https://docs.google.com/presentation/d/1XpOZjnhIT
enqokRUHIYQuXeWx8jpMSvr/edit?slide=id.p1#slid
e=id.p1

ANAPHOR

The Future of Conversational Intelligence

Elvis Valcarcel | Nafisa Ahmed | Salma Ghazi | James Mullins | Ashraf Aldekaim

Demo Block (6 mins)

1. APP OVERVIEW

Problem Statement

Remote teams face **"Context Bankruptcy"** in high-volume chat environments. Critical decisions are buried, making it impossible to catch up quickly without manual searching.

Target User

Project Teams: Needing standup recaps and quick context.
Admins: Managing multi-space noise and notification fatigue.
Casual Users: Catching up instantly via "What You Missed".

The Anaphor Solution

Anaphor leverages LLM summarization to bridge the gap between real-time noise and actionable intelligence.

Automated Previews: "What You Missed" summaries.
Manual Summary: User-triggered timeframe synthesis.
Organized Spaces: High-performance server hierarchy.

Demo Block (6 mins)

2. DEMO ROADMAP

01

Onboarding

Secure login and registration powered by apiService.ts and JWT persistence.

02

The Space

Messaging, reaction handling, and complex state via AppContext.

03

AI Logic

Execution of manual AI summaries and automated unread message processing.

Demo Link

3. CI/CD OVERVIEW

Automated Workflow

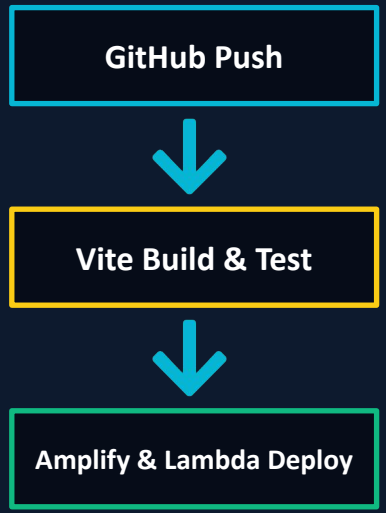
Every merge to **main** triggers a verified CI/CD deployment gate including the following tests:

- Deploy backend to AWS Lambda
- Deploy frontend via AWS Amplify
- Deploy websocket Lambda to AWS
- Lint
- Run backend tests
- Run Frontend Tests
- Run integration tests

Build: Automated Vite production bundling.

Verify: Vitest execution for contract and deployed health.

Sync: CD deployment to AWS Lambda and Amplify.

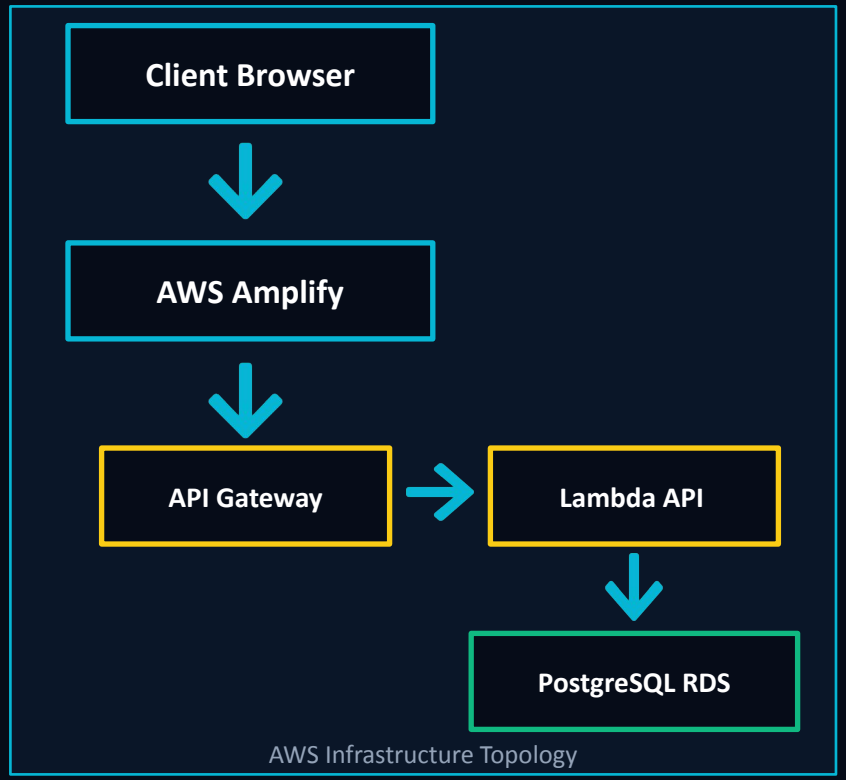


Automated Pipeline Verification

4. DEPLOYMENT OVERVIEW

AWS Serverless Stack

Frontend: Live Amplify App
API Layer: API Gateway & Lambda REST
Data: PostgreSQL on AWS RDS



AWS Infrastructure Topology

Postmortem Block (4 mins)

5. POSTMORTEM

Successes

AI Intelligence: Seamlessly integrated "What You Missed" and Manual Summaries as modular, scalable services.

Intuitive Interface over Feature Clutter: Instead of copying the overly complex layouts of existing chat apps, we kept our design minimal. This eliminated the usual learning curve and allowed users to test the core features immediately. We also built our UI with mobile in mind.

Struggles

Web Sockets: We were able to implement locally, but we couldn't configure AWS to cooperate with our codebase. We attribute this failure to it's introduction during the runup to P7.

Deployment: While Amplify frontend was smooth to deploy, configuring secrets, messing with VPC, and environment variables to all align with our backend APIs was more challenging, even with LLM help.

Lessons

Frontend Development: Our frontend development went through many iterations and the LLM struggled with UI bugs much more often than backend bugs.

Early CI/CD: Having an integrated testings and deployment pipeline earlier on would have helped ensure stability from the start of our project.

Postmortem Block (4 mins)

6. FUTURE WORK

WebRTC Audio

Adding low-latency audio rooms within Spaces for seamless standups.

Video Streaming

Implementing low-latency live streaming and screen sharing capabilities within Spaces.

RBAC Roles

Hierarchical permissions (Admin, Moderator) for community management.

QUESTIONS?

rD

rDebate

Reddit AI Debate Analyzer

A full-stack debate platform with AI-powered argument analysis.

Engy Masoud

React | Node · TS · Express | PostgreSQL | AWS Lambda · S3 · CloudFront

Live: https://d78kjfcaywen5.cloudfront.net

SECTION 01

App Overview

Problem

Online debates devolve into noise: weak arguments, missing evidence, and personal attacks. Readers lack a fast way to evaluate the reasoning quality of long comment threads.

Solution

A Reddit-style platform with 8 seeded subreddits, threaded comments, votes, emoji reactions, polls, debate stances, and deep-link sharing — where **every comment** can be sent through an LLM that returns a **Reasoning Summary** and **live Writing Feedback** in the composer.

Target Users

Debate Enthusiasts

Want quality threads & instant argument breakdowns

Students & Educators

Practice critical thinking and rhetoric with AI feedback

Subreddit Moderators

Surface high-signal discussion, flag weak reasoning

Casual Readers

Skim long debates via AI summaries instead of every reply

SECTION 02 · LIVE WALKTHROUGH

Demo Roadmap

https://youtu.be/CaAUNhcqWcY

01

Sign in (seed account)

Username & password shown on auth page

02

Browse 8 subreddits

Tech, Health, Games, Food, Music …

03

View AI Reasoning Summary

Primary claim · Evidence · Coherence

04

Live Writing Feedback

Unsupported claim · Weak evidence flags

05

Vote, react, pick a stance

Up/down · emoji · for/against

06

Polls & comment reporting

Inline polls · 4 report categories

07

Share → deep-link in new tab

`/post/:id` resolves cold

08

Notifications bell

Unread count · mark all read

Sample Thread: Should pineapples go on pizza?

Content

Pineapple does not belong on pizza. Pizza is fundamentally a savory dish built around the balance of dough, tomato sauce, cheese, and salty or umami-rich toppings, and adding a sweet fruit disrupts that core structure. While pineapple brings acidity and sweetness, those flavors tend to clash rather than complement when combined with melted cheese and seasoned meats, often overpowering the more subtle elements of the pizza. It can also affect texture in an unappealing way, releasing moisture that makes the crust soggy instead of crisp. Even though people are free to experiment with food, not every combination improves the dish, and pineapple on pizza feels more like a novelty than a meaningful culinary enhancement.

Image URL

https://itsavegworldafterall.com/wp-content/uploads/2020/07/Pineapple-Jalapeno-Pizza-5.jpg

High Level Response

Pineapple belongs on pizza because it improves flavor balance through a proven sweet–savory contrast. The sweetness and acidity of pineapple cut through fatty, salty toppings like ham and cheese, creating a more balanced bite instead of a heavy, one-note flavor. This isn't unusual in cooking—sweet-and-savory pairings like pork with fruit or honey-based sauces are widely used across global cuisines.There's also real-world evidence that people enjoy it: pineapple consistently ranks among the top 10–15 pizza toppings in major consumer surveys in North America, and "Hawaiian pizza" has remained on restaurant menus worldwide since the 1960s, showing long-term demand rather than a passing trend. Even critics can't deny its popularity—millions of pizzas with pineapple are sold every year globally.The texture concern is often overstated; when baked properly, pineapple caramelizes and loses excess moisture, similar to roasted vegetables used on pizza. Overall, its continued popularity and strong flavor chemistry make pineapple a valid, well-supported pizza topping rather than a mistake.

Low Level Response

Pineapple should not go on pizza because it ruins what pizza is supposed to taste like. Pizza is meant to be savory with cheese, sauce, and meat, and adding fruit makes it taste weird and too sweet. The texture is also bad because pineapple gets watery when baked and makes the crust soggy. Most people who like real pizza don't think fruit belongs on it, and it feels more like a joke topping than something serious. Pizza has worked fine for a long time without pineapple, so there's no reason to change it.

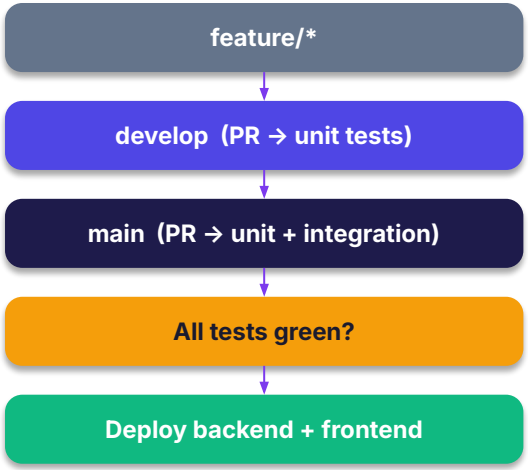
SECTION 03 · GITHUB ACTIONS

CI/CD Overview

5 GitHub Actions Workflows

- **`run-frontend-tests.yml`**
  PR/push → develop|main · Jest + RTL
- **`run-backend-tests.yml`**
  PR/push → develop|main · Jest + ts-jest
- **`run-integration-tests.yml`**
  PR/push → main · 14 tests vs deployed API
- **`deploy-backend.yml`**
  push → main (backend/**) · Build & deploy Lambda
- **`deploy-frontend.yml`**
  push → main (frontend/**) · S3 sync + CloudFront invalidation

Branching & Pipeline



Failing tests block merge → no broken deploy.

