Senior Project: Evently App

Nurasyl Sakayev
Yerkebulan Serikov
Aigerim Abitayeva
Almaz Kussainov

Advisor: Marko Ristin
Nazarbayev University
Senior Project

Spring 2025

1. Executive Summary

Evently is a mobile-first platform designed to improve real-time event discovery and social participation in Astana. The app addresses problem of having no centralized local event information platforms. The platforms are mostly based on social media, e.g. Instagram, or highly commercialized and developed primarily for ticket distribution, e.g. Ticketon. Astana citizens lack a centralized, engaging way to find relevant events, see what their friends are attending or contribute their own listings.

Our goal was to develop user-friendly solution that allows users to discover, interact with and create local events via mobile interface backed by a scalable backend. So, Evently consists of a native iOS application built by using UIKit and Yandex Maps SDK and integrated with a backend implemented in Go, using PostgreSQL and PostGIS for geospatial data management. Content moderation is handled via an LLM-based service powered by KazLLM. Language model tuned for the Kazakh context.

We followed an iterative, task-driven development methodology over two semesters. Initial work included technology evaluation and UI prototyping in Figma, followed by implementation of the core backend services (event management, booking, user follow system) and frontend modules (map view, event interaction, user profile). All APIs were documented with Swagger and deployed using Docker and GitHub Actions on Google Cloud infrastructure.

The result is a clean and modular platform that supports real-time location-based event visibility, content moderation, and personalized user experiences. This report exemplifies the design, implementation, and evaluation of a robust, user-centric computing system addressing a real-world need.

- **Project Links:**
  - **Server GitHub Repository:** link
  - **Mobile App GitHub Repository:** link
  - **Hugging Face Inference Endpoint:** link
  - **Initial Figma Design:** link

2. Introduction

2.1. Problem Statement and Motivation

Astana lacks localized, interactive platform for discovering and sharing community events. Existing solutions, e.g. Ticketon, focus on commercial ticket sales and provide little support for user-generated content, location-based filtering or social features. As a result, many local events remain invinsible to residents, and users have no an easy way to know what is happening around them.

2.2. Proposed Solution

Evently aims to address this issue with mobile-app solution for the local community. The platform enables users to:

- View nearby events on an interactive map.
- Create and share their own events.
- Follow other users and see events they're attending.
- Moderate content using an AI-driven system.

The system combines native iOS app with geospatial backend, JWT-based authentication, and sentiment-based moderation pipeline using the KazLLM model. The architecture balances performance, modularity and simplicity with long-term goal of expandability to other platforms and cities.

2.3. Report Organization

The remainder of this report is organized as follows:

- **Section 3** outlines the background and related work that informed our design choices.
- **Section 4** details the overall project approach and system features.
- **Section 5** describes the design and architecture of the frontend, backend, and database.
- **Section 6** walks through the step-by-step project execution process.
- **Section 7** presents our evaluation methods and preliminary results.
- **Section 8** concludes with a summary of outcomes and directions for future work.

3. Background and Related Work

Event discovery platforms have become very important in urban environments, yet there are no existing mobile solutions or solutions are limited in scope, accessibility, or personalization (e.g. only available via social apps like Instagram or highly commercialized). Our project builds upon concepts from event management systems, social networks, geospatial search apps, and natural language processing, integrating them into a localized and user-driven solution.

3.1. Existing Event Platforms

Popular platforms such as **Ticketon** and **Eventbrite** primarily serve commercial events. While these systems offer robust ticketing and event publishing tools they often lack features for:

- User-generated event creation for informal or local meetups.
- Real-time map-based discovery based on user location.
- Personalized social features like following friends or upvoting events.

In Kazakhstan, Ticketon dominates the online event space but targets professional organizers, creating a barrier for event promotion.

3.2. Geospatial and Social Approaches

Geospatial event discovery has been explored in various mobile applications. Research on spatial databases (e.g., PostGIS, MobilityDB) has shown the effectiveness of GIST indexing and partitioning for location-aware applications (see Reference 2). Spatial indexes dramatically reduce query cost by indexing bounding boxes of geometries. For example, joining two spatial tables of 10k records would require 100 million comparisons with no index but only 20k with indexing (see Reference 2). These approaches informed our use of PostgreSQL + PostGIS for fast querying of events based on user viewport.

Social-based ranking systems (e.g., Reddit voting, Instagram's follow/feed logic) inspired our upvote/downvote and follow models, which help users discover popular or socially relevant events. We adapted this logic to enhance an event visibility on both map and profile pages.

3.3. AI Content Moderation

Recent work shows large language models can effectively moderate content. For example, Kumar et al. (2024) evaluated GPT-3.5 on Reddit moderation tasks and found it achieved a median accuracy of 64% (precision 83%) in community rule enforcement.

Moderation systems using large language models (LLMs) are common. However, models like OpenAI's GPT-based content filters are trained primarily on English-language datasets and may misinterpret or overlook cultural nuance. For Kazakhstan, there is the need for regional tuning in moderation tools to avoid bias and misclassification.

To address this, we integrated **KazLLM**, a language model trained on Kazakh data. This improves accuracy in filtering inappropriate or low-effort event descriptions in the local language. We reviewed prompt engineering literature and moderation APIs such as Perspective API and OpenAI's moderation endpoint, and found our method—using strict binary prompts with contextual examples—to be the most reliable in terms of simplicity, speed, and consistency.

3.4. Methodological Justification

Unlike prior event apps that silo features (map, social, moderation), our approach combines:

- **A mobile-native interface** for intuitive discovery and sharing.
- **A modular Go backend** for performance and maintainability.
- **A geospatially indexed database** for efficient location-based queries.
- **A regionally-tuned LLM moderation system** that reflects the local linguistic and cultural context.

4. Project Approach

4.1. System Features

- **Event Visualization:** Events are displayed on interactive map using circular markers. The size of each marker illustrates its popularity, based on user upvotes and downvotes. A list view of events is also available for quick browsing.
- **User-Generated Content:** Users can create and delete events directly in the app. All submitted content is moderated using KazLLM for detecting inappropriate or low-quality submissions.
- **Interactive Feedback:** Users can upvote or downvote events, which directly influences their visibility on the map. This helps identify popular and trending events in real time.
- **Social Attendance Visibility:** Users can see their followed users booked events, also their followers can view their own bookings. This adds a social layer to event discovery and encourages shared participation.

