feature was added against a real, live URL — and that meant the team caught environment, secret, and config issues weeks before they would have shown up at the demo.

2

ARCHITECTURE

Treat external APIs as untrusted from day one

We assumed Muse and Groq responses would be stable, and they weren't. Wrapping every external call in a typed adapter, plus contract tests, cost us one extra day and saved roughly a week of debugging downstream.

3

QUALITY

Mutation testing exposes false coverage

Our line coverage was 96% on the frontend, but the first Stryker run killed only 41% of mutants. The gap was entire branches of business logic that no test actually asserted on. Mutation testing is now non-negotiable for the next project.

Team Process & Collaboration

How four people shipped 18 features in one semester.

245
TOTAL COMMITS

30+
MERGED PRS

5
CI/CD WORKFLOWS

P1–P6
ALL SPRINTS COMPLETED

Branching Strategy

- Feature branches off main per story or fix
- PR required to merge — no direct pushes
- Branch protection: CI checks must pass

Communication

- Messages for async coordination
- GitHub Issues for bug tracking
- PR comments for code-level discussion
- Daily check to review PRs and plan

Division of Labor

Krishi Shah
Backend, CI/CD, Debugging

Dhyani Soni
, Dev Specs, Conferences, Debugging

Saanvi Elaty
Profile, Testing, Jobs

Victor Jimenez
Figma Mockups, Documentation

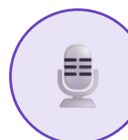
Future Work

What we'd build next, if we had another semester.

Reply Rate Analytics

FEATURED

Track outreach reply rates by template, tone, and target role. Surface which messages convert and feed insights back into the generator.



Voice Interview Prep

Live voice mock interviews with structured feedback on clarity, pacing, and STAR-method completeness.

Smart Job Matching

ML-driven ranking that learns from saves, rejections, and time-on-card to push more relevant roles to the top.

Connection Strength Scorer

Quantifies the strength of each connection by mutuals, recency, and shared context — making warm-intro paths visible.

Thank you — questions?

Link to backup screen recording:
https://drive.google.com/file/d/1AkS57aSHBQzu35NG06gR52sXLA4e0DH7/view?usp=sharing

FLOWBOARD

AI-Powered Workflow Management

CS 485 — AI-Assisted Software Engineering | Spring 2026

TEAM: GROUP 11

Luke Hill

Swechcha Ambati

Vishesh Raju

New Jersey Institute of Technology

May 2026 | Final Demo & Postmortem

AGENDA

01 **App Overview**
Problem statement, target users, and key value propositions

02 **Live Demo**
Walking through all three core user stories and real-time features

03 **CI/CD & Testing**
Automated pipelines, GitHub Actions workflows, and test coverage

04 **Deployment**
AWS infrastructure, Amplify, Lambda, and deployment process

05 **Postmortem**
Reflections on successes, challenges, and lessons learned

06 **Future Work**
Planned features and roadmap for continued development

01

CHAPTER ONE

APP OVERVIEW

Problem, Solution & Target Users

Problem Statement & Solution

THE PROBLEM

- Project managers spend **hours manually setting up** Kanban boards for new projects
- Meeting decisions and action items **get lost in unstructured notes**
- Team collaboration lacks **structured change review** and approval workflows
- No integrated solution connects **planning, meetings, and execution**

OUR SOLUTION

TrelloPlus — an intelligent, AI-powered project management platform

- **AI-generated boards** from project descriptions
- **Meeting transcript analysis** with structured summaries
- **Change review & approval** workflows with voting
- **Real-time collaboration** via WebSocket

TARGET USERS

Project Managers who need rapid project setup | **Meeting Facilitators** who capture decisions | **Team Members** in agile software teams

Roles: MANAGER (project creation) | MEMBER (card editing) | VIEWER (read-only)

KEY VALUE PROPOSITIONS

1. Save hours with AI board generation **2.** Never lose meeting insights **3.** Structured change governance **4.** Real-time team collaboration