SECTION 03A

How GitHub Actions Runs Our Tests

Frontend Unit Tests

- Trigger: PR/push to develop|main
- Runs on ubuntu-latest
- actions/setup-node@v4 (Node 18)
- **npm ci → npm test (Jest + jsdom)**
- Coverage uploaded as artifact
- Status check required to merge

Backend Unit Tests

- Trigger: PR/push to develop|main
- **ts-jest compiles TypeScript**
- Pure unit tests — no real DB
- pg & jwt mocked at module level
- Mutation tests (Stryker) on demand
- Status check required to merge

Integration Tests

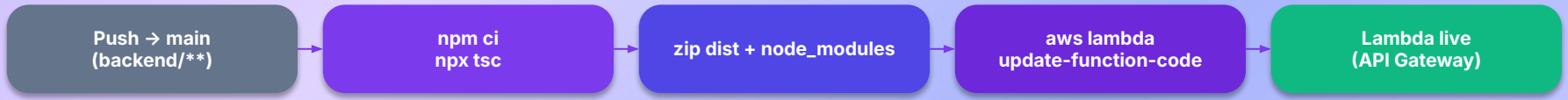
- Trigger: PR/push to main only
- Hits live API Gateway / Lambda
- ensureTestSubreddit before run
- 14 tests across auth, posts, votes
- Failure → deploy job is skipped
- Runs against deployed backend

Strategy: fast unit tests on every PR · slow, costly integration tests gated on main.

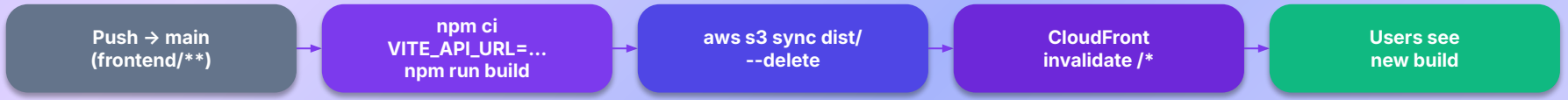
SECTION 03B · LAMBDA + S3/CLOUDFRONT

Continuous Deployment

Backend → AWS Lambda



Frontend → S3 + CloudFront



Path filters + concurrency

Each deploy workflow uses a `paths:` filter so a frontend-only commit doesn't redeploy the Lambda (and vice-versa). A `concurrency` group on the branch cancels in-flight runs when a new commit lands, so only the latest main ever ships.

SECTION 03C

Secrets & Permissions

GitHub Repository Secrets

AWS_ACCESS_KEY_ID	IAM user for CI/CD
AWS_SECRET_ACCESS_KEY	IAM user secret
LAMBDA_FUNCTION_NAME	Target Lambda for backend deploy
S3_BUCKET_NAME	Frontend static-site bucket
CLOUDFRONT_DISTRIBUTION_ID	For cache invalidation
API_GATEWAY_URL	Used by integration tests + FE build

Permission Model

Storage	Secrets live only in GitHub Actions secrets store — never committed, never echoed in logs (`***` masked).
Access	Workflows reference them via `${{ secrets.NAME }}`; only maintainers can read or rotate.
IAM Scope	A dedicated `github-deployer` IAM user has Lambda, S3, and CloudFront access only — no console login.
Runtime Secrets	DB credentials & JWT_SECRET are Lambda environment variables, not in the repo.
Network Isolation	Lambda and RDS share a private VPC/SG so the database is only reachable from the function — never from the public internet.

SECTION 03D · FAILURE SEMANTICS

Integration Tests in CI

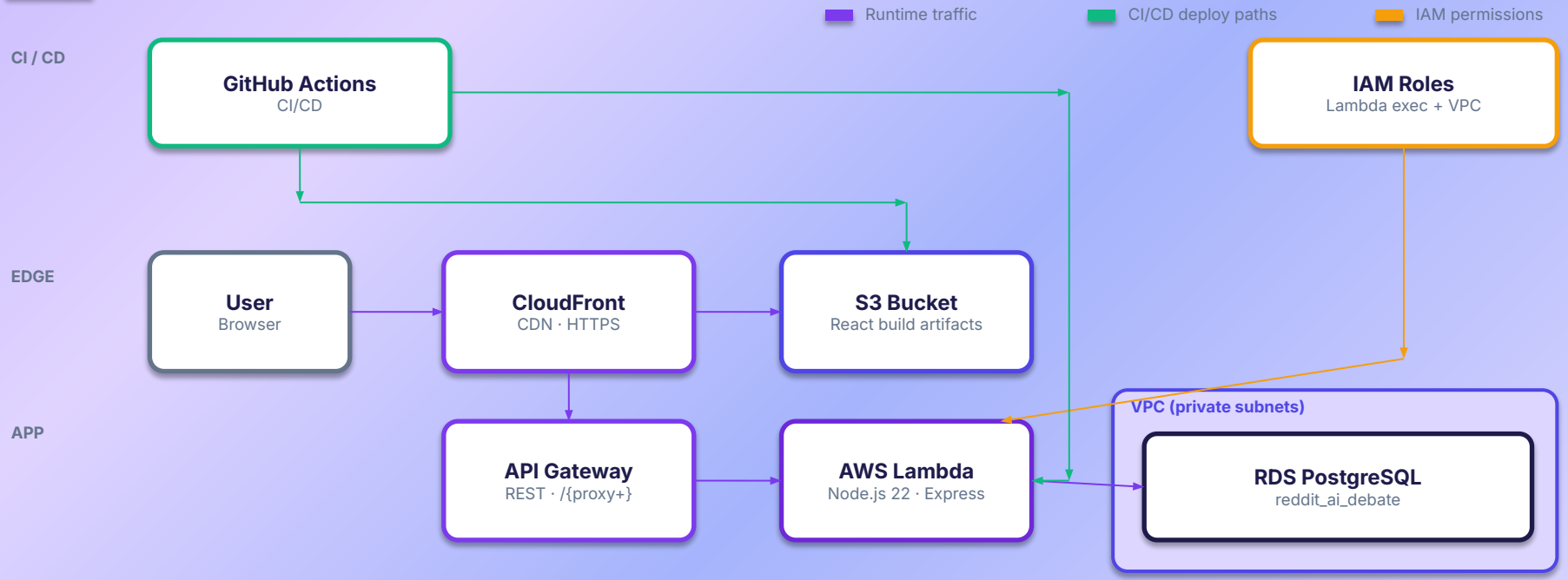
Decision flow on every push to main

If integration tests fail …

- Deploy job's `needs:` dependency is unmet → S3 sync + Lambda update never run
- Last successful deploy stays live — users keep the working version
- PR shows red status; merge button disabled until tests pass or are re-run
- 1 spec (Google OAuth) is intentionally skipped — mirrors prod, where the route returns 501 until a Google client ID is configured

SECTION 04

AWS Deployment Architecture



SECTION 05A · WHAT WENT WELL (SOLO BUILD)

Postmortem · Successes

Development	Clear DS/US specs in /docs/dev-specs broke the build into small, independent backend modules I could ship one at a time.
Testing	Heavy mocking of `pg` and `jwt` kept backend unit tests fast and deterministic; mutation testing (Stryker) caught silently-passing assertions.
Deployment	The Lambda + S3 + CloudFront stack deployed in minutes once IAM was right; rollbacks are just `aws lambda update-function-code` to a prior zip.
Workflow	Solo feature-branch + PR-to-self with required status checks kept `main` always deployable; integration tests caught several broken pushes before they shipped.

SECTION 05B · WHAT BROKE

Postmortem · Failures & Surprises

CORS config	CloudFront → API Gateway cookie/CORS handshake took an afternoon to dial in. Fix: explicit `FRONTEND_URL` env var and matching API Gateway CORS rules.
Cold starts	Initial Lambda cold start plus DB connection setup caused flaky early integration runs. Fix: lazy `pg.Pool`, single connection per invocation, retries in test client.
Share-link bug	Polls disappeared when a post was opened from a list because the list payload merged over the fetched detail (see SHARE_LINK_FIX.md). Fix: preserve the fetched poll on merge.
Test seed data	Integration tests assumed a `test` subreddit existed on fresh DBs. Fix: `ensureTestSubreddit.js` runs before the suite.
LLM drift	Generated code occasionally referenced stale package APIs. Fix: pin versions, treat every LLM diff as a PR to review.
Google config gap	Google Sign-In button shipped in UI; Lambda env var unset, so route returns 501. Code ready, configuration is the only missing step — noted on Slide 14.

SECTION 05D

Lessons Learned

If we started over...

- Wire CI/CD on day 1, not week 3
- Stand up the Lambda + RDS skeleton before writing any feature code
- Pick one API client (axios + interceptors) up-front and stick with it
- Adopt path-filtered deploy workflows from the first commit
- Write the integration test harness alongside the first endpoint
- Use IaC (CDK/Terraform) instead of Console clicks

Best practices to keep

- Required status checks even on a solo repo — your future self thanks you
- Paste errors instead of summarizing
- Path filters so unrelated deploys don't trigger
- Perfect the specs
- Mock at the module boundary; integration tests for the rest
- Write a short fix-forward note per major incident

SECTION 06 · WHAT'S NEXT

Future Work

1. **Activate Google Sign-In**
   Route, DB column, and UI button all in production today. Add `GOOGLE_CLIENT_ID` to the Lambda env to switch on the third sign-in path — zero code changes.

2. **Real-time WebSockets**
   Live comment updates, typing indicators, and instant notifications via API Gateway WebSockets. Backend skeleton exists; needs API Gateway side and client reconnect.

3. **Debate Score & Leaderboards**
   Aggregate the existing per-comment reasoning scores into a per-user 'argument quality' rating with weekly leaderboards per subreddit.

4. **Side-by-side Argument Diff**
   Pick two opposing comments and have the LLM produce a structured pro/con table with shared assumptions — ideal pairing for debate-side posts.

5. **Moderation Copilot**
   Report flow already ships with 4 categories; add LLM auto-triage to flag fallacies, toxicity, and low-effort comments with one-click actions.

6. **Mobile PWA + Push**
   Installable PWA with push notifications so debates follow you off the desktop.

rD

Thank you!

https://d78kjfcaywen5.cloudfront.net

github.com/engyMasoud/Reddit-AI-Debate-Analyzer

CRESCENDO

Final Demo & Postmortem

CS 485 AI-Assisted Software Engineering

PROJECT CRESCENDO

Music discovery with transparent AI community seeding

Live demo | CI/CD | Postmortem

Team members

Jossie Zamora Marcus Victor Hilario James Marciano Mark Youssef Paulo Bellame

App overview

Problem statement, target users, and what Crescendo provides.

PROBLEM

Independent artists often release music into an “empty room.” They may have talent and listeners, but not enough visible activity to keep conversation and discovery moving.

SOLUTION

- Music discovery pages for artists, albums, genres, and community discussion.
- Transparent bot personas that are labeled clearly when AI participates.
- Activity driven surfaces that help users find conversations worth joining.

TARGET USERS

- Independent artists who need early momentum.
- Music fans looking for new or niche artists.
- Authenticated users who want to post, reply, and create lists.

