GoClub

Architecture of a Location-Aware Social Platform

CS3305 Team Software Project — Team 8 University College Cork

February 2026

Abstract

This report documents the architecture of GoClub, a location-aware social platform built as part of CS3305 Team Software Project at University College Cork. It is written in the style of *The Architecture of Open Source Applications* (AOSA) — a format that asks not what a system does, but why it was built the way it was. That distinction is the lens through which every section of this report should be read.

Full service descriptions, API references, data models, and endpoint documentation are maintained as a living artefact at the project documentation site: theparliment.github.io/cs3305_2026_team_8. Readers seeking the internals of any individual service should go there. This report does not reproduce that material. It explains the thinking behind the architecture that documentation describes.

Three decisions shaped this project more than any others. First, how a team of four coordinated parallel development across five services without integration becoming a bottleneck — the answer involves Agile process, Git discipline, and documentation used as a development contract rather than a retrospective deliverable. Second, how a seven-container system was made to run identically on every developer's machine and demo reliably under pressure — the answer is Docker Compose, but the interesting part is the discipline behind it. Third, what happened when the team discovered that GPS coordinate data is fundamentally incompatible with a relational database under real-time load — the answer is Valkey, an in-memory geospatial store arrived at through deliberate research and tool selection rather than custom implementation. These three decisions are where the depth of this report lies. Everything else is supporting context.

Contents

> ** NOTE **
>
> I had a look through a few chapters of The AOSA — worth a glance if you have not seen it. The thing that struck me is that it is not an architecture spec. The system gets described, but only enough to set up the interesting part: why things were done the way they were, what went wrong, what got reconsidered. The lessons are the point. The tech is just the story they hang off. That is what I think we should aim for here. The stuff that is still worth reading in five years is not "we used FastAPI and Docker" — it is the thinking behind the decisions we made under pressure with the constraints we had. —

1 Introduction

Most student software projects, at their core, end up being a single service sitting on top of a database. GoClub is not that.

It is a location-aware social platform that depends on real-time data from active users, an event system with flexible visibility controls, a friend-circle model built around invitation state, and a wider community layer for broader group membership. All of these pieces need to work together, built by a team of four students under academic time pressure, across development environments that ranged from one laptop to another.

At the centre of the application is the idea of the **inner circle**: a small group of close contacts who share live location, making it possible to answer a simple spontaneous question — who is nearby right now, and do they want to meet? Layered on top of this are Groups (communities based on shared interests) and Events (structured gatherings with RSVP and configurable visibility, similar in concept to Eventbrite but aimed at a university setting).

However, the most interesting part of this project is not the feature set. It is the engineering decisions the team was pushed into making by three problems that turned out to be far more difficult than expected:

- coordinating parallel development across multiple services
- achieving reproducible environments and deployments
- handling high-frequency, real-time GPS data that did not fit the assumptions of a traditional relational database

This report tells the story of those three problems, and how the architecture evolved in response to them.

2 Requirements and Constraints

Intent: Establish the forces that shaped every decision so the reader can evaluate the choices that follow.

2.1 Functional Requirements

Intent: Name the five functions and flag proximity as the one that breaks the standard assumptions.

2.2 Non-Functional Constraints

Intent: Cover team size, deployment environment, privacy/security, and scope — each linked forward to where it becomes a real decision.

3 System Overview

GoClub is composed of five backend microservices, an API gateway, a shared common library, a Valkey in-memory cache, and a Jinja-templated frontend. All components are orchestrated by Docker Compose into a single deployable stack. The service decomposition maps directly to the six functional domains: Frontend, Auth, Circles, Groups, Events and Proximity.

