Architectural Requirements
Taxi Tap by Git It Done

Contents

1 Introduction

TaxiTap is a comprehensive taxi booking and management application designed to connect passengers with drivers in real-time. The system provides a seamless, secure, and scalable solution for urban transportation needs. This document outlines the architectural decisions, quality requirements, and technical strategies employed to deliver a robust, high-performance mobile application. The system facilitates ride booking, real-time tracking, and driver management through an intuitive mobile interface backed by a serverless architecture.

2 Architectural Design Strategy

Strategy Chosen: *Decomposition via Feature-Driven Development (FDD)*

TaxiTap is built using a modular, feature-based decomposition strategy that aligns with our quality requirements of availability, scalability, and usability. Each functional system (e.g., User System, Vehicle System, Trip System) is designed, tested, and deployed independently.

Benefits of this strategy:

- Clear modularity: Separation of concerns enables focused development and maintenance.
- Parallel development: Multiple features can be developed simultaneously by different team members.
- Changes to one feature have minimal impact on others
- Scalability: Features can be scaled based on demand without affecting the entire system
- Reduced risk: Changes to one feature have minimal impact on others.

This strategy was chosen because it directly supports our top three quality requirements:

- Availability - Failure in one module doesn't bring down the entire system
- Scalability - Individual modules can be scaled independently
- Usability - Clear feature boundaries lead to better user experience design

3 Architectural Strategies

Chosen Style: *Event-Driven Architecture*

Event-driven architecture is centered around asynchronous communication between components. Components emit and react to events, allowing for real-time responsiveness, loose coupling and scalability. Why this is the best fit for our system:

- Real-time Interaction: Location updates, ride requests, driver availability, and notifications all benefit from real-time triggers and updates. Convex supports reactive data and background functions, making it a natural fit for an event-driven model.
- Asynchronous Processing: Tasks like sending push notifications, updating seat availability, or logging analytics should not block the main user flow. EDA allows these to run in the background, improving app responsiveness.
- Loose Coupling: With EDA, components (like driver matching and notifications) can be developed and deployed independently. This aligns well with Convex's function-based model, which is modular and reactive.
- Scalability and Maintainability: New features can easily be added by listening to events without modifying core components.

3.1 Architectural Strategies extended

- **Availability:**
  Problem: If the system becomes unavailable, users cannot book or accept rides, damaging both reliability and trust.
  Solution:
  - **Replication:** Convex provides high availability by replicating data and functions across multiple geographic zones, ensuring continuity during failures.
- **Scalability:**
  Problem: To support growing demand and simultaneous ride requests, the system must dynamically handle increased load.
  Solution:
  - **Horizontal scale-out:** Convex automatically scales infrastructure horizontally to meet demand. Convex handles horizontal scaling of function execution through a service called Funrun. This service allows for the distribution of function execution across multiple machines, enabling a more scalable and performant system.
  - **Data sharding:** Data is partitioned across shards by Convex without manual configuration, improving scalability. Convex handles data sharding primarily through Components and Sharded Counters. Components, like the Sharded Counter, are isolated units of logic that handle specific tasks and provide transactional consistency, even when dealing with potentially conflicting operations.
  - **Asynchronous processing:** Non-critical tasks (e.g., sending notifications) are offloaded to background jobs using Convex async functions, freeing up system resources.
- **Usability:**
  Problem: An intuitive and responsive interface is crucial to support users with varying levels of digital literacy.
  Solution:
  - **Real-time UI:** Keeps location and notification data live using subscriptions or efficient polling mechanisms.

  - **Responsiveness:** Improves user experience by minimizing UI latency and limiting the use of loading spinners to essential operations.

- **Security:**
  Problem: The system manages sensitive user data, including real-time locations and account details. Breaches can result in legal and reputational harm.
  Solution:

  - **Secure communication:** All data in transit is encrypted using TLS by default.
  - **Role-based access control (RBAC):** Server functions enforce strict access controls based on user roles (e.g., driver vs passenger).