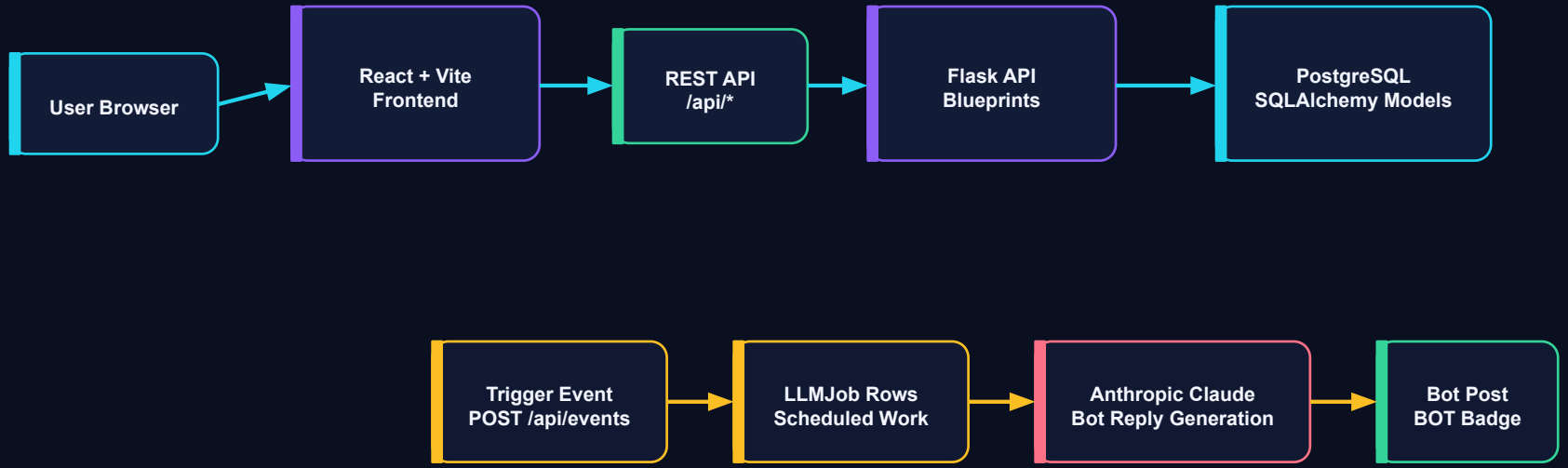
Demo roadmap

https://main.d291kg32gzfrfc.amplifyapp.com/

0:00 - 0:45	**Discovery page**	Show artist grid, filtering, sorting, and activity based discovery.
0:45 - 1:30	**Artist profile**	Open an artist page and show profile context, activity score, and discussions.
1:30 - 2:30	**Discussion flow**	Open a discussion, show BOT flair, then post a reply as an authenticated user.
2:30 - 3:15	**AI activity trigger**	Use developer control to trigger LLM activity and explain scheduled bot participation.
3:15 - 4:15	**Album ecosystem**	Show best albums, new releases, and genres as additional discovery surfaces.
4:15 - 5:10	**Lists workflow**	Create or open a list and demonstrate user curation.
5:10 - 6:00	**Credibility close**	Show community stats, deployed app, and CI/CD status.

System architecture

React frontend, Flask backend, database models, and AI generated bot participation.



Key data models: Artist, Album, Discussion, Post, User, LLMPersona, LLMJob, List, ListAlbum

Testing strategy

The project uses layered tests to protect frontend, backend, and full app flows.

Backend tests

- pytest
- Route and model coverage
- SQLite in-memory during tests
- Coverage reports in CI

Frontend tests

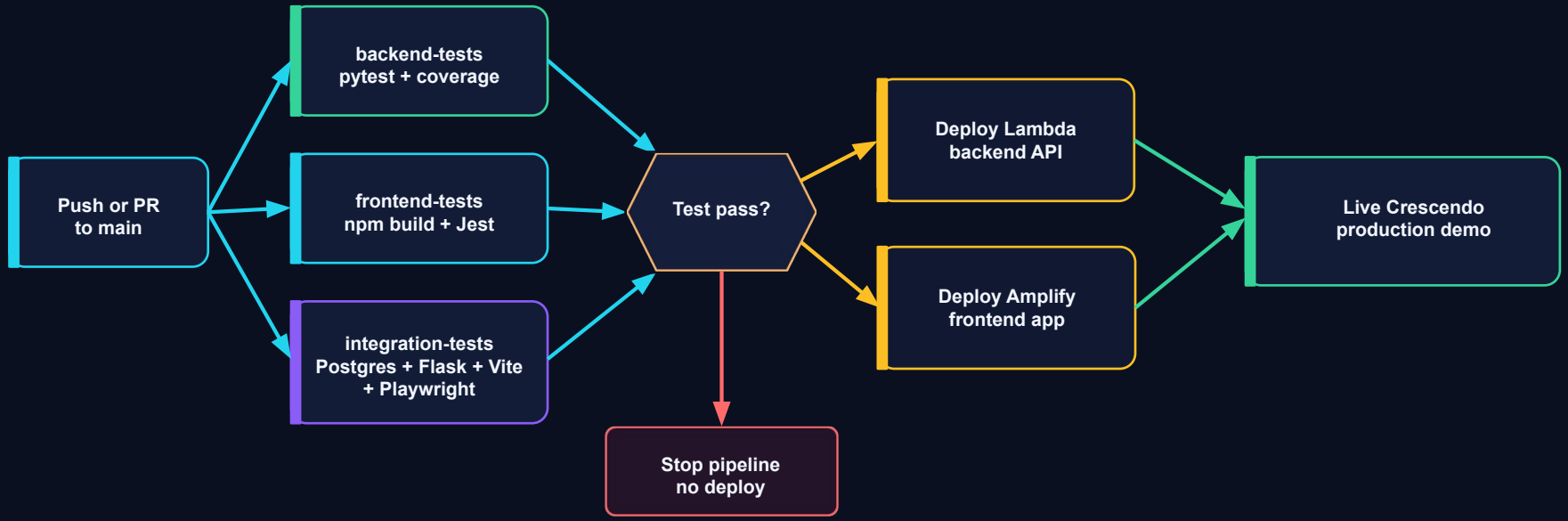
- Jest and React Testing Library verify core React pages and important rendering flows.
- Vite build step catches TypeScript/bundling problems before deployment.

Integration tests

- Playwright
- Flask + Vite + Postgres
- Auth, lists, search, route flows
- Deployment blocked on failures

CI/CD overview

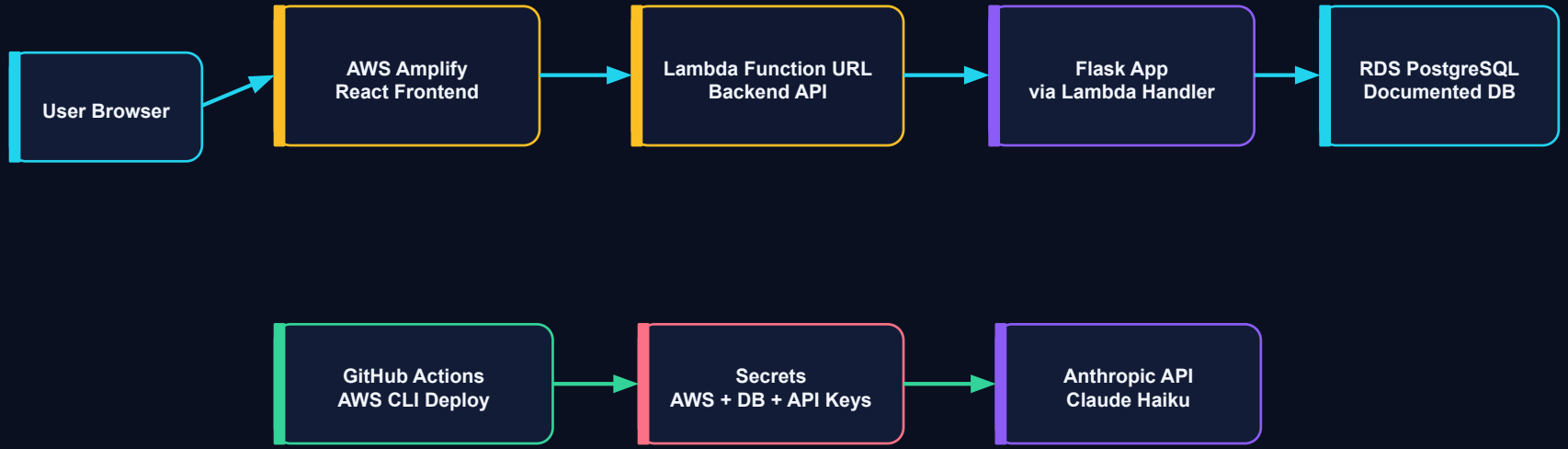
GitHub Actions runs tests first, then deploys only from main after required jobs pass.



Deployment gate: deploy jobs require successful backend, frontend, and integration jobs. Secrets are stored in GitHub Actions, not in the repo.

Deployment overview

AWS Amplify serves the frontend while AWS Lambda runs the Flask backend API.



Frontend URL: https://main.d291kg32gzfrfc.amplifyapp.com

Backend API URL: https://ue039qft5b.execute-api.us-east-1.amazonaws.com/prod

Postmortem: successes, failures, lessons

What we learned from building, testing, and deploying Crescendo.

Successes

- Full-stack app reached a live deployed demo state.
- Core routes support discovery, discussions, auth, lists, search, and stats.
- CI/CD pipeline became more reliable over the sprint.
- Bot transparency stayed central through visible BOT labeling.

Failures and surprises

- Serverless deployment exposed a mismatch with in-process LLM scheduling.
- Serverless LLM Scheduling needs stronger integration testing.
- CI/CD required more iteration than expected.
- Some documentation drifted from the current workflow.
- Developer controls remained visible longer than ideal.

Lessons

- Design background jobs around deployment reality early.
- Keep architecture docs synced with code and workflows.
- Test deployment behavior, not just local behavior.
- Use LLMs as accelerators, but verify generated assumptions.

LLM usage analysis

LLMs helped both the product experience and the engineering workflow, but required verification.

PRODUCT USE

- Anthropic Claude Haiku generates synthetic discussion participation.
- Bot comments are labeled with visible BOT flair for transparency.
- The goal is to seed conversation without pretending AI users are human.

ENGINEERING USE

- LLMs helped draft materials, diagrams, deployment explanations, and implementation plans.
- Generated outputs still needed repo-level verification.
- A recurring issue was confident output that did not always match the deployed architecture.

Core lesson: LLM speed is valuable only when paired with human review, tests, and deployment proof.

Future work

Next features that would make Crescendo more reliable, useful, and launch-ready.

Artist analytics dashboard	Show engagement trends, listener activity, and discussion growth over time.
Queue-based LLM worker system	Move scheduled bot jobs to SQS or EventBridge for reliable serverless execution.
Moderation and reporting	Add safety controls for comments, bot tone, and community guidelines.
Role-based authorization	Separate guest, user, artist, and admin permissions more clearly.
Real-time updates	Use WebSockets or SSE so comments and activity update without manual refresh.

VSClone

A Cursor-style editor that runs on the AI subscriptions you already pay for.