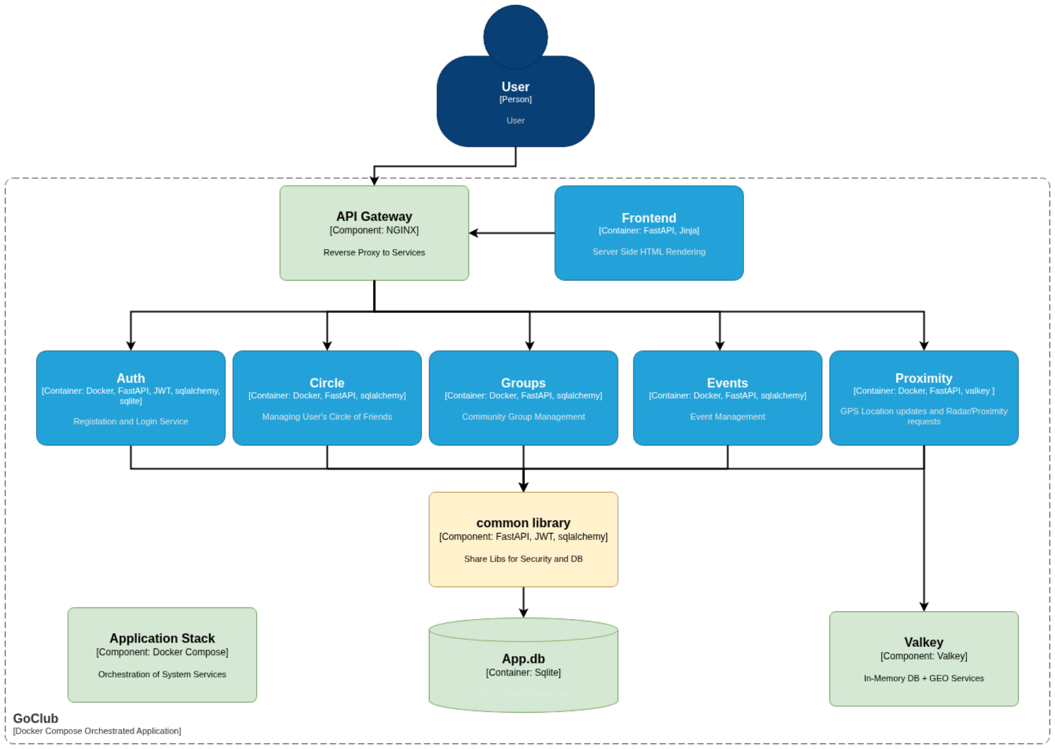

User
[Person]
User
API Gateway
[Component: NGINX]
Reverse Proxy to Services
Frontend
[Container: FastAPI, Jinja]
Server Side HTML Rendering
Auth
[Container: Docker, FastAPI, JWT, sqlalchemy, sqlite]
Registation and Login Service
Circle
[Container: Docker, FastAPI, sqlalchemy]
Managing User's Circle of Friends
Groups
[Container: Docker, FastAPI, sqlalchemy]
Community Group Management
Events
[Container: Docker, FastAPI, sqlalchemy]
Event Management
Proximity
[Container: Docker, FastAPI, valkey]
GPS Location updates and Radar/Proximity requests
common library
[Component: FastAPI, JWT, sqlalchemy]
Share Libs for Security and DB
Application Stack
[Component: Docker Compose]
Orchestration of System Services
App.db
[Container: Sqlite]
Valkey
[Component: Valkey]
In-Memory DB + GEO Services
GoClub
[Docker Compose Orchestrated Application]

Figure 1: GoClub Archiecture

Each service is a self-contained Python FastAPI application packaged as a Docker container, communicating with the outside world exclusively through the NGINX gateway. The common library is not a service; it is a shared Python package mounted into each service container at runtime, providing database session management, SQLAlchemy base models, and JWT processing utilities.

The gateway routes requests by URL path prefix: any request to `/auth/*` is forwarded to the Auth service, `/circles/*` to the Circles service, and so on. This design means adding a new service requires one additional to the NGINX configuration block.

Valkey sits alongside the relational database as a second data tier, serving exclusively the Proximity service. Its role and the reasoning behind its selection are the subject of Section XXX.