- **Performance:**
  Problem: Real-time systems require fast responses. Delays in booking, tracking, or communication reduce system usability.
  Solution:

  - **Database indexing:** Convex uses optimized queries with `withIndex(...)` for fast access to common fields like driver ID or ride ID.
  - **Asynchronous tasks:** Operations like sending notifications are processed asynchronously to reduce response time for users.

4 Architectural Quality Requirements

4.1 Quality attributes

The following quality requirements are prioritized based on client needs and system criticality:

- **Availability:**
  Why it's a top quality requirement: Ensures continuous service. If the app is down, users cannot book or accept rides, which impacts revenue and reputation.

Stimulus Source	System failures, network issues, or high load
Stimulus	System components become unavailable or unresponsive
Response	System maintains service continuity with graceful degradation
Response Measure	– 99.5% uptime under normal usage
Environment	Normal and peak operating conditions
Artifact	Entire system infrastructure including database and backend

- **Scalability:**
  Why it's a top quality requirement: Supports many users using the app simultaneously and allows the system to handle growth in demand.

Stimulus Source	Increasing user base and concurrent ride requests
Stimulus	System must handle growing demand without performance degradation
Response	Horizontal scaling of services and database sharding
Response Measure	– Support 100+ concurrent ride requests – Maintain response times under increased load – Auto-scale infrastructure based on demand
Environment	Variable load conditions from low to peak usage
Artifact	Backend services, database, and infrastructure

- **Usability:**
  Why it's a top quality requirement: Users must be able to complete essential tasks easily, regardless of their technical skill level. Poor usability leads to frustration and abandonment.

Stimulus Source	Users with varying technical skills using the app
Stimulus	Users need to complete ride booking and payment tasks easily
Response	Use simple language, clear layout, and high-contrast color schemes. Employ interface metaphors such as intuitive icons to make navigation easy for users, including those who are technologically inexperienced
Response Measure	– Task completion within a few clicks
Environment	Mobile device usage in various conditions
Artifact	User interface, navigation flow, visual design

- **Security:**
  Why it's a top quality requirement: The system manages sensitive user data, including locations and contact details. Breaches can result in legal issues and loss of trust.

Stimulus Source	Unauthorized access attempts or data breaches
Stimulus	Users/attackers attempting to access sensitive data
Response	System enforces strict access controls and data protection
Response Measure	– Role-based access control (RBAC) enforced – Data encrypted in transit and at rest – Authentication tokens validated
Environment	All system interactions and data access
Artifact	User data, location information, payment details

- **Performance:**
  Why it's a top quality requirement: Real-time features must respond quickly to meet user expectations. Delays in booking, navigation, or messaging reduce usability during peak hours.

Stimulus Source	User/Driver requesting ride booking or location updates
Stimulus	User wants real-time responses for booking, tracking, and navigation
Response	System provides immediate feedback and updates
Response Measure	– Backend functions respond within 20ms average – Location updates occur every time the user moves – Ride requests processed within 100ms
Environment	Peak usage hours with high concurrent users
Artifact	Mobile app, backend functions, database queries

5 Architectural Patterns

This section discusses the key architectural patterns employed in our system, along with their motivations, the quality requirements they support, and their specific application in our implementation.

5.1 Leader-Follower (Replication)

Purpose: The Leader-Follower pattern replicates services or data across nodes to ensure availability and failover.

Why We Use It: To provide high availability and data redundancy. In case one instance fails, another replica can take over.

Quality Requirement Supported:

- **Availability**

Application in System: Convex's serverless backend infrastructure employs automatic replication of data and functions across multiple zones. This ensures continued access to services even if a zone experiences downtime. It also supports strong consistency guarantees without requiring developers to manage replicas manually.

5.2 Service-Oriented Architecture

Purpose: To structure the system as a set of loosely coupled, reusable services that communicate over well-defined interfaces. Each service encapsulates a specific business capability, enabling flexibility in deployment and evolution.