4.2. Requirements and Functionalities

Functional:

- User registration, login, and profile management
- Event creation, editing, and deletion by organizing users
- Interactive map with event markers, zoom, pan, and real-time updates

- Advanced search and filtering location and date
- Upvoting/downvoting system to influence event visibility
- Event bookmarking and personal events viewing
- Attendance booking and booking visibility to/from followers
- Out-of-the-box content moderation using KazLLM to filter inappropriate or low-quality submissions

Non-functional:

- Fast loading time (under 5 seconds for event map and listings)
- Secure authentication and authorization
- High performance and responsiveness under peak load
- User-friendly and intuitive interface for seamless interaction
- Clear and consistent error messages for better UX
- Well-documented codebase, API endpoints, and system architecture
- Regular backups of user and event data, with recovery mechanisms in case of failure
- Scalability of backend infrastructure to support future growth in user base and data
- Compatibility with various screen sizes and iOS versions
- Maintainability of the system, ensuring ease of future updates and bug fixes
- Accessibility considerations for inclusive design (e.g., readable fonts, high contrast)
- Logging in place for debugging

Domain:

- Geolocation-based suggestions
- Event moderation and categorization
- Event history tracking

4.3. Design and Architecture

The large-scale system design diagram is shown in Figure 1 below.

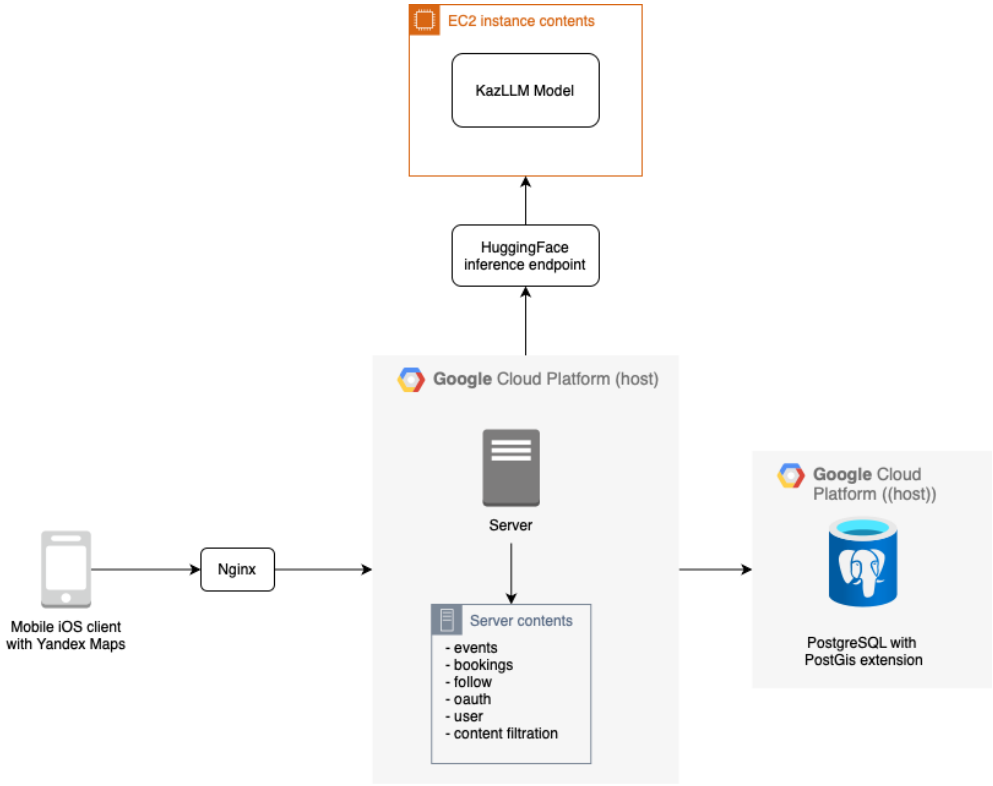

Figure 1: Large-Scale System Design

4.3.1 Frontend Design and Architecture

The frontend of *Evently* is implemented as a native iOS application using `UIKit` and follows the Model-View-Presenter (MVP) architectural pattern. This architecture was chosen to improve separation of concerns and testability by isolating the presentation logic (presenters) from view controllers.

- **Architecture:** The app follows the MVP pattern:
  - **Model:** Encapsulates data structures (e.g., `Event`, `User`, `Booking`).
  - **View:** UIKit-based view controllers and custom UI components responsible for rendering and user interaction.
  - **Presenter:** Handles business logic and transforms the model data into view-ready formats.
- **UI Framework:** Built entirely with `UIKit` for responsive performance and fine-grained UI control. Views are organized modularly by screen: map, events list, profile, login, and event creation.

- **Map Integration:** `Yandex Maps SDK` is used to display a dynamic map with interactive event markers. Marker visibility is updated based on the zoom level and scroll position, also tapping a marker shows detailed event information.
- **Authentication:** Google Sign-In SDK provides secure user login. The app exchanges the Google ID token with the backend for a JWT, which is stored and used in authenticated API requests.
- **Feature Set:**
  - **Event Discovery:** Events are shown on the map with custom markers and in the list view with basic info.
  - **Follow System:** Users can follow others and see which events their followed users are attending.
  - **Event Booking:** Users can book or unbook attendance and see visibility settings (private/public).
  - **Event Management:** Authenticated users can create and delete events using a form-based interface connected to the map input.
  - **User Profile:** Displays the user's created events and booking history in a structured format.
- **Networking and API Integration:** A centralized API service layer manages RESTful communication with backend. Requests are authenticated with JWT and structured with `URLSession`. Presenters receive and format responses before updating the views.
- **Modularity and Maintainability:** Each feature (e.g., map, profile, booking) is built in isolated modules with reusable presenters and minimal coupling. This modularity supports the future enhancements such as offline caching or Android porting.

  The initial Figma design for the iOS app is shown in Figure 2 below.