02

CHAPTER TWO

LIVE DEMO

Walking Through Core User Stories

Demo

US1

AI Board Generation

Project manager describes a project

AI analyzes and generates Kanban board with stages and starter tasks

Full CRUD operations supported

US2

Meeting Summary

Facilitator captures meeting notes

AI extracts action items, decisions, and proposed changes

Structured approval checklist

US3

Change Review & Approval

Changes from meetings flow to review

Team votes on approval with consensus engine

Approved changes auto-applied to board

Links:
Frontend
Backend
Transcript

03

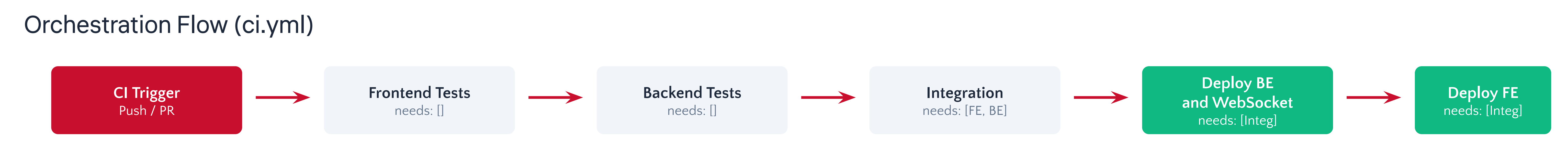
CHAPTER THREE

CI/CD & TESTING

Automated Pipelines & Quality Assurance

Testing: GitHub Actions Workflows

Workflow File	Purpose	Details
ci.yml	Main Orchestrator	Triggers frontend-tests, backend-tests, integration-tests, deploy-backend, deploy-frontend sequentially
run-frontend-tests.yml	Frontend Unit Tests	Jest + React Testing Library with targeted coverage for KanbanBoard, MeetingSummary, MeetingChanges
run-backend-tests.yml	Backend Unit Tests	Maven/JUnit 5 + Mockito with JaCoCo coverage for ProjectService, SummaryService, ChangeApplicationService
run-integration-tests.yml	E2E Integration Tests	Full end-to-end API testing validating frontend-backend contract
deploy-aws-lambda.yml	Backend Deployment	Packages Spring Boot JAR with Maven Shade, uploads to S3, deploys to Lambda
deploy-aws-lambda.yml	WebSocket Lambda Deployment	Reuses the shaded Spring Boot backend JAR from S3, updates the WebSocket Lambda
deploy-aws-amplify.yml	Frontend Deployment	npm build, uploads artifacts, triggers Amplify start-job via AWS CLI



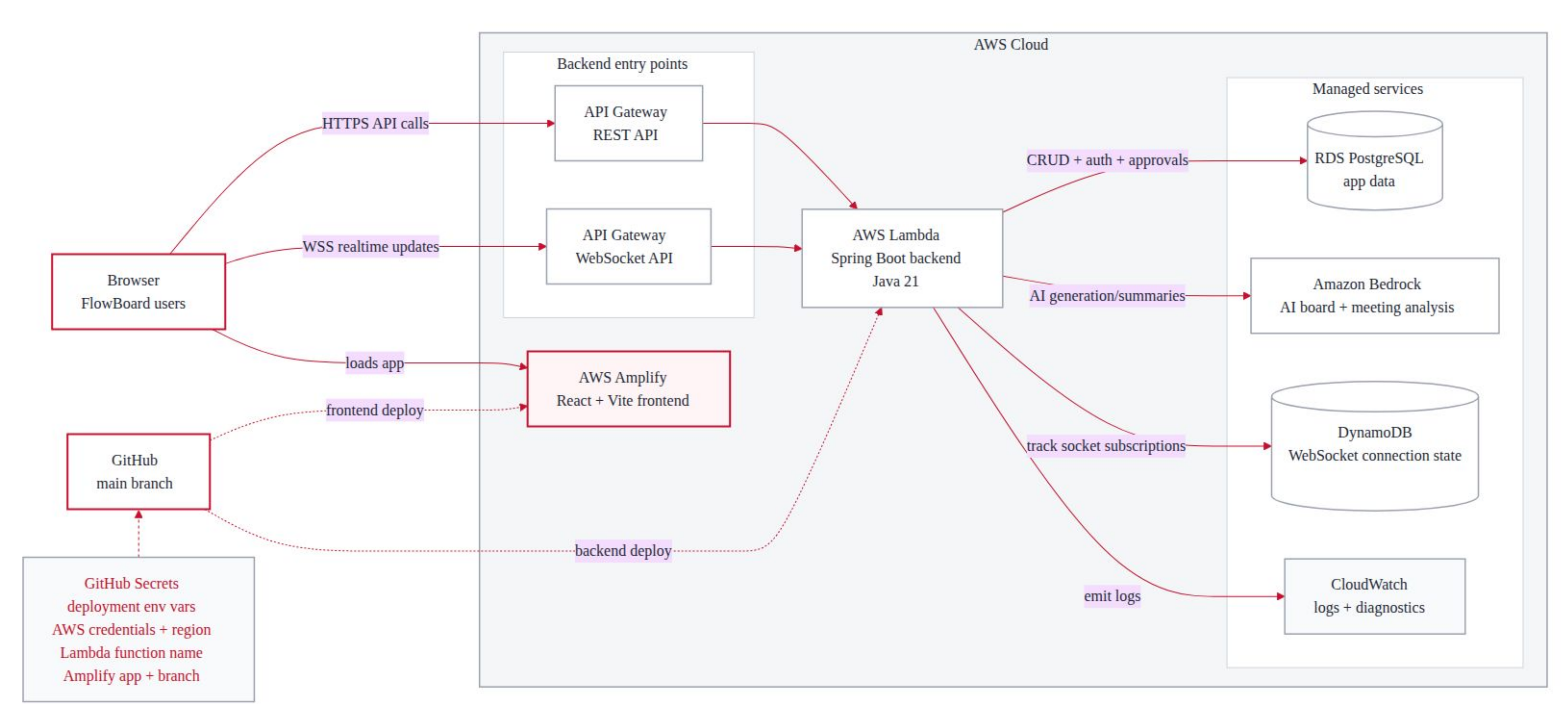
04

CHAPTER FOUR

DEPLOYMENT

AWS Infrastructure & Deployment Process

AWS Deployment Architecture



05

CHAPTER FIVE

POSTMORTEM

Reflections on Our Development Journey

What Went Well

Workload distribution

Other than deployment, throughout the development, testing, the workload was distributed well.

Realistic goals

We aimed to implement the three user stories fully, and we were able to do that.

Diverse LLM tools brought diverse ideas

All of us used used different tools, so we always had a pool of outputs and ideas to choose from.

Real Time Board changes works!

Took one of the team members (and the LLM) two whole days to make it work. And it works!

