International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 09 Issue: 10 | Oct - 2025 SJIF Rating: 8.586 ISSN: 2582-3930

Technical Research and Architectural Plan for 'Classic Union': A Scalable Event Organizing Platform using Flutter and Firebase

Dr. M. Sengaliappan1, Madhana Raja Vel .R 2

1Professor & Head, Department of Computer Applications, Nehru College of Management, Bharathiar University,Coimbatore,Tamilnadu,India

Cmsengs.7@gmail.com

2Master of Computer Applications, Nehru College of Management,

Bharathiar University,Coimbatore,Tamilnadu,India rmrajavel28@gmail.com

Abstract

This paper details the technical architecture for 'Classic Union,' a highly scalable, event-driven application designed to streamline event organization and management. The system employs a robust hybrid technology stack, utilizing Flutter for cross-platform presentation and a serverless Firebase backend (comprising Firestore, Authentication, and Cloud Functions). A local SQLite database is strategically integrated to facilitate complex querying and advanced offline data persistence. The primary architectural contribution addresses the critical challenge of high-contention concurrency risks inherent in distributed booking systems. The methodology leverages a modified Clean Architecture pattern, specifically adopting the BLoC/Cubit pattern to ensure a rigorous separation of concerns. Key transactional logic, specifically event slot reservation and secure payment processing, is securely elevated to Firebase Cloud Functions (CF). This server-side approach guarantees data integrity by implementing atomic transactions for concurrency mitigation and secure webhook integration (Stripe) for payment finalization. The resultant system is engineered for exceptional maintainability, resilience against typical race conditions, cost optimization through selective local data caching, and seamless scalability necessary for supporting large-scale event operations.

1. Introduction

The contemporary event management sector requires digital solutions capable of handling large volumes of simultaneous transactions, ensuring real-time data synchronization, and maintaining stringent security standards for both financial and proprietary user information. The 'Classic Union' application is positioned to address these requirements by combining the efficiency of the Flutter framework with the distributed computing capabilities of Google's Firebase ecosystem.

Flutter is selected as the development framework due to its inherent cross-platform compatibility and its design as an extensible, layered system. This structure, built upon a series of independent libraries, ensures that the framework level is modular and replaceable, providing the necessary foundation for rapid, uniform development across mobile operating systems. At its core, the Flutter engine, largely written in C++, handles primitives like rendering, text layout, and the plugin architecture. Meanwhile, Firebase offers a suite of managed, scalable backend services, allowing development teams to concentrate resources on application-specific business logic rather than infrastructure management. This combined approach is fundamental to meetingthe scalability and performance benchmarks required for a high-traffic event application.

1.1 Problem Statement: Addressing Distributed System Challenges

The most complex functional requirement for 'Classic Union' resides in reliably managing concurrent requests for limited, time-sensitive resources, such as event tickets or specific seating slots.

The Concurrency Paradox

In high-contention scenarios typical of popular event launches, multiple users attempt to secure the same resource concurrently. Direct handling of these operations using standard client-side Firebase transactions presents a significant risk. The mobile and web Software Development Kits (SDKs) for Firestore utilize an optimistic concurrency control mechanism. This mechanism operates under the assumption that data contention is unlikely, avoiding database locks to maintain speed. If a conflict is detected—meaning a

document read within the transaction was modified by an external operation—the client-side transaction automatically retries. However, under severe contention, this retry loop may fail after a finite number of attempts, resulting in an ABORTED: Too much contention on these documents. Please try again error. This unpredictable failure mode is unacceptable for the core business function of event booking, which demands deterministic, guaranteed resource allocation.

Security and Compliance

A secondary, yet equally critical, challenge is ensuring strong security throughout the system. This encompasses two main areas: first, establishing robust Role-Based Access Control (RBAC) to differentiate privileges among Admins, Organizers, and Attendees ; and second, securing all financial transactions. Payment processing requires protecting sensitive API keys and handling the inherently asynchronous nature of payment state changes reliably, thereby guaranteeing financial data integrity and compliance.

Justification for Technical Rigor

To address these challenges, the architectural plan must establish a predictable and deterministic solution that places data integrity above simple read speed, particularly for mutable, critical operations. A highly rigorous technical blueprint is required to outline the necessary server-side delegation and security enforcement mechanisms.