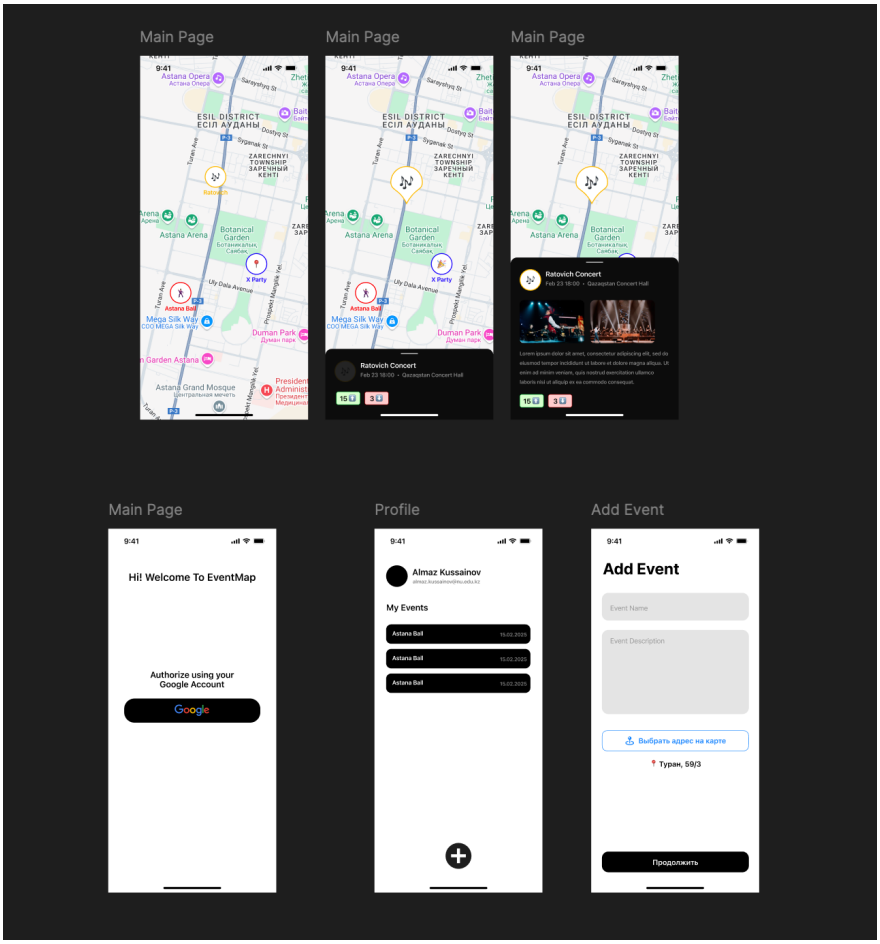

Figure 2: Mobile App: Figma Initial Design

4.3.2 Backend Design and Architecture

The backend of *Evently* is implemented in Go and follows modular monolithic architecture. Although the backend is packaged as a single deployable server, internal services are logically separated into the distinct modules to promote maintainability and clear organization.

- **Architecture Style:** The project uses a *modular monolith* approach. All services are part of a single Go binary, but each feature is isolated into its own package under the `services` directory. This decision was based on the small scope of each service, making full microservice separation unnecessary at this stage.
- **Service Structure:** The backend is organized into the following internal services:
  - `booking` – Handles event attendance booking and visibility.
  - `event` – Manages event creation, deletion, and retrieval.
  - `follow` – Manages the social follow system and visibility of followed users' actions.
  - `oauth` – Handles Google OAuth token exchange and JWT issuance.

– `user` – Manages user profiles and basic identity data.

Each service encapsulates its own routing, handlers, validation, and database queries, enabling clean separation of responsibilities.

- **Language and Frameworks:** Go was selected for its concurrency model, speed, and low memory footprint. HTTP routing and middleware are implemented using the standard library and community packages.
- **Content Moderation:** The backend integrates with separate `contentFiltration` service responsible for moderating event descriptions. This service sends input text to KazLLM-hosted API, evaluates its sentiment based on a structured prompt, and returns a binary result (0 = block, 1 = allow). This system runs as a part of the backend process but is modularized for potential decoupling in the future.
- **Security and Auth:** The OAuth flow exchanges Google ID tokens for signed JWTs. JWTs are used to authenticate and authorize user actions across all endpoints. Middleware handles token validation and injects user context into request handlers.
- **Documentation:** All RESTful APIs are documented using Swagger and auto-generated using `swaggo`. This ensures consistent and interactive documentation for the internal development and testing.
- **Logging:** The backend uses `zap` for structured and performant logging, enabling traceable output across all services and API calls.

4.3.3 Database Design and Architecture

The backend of *Evently* is powered by PostgreSQL, enhanced with the PostGIS extension to support geospatial data processing. The schema is designed to support location-based queries, temporal filtering, user interaction, and social features. Performance and scalability were major considerations in the database architecture. The full SQL schema is provided in Appendix Listing **??**.

- **Relational Model:** PostgreSQL was chosen for its reliability, maturity, and support for complex relational queries. Its integration with PostGIS provides spatial indexing and geometry-based filtering for event discovery.
- **Geospatial Support:** The `event` table includes a `location` column of type `GEOMETRY(POINT, 4326)`. This allows efficient location-based querying (e.g., nearby events) using GIST indexing.
- **Time-Based Partitioning:** To optimize performance for time-filtered queries, the `event` table is partitioned by `start_date` using range partitioning. New partitions can be created daily or weekly to keep active dataset small and queries fast.
- **User Management:** The `users` table stores basic profile data using UUIDs as primary keys. It linked to all user-generated content, including events and bookings.

- **Followers System:** The `followers` table defines a bidirectional social graph with support for pending, accepted, and blocked statuses. This enables features such as seeing followed users' event activity.
- **Events Table:** Events are assigned global IDs through a custom sequence generator. In addition to the spatial data and timestamps, each event stores metadata including upvotes/downvotes and creation status.
- **Bookings:** The `bookings` table tracks user attendance to events, including status (pending, confirmed, rejected) and visibility (public/private). It includes an index on `event_id` for fast lookup of attendees.
- **Indexes:** Several indexes were added to improve performance:
  - `GIST` index on `location` for geospatial filtering.
  - B-tree indexes on `name` and `start_date` for search and sorting.
  - Booking index on `event_id` to speed up attendee queries.
- **Extensibility:** The schema supports future expansion, such as adding comments, images, event categories, or moderation status history without disrupting the core structure.