Why We Use It: Taxi Tap uses SOA to separate key business functions—such as ride booking, driver management, payment processing, and notifications—into independent services. This allows teams to develop, maintain, and deploy each service without impacting others, while still integrating them into a cohesive platform through standard service contracts.

Quality Requirements Supported:

- **Scalability** – Services can scale independently depending on load. For example, the ride booking service can be scaled up during peak hours without affecting payment processing.
- **Performance** – Limits cross-service interference; e.g., high load on the booking service does not slow down authentication or driver location tracking.

Application in System: In Taxi Tap, the SOA approach ensures that each core function (e.g., Booking Service, Authentication Service, Payment Service, Location Service) operates as an independent service. These services communicate via a defined API, enabling the system to replace or update services without a full system redeployment. For example, the Payment Service manages transaction records independently from the Ride Service's booking data, connected only through shared ride identifiers.

5.3 Event-Driven Architecture

Purpose: Decouple services by communicating through asynchronous events.

Why We Use It: To handle asynchronous workflows, reduce coupling, and improve responsiveness under load.

Quality Requirements Supported:

- **Scalability** – Events allow services to scale independently.
- **Performance** – Asynchronous processing ensures quick responses.
- **Availability** – Temporary service disruptions don't block upstream operations.

Application in System: Convex background functions are used to handle side effects such as updating ride history, sending notifications, and recalculating driver ratings. These tasks are decoupled from the main user actions (e.g., booking a ride), improving perceived responsiveness. Events like `rideRequested`, `rideCompleted`, and `locationUpdated` are used to trigger background logic.

5.4 Model-View-ViewModel (MVVM)

Purpose: Promote separation of concerns between UI, state, and business logic.

Why We Use It: To create maintainable and testable frontend code, and to simplify UI state synchronization.

Quality Requirement Supported:

- **Usability**

Application in System: Frontend components (built using React Native) follow MVVM principles:

- **Model:** Convex queries and mutations represent data.
- **ViewModel:** Hooks like `useQuery`, `useState`, and custom context (e.g., `ThemeContext`) manage UI logic.
- **View:** React components render based on ViewModel state.

This enables real-time updates (e.g. seat availability or route filtering) with a clean separation of concerns, enhancing user experience.

6 Design patterns

Observer Pattern

- Pattern Type: Behavioural
- Participants:
  - Subject: Notification System
  - Observer: User
  - Concrete Observer: Passenger, Driver
- Explanation: The Observer pattern allows an object (User) to be notified automatically of state changes in another object (Notification System). This is ideal for handling events like route updates or ride status.
- Example:
  - User receives alerts from the Notification System.
  - Notification System initiates a notification when a route is announced.

Mediator Pattern

- Pattern Type: Behavioural
- Participants:
  - Mediator: Taxi Request System
  - Colleague: Passenger, Driver

- Explanation: The Mediator pattern centralizes complex communication between objects. Instead of Passenger directly interacting with Driver, requests are handled through the Taxi Request System.
- Example:
  - Taxi Request System acts as an intermediary between Passenger and Driver.
  - Passenger makes a request for pickup to a driver, but the Taxi request system acts as the middleman for this request.

7 Architectural Design

Overview: Taxi Tap is structured using a feature-driven and event-driven architecture. The diagram below illustrates this architecture.