2. Architectural Framework and Design Rationale

2.1 Layered Flutter Architecture (Clean/BLoC Adaptation)

The 'Classic Union' application employs a modified Clean Architecture structure, often referred to as the Model-View-ViewModel (MVVM) pattern in the Flutter context, with a focus on strict separation of concerns. This approach divides the application into distinct, responsibility-bound layers: Presentation, Domain (Business Logic/View Models), and Data (Repositories/Services).

Component Breakdown

- **Presentation Layer (UI/Views):** This layer consists of the visual elements and user interface components. Views, typically built around the Scaffold widget, are solely responsible for rendering the current UI state and forwarding user interactions (events) to the view model.
- **Domain Layer (BLoC/Cubit):** This layer encapsulates the application's core business logic. Functioning as the view model, the BLoC or Cubit components handle the logic necessary to convert raw application data retrieved from the repository into specific UI States. They act as the mediator, interacting exclusively with the Data Layer to fetch or submit information.
- **DataLayer (Repositories/Services):** The Data Layer serves as the source of truth abstraction. Repositories define the necessary operations (e.g., fetch event details, submit new reservation) and abstract the data sources from the business logic. The Services, or Data Sources, implement these operations, containing the specific code to interact with external systems, such as Firebase Firestore API calls or local database commands (sqflite).

State Management Rationale (BLoC/Cubit)

The Business Logic Component (BLoC) pattern, specifically its simplified counterpart, Cubit, is selected as the primary state management solution. This choice provides a highly structured and reactive data flow while minimizing the boilerplate code typically associated with full BLoC implementations. Cubit offers a scalable solution for managing complex, data-driven forms and real-time UI updates stemming from the backend, providing more structure than standard Provider or low- level setState mechanisms.

The use of BLoC inherently aligns with the event-driven nature of the Firebase backend. Firestore listeners generate real-time data snapshots, which are essentially events capturing state changes on the cloud. The BLoC pattern, built upon processing UI **Events** and emitting application **State** changes , provides a clean, predictable mechanism for processing these asynchronous Firestore snapshots. This synergy between the architectural pattern (BLoC) and the data source behavior (Firestore real-time events) is critical for maintaining consistent user interfaces, mitigating the risk of inconsistent UI updates that can plague less structured approaches when consuming live data streams. Dedicated widgets such as BlocProvider manage BLoC access within the widget tree, while BlocBuilder and BlocListener efficiently handle UI rendering based on state changes and execute necessary side effects, such as navigating to a confirmation screen after a successful operation.

3. RELATED WORK AND LITERATURE REVIEW

Mobile event–organizing applications have evolved from simple schedule tools into integrated digital ecosystems that manage registration, ticketing, live updates, and analytics. In particular, campus and community event systems now emphasize cross-platform support, real-time communication, and offline reliability [1]. The "Classic Union" application is designed within this paradigm, employing Flutter for cross-platform development, SQLite for local data persistence, and Firebase for cloud synchronization.

3.1 Cross-Platform Frameworks and Flutter

Flutter, introduced by Google, enables developers to build visually rich, native- compiled applications for Android and iOS from a single code base. Studies demonstrate that Flutter provides high performance, expressive UI capabilities, and reduced development time compared with native approaches [2]. Empirical analyses further highlight Flutter's consistent frame rendering rate and widget reusability, making it suitable for responsive, interactive event-management UIs [3].

3. 2Local Storage with SQLite

SQLite is a lightweight, serverless relational database engine that provides ACID-compliant transactions and rapid local querying. It is widely adopted in mobile development due to its small footprint and low latency [4]. Comparative tests reveal that SQLite excels at handling frequent read/write operations and supports efficient caching when network connectivity is unavailable [5]. In event applications, it ensures smooth offline access to participant lists, schedules, and QR-based tickets.

3.3 Cloud Integration and Real-Time Data via Firebase

Firebase is a Backend-as-a-Service (BaaS) platform offering Realtime Database and Cloud Firestore for synchronized cloud storage, authentication, and push notifications [6]. These features are essential for dynamic environments such as event management, where live updates, user presence, and announcements must be reflected across devices instantly [7]. Firebase's built-in offline caching complements local storage, ensuring data consistency during intermittent connectivity [8].