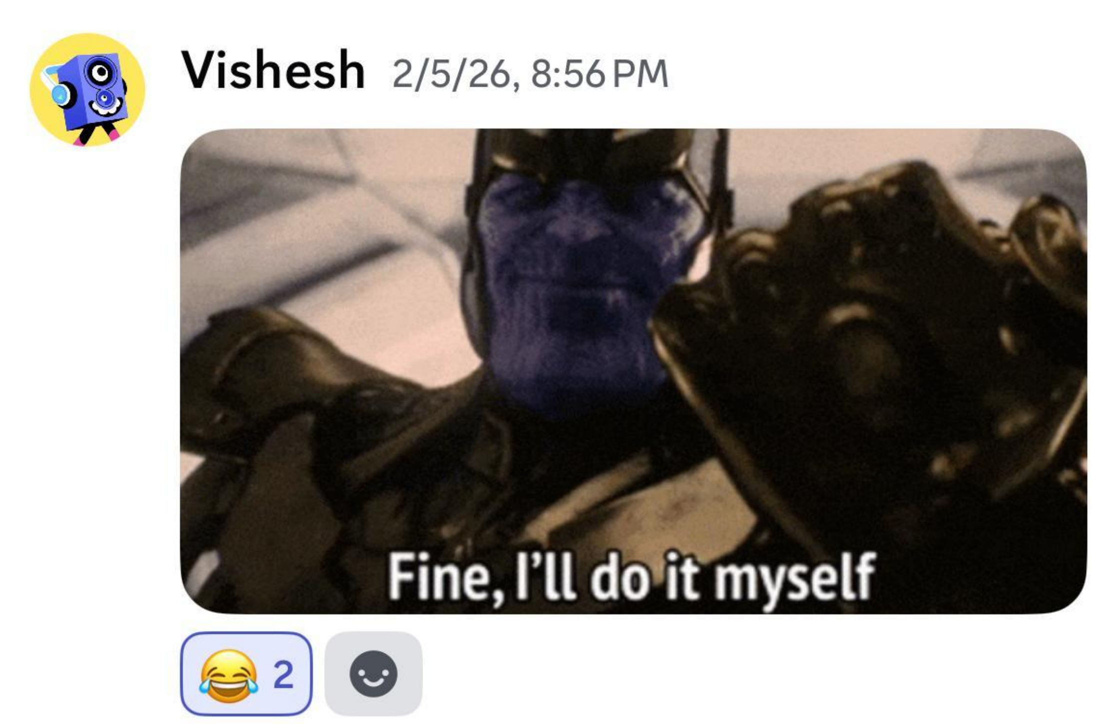
Challenges & Surprises

Technology Fit

Following the LLM's initial stack choices led to extra integration and AWS deployment complexity

Slow Deploy Feedback Loop

Small bugs often required a full CI/CD cycle before the team could verify fixes in the deployed app



AWS/GitHub Permission Friction

Repo ownership and missing organization setup made some GitHub and deployment permissions harder to manage

Single Deployment Owner

Only one person had full deployment access, concentrating AWS fixes and release work on one teammate

Sprint Planning Gaps

Class-time meetings were less effective when sprint specs were not reviewed early enough to plan together

Surprise: Deployment Issue quite late into the sprint

After adding our final feature, we found out that the frontend changes were not getting deployed properly, so we had to switch to our back-up deployment.

Lessons Learned

1 Share Deployment Ownership Early

More than one teammate should have AWS/GitHub release access so deployment work and debugging are distributed.

2 Plan Sprints Before Coding

Sprint meetings only help when the next specifications are reviewed beforehand and decisions are ready to make.

3 Keep Context Current

Stale docs and tests confused Copilot, so generated code quality depended on cleaning up old context.

4 Choose the Stack for Scope

Using technologies because an LLM suggested them created avoidable deployment cost for a smaller project.

5 Test the Deployed Path, and improve the testing

Local success was not enough; WebSocket behavior, AWS config, and runtime endpoints had to be validated after deployment.

06

CHAPTER SIX

FUTURE WORK

What's Next for Flowboard

Future Features & Roadmap

Meetings hosted on FlowBoard

Meeting transcripts can be generated in the application itself, and used for generating action items

Email Notifications

Send updates for approvals, assignments, meeting summaries, and important board changes

Transcript Assignee Updates

Allow meeting transcripts to propose changes to task assignees, not just cards or checklist items

Approval Reminders

Nudge reviewers when meeting-derived changes are waiting for approval

Delete Account

Let users permanently remove their profile and associated account data from Flowboard

Progress monitoring

Monitor project item progress and passively adjust project progression and blockers.

THANK YOU

FlowBoard — AI-Powered Workflow Management

QUESTIONS?

TEAM

Luke Hill | Swechcha Ambati | Vishesh Raju

CS 485 — AI-Assisted Software Engineering | New Jersey Institute of Technology | Spring 2026

github.com/Luke-Hill-325/CS698_Project_TrelloPlus

Group 9 Presents

Chatlite

Intelligent Question Detection and Discussion Tracking for Small Teams and Classes.

Team Members: Eric Perez

Rethinking Modern Chat

The Problem

In fast-moving channels, critical questions get buried under message volume. Discussions lose context, and team members miss actionable items without constant monitoring.

The Solution

Chatlite surfaces questions and active discussions automatically. It transforms a chaotic stream into an organized, actionable dashboard for small teams and classes.

Target Users

As more and more classes are switching to discord for classes, it can be hard to keep track. Students are eager to know more about deadlines and details so having a question detection software can help professors who can't spend all day on the discord server. Same goes for students and the discussion detection software. This is just one of the many real world implications of this app.

Core Functionality

Question Detection

Heuristic-based detection of unanswered questions in real-time channel history.