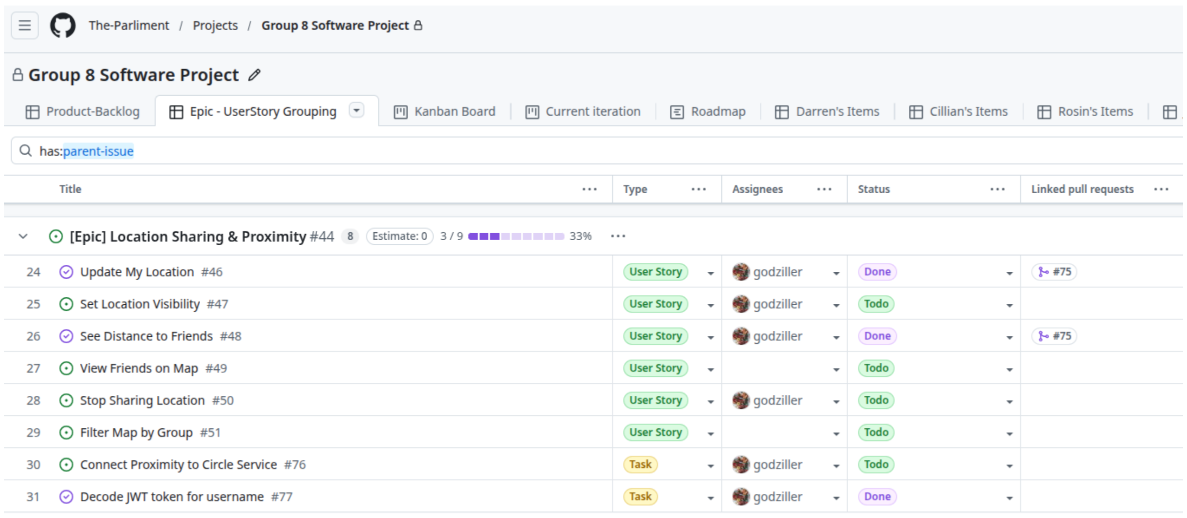
4 Challenge One — Coordinating Parallel Development

The microservices decomposition was not chosen because it is architecturally fashionable. It was chosen because the team needed to build six microservices simultaneously without each developer's work becoming a bottleneck for everyone else. That goal is only achievable if the coordination problem is solved. A microservices architecture with poor coordination is worse than a monolith: all the integration headache, none of the parallel development.

The team's response was three-layered: Agile process to align work to service boundaries, Git discipline to protect integration points, and documentation as a contract that enabled work to proceed before all services existed.

4.1 Agile Process and the Mapping of Stories to Services

The team used Google Projects to track work using epics and user stories. Effort was made to ensure story boundaries matched service boundaries. An epic for "Proximity Feature" decomposed into stories that could each be owned by a single developer: the Valkey integration, the location update endpoint, the nearby-users query, and the frontend map component. These stories could be worked on in parallel because their integration points were defined upfront as API contracts, not discovered at the end as merge conflicts.


The-Parliment / Projects / Group 8 Software Project

Group 8 Software Project

has:parent-issue

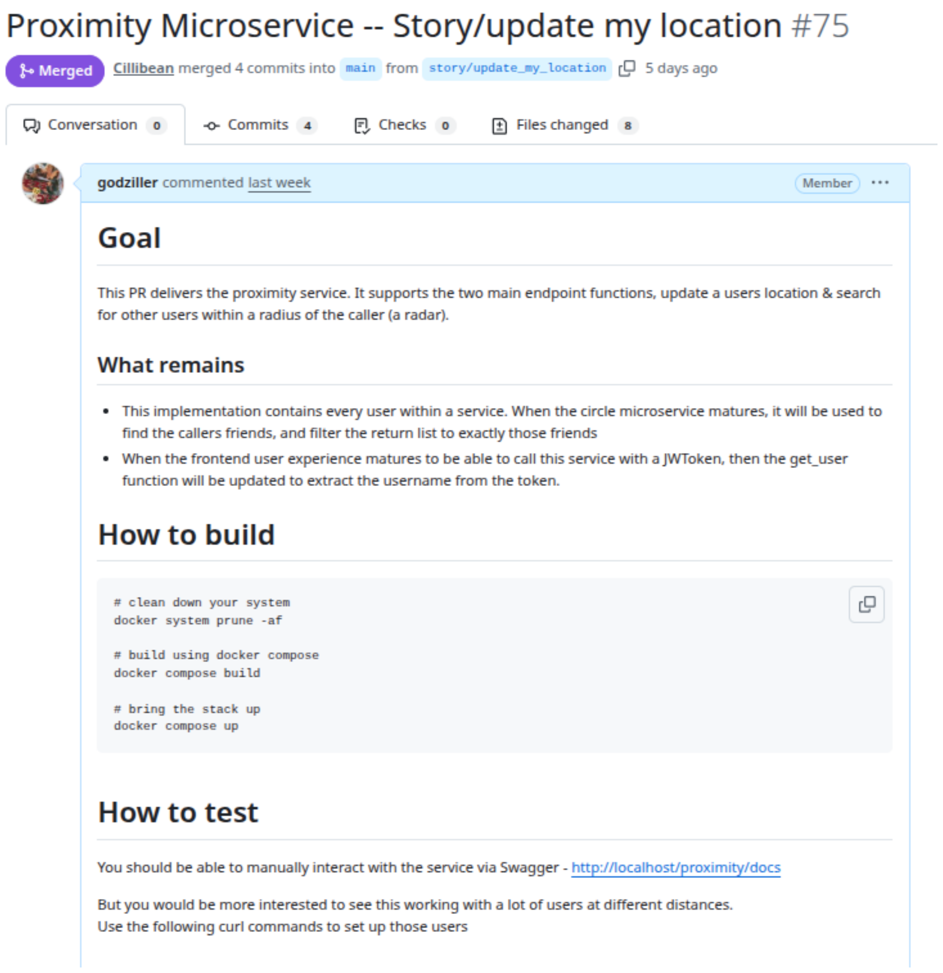
[Epic] Location Sharing & Proximity #44 8 Estimate: 0 3 / 9 33%