Team

VSClone

Members

Brandon Howe

App Overview

THE PROBLEM

IDE has the UX. CLI has the value. Neither has both.

Cursor: polished editor UX, but the $20 plan is exactly $20 of API credits at retail rates.

Claude Code, Codex, Gemini CLI: vastly more capacity per dollar — but the terminal is not an IDE.

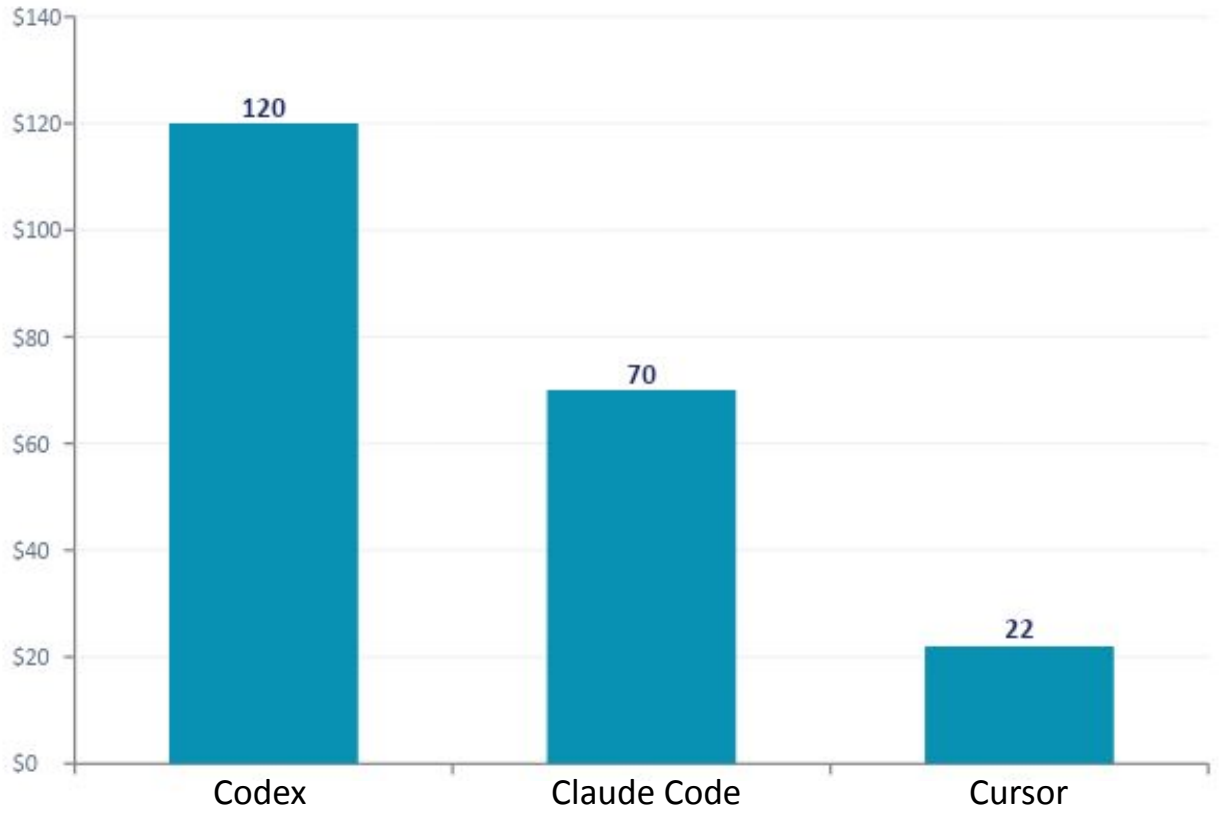
TARGET USER

Anyone who already pays for an AI subscription and doesn't want to double-pay for an IDE.

- Claude Pro / Max
- ChatGPT Plus / Pro
- Gemini Advanced

WHAT $20/MONTH ACTUALLY BUYS YOU

Estimated API-equivalent value, retail rates



Demo Roadmap

1

OAuth Login

Authenticate against Google with PKCE. Tokens land in the OS keychain — never in the renderer.

2

Build a Game with Gemini

Prompt the agent to scaffold a small browser game; watch it read, write, and run files end-to-end.

3

Chat & Multiple Threads

Switch between parallel conversations with persistent context and per-thread history.

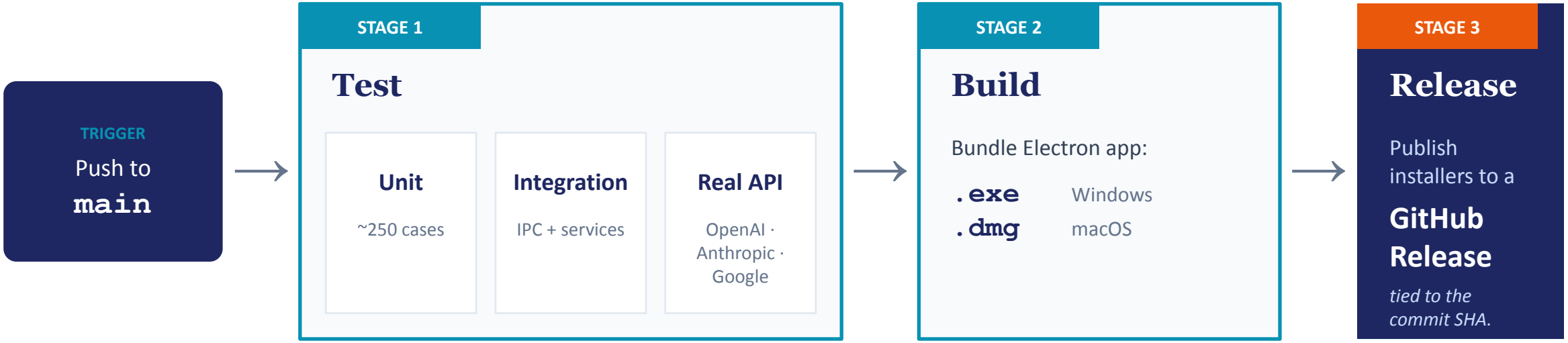
4

Multimodal + Permissions

Drop in a screenshot, then watch the permission gate before any destructive shell command runs.

CI/CD & Deployment

Non-traditional. *VSClone is a desktop application — there is no AWS or backend to deploy. The release pipeline runs on GitHub Actions and ships installers directly to users.*



SECRETS & PERMISSIONS

API keys for Anthropic, OpenAI, and Google are stored as GitHub Actions repository secrets and only injected for the real-API test job — never baked into release artifacts. Test failures fail the workflow and block the release stage.

Postmortem · Successes

Secure OAuth across all three providers

PKCE, token refresh, and OS-keychain-backed credential storage. Tokens never reach the renderer.

Cursor-style tab completion

Inline ghost-text suggestions powered by the user's own subscription.

Working agent harness

Read files, write files, explore directories, run shell commands — gated by a permission prompt.

Successful late-semester refactor

Swapped raw-JS DOM rendering for Preact and restructured the chat data model — paid off immediately.

48 test files

~250 test cases

80% line coverage

3 OAuth providers

Postmortem · Failures & Challenges

Codebase scale ate context windows

- VSCode's source is enormous and verbose in raw JS
- Lower-tier models had their context filled before producing useful output
- Switched UI to Preact mid-semester for a representation models could reason over – should have done at the beginning

Architectural drift forced a late refactor

- No clear up-front design caused random chat state and IPC bugs
- Rewrote chat data model in P6 — risky timing, but tractable with LLMs
- Definitely was a good idea, but I would do this earlier in the future

Anthropic OAuth gates Sonnet/Opus

- Anthropic doesn't expose its top models through OAuth
- Limits VSClone's usefulness for Claude users
- T3Code-style SDK spoofing would unlock it

Postmortem · Process Evaluation

YES — significantly

Did LLM-driven development help?

- Worked several times faster than I could have alone
- Opus's 1M context window was critical for VSCode's verbose codebase
- Frontier models handled most features in 2–5 prompts of iteration

N/A — solo project

Did the feature-branch workflow work smoothly?

- Worked alone, so branching and PR review didn't really apply
- Mostly committed to main with CI as the safety net
- Skipped the GitHub Projects board after early weeks — kept context in my head
- Both would matter on a team or larger codebase

EVOLVED throughout

How did your testing strategy evolve?

- Started with plain unit tests in common/
- Added IPC integration tests once the renderer/main split solidified
- Layered real-API tests against OpenAI, Anthropic, Google late in the semester
- GitHub Actions secrets injected only for the real-API job

Postmortem · Lessons Learned

01 Frontier models pay for themselves

Opus's 1M-token context window made working in VSCode practical. The same project with Haiku-tier models would have been dramatically harder.

02 LLMs struggle where humans do

Massive codebases, tangled architectures, undocumented invariants. Most failures weren't model failures — they were complexity failures.

03 Late refactors are tractable with LLMs

What used to be a multi-week risk is now a multi-day sprint. Don't anchor on architectural decisions just because the project is mid-flight. The refactor in this case was done too late and should have been completed earlier.

Future Work

1

Refine the agent harness

Functional today, but unrefined. Improve diff UX, tool-call ergonomics, error recovery, and streaming feedback to close the gap with Cursor and Antigravity.

2

Spoof the Claude Code SDK

Anthropic's OAuth gates Sonnet and Opus. Following T3Code's pattern of impersonating the Claude Code SDK would unlock Opus 4.7 for Claude subscribers.

3

Specialized tab-completion model

Current implementation is a general-purpose chat model. A small model trained or distilled specifically for code completion would be faster and cheaper.

4

Smarter codebase indexing

Embedding-based retrieval would let the agent navigate large repos without filling the context window on every query — a real bottleneck during this project.

CS 485 · SPRING 2026

Harmony

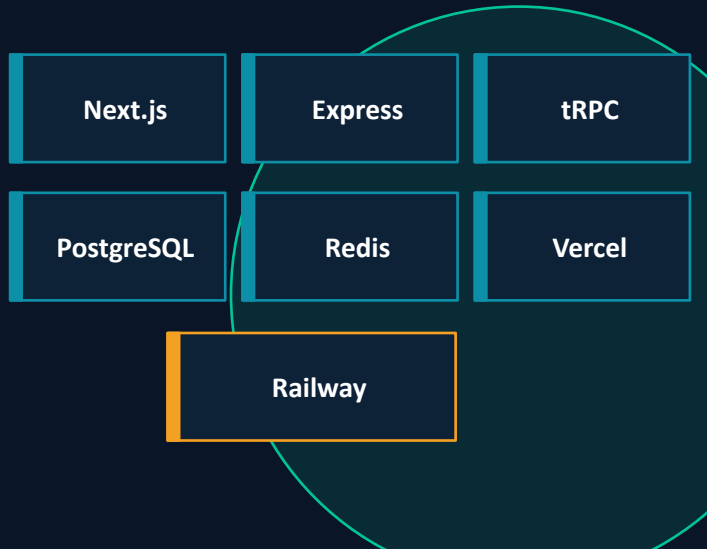
A Search-Engine-Indexable Chat Application

CS485-Harmony

- Allen Cabrera
- Aiden Barrera
- Avanish Kulkarni
- Declan Blanchard
- Fardeen Iqbal

Frontend: harmony-dun-omega.vercel.app

Backend API: harmony-production-13e3.up.railway.app



App Overview

Problem · Solution · Target User

The Problem

Chat apps are walled gardens.