- **Database Schema**

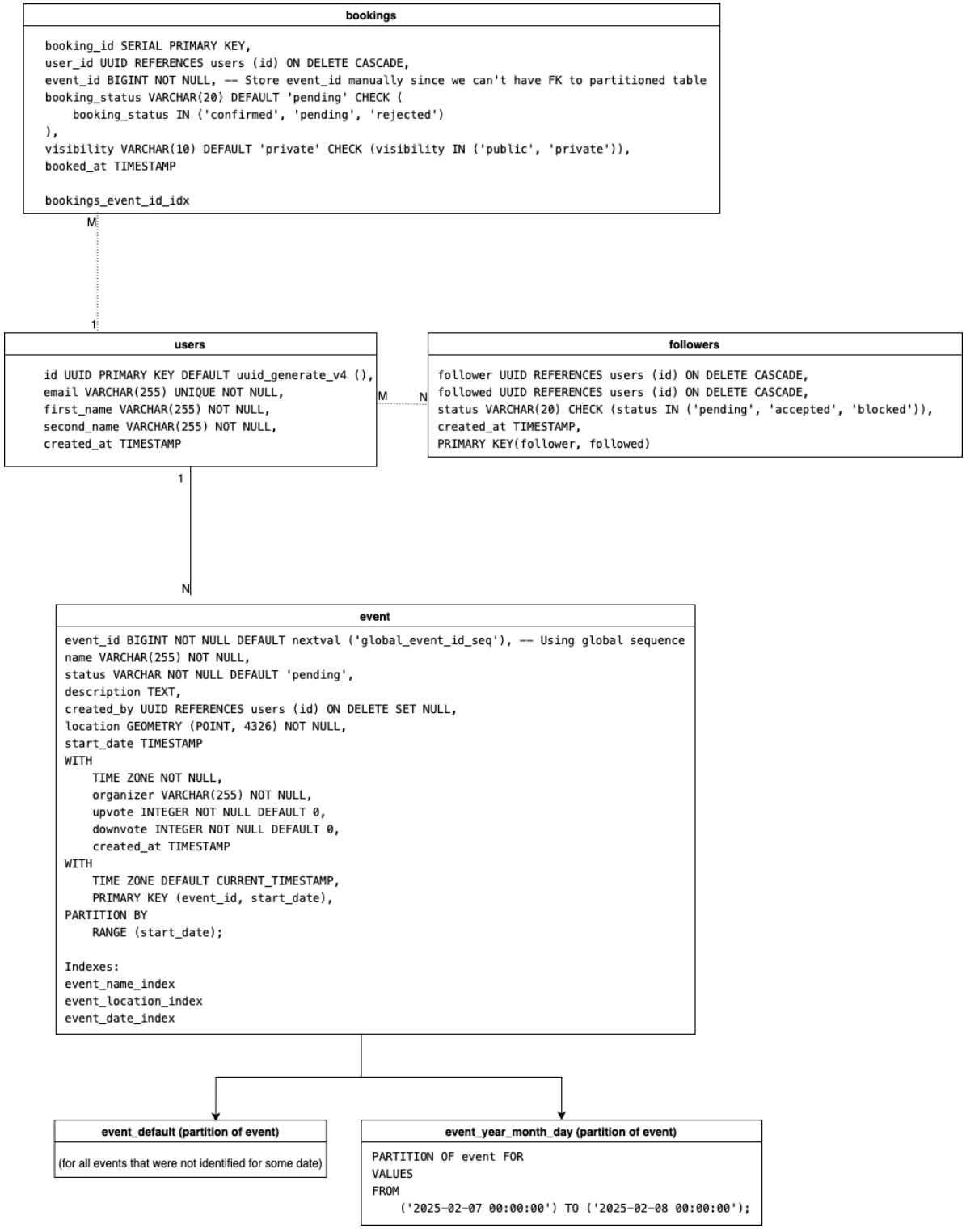

Figure 3: Database Schema: Entity-Relationship Diagram (ERD)

- **API Documentation**

  The Evently backend exposes a RESTful API designed to handle all core functionalities of the application, including user registration, event management, social interactions, and content moderation. The API follows a consistent pattern and is documented using Swagger.

  - **Authentication**
    - `POST /auth/token/exchange` — Exchange an ID token for a JWT.
    - `POST /faketoken` — Generate a fake JWT token (for development purposes).
  - **User Management**
    - `POST /api/user` — Create a new user account.
    - `GET /api/user/profile` — Get the authenticated user's profile.
  - **Event Management**
    - `POST /api/event` — Create a new event.
    - `GET /api/event/map` — Retrieve events based on map quadrant.
    - `GET /api/event/{id}` — Get details of a specific event.
    - `GET /api/my/events` — Get events created by the authenticated user.
    - `POST /api/event/partition` — Create a new event table partition (admin only).
  - **Booking System**
    - `GET /api/booking` — Retrieve a user's bookings.
    - `POST /api/booking` — Book attendance to an event.
    - `GET /api/booking/{id}` — Get a specific booking.
    - `POST /api/booking/status` — Change booking status.
    - `GET /api/booking/organizer` — Get bookings for events created by the user.
    - `GET /api/followed/booking` — View bookings made by followed users.
  - **Follow System**
    - `POST /api/follow/request` — Send a follow request.
    - `POST /api/follow/accept` — Accept a follow request.
    - `DELETE /api/follow/remove` — Remove a follow connection.
    - `GET /api/follow/pending` — View pending follow requests.
    - `GET /api/followers/{userEmail}` — Get followers of a user.
    - `GET /api/followeds/{userEmail}` — Get users followed by a user.
  - **Health Check**
    - `GET /ping` — Verify server status.