	Title	Type	Assignees	Status	Linked pull requests
24	Update My Location #46	User Story	godziller	Done	#75
25	Set Location Visibility #47	User Story	godziller	Todo	
26	See Distance to Friends #48	User Story	godziller	Done	#75
27	View Friends on Map #49	User Story		Todo	
28	Stop Sharing Location #50	User Story	godziller	Todo	
29	Filter Map by Group #51	User Story		Todo	
30	Connect Proximity to Circle Service #76	Task	godziller	Todo	
31	Decode JWT token for username #77	Task	godziller	Done	

Figure 2: Agile Epics and Stories

4.2 Git Discipline and the Pull Request as Integration Gate

Each story lived on its own feature branch. Code only reached main through a pull request, which acted as the team's integration gate — the point where a change went from "works on my machine" to "works in the system." With 50+ tracked issues across the project, this wasn't optional. Branch names followed the story structure, so there was always a clear line from a tracked issue to the code that closed it. Keeping this up consistently across four people over 8 weeks, especially under deadline pressure, was harder than it sounds.


Proximity Microservice -- Story/update my location #75

Cillibean merged 4 commits into `main` from `story/update_my_location` 5 days ago

godziller commented last week

Goal

This PR delivers the proximity service. It supports the two main endpoint functions, update a users location & search for other users within a radius of the caller (a radar).

What remains

- This implementation contains every user within a service. When the circle microservice matures, it will be used to find the callers friends, and filter the return list to exactly those friends
- When the frontend user experience matures to be able to call this service with a JWToken, then the get_user function will be updated to extract the username from the token.

How to build

```
# clean down your system
docker system prune -af

# build using docker compose
docker compose build

# bring the stack up
docker compose up
```

How to test

You should be able to manually interact with the service via Swagger - http://localhost/proximity/docs

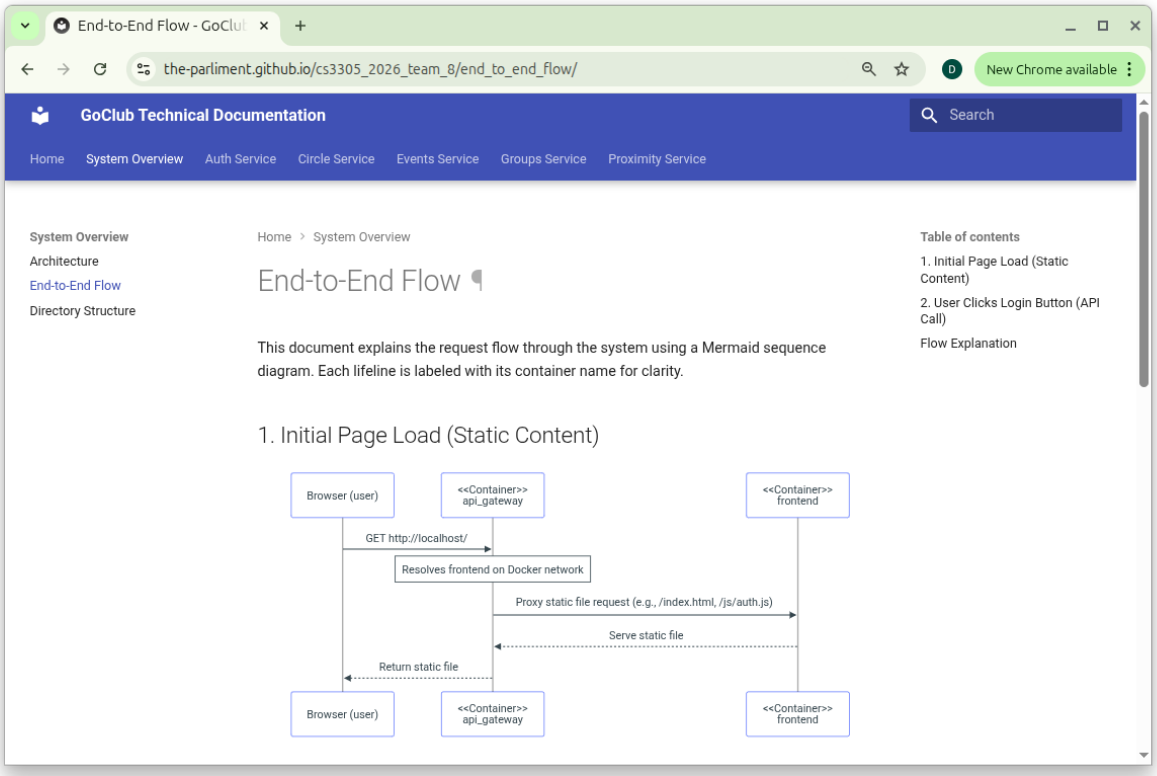
But you would be more interested to see this working with a lot of users at different distances.
Use the following curl commands to set up those users