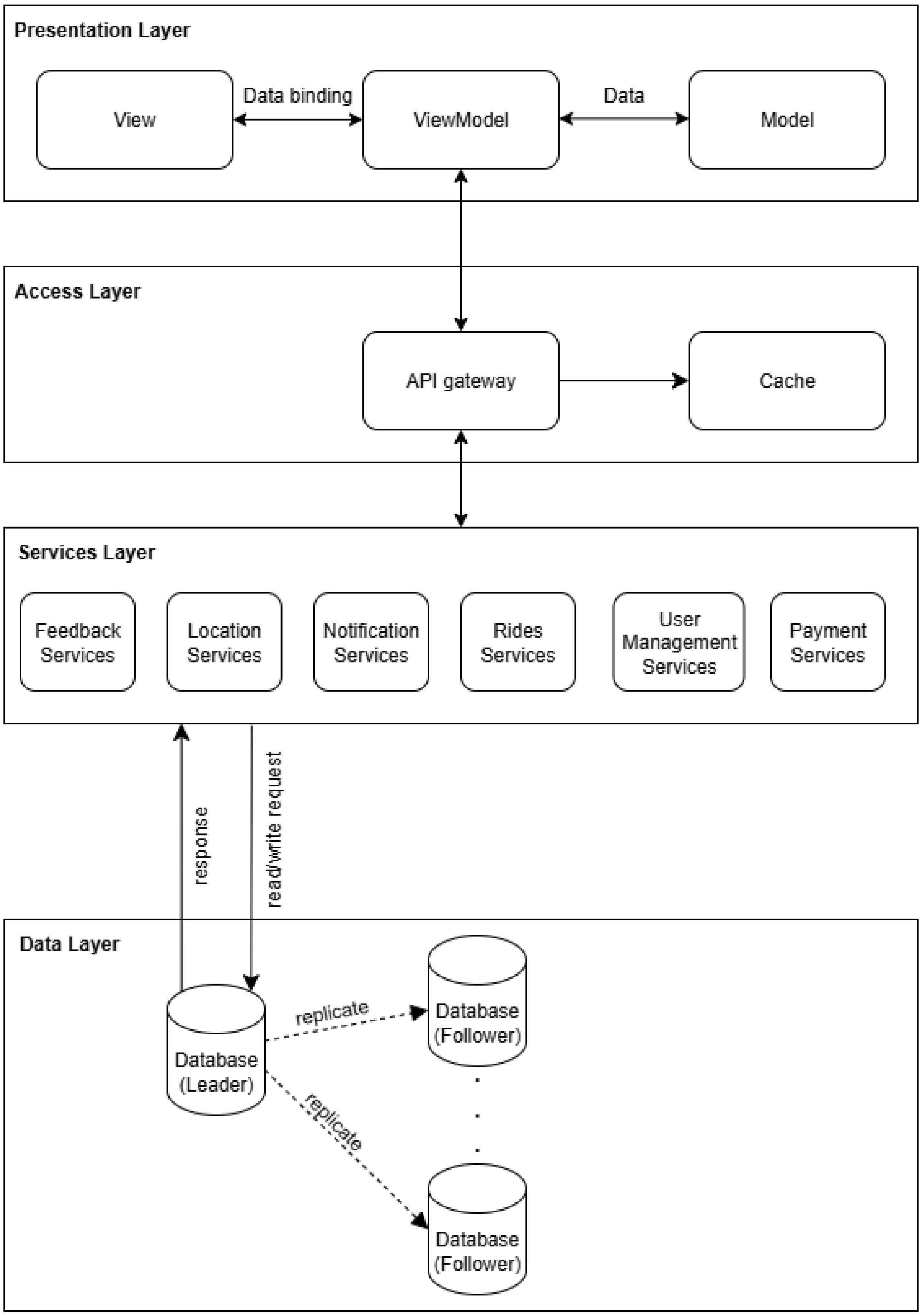

Figure 1: Architecture Diagram

Components:

- **Expo Frontend:** Mobile-first interface using React Native. The user intefraces are built with MVVM.
- **Convex Backend:** Serverless backend with modular mutations and schema.
- **Convex Database:** Strongly-typed database used by each module.

This design provides modularity, scalability, and testability with minimal DevOps complexity.

8 Architectural Constraints

- **Client Constraints:** Must remain within the AWS Free Tier; performance must be maintained under low-cost infrastructure.
- **Deployment Constraints:** Fully serverless; no Docker/Kubernetes; must deploy via CI/CD with minimal setup.
- **Security Constraints:** Only verified users may access trip, payment, or GPS functionality.
- **Latency Constraints:** Real-time location updates must occur.
- **Scalability Constraints:** Design must accommodate scaling to 1,000+ users without architectural changes.