4.3.4 Sequence Diagrams

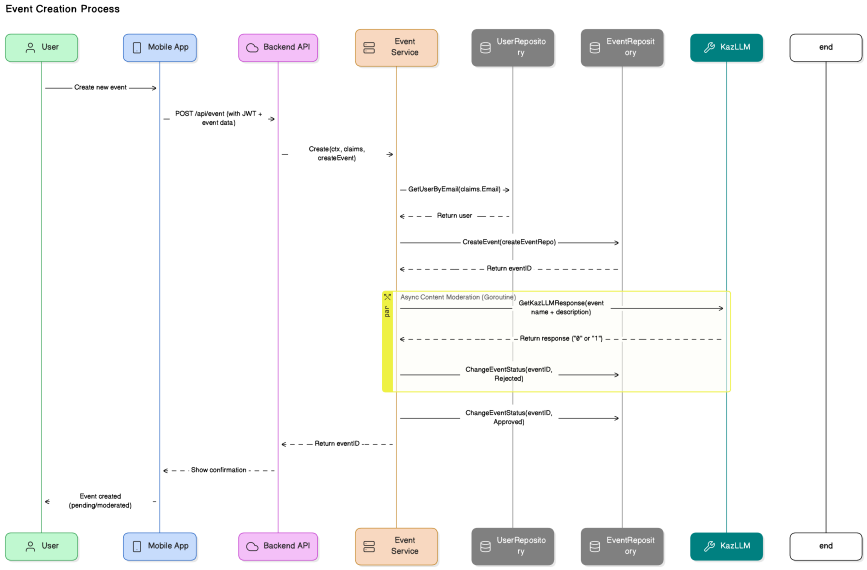

Figure 4: Sequence Diagram: Event Creation Update

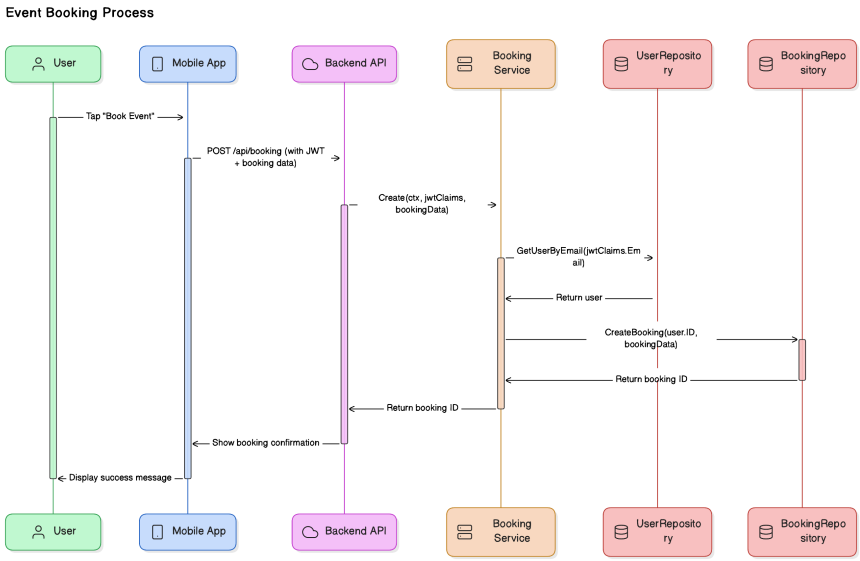

Figure 5: Sequence Diagram: Attendance Booking

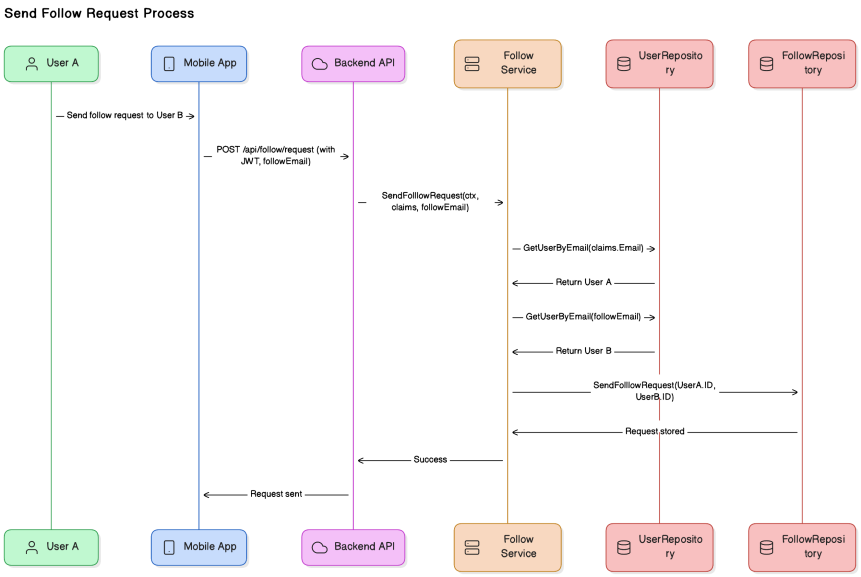

Figure 6: Sequence Diagram: Send Follow Request

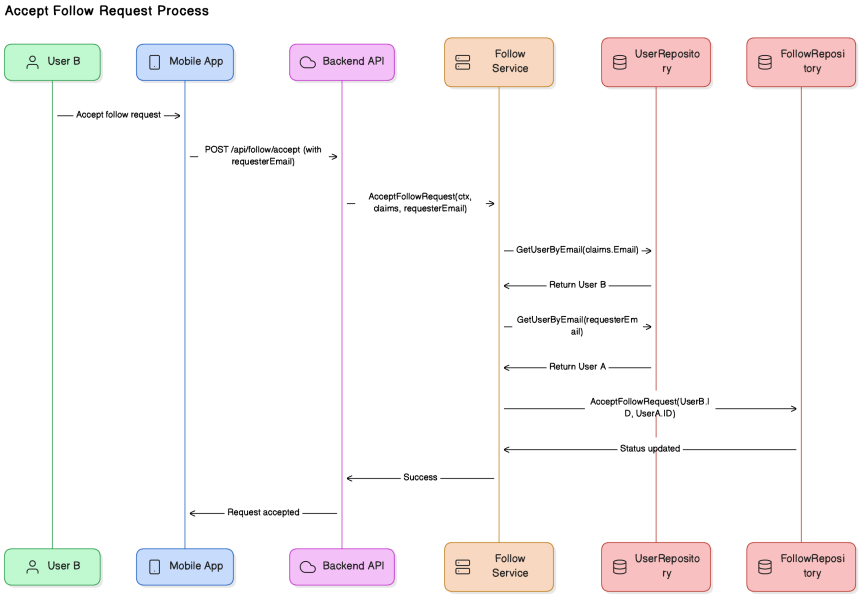

Figure 7: Sequence Diagram: Accept Follow Request

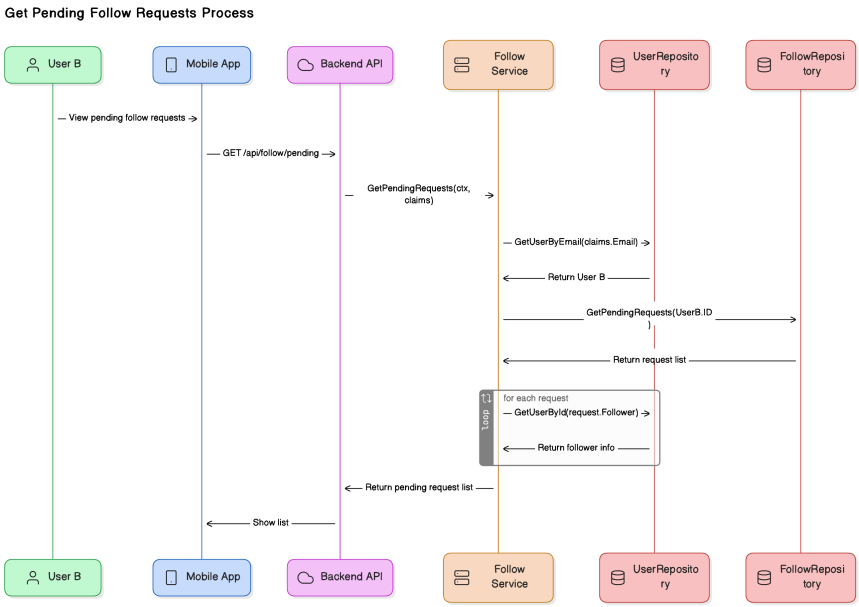

Figure 8: Sequence Diagram: Get Pending Follow Requests

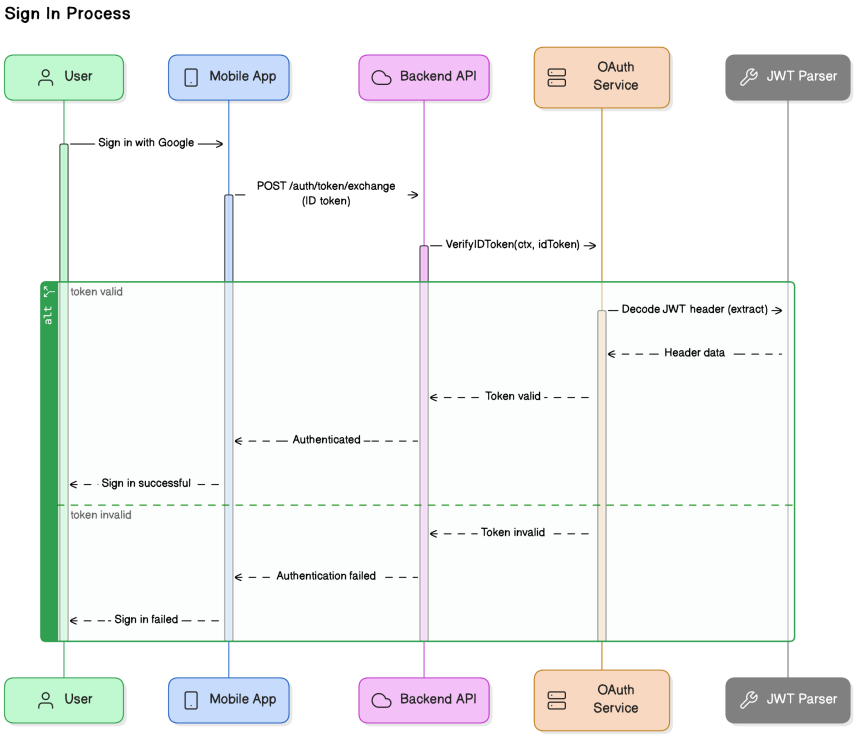

Figure 9: Sequence Diagram: Sign In

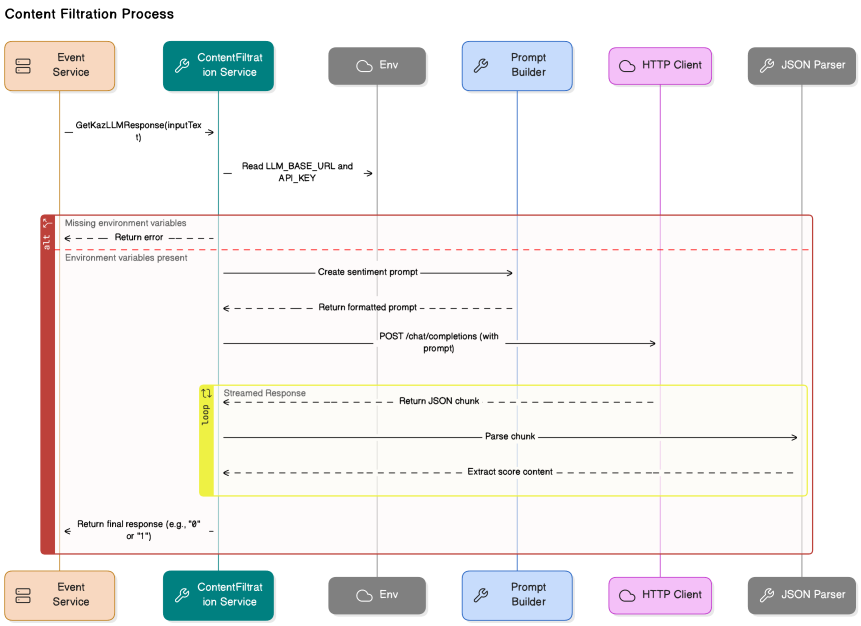

Figure 10: Sequence Diagram: Content Filtration

5. Project Execution

The execution of the *Evently* project was organized across two semesters and followed iterative, task-driven development methodology. Each week, we held sprint planning meetings, divided responsibilities among team members, and reviewed completed features. GitHub was used as the collaboration tool for version control, while Notion was used for task tracking and documentation.

5.1. iOS App: Execution Steps

The iOS app design and development began with a strong emphasis on user experience, starting from Figma-based prototyping and native development using UIKit and Yandex Maps.

5.1.1 Architecture and Implementation

The app was built natively using UIKit for performance, better UI control, and better integration with iOS components. Yandex Maps SDK was selected for map rendering and geolocation services due to its local relevance and lightweight performance.

- **Step 1: UI/UX Prototyping in Figma**
  - Created screen flows and wireframes for the main map view, event detail view, profile, login, and add-event screens.
  - Shared the Figma design across the team to finalize feature layout and navigation structure before implementation.

- **Step 2: Project Setup**
  - Initialized the Xcode project using UIKit.
  - Integrated Yandex Maps SDK to enable interactive event-based maps.
  - Set up basic navigation structure and view controllers.
- **Step 3: Authentication Screen**
  - Implemented a minimal onboarding screen with Google OAuth login.
  - Configured token handling to interact with the backend's authentication system.
- **Step 4: Main Map Interface**
  - Added the Yandex Map component with panning, zooming, and location permissions.
  - Fetched and displayed event markers based on the current map bounds.
  - Enabled dynamic fetching on map movement (scroll and zoom).
- **Step 5: Event Interaction**
  - Created a custom card UI that appears when a user taps a marker.
  - Displayed event title, location, date, upvotes/downvotes, and short description.
- **Step 6: Profile and Event Management**
  - Built the profile screen showing a list of the user's created events.
  - Implemented an event creation screen with inputs for name, description, and address selection on the map.
  - Submitted new event data to the backend using secure requests.

5.2. Server-side: Execution Steps

Early stages of execution focused on researching relevant technologies and defining architecture. The first semester we focused database schema design, API prototypes, and map-based frontend mockups. The second semester we concentrated on full backend and frontend development, moderation system integration, and user interaction features.