Team conversations, help threads, and community knowledge live inside closed platforms — invisible to Google, Bing, and other crawlers.

Users can't find answers without an account, and knowledge never surfaces in organic search.

Our Solution

- **Public Channels**

  Any channel toggles to PUBLIC_INDEXABLE — full SSR with Open Graph, Twitter Card & JSON-LD.

- **Guest View**

  Unauthenticated visitors browse public channels without logging in.

- **SEO Meta Tags**

  Auto-generated titles & descriptions via NLP. Admin overrides available.

- **Real-Time Messaging**

  SSE-based live updates; no WebSocket required.

Target Users: Open-source communities · Dev teams · Public support forums · Knowledge-sharing organizations

Demo Roadmap

Live walkthrough of the deployed application: harmony-dun-omega.vercel.app

01 Landing Page

Browse the public server landing — no login needed. Shows indexed channels discoverable by search engines.

02 Guest Channel View

Open a PUBLIC_INDEXABLE channel. View messages, scroll history, and see real-time updates as a guest.

03 SEO Meta Tags

Inspect <head> tags in DevTools — title, description, Open Graph, Twitter Card, and JSON-LD structured data.

04 Visibility Toggle

Admin toggles a channel from PUBLIC_INDEXABLE to PRIVATE. Guest link returns 403. Sitemap updates.

05 Authenticated Chat

Log into Harmony. Send a message — SSE delivers it live to all connected clients in under 100ms.

06 Admin Meta Override

Override auto-generated title & description. Trigger background regeneration job — poll status to succeeded.

CI/CD Pipeline

GitHub Actions · 6 workflows · Vercel + Railway

On Every PR

ci.yml

Frontend lint + build
Backend lint + build + Prisma generate

run-frontend-tests.yml

Jest unit tests (Next.js components)

run-backend-tests.yml

Jest unit tests (Express services)

On Every PR + Main

run-integration-tests.yml

Spins up Postgres + Redis
Seeds mock data — runs full API + SSR suite

On Main / Dispatch

demo-seed.yml

Re-seeds demo data in production Railway DB

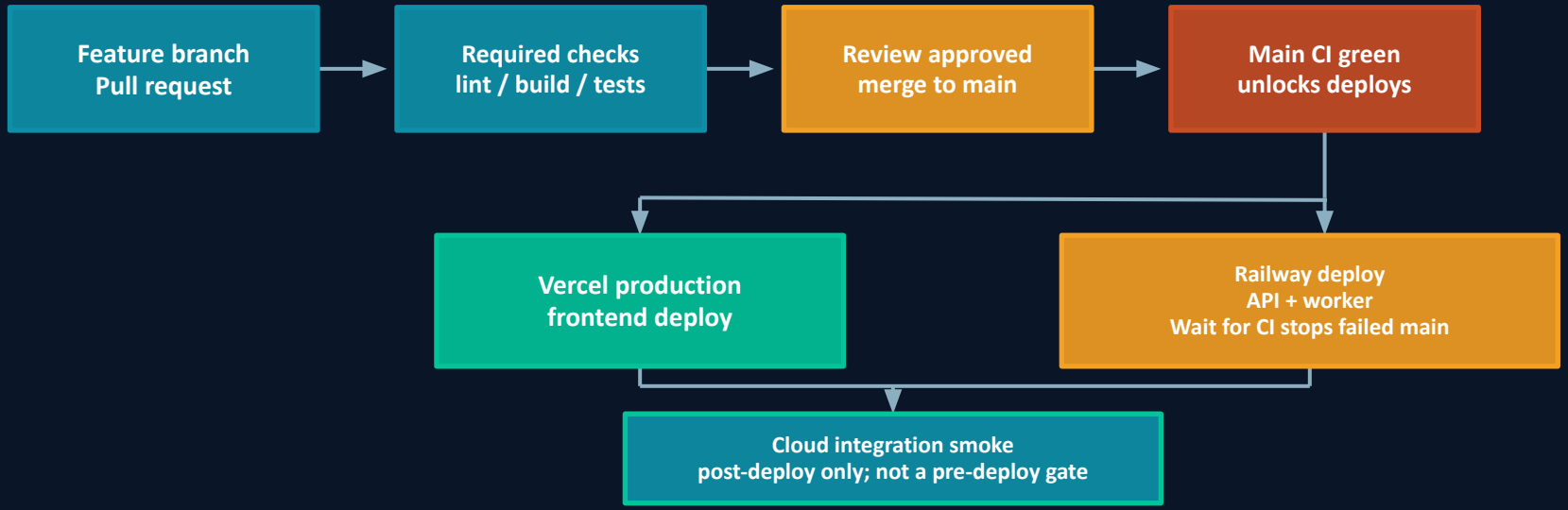
cloud integration tests

Read-only smoke tests against deployed URLs

Failure impact: main is protected by CI + Integration Tests; Railway Wait for CI blocks deploys until GitHub Actions pass

CI/CD Workflow Chart

After merge to main: Vercel + Railway deploy in parallel; cloud smoke runs post-deploy



6
GitHub Actions Workflows

2+
API Replicas (Railway)

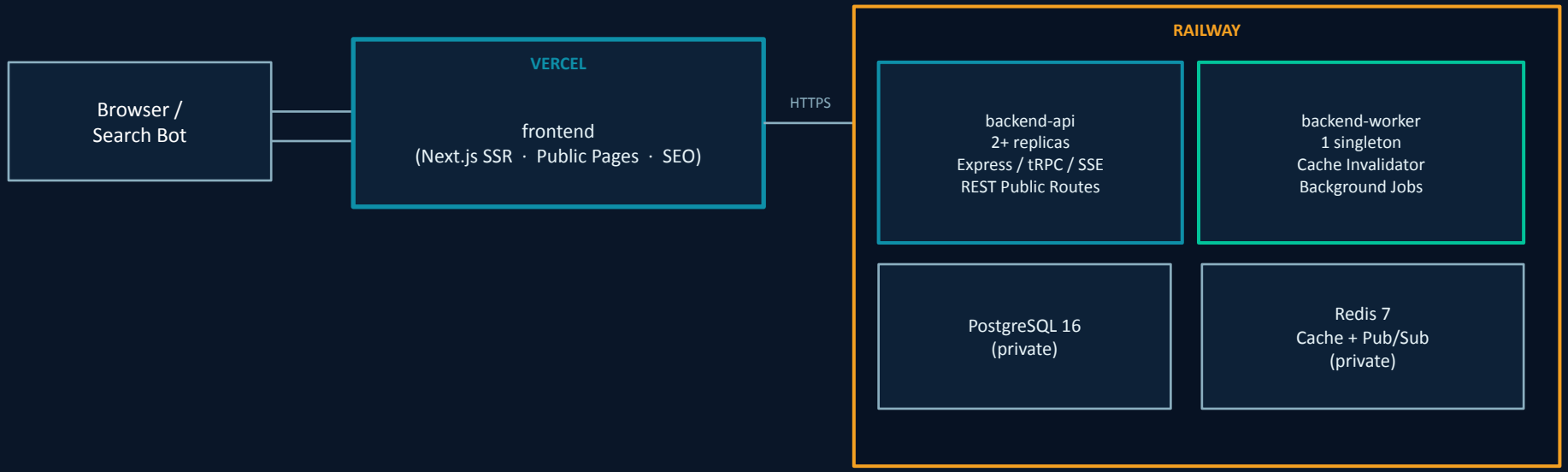
1
Singleton Worker

100+
Backend Unit Tests

12+
Integration Test Cases

Deployment Architecture

Vercel (frontend) + Railway (backend-api · backend-worker · Postgres · Redis)



Auth: Bearer JWT tokens · httpOnly cookie (frontend only) · CORS origin allowlist — no wildcard

File Storage: Cloudflare R2 (S3-compatible) via STORAGE_PROVIDER=s3 · SSE fan-out via Redis Pub/Sub — each API replica subscribes independently (no sticky sessions)

Secrets & Permissions

Where credentials live · how production access is scoped · what can block deployment

GitHub Actions

Repository secrets provide CI-only credentials. Branch protection requires CI + integration tests before main can merge.

Railway Production

Service variables live in Railway for backend-api and backend-worker. Wait for CI delays deploys until GitHub Actions pass.

Vercel Frontend

Frontend env vars are scoped by environment. httpOnly cookies stay browser-side; public config is separated from server secrets.

Scoped External Access

Postgres + Redis use private networking. R2 keys, JWT secrets, and CORS origins are least-privilege production variables.

No credentials are stored in the repo. Production-only secrets stay in provider-managed variable stores.

Postmortem — Successes

What went well across 5 sprints

LD

Live Production Deployment

Full Vercel + Railway stack live with 2+ API replicas, singleton worker, Redis pub/sub fan-out, and Cloudflare R2 attachments.

AI

AI-Assisted Development

Claude Code and GitHub Copilot used throughout: PR reviews, dev-spec authoring, feature scaffolding, and test writing (100+ sessions logged).

CI

Robust CI/CD with Branch Protection

6 GitHub Actions workflows enforce lint, build, unit tests, and integration tests before any merge to main.

SSE

Real-Time SSE Architecture

SSE delivers live message updates via Redis pub/sub — no WebSocket complexity, works through CDN proxies, scales across replicas.

SEO

SEO-First Design

NLP-driven meta tag generation (TF-IDF keywords, extractive summarization), Open Graph, Twitter Card, JSON-LD, and visibility-aware sitemap.

DS

Thorough Dev Specs

Three formal dev specs plus a unified backend architecture doc authored before a line of code was written.

Postmortem — Failures & Lessons

What we'd do differently

Failures / Pain Points

E2E Flakiness in CI

WebKit-only auth failures caused by running Next.js in dev mode inside GitHub Actions. Fixed by switching to production build + start.

Auth Crypto Divergence

PBKDF2 byte-encoding contract differed between backend helpers, seed data, and E2E bootstrap — browser auth and CI diverged silently.

Shared Port Collision

backend-api and backend-worker both read PORT env — worker would win the race and crash the API. Required explicit port separation.

Vacuous Integration Assertions

Early helpers used exclusion-only assertions without first checking the response shape — masking actual failures in CI.

Key Lessons

Red-Green-Refactor Strictly

For security or contract-breaking changes, write the failing test first before touching implementation.

E2E Needs Prod-Mode Frontend

For browser E2E in CI, prefer build+start over next dev — dev-mode tooling creates non-product flakiness.

Centralize Byte-Level Contracts

When a client/server crypto contract exists, mirror encoding rules across service, seed, and E2E bootstrap.

Rate Limits Need E2E Config

Shared rate limit state interacts badly with concurrent browser runs. Use NODE_ENV=e2e to raise thresholds.

Future Work

6 next features we'd ship with more time

01 Full-Text Search

Add Postgres tsvector/tsquery search on messages — lets users find relevant threads without leaving the app, and surfaces ranked results for public channels.

02 Custom Domain Mapping

Let admins point their own domain (e.g. chat.myproject.org) at a Harmony server. Unlocks branded canonical URLs and improves SEO authority.

03 Threaded Replies

Organize discussions Slack-style. Threads reduce noise in public channels and produce richer JSON-LD structured data for search crawlers.