Discussion Detection

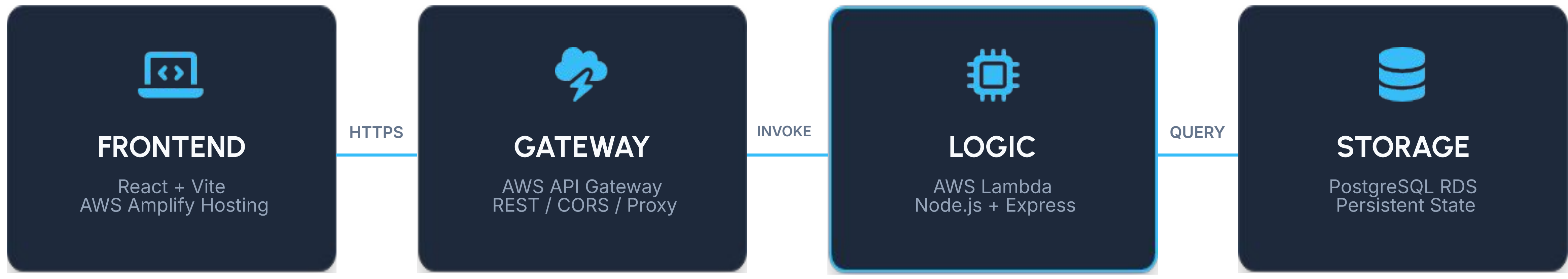
Multiple questions and users required to allow threads to go unmissed.

Simulated Users

Automated responders with typing indicators to test these features.

Systems Architecture Blueprint

End-to-end serverless infrastructure for the Chatlite platform.



Live Demo Roadmap

Primary Stories

Open live frontend on Amplify. Show channel list & seeded history.

Question Detect

Post a question; verify it appears in the "Unanswered" list instantly.

Full Loop

Reply to question; observe status change and notification trigger.

System Proof

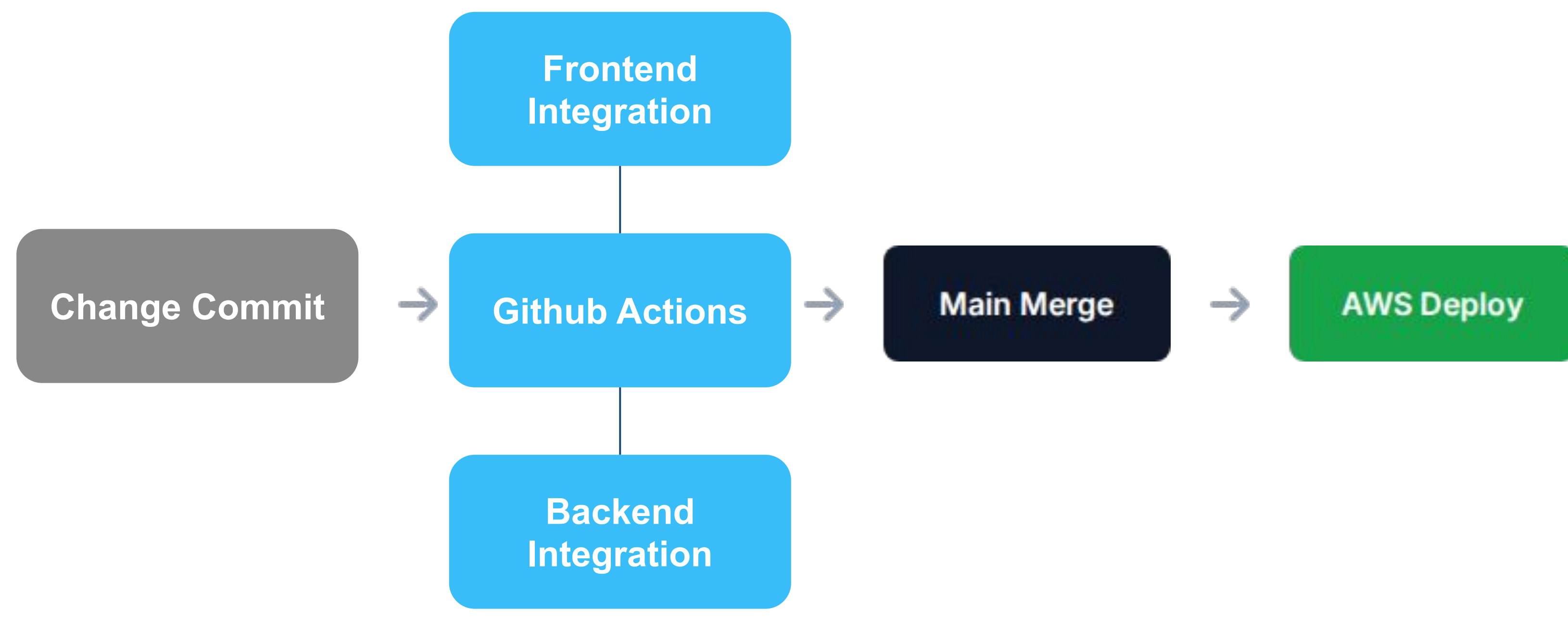
Verify data persistence via refresh and API health checks.