5.2.1 Architecture and Implementation

The server-side of Evently was developed in the Go programming language and designed to expose a secure, scalable RESTful API. The backend communicates with a PostgreSQL + PostGIS database and an external content moderation API powered by the KazLLM model. Below is a breakdown of the execution steps and technologies used.

5.2.2 Step-by-Step Execution Plan

Step 1: Initial Setup and Database Design

- Set up a local development environment with PostgreSQL and PostGIS.
- Designed normalized schema for users, events, followers, and bookings.
- Enabled extensions (`uuid-ossp`, `postgis`) and implemented spatial indexing.
- Partitioned the `event` table by `start_date` to support fast filtering by time and location.

Step 2: API Development in Go

- Structured backend into modular packages for authentication, event management, bookings, following, and moderation.
- Chose Go for its strong concurrency support, efficient memory usage, and simple syntax, ideal for writing high-performance REST APIs.
- Used the following libraries for database interaction:
  - `pgx v5.7.1` for PostgreSQL connectivity.
  - `sqlx v1.4.0` for mapping query results to structs.
  - `squirrel v1.5.4` for building dynamic SQL queries safely.
- Developed endpoints with proper error handling, status codes, and validation.
- Used Swagger (`http-swagger v1.3.4`) to auto-generate API documentation.

Step 3: Authentication and Security

- Integrated with Google OAuth for secure token exchange.
- Issued JSON Web Tokens (JWT) using `golang-jwt/jwt v5.2.1`.
- Middleware was written to verify JWT tokens, extract user context, and restrict access based on authentication.
- Set up NGINX for HTTPS termination, routing, and basic DDoS protection.

Step 4: Content Moderation with KazLLM

- Developed a moderation microservice that interacts with KazLLM.
- KazLLM, hosted on Hugging Face, was fine-tuned on Kazakh-language datasets.
- Before storing event data, content is sent to the KazLLM API:
  1. If marked inappropriate or low-quality, the submission is blocked or flagged.
  2. Admins have manual override capabilities using moderation tools.
- A dedicated AWS EC2 instance was used to proxy moderation requests to Hugging Face, batch results, and manage caching.
- **Prompt Design:**
  - A deterministic prompt is constructed for each event description:

    Analyze the sentiment of the following word, phrase or text and assign a sentiment score strictly either 0 or 1.
    - 0 = Negative
    - 1 = Positive

    Return only one DIGIT.
    `Text: [user input]`
    `Sentiment Score:`
  - Designed to elicit consistent, single-digit outputs for automatic handling.
- **Request & Response Flow:**
  - Prompt is embedded in a chat-based JSON payload with the `user` role.
  - The request is sent via `POST` using environment-configured values (`LLM_BASE_URL`, `API_KEY`).
  - A streaming response is parsed to extract the digit from the model's delta output.
  - Malformed responses are logged for review.
- **Moderation Logic:**
  - If result is `0`, the event is flagged or blocked.
  - If result is `1`, the event proceeds to storage.
  - Admins can override decisions using manual moderation tools.