9 Technology Choices

Backend Platform

Option	**Pros**	**Cons**
Convex	Fully serverless, fast dev, native React support	New ecosystem, TypeScript only
Firebase	Realtime syncing, easy integration	Poor test tooling, security rule complexity
AWS Lambda	Highly scalable, mature	Complex CI/CD, requires DevOps setup
Chosen: Convex	Perfect fit for modular, testable architecture. Free tier-friendly.	

Frontend Platform

Option	**Pros**	**Cons**
Expo (React Native)	Fast prototyping, hot reload, cross-platform	Slightly heavier bundles
Flutter	Beautiful UI, good performance	Slower iteration, Dart-only
Native iOS/Android	Highest performance	High dev effort, no code sharing
Chosen: Expo	Fastest mobile-first path with TypeScript and Convex integration.	

Database

Option	**Pros**	**Cons**
Convex DB	Type-safe, built for Convex, no config	Smaller community
Firestore	Realtime, battle-tested	Complex security model
Supabase	Postgres-based, open source	Overhead for micro-systems
Chosen: Convex DB	Natively integrated with our serverless logic.	

10 Deployment Model

The system will be deployed following a cloud-based deployment model to ensure scalability, availability, and ease of access. The deployment environment will include:

- **Frontend:** Deployed via Expo Go.
- **Backend:** Hosted on Convex cloud (Convex backend as a service).
- **Database:** Managed by Convex (integrated serverless database).
- **CI/CD:** Managed via GitHub Actions to automate linting, testing and deployment.

The system is deployed using a cloud-based deployment model. The deployment pipeline follows a CI/CD workflow that integrates development, continuous integration, and continuous deployment. During development, source code is managed through GitHub, with developers using Expo Go for mobile app testing and Convex Dev for backend testing in a local environment. Once changes are committed and pushed to GitHub, the CI pipeline is triggered via GitHub Actions, where automated steps such as linting, formatting, and testing are executed to ensure code quality and consistency. After successful integration, the CD pipeline automatically deploys the validated build to the target production environment hosted in the Convex cloud platform. The deployment topology follows a multi-tier architecture, where the frontend (mobile client) communicates with backend services hosted in Convex, which provides serverless functions, database storage, and scaling capabilities. The diagram illustrates this process, showing the transition from local development through GitHub-based integration pipelines to final deployment in Convex Production.

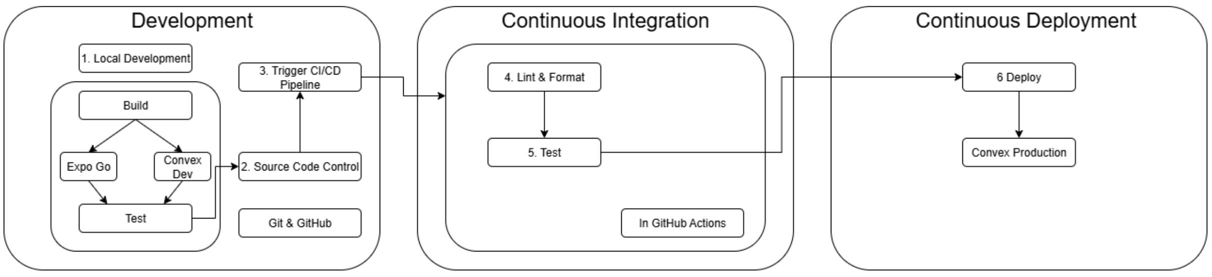

Figure 2: Deployment Diagram

11 Live Deployed System

A fully functional live version of the system will be accessible for demonstration purposes. This deployed system will allow real-time interaction by both drivers and passengers via mobile devices. The system will include:

- User registration and authentication.
- Route selection and ride request.
- Real-time updates.
- Feedback and rating system.
- User account management.
- Cash payment support.