Simulated Users

turn on simulated users to show thread detection

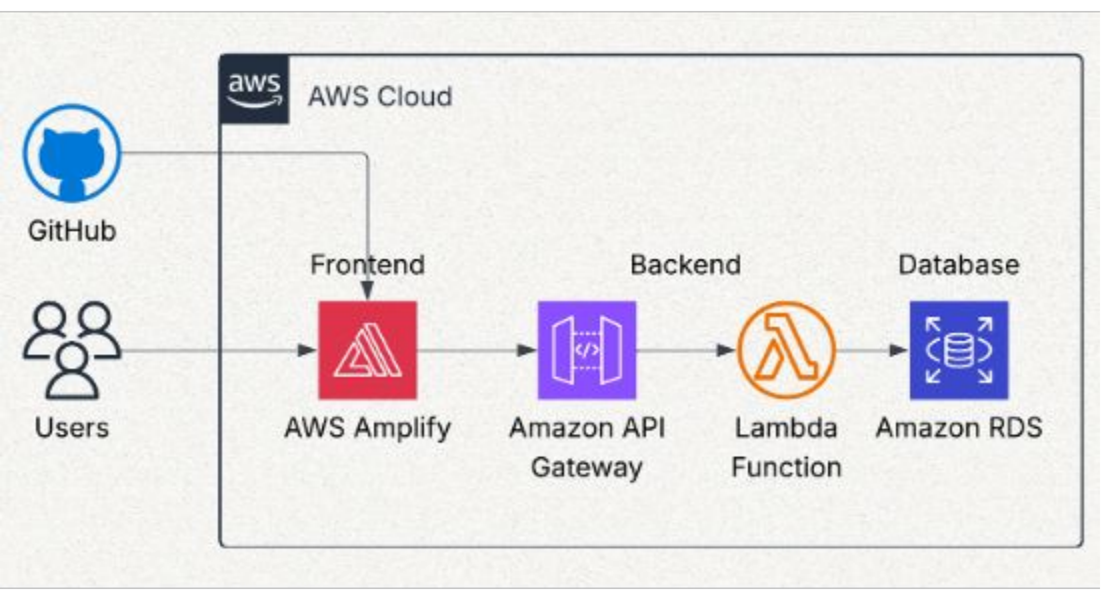
Deployed Link: https://main.d2iu69duubq7ln.amplifyapp.com

CI/CD Pipeline

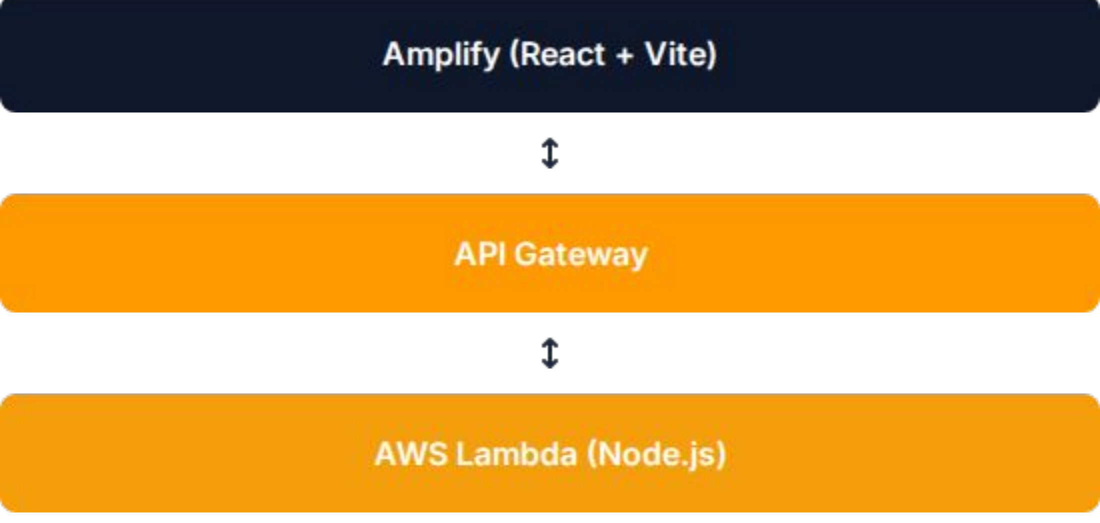


Automated gates ensure that every commit is validated via Unit and Integration tests before reaching the Lambda/Amplify production environment.

AWS Infrastructure



Serverless Delivery



Scalable, pay-per-use architecture that handles frontend hosting and backend processing end-to-end.

What Went Well

- **Architectural Boundaries:** Clean separation between FE/BE allowed for parallel development and faster feature builds.
- **Frontend development:** Clean UI with functionality to keep discords friendly design with Teams more professional elements
- **Test-Driven Reliability:** Automated regression suites saved the demo multiple times during the final polish phase.

Challenges & Lessons

Deployment

Environment configuration took significantly longer than logic implementation. Matching local variables with Lambda-secret environments required multiple iterations.

Frontend Backend Wiring

As the backend was developed, the wiring became more and more faulty between the static frontend and the dynamic backend. Many bugs arose form this like loading issues and delays.

Deployment Lesson

Build the deployment and integration checks *earlier*. Cloud-native issues don't show up in local Jest suites—verify the live endpoint as soon as the first route is ready.

wiring lesson

Its important to improve both ends at the same time as the project grows more and more. Frontend developed months ago should be updated as the backend is still being worked on.

The Roadmap Ahead

Advanced Auth

Account creation with advanced authorization for security

ML-Driven Detection

Using Natural Language Processing for higher accuracy discussion detection.

LLM Responses

LLM generated responses for a more natural flow of conversation.

GlossaDocs

A multilingual-first writing app

Tommy Kolb

γλῶσσα • tongue • language

glossadocs.com

The Typing Problem

Overview

Personal problem → personal project

Keep language choice, typing help, and the document in the same workflow.