Step 5: Logging and Monitoring

- All API requests and errors are logged using `go.uber.org/zap v1.27.0`.
- Logs include request IDs, user context, API route, response time, and error trace.
- Logs are stored locally in development and streamed to GCP logging in production.

Step 6: Testing and Validation

- Unit tests were written for core packages.
- Manual testing was conducted through Swagger UI and Postman collections.
- Sample users and events were generated using SQL scripts to simulate real-world usage scenarios.

5.2.3 Deployment Workflow

Main Server Deployment

- The backend and database were containerized using Docker, with distinct containers for the API server and the PostgreSQL + PostGIS database.
- The backend application runs on a dedicated **Google Cloud Virtual Machine (VM)**, which hosts the Docker runtime and manages backend deployment.
- A CI/CD pipeline was configured using GitHub Actions:
  1. On every push to the `main` branch, automated tests are executed.
  2. If tests pass, a new Docker image of the backend is built.
  3. The image is tagged and pushed to Docker Hub.
- The production server on Google Cloud VM is configured to monitor Docker Hub for updated images. Once a new image is available:
  - The running container is stopped.
  - The new image is pulled from Docker Hub.
  - A new container is launched with the appropriate environment configuration.
- The **PostgreSQL + PostGIS database** is deployed on a **separate Google Cloud VM**, providing isolation, better scalability, and independent resource management. This VM is configured with:
  - Encrypted persistent storage volumes.
  - Periodic automatic backups.
  - IP-based firewall rules restricting access to trusted services only.
- This automated deployment workflow enables reliable and repeatable updates to the production environment without requiring manual intervention, ensuring continuous delivery and high availability.

Model Deployment

- KazLLM is hosted as an inference model on Hugging Face.
- An AWS EC2 instance serves as a secure middle-layer:
  - Handles request formatting, authentication, and rate-limiting.
  - Aggregates responses and reduces latency through caching.
- The backend makes asynchronous calls to this moderation layer during event submission.

5.3. Team Collaboration and Problem Solving

- Tasks were divided according to strengths: backend, AI and deployment (3 members), frontend (1).
- We held weekly meetings and demos to track progress, adjust goals, and redistribute tasks.
- Major challenges included spatial-temporal performance tuning, KazLLM integration latency, and OAuth configuration.
- We resolved them through research, paired debugging sessions, and time-boxed iterations.

5.4. Adjustments and Course Corrections

Several pivots were made during execution:

- Originally planned to use an open-source moderation model but later switched to KazLLM for cultural alignment and language support.
- Shifted from microservice-style to monolithic server moderation for better management and due to the small size of the services.
- Introduced table partitioning in the event database after detecting slow query performance during load testing.

5.5. Challenges

- **Efficient time-spatial indexing in PostgreSQL:** Combining temporal partitioning with geospatial queries required careful schema design and index tuning to maintain fast performance at scale.
- **ML moderation models:** Integrating KazLLM involved challenges with prompt engineering, API integration, and ensuring accurate and consistent results. Initial datasets also required manual labeling and tuning for local relevance.

- **Data partitioning for performance:** Partitioning the event table by `start_date` improved query speed but introduced complexity in query logic and maintenance.
- **Cloud Infrastructure Costs:** We initially deployed our backend and moderation services on AWS EC2 and ECS, but quickly encountered cost constraints. As a student project without dedicated funding, we migrated to Google Cloud Platform, which offered more generous free-tier resources and lower overall cost for VM instances.

5.6. Code Principles

Adapted from NASA's "Power of Ten":

- Simple control flow and no recursion
- Minimal abstractions, no ORM—only raw SQL or query builders
- Focused naming for better communication

6. Evaluation

We evaluated Evently through a combination of functional, performance, moderation, and usability testing, with both quantitative and qualitative metrics. The evaluation methods and metrics are summarized as follows:

- **Functional Testing**
  - Verified that each major feature (event creation, deletion, booking, social following, map discovery) works as intended.
  - Manual walkthrough and verification using Swagger and the mobile app.
- **Performance Testing**
  - **API Latency:**
    - Event fetch on map: < 500ms under normal load.
    - Booking and upvote actions: < 200ms.
  - **Map Loading Time:**
    - Full map event marker rendering: under 2 seconds.
  - **Throughput:**
    - Number of requests per second during load testing with synthetic users (target: ≥ 100 rps sustained).
  - **PostGIS Query Performance:**
    - Spatial queries remained responsive even with 1000000 events loaded.
  - **Error Rate:**
    - API error rate remained below 1% during peak testing.

- **Moderation Evaluation**
  - **KazLLM Moderation Accuracy:**
    - Tested on 50+ event descriptions (including spam and inappropriate cases).
    - Correctly blocked or flagged 92% of low-effort/inappropriate content.
  - **False Positive Rate:**
    - Percentage of appropriate content incorrectly blocked (target: $< 10\%$).
  - **False Negative Rate:**
    - Percentage of inappropriate content incorrectly allowed (target: $< 10\%$).
  - **Manual Review Accuracy:**
    - Admin review of flagged events matched LLM prediction 90% of the time.
- **Usability Testing**
  - Internal team testing using iOS App simulations.
  - **System Usability Scale (Planned):**
    - SUS survey to be administered to external users in next semester testing phase.
  - **Task Success Rate:**
    - Early internal testing showed $> 85\%$ success rate in completing key tasks (event discovery, creation, booking).
  - **Average Time-on-Task:**
    - Event creation completed in under 1 minute by most users.
- **Voting System Validation**
  - **Correlation Analysis:**
    - Strong positive correlation between upvote counts and event visibility on the map/list.
  - **Discovery Rate:**
    - High-upvoted events received significantly more bookings compared to low-upvoted events (preliminary validation).
- **Manual Moderation Review Sessions**
  - **Review Latency:**
    - Average time from event submission to moderation decision: under 5 seconds (automated) or under 24 hours (manual review).
  - **Automatic Moderation Accuracy:**
    - Automatic moderation achieved inter-rater reliability of 0.92 with manual moderation, indicating strong agreement.