3. 4Hybrid Approach: SQLite + Firebase

Several studies propose combining SQLite and Firebase to achieve an offline-first design, enabling users to interact with data locally while synchronizing with the cloud once a connection is re-established [9]. This hybrid architecture ensures low-latency interaction and seamless cross-device data updates. Such synchronization models have been shown to enhance user experience and reliability in mobile event applications [10].

3.5 Security, Privacy, and Scalability

Data security and user privacy are crucial in event applications that handle personal information. Firebase offers authentication, access control, and encryption mechanisms [11]. However, sensitive data stored locally must also be secured via encrypted SQLite storage. Studies emphasize efficient Firestore query design and rule-based data access to reduce latency and cost [12]. Proper data partitioning and real-time sync conflict resolution techniques further improve scalability and consistency [13].

3. 6Research Gap and Application Implications

Although prior works have demonstrated Flutter-Firebase integration, limited empirical studies assess hybrid offline-first architectures that combine both SQLite and Firebase under variable network conditions [14]. The "Classic Union" application addresses this gap by implementing a dual-storage system to ensure reliability, performance, and user convenience in community-level event management.

4. HybridPersistence Strategy:Firestore and SQLite Coexistence

4.1 Firebase Firestore as the Source of Truth

Firestore is mandated for handling critical, mutable data requiring real-time synchronization, massive scalability, and reliable persistence. It serves as the single, immutable source of truth for all transactional data, including reservations, user roles, and event capacities. Furthermore, Firestore provides built-in offline data persistence by caching a local copy of data that the application actively uses, which generally suffices for basic read latency reduction.

4.2 Strategic Use of Local SQLite

While Firestore caching supports basic offline operation, local SQLite integration, via the sqflite plugin , is strategically employed to meet advanced data management and cost optimization requirements that transcend the capabilities of the native Firestore cache.

The need for a dedicated local relational database stems from two major requirements:

1. **Complex Relational Queries:** SQLite is a proven solution for handling relational data models with multiple tables and complex relationships. If the application needs to execute advanced local queries—such as correlating user demographic data (local preference settings) with detailed event schedules (cached from Firestore) or generating organizer reports based on joined local data sets—the relational structure and SQL querying capabilities of SQLite are significantly superior to a flat document store cache.

2. **Read Optimization and Cost Reduction:** For high-volume data that is static or common across users (e.g., localized content, large catalogs of historical events, or large, rarely changing configuration parameters), repeatedly fetching this data from Firebase generates unnecessary read charges. By implementing a cache-first strategy in the Repository layer— retrieving data once, writing it to SQLite, and serving subsequent requests locally—the number of Firebase read operations is drastically reduced, optimizing cloud costs and improving access speed for common data sets.

The Repository layer is responsible for defining a clear synchronization model: it checks the local SQLite cache first and only performs a Firestore fetch if the local data is stale or absent. Any critical update is then propagated from Firestore back to the SQLite cache.

Table Title: Hybrid Data Persistence Strategy for 'Classic Union'

The coexistence of SQLite and Firestore necessitates disciplined data management within the Data Layer. While Firestore provides eventual consistency across devices, SQLite provides immediate consistency locally. To prevent compromise of distributed data integrity, Firestore is designated as the sole source of truth for critical, mutable data (e.g., ticket availability). SQLite must function strictly as an immutable, read-optimized cache for non-critical or static data. The Data Layer must enforce that all write operations related to critical inventory bypass SQLite entirely and are directed through the secure Cloud Function endpoint to Firestore.

5. Critical System Design and Implementation

5.1 High-Concurrency Booking Mechanism (Mitigating Race Conditions)

The inherent risk of concurrent resource access must be addressed by moving the booking logic away from the client-side environment. Since standard Firestore client transactions use optimistic locking—prone to retries and ultimate failure (ABORTED errors) under high contention —the core booking function is delegated to Firebase Cloud Functions (CF).

Mandatory Server-Side Enforcement

Cloud Functions provide a serverless environment where code executes securely within Google's data centers. This environment is utilized to implement the critical transactional logic:

1. **Client Request:** The Flutter client calls an HTTPS Cloud Function, providing non-sensitive data such as the user ID and the target event ID.