Target users: multilingual learners, writers, and anyone working across scripts

From Idea to App

Demo

1. First version

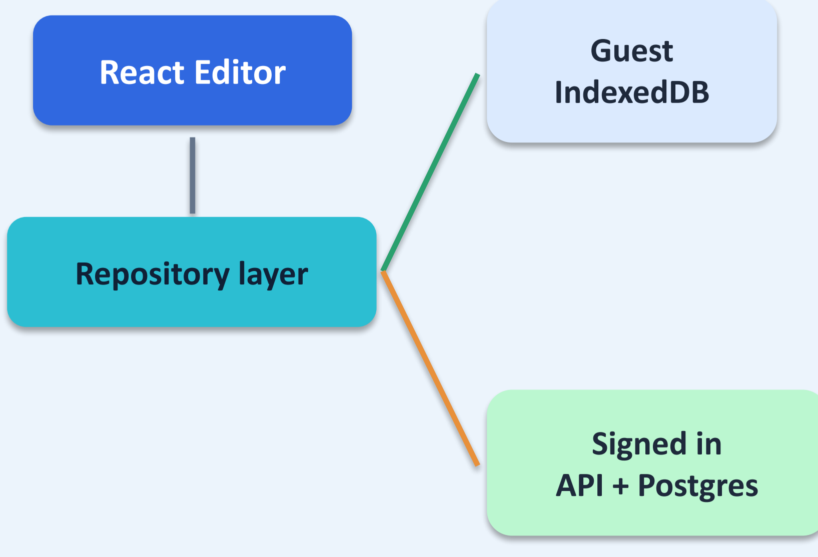
English, German, Russian frontend prototype.

2. Real accounts

Backend, auth, documents, settings, persistence.

3. More languages

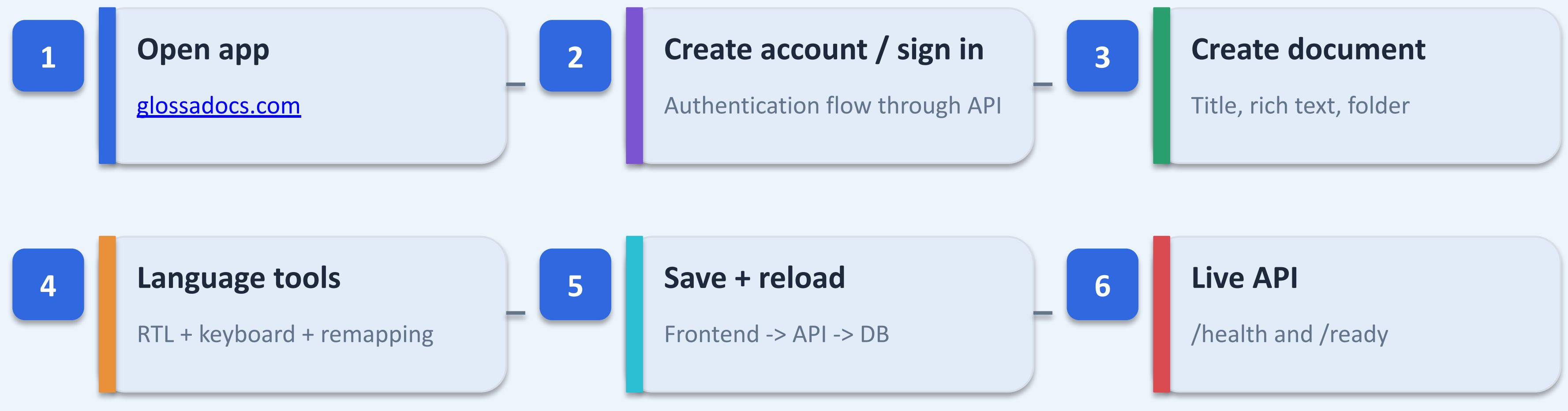
The hard part: every script has different typing rules.