Figure 3: Example PR

4.3 MkDocs as a Development Contract

This is the most underappreciated decision in the project's process story. The team adopted MkDocs not as a documentation deliverable to be produced at the end, but as a coordination tool used during development. Each service owner documented their API design — endpoints, request schemas, response structures — before or alongside implementation.

As the team had already been documenting APIs and designs in a docs/ folder, adding a mkdocs.yaml took minutes — but the benefits far outweighed the effort.



End-to-End Flow

This document explains the request flow through the system using a Mermaid sequence diagram. Each lifeline is labeled with its container name for clarity.

1. Initial Page Load (Static Content)

Figure 4: GoClub Website

The real power of MkDocs here was that it packaged everyone's APIs and designs into a single searchable site — one place to check rather than hunting through files or waiting on a teammate to reply. The consequence was significant: a developer building the Events service could code against the Circles API without the Circles service being complete, because the contract was written down and easily searchable. The documentation site became a substitute for a developer being physically available to answer questions. In a team working across different schedules and locations, this is not a minor convenience — it is the difference between parallel work and sequential work.

> **Key insight:** Documentation written *during* development is a design tool. Documentation written *after* development is a historical record. Only the former enables parallel work.

5 Challenge Two — Reproducible Deployment of a Multi-Service Stack

> **Intent:** Explain the deployment problem and the discipline behind the Docker Compose solution — not what Docker is, but why it mattered here.

5.1 Container-Per-Service and the Isolation Benefit

Intent: Explain what containerisation bought the team in terms of consistency, isolation, and conflict-free collaboration.

5.2 NGINX as the Unifying Gateway

Intent: Explain why a single gateway — single origin, extensible by addition — was the right choice.

```
server {
    listen 80;
    server_name _;

    # Auth service: /auth/* -> auth container, strip /auth prefix
    location /auth/ {
        rewrite ^/auth/(.*)$ /$1 break;
        proxy_pass http://auth:8000/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
```

5.3 The Two-Phase Development Loop

Intent: Describe the local-dev/full-stack-dev split and why keeping both modes viable avoided slow feedback loops.

5.4 The Nuclear Option

Intent: Frame the full state-reset command as a first-class operational tool, not an admission of failure.

```
docker compose down --volumes --remove-orphans
docker system prune -af
sudo rm common/app.db
```

6 The Core Services

Four of the five services — Auth, Circles, Groups, and Events — share a common character: they manage entities that change infrequently, have clear relational structure, and benefit from transactional consistency. This section describes each concisely, focusing on boundary decisions and architecturally notable design choices. Full API references and data model details are available at the MkDocs documentation site.

6.1 Auth Service

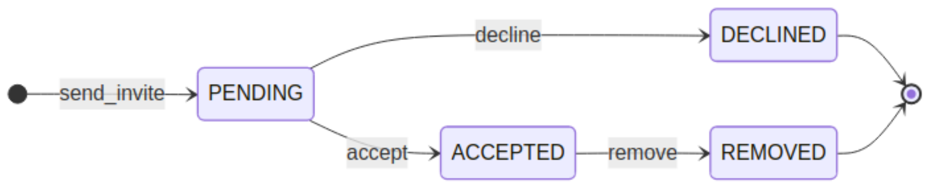
The Auth service owns user identity: registration, login, and profile management. Its core architectural responsibility is JWT issuance. Token verification is local — each service validates the JWT signature using the shared secret from the common library, without a network call back to Auth on every request. This removes both a single point of failure and a latency cost that would otherwise appear on every authenticated API call across the system.