04 Webhook / Zapier Integration

Outbound webhooks on message events let teams pipe Harmony into CI systems, Slack bridges, or ticketing tools without built-in complexity.

05 AI Channel Summaries

LLM-generated weekly digests pinned to each public channel — improving first impression for new visitors and enriching SEO meta descriptions.

06 Security Issues

Many security issues were deprioritized in the interest of being feature complete during the final sprint. We would like to revisit and fix these later.

CS 485 · SPRING 2026

Thank You

Harmony — Making chat conversations findable, shareable, and open.

Harmony

github.com/CS485-Harmony/Harmony

Allen Cabrera

Aiden Barrera

Avanish Kulkarni

Declan Blanchard

Fardeen Iqbal

Spiritual Q&A Platform

CS 698 · Software Engineering

Group 12 · Balaji Shashipreeth Racherla · UCID: BR426 · P7 Final Demo

Flutter · FastAPI · PostgreSQL · Redis · Qdrant · OpenAI · Terraform · AWS

App Overview

RAG-powered spiritual Q&A — answers grounded in authentic organizational texts with source citations

The Problem

Spiritual seekers cannot get accurate, source-grounded answers from authentic organizational texts. Public search mixes unvetted content with genuine teaching — generic AI hallucinates or draws from out-of-canon sources.

Target User

Members and students seeking citation-backed answers from proprietary texts — without relying on generic web search or hallucinating AI assistants.

Primary User Stories

- Guest: Ask a question, receive a cited answer without login
- Member: Register, login securely, maintain conversation history
- Admin: Upload PDFs, trigger ingestion, manage documents

Our Solution

1. RAG pipeline over proprietary spiritual texts
2. OpenAI embeddings + Qdrant vector similarity search
3. Cited answers with exact page / paragraph references
4. Flutter web/mobile frontend on AWS Amplify
5. FastAPI backend on EC2 · RS256 JWT auth · CSRF protection
6. Admin panel: PDF upload and ingestion management

Live Demo

Walking through 5 user stories on the deployed application

US1

Guest Q&A — Frontend Live on Amplify

"As a guest, I can ask a spiritual question and receive a cited answer without logging in"

DEMO STEPS

1. **Open Amplify URL**
   [AMPLIFY_URL] — confirm HTTPS, hosted on Amplify CDN globally
2. **Observe guest access**
   No login required — Redis rate-limited session (IP + guest_session_id cookie)
3. **Type a spiritual question**
4. **Submit and watch response**
   FastAPI: embed query → Qdrant search → GPT-4.1-mini → Flutter renders answer
5. **Show citations panel**
   Each answer cites: document name, page number, paragraph — not hallucinated

WHAT THIS PROVES

Frontend LIVE
Active on AWS Amplify CDN — globally served, HTTPS, no localhost

Frontend → Backend
Dio HTTP client calls EC2 /api/chat endpoint over HTTPS

RAG pipeline
End-to-end: query → embed → retrieve → generate → cite

Guest rate limit
Redis guards against abuse without requiring authentication

US2

Registration & Secure Login

"As a new user, I can register and log in — RS256 JWT tokens are issued and stored securely"

DEMO STEPS

1. **Navigate to Register screen**
   Click "Create Account" — GoRouter navigates to /register; no auth guard on this route
2. **Fill form and submit**
   POST /api/auth/register — user created in PostgreSQL, password bcrypt-hashed
3. **Login with credentials**
   POST /api/auth/login — RS256 JWT issued: access (15 min) + refresh (7 days)
4. **Show auth guard in action**
   Navigate to /chat while logged out → GoRouter auth guard redirects to /login
5. **Attempt wrong password**
   POST /api/auth/login → 401 Unauthorized — error displayed in Flutter UI

AUTH ARCHITECTURE

RS256 JWT (asymmetric)

Private key signs on backend; public key verifies — key never leaves server

flutter_secure_storage

Tokens in OS Keychain / Keystore — never in localStorage or cookies

CSRF protection

HMAC-signed double-submit cookie threaded through Dio interceptor

Token refresh

Refresh call is transparent to user; revoked tokens checked in PostgreSQL

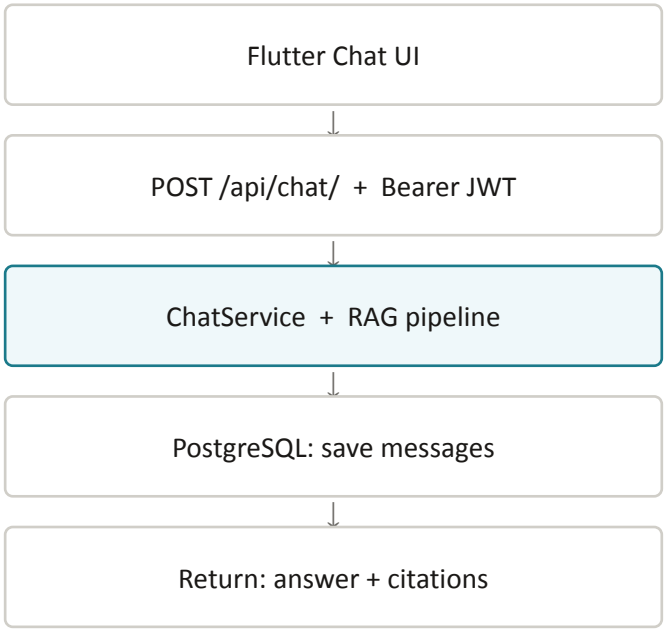
US3 Authenticated Q&A + Conversation History

"As a logged-in user, my questions and answers are saved and I can review past conversations"

DEMO STEPS

1. **Log in with test account**
   JWT issued — Riverpod auth state updated, GoRouter navigates to /chat
2. **Ask a question (authenticated)**
   POST /api/chat/ with Authorization: Bearer — conversation_id assigned in PostgreSQL
3. **Ask a follow-up**
   Last 5 turns injected into RAG prompt for coherent multi-turn context
4. **Navigate to History tab**
   GET /api/chat/history — all past sessions loaded from PostgreSQL
5. **Tap a past conversation**
   Full message thread re-loaded — persisted server-side, not in local storage

DATA FLOW



US4

Admin Panel — PDF Upload & Ingestion

"As an admin, I can upload a spiritual text PDF and make it searchable in seconds"

DEMO STEPS

1. **Login as admin user**
   Admin role in PostgreSQL — GoRouter shows Admin tab only for admin role (RBAC)
2. **Upload a PDF**
   Admin panel: Upload Document → multipart POST /api/admin/documents/upload
3. **Trigger ingestion**
   POST /api/admin/documents/{id}/ingest → parse → chunk → embed → upsert Qdrant
4. **Verify in document list**
   GET /api/admin/documents → new doc with status "ingested" and chunk count
5. **Ask a question about it**
   Switch to chat → ask about the new PDF → citations reference the new document

INGESTION PIPELINE

1. PDF Parser

pdfminer — extracts raw text, preserves page structure

2. Chunker

500-token chunks, 50-token overlap, paragraph-aware

3. Embedder

OpenAI text-embedding-ada-002 → 1536-dim vectors

4. Qdrant Upsert

spiritual_docs collection — cosine similarity index

US5

Live Backend — Endpoints on EC2

"Show the backend API is live, all services are healthy, and the OpenAPI spec is reachable"

DEMO STEPS

1

Open /health/full in browser

GET [EC2_URL]/health/full — confirm JSON: all 4 services return "ok"

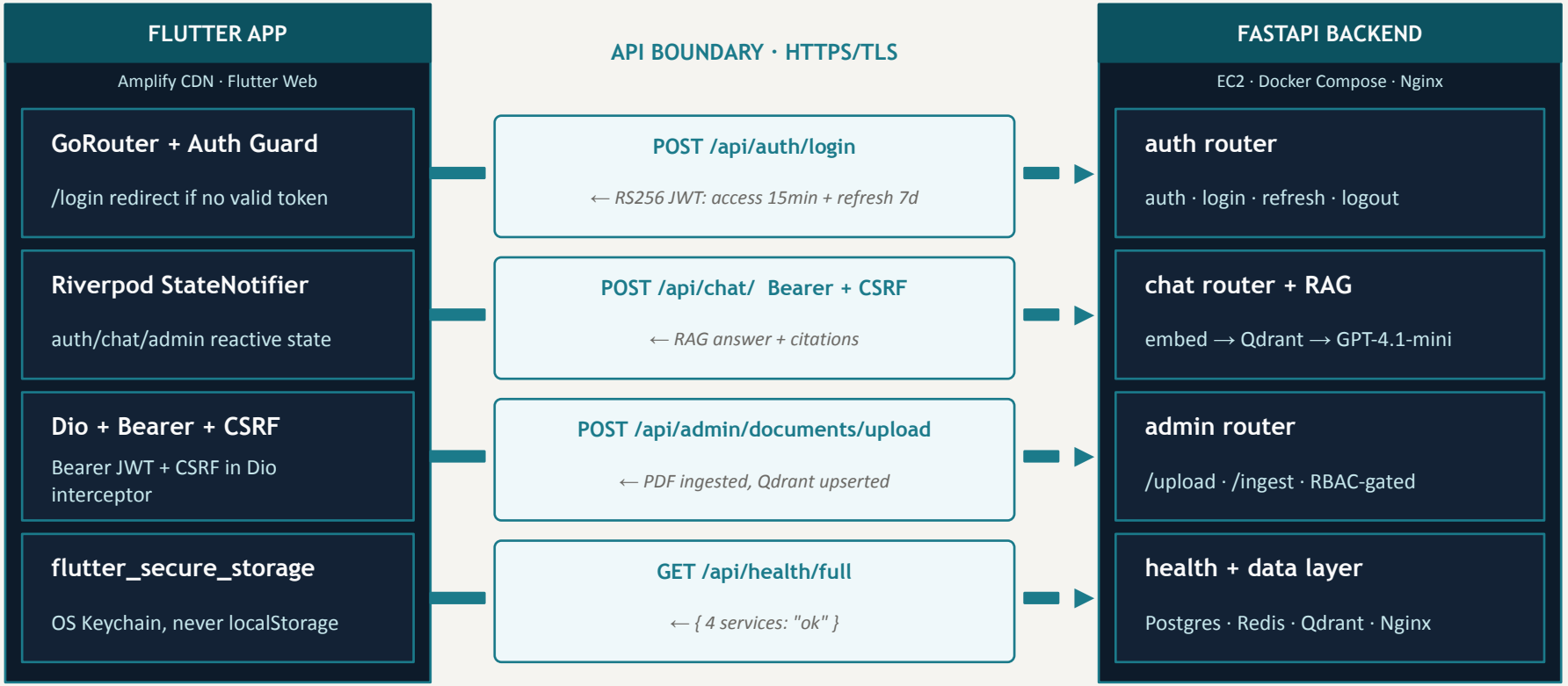
EXPECTED: GET /health/full

```
{
 "status": "healthy",
 "postgres": "ok",
 "redis": "ok",
 "qdrant": "ok",
 "openai": "ok",
 "version": "1.0.0"
}
```

Verify ALL 4 services green before presenting

Frontend → Backend Integration

Flutter (Amplify CDN) talks to FastAPI (EC2 Docker) over HTTPS — Dio HTTP client with JWT + CSRF on every authenticated call