Demo Link

Live Demo Roadmap

Demo



Launch App

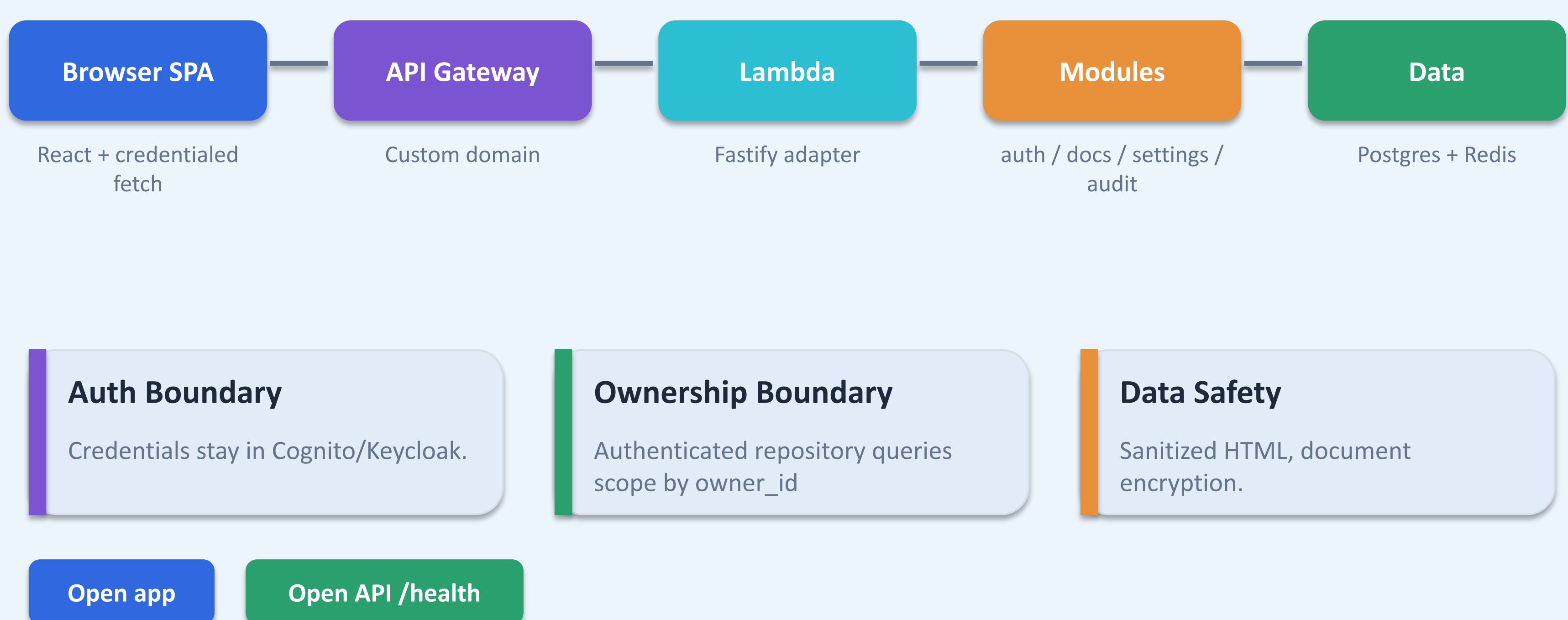
Show /health

Show /ready

Frontend → Backend Integration

Demo

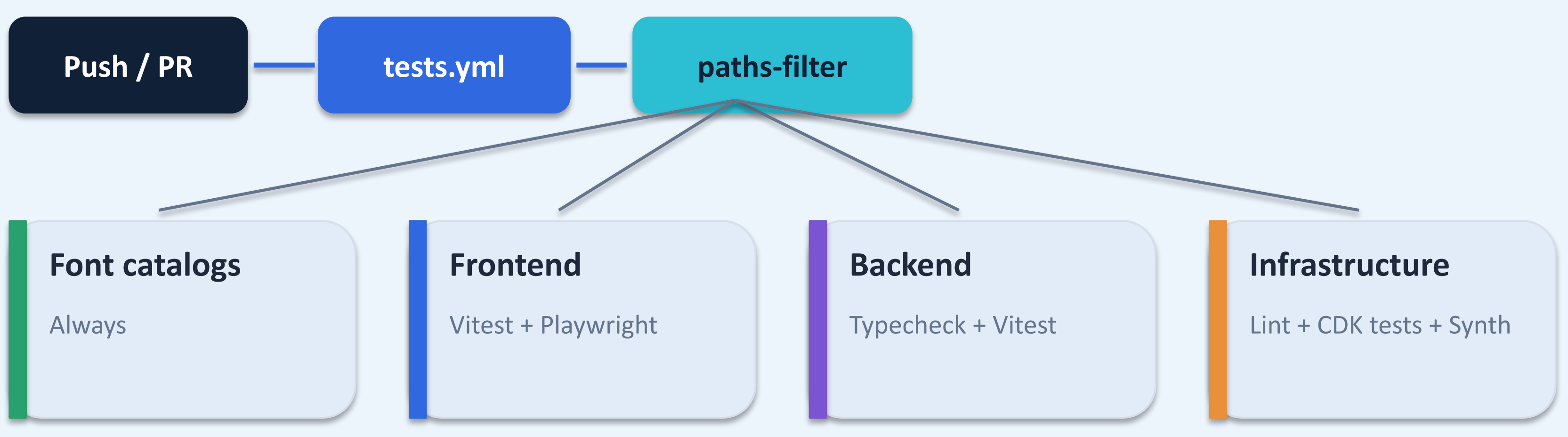
Same API contract in local Docker and AWS Lambda