The scope boundary is deliberate: Auth answers "who are you?" It does not answer "what are you allowed to do here?" Keeping authorisation logic in the relevant domain service means Auth stays small and focused, and avoids becoming a bottleneck as the system grows.

6.2 Circles Service

Circles are the social core of the application: small, invitation-based groups representing close friendships. The defining data model decision is the invitation state machine. A circle membership passes through explicit states — invited, accepted, declined, removed — and the service enforces valid transitions. This prevents a user from appearing in circle queries before their invitation is accepted.

The circle boundary is also the privacy boundary for location sharing. Onlyaccepted circle members can see each other's positions in the Proximity service. The Circles service is the authority on membership; the Proximity service delegates that question rather than reimplementing it.



6.3 Groups Service

Groups serve a different social function from Circles. Where a Circle is intimate and invitation-only, a Group is a community of shared interest that can be public or private. The distinction is set by the creator at the time of group creation.

A public group is discoverable by any user via `/list` and open to join without any approval — any authenticated user can add themselves.

A private group is invisible to non-members entirely; it does not appear in discovery at all. Rather than a request-and-approve flow, membership in a private group is entirely owner-controlled: only the creator can add users via `/join` on their behalf. This keeps the permission model simple — there is no pending state, no invite negotiation, the owner either adds someone or they are not in the group.

The same owner authority extends to removal. The `/removemember` endpoint is restricted to the group owner — members cannot remove each other, only the creator can manage the roster. This binary owner/member model is intentionally lightweight for the scope of the project; a production system might introduce moderator roles, but for GoClub the distinction was sufficient.

The public/private flag also propagates into the Events service: an event scoped to a group is only visible to its members, so the Events service must resolve group membership at query time. The Groups service remains the single source of truth for membership, and the Events service defers to it rather than duplicating that logic.

6.4 Events Service

Roisin to add…….

6.5 Frontend Service

Cillian to add - benefit of server side html rendering with jinja

---

7 Challenge Three — When the Database Is the Wrong Tool

Intent: Tell the story of discovering that GPS data does not fit a relational store, characterising the mismatch precisely, and finding Valkey as the purpose-built solution.

7.1 The Problem with Location in a Relational Database

Intent: Identify the three specific mismatches precisely — this is the diagnosis that justifies everything that follows.

Three specific mismatches identified: high write frequency against a single-writer store; spatial calculation against a relational query engine; volatile state forced into durable storage. Each pointed toward the same conclusion: location data needs a different tool.

7.2 The Research Process and the Discovery of Valkey

Intent: Show how characterising data properties first led to finding an existing tool rather than building a custom solution.

7.3 The Resulting Two-Tier Data Architecture

Intent: Name the pattern and summarise the role of each tier.

The transferable lesson: Valuable engineering move in this project was the decision not to write code. Recognising that a problem has prior art — and knowing how to search for it — is a more valuable skill than the ability to implement a solution from scratch.

---

8 Cross-Cutting Concerns

Intent: Cover the two elements that span all services and explain the trade-offs of each choice. Perhaps we present the schema?

8.1 The Common Library

Intent: Justify centralised shared code as the right coupling decision at this scale, while honestly acknowledging the trade-off.

8.2 JWT and Distributed Authentication

Intent: Explain local token verification and the benefits as a shared library - aka the verify & decode commonality.

---

9 Future Improvements

Intent: Record chapter per future work — not as apologies.

9.1 User Lifecycle Event Bus and DB per service

Intent: Identify the user-deletion cleanup problem and describe an event bus as the correct solution using existing infrastructure.

9.2 Priority Two API Completion

Intent: Note that descoped work is documented and intentional, not a gap.

---

10 Lessons Learned

10.1 On Deployment: Make the Environment a First-Class Artefact

Intent: Argue that containerising early and treating docker-compose.yml as canonical saved the team from an entire class of failures.

---

11 Conclusion

Intent: Close by arguing that the problems encountered — and the thinking required — are the same ones that appear at any scale.

- **GitHub:** https://github.com/The-Parliment/cs3305_2026_team_8
- **Documentation:** https://the-parliment.github.io/cs3305_2026_team_8/