Nontrivial Workflows

RS256 JWT authentication flow and RAG query pipeline — the two most complex integration paths

RS256 JWT AUTHENTICATION

1. **Register**
   POST /api/auth/register ⟶ bcrypt hash ⟶ PostgreSQL
2. **Login**
   POST /api/auth/login ⟶ bcrypt.verify ⟶ RS256 sign
3. **Tokens issued**
   access 15 min + refresh 7 days — RS256-signed
4. **Stored securely**
   flutter_secure_storage ⟶ OS Keychain / Keystore
5. **Authenticated request**
   Bearer token header + HMAC CSRF double-submit cookie

RAG QUERY PIPELINE

1. **Question submitted**
   POST /api/chat/ + Bearer JWT + conversation_id
2. **Embed query**
   text-embedding-ada-002 ⟶ 1536-dim float vector
3. **Vector search**
   Qdrant cosine ⟶ top-5 chunks (threshold ≥ 0.7)
4. **Generate answer**
   GPT-4.1-mini: context + last 5 turns ⟶ cited
5. **Persist & return**
   PostgreSQL save + Flutter renders answer + citations

Application URLs

Frontend (AWS Amplify):

https://main.d2pblhjs4pjp2h.amplifyapp.com

Flutter web — globally served on Amplify CDN

Backend API (EC2):

https://d30olv28vis4z6.cloudfront.net/health/full

FastAPI — shows all 4 services: ok

Repository:

github.com/shashigemini/cs698-repo

All 7 GitHub Actions workflows + Terraform infra

Testing & Deployment

How tests run · How CD works · Secrets · Failure gates

CI Gate Architecture

Trigger conditions and deploy gates — read directly from .github/workflows/.yml*

ON EVERY PULL REQUEST (path-filtered)

run-backend-tests

*Trigger: apps/backend/** changed*

ruff lint · pytest + real Postgres/Redis/Qdrant in Docker

Required PR check

run-frontend-tests

*Trigger: apps/frontend/** changed*

flutter analyze · flutter test --coverage · flutter build web

Required PR check

run-integration-tests

*Trigger: backend/** OR frontend/** changed*

Full-stack E2E · OpenAI stubbed · Flutter Linux driver

Required PR check — DOES NOT TRIGGER DEPLOY

ON PUSH TO MAIN (PR merged) — integration-tests does NOT run on main

run-backend-tests ✓ passes

conclusion == success (workflow_run trigger)

No branch restriction in job condition

→

deploy-aws-ec2

Terraform init/plan/apply → SSH EC2 → git pull → docker compose up --build → /health/full (60 retries × 10s)

run-frontend-tests ✓ passes

conclusion == success + event == push

branch == main → production
branch != main → staging

→

deploy-aws-amplify

Production (main): POST webhook → Amplify CDN rebuild → flutter build web --release → live in ~3 min

Staging (non-main): POST $AMPLIFY_STAGING_WEBHOOK_URL → staging CDN rebuild

How Tests Run

Each workflow runs in an isolated GitHub Actions runner — exact steps for each workflow

run-backend-tests

1. Checkout + set up Python 3.11
2. docker compose up postgres redis qdrant -d
3. poetry install + ruff check app/
4. pytest tests/ -v --cov=app --cov-report=xml
5. Upload coverage report as workflow artifact

run-frontend-tests

1. Checkout + set up Flutter (stable)
2. flutter pub get + build_runner build
3. flutter analyze (any warning = fail)
4. flutter test --coverage lib/
5. flutter build web --release (build verif.)

run-integration-tests

1. Checkout + start backend (E2E_TESTING=true)
2. OpenAI client stubbed — zero API cost
3. docker compose up postgres redis qdrant -d
4. seed_qdrant.sh — preload test vectors
5. Flutter Linux driver runs integration_test/

Integration Tests & Failure Impact

How CI failures block deployment and what happens when a deploy goes wrong

HOW INTEGRATION TESTS WORK

E2E_TESTING mode

Env var disables real OpenAI — LLM client returns deterministic stub responses. Zero API cost in CI.

Full stack in one runner

FastAPI, Postgres, Redis, Qdrant all start in the GitHub Actions runner; seed data loaded via script.

Robot pattern (Flutter)

Each screen has a Robot class encapsulating interactions — tests are high-level and readable.

Coverage

auth: register + login + logout · chat: Q&A round-trip + history · admin: PDF upload + status

FAILURE IMPACT

Test job fails on PR

Merge blocked by required status check — code never reaches main branch

Test fails on push to main

needs: [run-backend-tests] in deploy workflow — deploy never triggers if tests fail

/health/full fails after deploy

Healthcheck step exits 1 — deploy job fails; previous container keeps running (no downtime)

Rollback strategy

git revert main → push → tests run → pass → deploy restores previous known-good image

CD Pipeline — Backend (EC2)

deploy-aws-ec2: triggered automatically after run-backend-tests succeeds on main branch

WHAT deploy-aws-ec2 DOES (step by step)

1. **Checkout + configure AWS credentials**
   AWS_ACCESS_KEY_ID + SECRET_ACCESS_KEY from GitHub Secrets — IAM user with least-privilege EC2 + Amplify policy
2. **Terraform init**
   Downloads providers, configures S3 backend (TF_STATE_BUCKET) — prevents state drift across team members
3. **Terraform plan + apply**
   Detects EC2, security group, and elastic IP changes. Creates or updates infra. State written back to S3.
4. **SSH into EC2 + git pull**
   Uses EC2_SSH_PRIVATE_KEY secret. Pulls latest merged code onto the server from main branch.
5. **docker compose up -d --build**
   docker compose -f docker-compose.prod.yml up -d --build — rebuilds images, replaces containers
6. **Readiness gate: /health/full**
   curl -f [URL]/health/full — retried 60 times × 10 sec (10 min max). All 4 services must return ok.

CD Pipeline — Frontend & Secrets

deploy-aws-amplify + how all secrets are stored and permissions granted

FRONTEND CD: deploy-aws-amplify

1. **Trigger**
   Runs after run-frontend-tests succeeds on main (needs: dependency in workflow YAML)

2. **Single webhook POST**
   HTTP POST to $AMPLIFY_PROD_WEBHOOK_URL from GitHub Secrets — triggers Amplify build

3. **Amplify build**
   Amplify pulls GitHub → flutter build web --release in build environment → deploys to CDN

4. **Live in ~3 min**
   Commit to main → CI passes → Amplify deploy auto-triggered — no manual steps needed

GITHUB REPOSITORY SECRETS	
AWS_ACCESS_KEY_ID / SECRET	IAM user — least-privilege EC2 + Amplify
OPENAI_API_KEY	Backend only — never sent to Flutter client
JWT_PRIVATE_KEY (RS256)	Signs tokens on backend; never in repo
JWT_PUBLIC_KEY	Used in CI for token verification tests
EC2_SSH_PRIVATE_KEY	Deploy workflow SSHs to run docker compose
CSRF_SECRET	HMAC key for CSRF cookie signing
AMPLIFY_WEBHOOK_URL	Webhook URL to trigger Amplify CDN rebuild
TF_STATE_BUCKET / REGION	S3 bucket for Terraform remote state

No secrets committed to code — all in GitHub Repository Secrets

Postmortem

Successes · Failures · Lessons Learned

Successes

What went well — architecture, CI/CD, testing, and team process

Architecture

✓ Router → Service → Repository pattern kept FastAPI clean and testable across all 5 routers

✓ Feature-slice frontend scaled well — admin, settings, startup screens added without regressions

Testing

✓ 22 backend test files covering every layer from auth service to RAG chunking

✓ E2E with OpenAI stubbed: zero flakiness, zero API cost in CI

✓ mutmut revealed real logic gaps in token decode and CSRF boundary conditions

CI/CD & Deployment

✓ Commit → test → deploy: fully automated — no manual steps after code review

✓ Terraform S3 remote state prevented infra drift across all team members

✓ /health/full gate caught broken deploys before users were affected

LLM-Driven Development

✓ Terraform + workflow YAMLs: 1-3 prompt iterations each — very high ROI

✓ RAG prompt and E2E test script: 4-7 iterations but saved significant wall-clock time

Failures & Surprises

Incidents and delays that cost us time

Critical failure Major delay Surprise

! **Qdrant volume corruption on EC2**

Forced container restart corrupted Docker volume — required manual rm + full re-ingestion (~90 min). Fixed with volume health checks in deploy workflow.

! **Flutter build_runner codegen drift**

Riverpod + freezed codegen drifted during iteration — cryptic compile errors. Cost ~6 hours. Fixed by adding build_runner to pre-PR checklist.

! **CSRF token flow misaligned with Flutter web SPA**

CSRF middleware blocked Dio requests — cookies not forwarded across redirects. 2 days to diagnose via detailed request/response logging.

! **EC2 SSH readiness vs. cloud-init race condition**

Workflow SSH'd into EC2 before Docker finished bootstrapping. Fixed with cloud-init status --wait step. Lost two deploy attempts.

~ **Mutation testing revealed untested boundary conditions**

mutmut showed token decode boundaries and CSRF generation were undertested despite high line coverage. Added targeted tests after report.

Process Evaluation

LLM-driven development · Feature branches · Testing strategy evolution

LLM Usage Analysis

- High ROI: Terraform HCL, CI workflow YAMLs, FastAPI boilerplate (1-3 iterations)
- Medium ROI: RAG prompt tuning, Riverpod scaffolding, test fixtures (4-7 iterations)
- Low ROI: Business logic edge cases — outputs required heavy correctness verification
- Fix strategy: add concrete file context, decompose large prompts into subtasks
- Overall: LLMs saved time on infrastructure and scaffolding tasks

Workflow & Team Process

Feature branches worked well

PRs triggered CI — tests must pass before merge. Caught several regressions before they reached main.

Path-filtered CI cut cost

Backend CI only runs on apps/backend/** changes. Frontend for apps/frontend/**. ~50% faster CI.

Testing strategy evolved

Started with unit tests; added E2E after backend stabilized; added mutation testing in final sprint.

Lessons Learned

What we would do differently starting from scratch

1

Add /health/full from day one

Retrofitted late. Starting with readiness checks for every dependency would have caught issues weeks earlier.

2

Catch codegen drift in CI

build_runner drift should fail a CI job immediately — not be discovered during development.

3

Stub third-party services by design

LLM client behind an interface from day one makes testing faster and completely free in CI.

4

Run mutation tests per module in CI

Line coverage masks real logic gaps. mutmut per module in CI (not one-time audit) catches issues sooner.

5

IaC-first — Terraform before manual AWS

Early manual EC2/Amplify setup caused drift that took days to reconcile. IaC from day one is non-negotiable.

Future Work

5 features for the next iteration of the platform

1 **Multi-collection Qdrant support**

Partition documents into separate vector collections by book or topic — targeted retrieval, less noise in answers.

2 **Conversation memory across sessions**

Persist a summarized conversation buffer in Redis for long-running Q&A threads across multiple sessions.

3 **Real-time streaming answers (SSE)**

Server-Sent Events endpoint in FastAPI — render tokens progressively in Flutter for a GPT-like experience.

4 **Answer quality feedback loop**

Thumbs-up/down signals logged to PostgreSQL, used to tune retrieval parameters and eventually fine-tune a domain-specific model over time.