CI Overview

Testing

Path-filtered test workflows keep feedback relevant



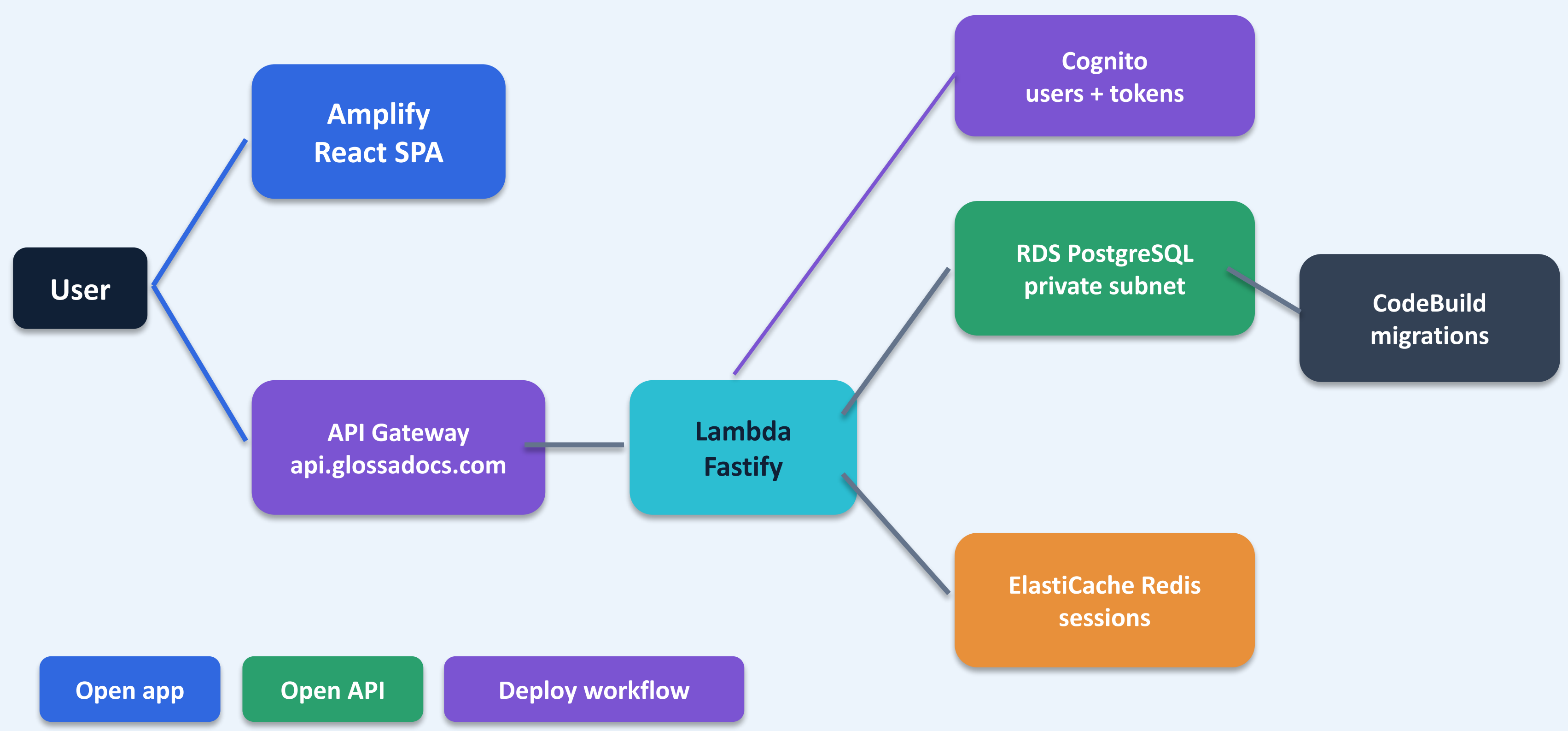
Failing required gates block merge

Open tests workflow

All Actions

AWS Deployment Architecture

Deployment



CD Pipeline

Quality gates — Test, Typecheck, Synth

Deploy infra — CDK with GitHub OIDC

Migrate DB — CodeBuild inside VPC

Deploy app — Lambda/API + Amplify release

Smoke tests — API health checks

Integration tests

Playwright against production URLs with a dedicated E2E user.

Secrets

GitHub vars + secrets; AWS role via OIDC.

Failure behavior

Any failed gate stops the next stage.

Deploy workflow | Deployed tests | Actions

Postmortem: What Worked

A Real Problem

The initial motivation stayed clear: writing across languages should feel less fragmented.

Learning Through Creating

Made me realize how much I don’t know about different languages, scripts, fonts, and input methods.

Deployable Architecture

More than a prototype: accounts, API, database, AWS deployment, and CI gates.

Modular Backend

Fastify modules kept auth, documents, settings, and audit separate without microservices.

LLM Leverage

Useful for docs, AWS wiring, and iteration, as long as CI and review stayed in the loop.

It’s Real! It Works!

Satisfaction from seeing a real, tenable project

Postmortem: What Surprised Me

Languages, Not Just Labels

Every new language raised questions about fonts, direction, keyboard behavior, and UX limits.

Authentication Edge Cases

Cognito policies, email verification, password reset, and local Keycloak parity took more care than expected.

AWS Setup

RDS TLS CA bundles, Lambda networking, Redis sessions, and migration order created deployment work.

Feature Branches

Helpful for isolation, but cross-cutting changes still need to stay reviewably small.

About Code Review…

Fast generation overrode verification; docs and tests became the guardrails - I didn’t review code.

Surprisingly Costly

Thank goodness for AWS credits

Close

Where GlossaDocs Goes Next

Lessons Learned

1. Design the deployment path earlier.
2. Local development can only take you so far
3. Write tests for cross-layer contracts first.
4. Heavy documentation is necessary for successful LLM development

Future Work

- **Full IME integrations** — Better ranking / segmentation and richer composition UX
- **Language-Adaptive Behavior** — Spellcheck? What spellcheck?
- **Version History** — Recover earlier document states
- **Collaboration** — Shared folders and real-time editing
- **More Emails** — Cognito has a free 50 emails per day limit - will that always be enough?

Try It For Yourself!

The multilingual-first writing app

γλῶσσα • tongue • language

Problems? Email glossadocs@gmail.com or raise a GitHub issue

Backup Links Safety Net

Links

Application	glossadocs.com
API health	api.glossadocs.com/health
API readiness	api.glossadocs.com/ready
Repository	github.com/TommyKolb/GlossaDocs
All Actions	github.com/TommyKolb/GlossaDocs/actions
Deploy workflow	github.com/TommyKolb/GlossaDocs/actions/workflows/deploy-production.yml
Tests workflow	github.com/TommyKolb/GlossaDocs/actions/workflows/tests.yml
Deployed integration tests	github.com/TommyKolb/GlossaDocs/actions/workflows/deployed-integration-tests.yml