7. Conclusion and Future Work

Conclusion

In conclusion, the *Evently* project delivers a location-based event discovery platform designed specifically for the Astana community. Our key contributions include:

- Development of a scalable and modular backend in Go, with a geospatial database using PostgreSQL and PostGIS.
- A real-time, sentiment-based content moderation system powered by KazLLM, tailored for Kazakh, Russian, English-language inputs.
- A social layer enabling user engagement through following, booking visibility, and voting.
- A clean architecture that separates concerns across API design, AI moderation, and database management.
- A fully automated CI/CD pipeline with containerized deployment using Docker, GitHub Actions, and Google Cloud Platform.
- Full integration of the frontend iOS application with backend APIs and the moderation system.
- Implementation of secure authentication mechanisms (OAuth) and user session management with JWT tokens.

These components work together to address the lack of localized, interactive event platforms in the region, providing a foundation for community-driven discovery and participation.

Future Work

The future evolution roadmap might include:

- Comprehensive integration and user testing to identify edge cases and usability bottlenecks.
- Performance tuning of API endpoints and database queries for production-readiness.

Additional areas for improvement include improving moderation accuracy through more nuanced classification (beyond binary sentiment), expanding language support to Russian and Kazakh for the UI, and developing an Android version to broaden accessibility.

The current architecture lays a strong foundation for these extensions and demonstrates our ability to design, implement, and evaluate a computing-based solution for real-world social interaction needs.

8. Ethical and Legal Considerations

We prioritize user safety and legal compliance:

- ML and manual moderation for content violations
- Consent forms outlining community rules
- Banning and blacklisting for repeat offenders
- Voting system for community-based filtering

9. References

1. KazLLM Model: https://huggingface.co/issai/LLama-3.1-KazLLM-1.0-8B
2. Kumar, R., Singh, S., & Xu, M. (2024). Evaluating LLMs for Community Moderation Tasks. https://arxiv.org/abs/2402.01987
3. PostGIS: https://postgis.net/
4. Spatial Indexing – Introduction to PostGIS: https://www.google.com/search?client=safari&rls=en&q=spatial+indexing+-+introduction+to+postgis&ie=UTF-8&oe=UTF-8
5. MobilityDB: https://mobilitydb.com/documentation.html
6. Power of Ten: https://spinroot.com/gerard/pdf/P10.pdf
7. UIKit Documentation: https://developer.apple.com/documentation/uikit
8. Yandex Maps SDK for iOS: https://yandex.com/dev/maps/mapkit/doc/ios/quickstart/
9. Google Sign-In for iOS: https://developers.google.com/identity/sign-in/ios
10. Hugging Face Inference API: https://huggingface.co/docs/api-inference/index
11. Go Programming Language Documentation: https://go.dev/doc/
12. pgx PostgreSQL Driver for Go: https://pkg.go.dev/github.com/jackc/pgx/v5
13. sqlx Go Extensions: https://pkg.go.dev/github.com/jmoiron/sqlx
14. Squirrel SQL Builder for Go: https://pkg.go.dev/github.com/Masterminds/squirrel
15. Zap Logging for Go: https://pkg.go.dev/go.uber.org/zap
16. Swagger for Go (swaggo): https://github.com/swaggo/swag
17. JWT with Go (golang-jwt): https://pkg.go.dev/github.com/golang-jwt/jwt/v5
18. PostgreSQL Documentation: https://www.postgresql.org/docs/

19. Spatial Indexing – Introduction to PostGIS: https://postgis.net/workshops/postgis-intro/indexing.html

20. UUID-OSSP Extension (PostgreSQL): https://www.postgresql.org/docs/current/uuid-ossp.html

21. MobilityDB User Manual: https://mobilitydb.github.io/MobilityDB/master/

22. Docker Documentation: https://docs.docker.com/

23. Docker Hub: https://hub.docker.com/

24. GitHub Actions Documentation: https://docs.github.com/en/actions

25. Google Cloud VM Instances: https://cloud.google.com/compute/docs/instances

26. AWS EC2 Documentation: https://docs.aws.amazon.com/ec2/index.html

27. Figma Help Center: https://help.figma.com/hc/en-us

28. GitHub Documentation: https://docs.github.com/

29. Notion Help: https://www.notion.so/help

30. Postman Learning Center: https://learning.postman.com/

10. Appendix

```
-- Enable necessary extensions
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
CREATE EXTENSION IF NOT EXISTS postgis;

-- Users table
CREATE TABLE IF NOT EXISTS users (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    email VARCHAR(255) UNIQUE NOT NULL,
    first_name VARCHAR(255) NOT NULL,
    second_name VARCHAR(255) NOT NULL,
    created_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP
);

-- Followers (social graph)
CREATE TABLE IF NOT EXISTS followers (
    follower UUID REFERENCES users(id) ON DELETE CASCADE,
    followed UUID REFERENCES users(id) ON DELETE CASCADE,
    status VARCHAR(20) CHECK (status IN ('pending', 'accepted', 'blocked')),
    created_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (follower, followed)
);

-- Global event ID sequence
CREATE SEQUENCE IF NOT EXISTS global_event_id_seq START 1 INCREMENT BY 1;
```

```sql
-- Partitioned event table by start_date
CREATE TABLE IF NOT EXISTS event (
    event_id BIGINT NOT NULL DEFAULT nextval('global_event_id_seq'),
    name VARCHAR(255) NOT NULL,
    status VARCHAR NOT NULL DEFAULT 'pending',
    description TEXT,
    created_by UUID REFERENCES users(id) ON DELETE SET NULL,
    location GEOMETRY(POINT, 4326) NOT NULL,
    start_date TIMESTAMPTZ NOT NULL,
    organizer VARCHAR(255) NOT NULL,
    upvote INTEGER NOT NULL DEFAULT 0,
    downvote INTEGER NOT NULL DEFAULT 0,
    created_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (event_id, start_date)
) PARTITION BY RANGE (start_date);

-- Default partition
CREATE TABLE IF NOT EXISTS event_default PARTITION OF event DEFAULT;

-- Example daily partition
CREATE TABLE IF NOT EXISTS event_2025_02_07 PARTITION OF event
    FOR VALUES FROM ('2025-02-07 00:00:00') TO ('2025-02-08 00:00:00');

-- Indexes
CREATE INDEX IF NOT EXISTS event_name_index ON event (name);
CREATE INDEX IF NOT EXISTS event_location_index ON event USING GIST (location);
CREATE INDEX IF NOT EXISTS event_date_index ON event (start_date);

-- Bookings table
CREATE TABLE IF NOT EXISTS bookings (
    booking_id SERIAL PRIMARY KEY,
    user_id UUID REFERENCES users(id) ON DELETE CASCADE,
    event_id BIGINT NOT NULL,
    booking_status VARCHAR(20) DEFAULT 'pending'
        CHECK (booking_status IN ('confirmed', 'pending', 'rejected')),
    visibility VARCHAR(10) DEFAULT 'private'
        CHECK (visibility IN ('public', 'private')),
    booked_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX IF NOT EXISTS bookings_event_id_idx ON bookings (event_id);
```

Listing 1: Core Database Schema