5 **iOS / Android native distribution**

Submit to App Store and Google Play with full native authentication, push notifications, and background sync for community updates.

Thank You

Frontend (AWS Amplify):	https://main.d2pblhjs4pjp2h.amplifyapp.com
Backend API (EC2):	https://d30olv28vis4z6.cloudfront.net/health/full
Repository:	github.com/shashigemini/cs698-repo

Team BR426 · CS 698 · Spring 2026

OutlookPlus

The AI sidebar that reads your inbox so you don't have to.

CS 698-004 · NJIT · Spring 2026

Group 13: Xun Song & ZhiRong Zhang

Two questions before we start.

1. Who here has ever asked ChatGPT, Claude, or Gemini to help write an email reply?
2. Who has more than 100 unread emails right now?

> *Please raise your hands.*

The problem and the user we built for.

The problem

Busy users get more than 100 unread emails. Reading, sorting, and replying takes hours every week. Today's AI tools live in a different tab, so you have to copy and paste every email to get help.

Target user

Students, researchers, and working professionals who use Gmail or another email mailbox daily — and want an LLM to help them read, organize, and draft replies right inside their inbox.

What this costs them

A typical user spends ~20 minutes a day skimming 50 unread messages, writes the same kind of reply 5 times a week, and switches tabs every time they want an AI summary.

What we promise

Open the inbox once. Every email gets a summary, a category, and a draft reply right inside the same view — no copy, no paste, no second tab.

1 · LIVE DEMO · ≤ 6 MIN

OutlookPlus is an AI manager for your inbox.

Auto-summary on open

Every email gets a one-line summary, a sentiment label, and a category tag the moment you open it.

One reply draft, two suggested actions

LLM drafts a reply you can send in one click, and gives you two suggested next steps below it.

Auto-classified inbox

Every message is sorted into Work, Personal, Finance, Social, Promotions, or Urgent — no manual rules required.

AI polish before you send

Write your reply in plain words, click the polish button, and Gemini rewrites it into a clean, professional email before you send it.

Stack: React + Vite · FastAPI + Mangum · AWS Lambda · S3 · Gemini 3 Flash

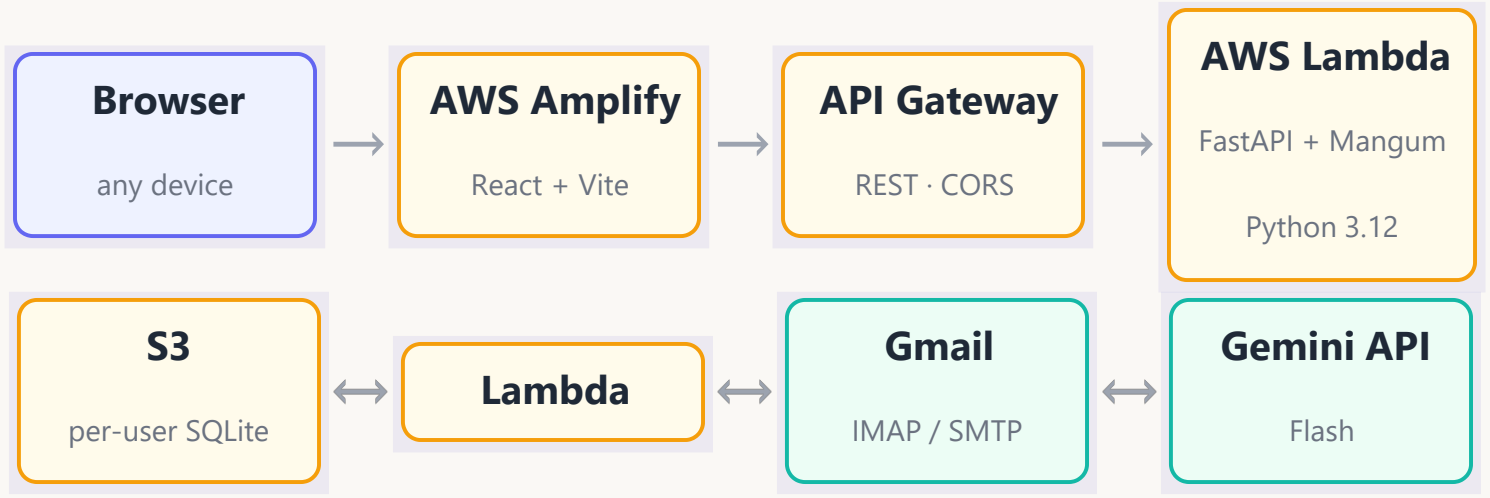
LIVE DEMO

https://main.d3sfc324cuhh5w.amplifyapp.com/inbox

https://www.youtube.com/@zhirongzhang-0913

2 · TESTING & DEPLOYMENT · ≤ 2 MIN

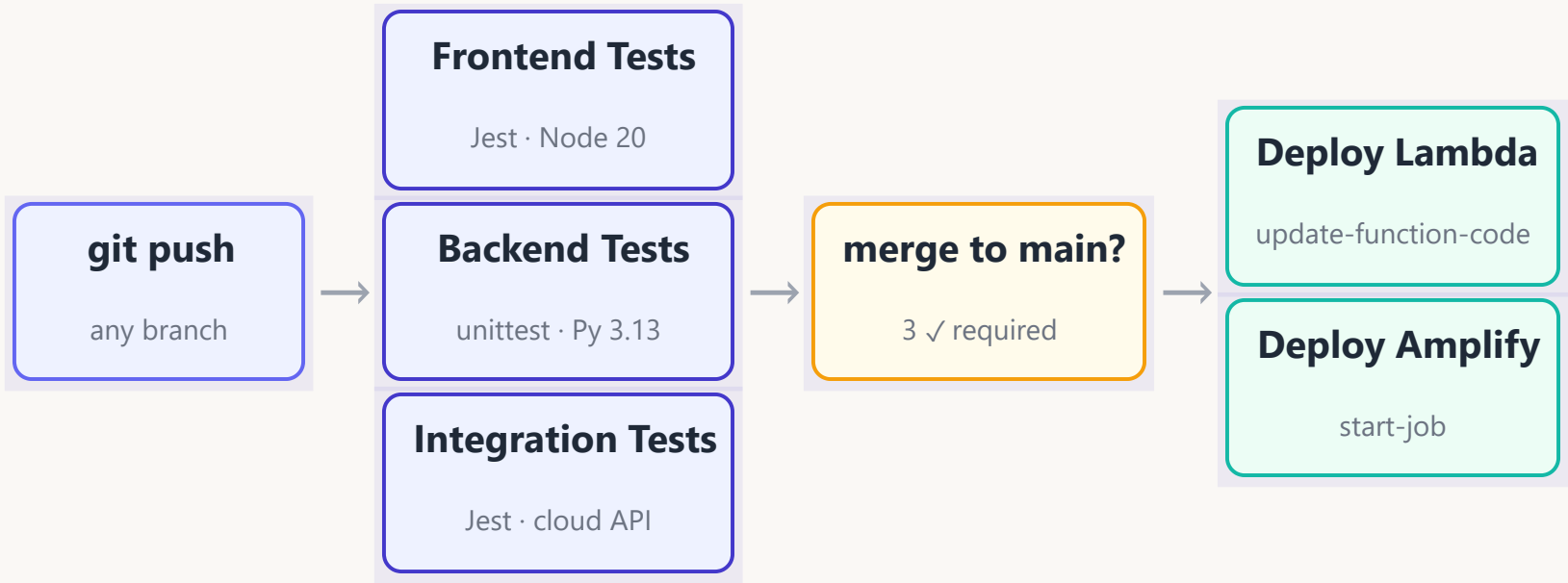
The architecture is serverless



Every request carries an `X-User-Email` header.
Lambda loads that user's data, runs, then saves it back — nothing is shared.

2 · TESTING & DEPLOYMENT · ≤ 2 MIN

GitHub Actions runs our tests on every push.



Three test suites run in parallel — frontend, backend, and integration.
All three have to pass before anything can merge into main.

2 · TESTING & DEPLOYMENT · ≤ 2 MIN

After merging to main, two workflows deploy.

Backend → AWS Lambda

The workflow zips up our Python code,

uploads it to AWS Lambda, and tells Lambda

to start using the new version.

Frontend → AWS Amplify

The workflow asks AWS Amplify to start a release

build, which rebuilds the React app and publishes

it to our public URL.

Both log in with `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` from GitHub Secrets — no developer ever sees the raw keys.

3 · POSTMORTEM · ≤ 4 MIN

Looking back, here is what went well.

We shipped end-to-end in ten weeks

We merged 97 commits across 9 pull requests with zero rollbacks, going from the first commit to a live URL in ten weeks.

AI wrote almost all of the code, even Amplify, which I had never used.

I barely typed any code by hand. The LLM wrote the frontend, the Lambda handlers, and even the Amplify config — a tool I had never opened before.

Our scope stayed clear from day one

We locked the feature set in week one and never pivoted. Every commit pushed the same target forward — no rework, no wasted branches.

Branch protection worked well

GitHub Actions ran our three test suites on every push, and all three had to pass before we could merge — that caught a lot of real bugs.

3 · POSTMORTEM · ≤ 4 MIN

Now let's look at what actually broke.

Lambda kept forgetting users' data

Every time Lambda woke up cold, it wiped its own scratch disk — so users lost data. We fixed it by saving back to S3 after every write.

The AI sidebar fought us for a whole day

One day in April we made twelve commits trying to fix the sidebar — every time we patched one timing bug, another one popped up.

The AI built buttons that didn't do anything

The LLM happily added buttons for features we hadn't actually built — the page looked finished, but clicking the buttons did nothing.

Fake "loading" text fooled real users

Friendly placeholder text fooled real users — so we pulled every fake message out of the app.

3 · POSTMORTEM · ≤ 4 MIN

Here are three honest opinions on our process.

LLM-driven development

The LLM wrote roughly 80% of our code, our tests, and our infrastructure setup. Where it kept tripping up was small React `useEffect` bugs — it brought back the same race condition four times.

Feature-branch workflow

Our feature-branch workflow worked really well: every fix went through a pull request, and branch protection caught the regressions before they hit main.

Testing strategy

We started with no tests at all, then moved to clicking through the app by hand, then added unit tests, and finally integration tests — which caught the bugs the earlier stages missed.

3 · POSTMORTEM · ≤ 4 MIN

If we started over, here are three things we would do differently.

1. Bring tests in earlier.

If we had written tests when the codebase was small, we would have caught bugs while they were still cheap to fix.

2. Never fake the loading state.

A spinner over an empty space is always better than placeholder text — every fake message we shipped cost us hours to track down and remove later.

3. Check that the LLM actually wired up every feature it drew.

The AI loved generating UI for things that had no backend. Click through every button before trusting the screen.

Looking forward, here are three features we could add.

OAuth instead of App Passwords

Right now users have to set up a Gmail App Password. OAuth would replace that with a single "Sign in with Google" click.

Background ingest worker

A background job could fetch new mail on a schedule, so the inbox is already fresh when the user opens the app.

Weekly digest

Instead of staring at 200 unread emails, the user gets one weekly message that explains what mattered and why.

Thank you.

Questions?

Code: github.com/XunSong02/OutlookPlus