2. **Atomic Operation Execution:** The Cloud Function acts as a secure "manual secretary" for the resource. It uses the Firebase Admin SDK to execute a server-side atomic transaction on the event availability document in Firestore.

3. **Core Logic:** Within the atomic transaction block, the function first **Reads** the current capacity. It then **Verifies** that the capacity is greater than zero and confirms the user's eligibility. If successful, it **Decrements** the capacity and **Writes** the updated capacity back to the event document while simultaneously creating the reservation document.

4. **Concurrency Resolution:** Server-side atomic transactions manage data contention internally. If a document read is modified by a concurrent operation, the transaction automatically retries. This ensures that only one operation successfully commits the required capacity decrement, providing the necessary guarantee of data integrity for resource allocation.

5.2 Role-Based Access Control (RBAC) and Security Rules

A multi-tiered access control system is implemented to separate privileges effectively. The primary roles defined are **Admin** (system oversight), **Organizer** (event management), and **Attendee** (consumer access).

Implementation Strategy

1. **Role Storage:** User roles are linked to the Firebase Authentication system. The definitive role information is stored within the user's document in Firestore (e.g., /users/{uid}) as a map of roles. Crucially, the assignment of these roles is restricted to a trusted environment (such as an Admin panel backed by Cloud Functions or the Firebase Admin SDK) to prevent unauthorized privilege escalation.

2. **Enforcement:** Access control is rigorously enforced using Firestore Security Rules. These rules validate every read and write request by checking the

authenticated user's ID (request.auth.uid) and cross-referencing it with the user's stored role in the database.

The effectiveness of security rules as a non- negotiable firewall is paramount. Client-side navigation or feature toggles based on locally stored roles are insufficient and easily bypassed. Therefore, the architecture ensures that the final permission check always occurs within the security rules, thereby protecting the underlying data.

A common scenario involves granular control for Organizers: Security rules must permit an Organizer to update specific fields (e.g., event title or content) on an event document only if their UID is present as the hostUID or within a designated roles map on that document. Conversely, the rules must explicitly deny the Organizer permission to modify critical fields like the event's roles map or financial data, reserving those permissions for the Admin role.

5.3 Secure Payment Processing (Stripe Integration)

Stripe is chosen as the payment gateway due to its robust, developer-friendly official Flutter SDK (flutter_stripe) and its built-in security features, including PCI-DSS Level 1 compliance and automatic tokenization, which simplifies security burdens.

Security Protocol: Backend-Centric Model

To maintain the highest level of security, the Stripe Secret Key is strictly confined to the secure server environment of Firebase Cloud Functions. The payment process follows a three-step server-centric workflow:

1. **Client Request:** The Flutter application initiates the purchase, sending the item ID and quantity to a dedicated HTTPS Cloud Function endpoint. Crucially, the actual price is retrieved or validated server-side using trusted data sources to prevent any client-side manipulation of the transaction amount.

2. **Payment Intent Creation (CF):** The Cloud Function utilizes the secure, private Stripe Secret Key to interact with the Stripe API, creating a PaymentIntent. It then securely transmits only the public, non- sensitive client_secret back to the Flutter client.

3. **Client Payment:** The Flutter application uses the official flutter_stripe package and the received client_secret to present the Payment Sheet (e.g., Apple Pay, Google Pay, Card Form). All sensitive financial data is collected and tokenized directly by Stripe, bypassing the 'Classic Union' servers entirely, thus fulfilling PCI compliance requirements.

Asynchronous Transaction Finalization (Webhooks)

Payment finalization is an asynchronous process, often completing after the client-side UI confirmation. Relying solely on the client's connectivity at the moment of payment success is fragile. To guarantee reliable reservation status updates (e.g., moving a reservation from PENDING to PAID), a dedicated, secure Cloud Function is deployed as a webhook listener. Stripe sends real-time events (e.g., checkout.session.completed) to this webhook. The function validates the signature of the incoming webhook and, using the secure Admin SDK, updates the corresponding reservation document in Firestore, providing an out-of-band, tamper-proof mechanism for transaction finality.

6. Architectural Diagrams and Data Flow Analysis

6.1 High-Level System Architecture Diagram (Conceptual Model)

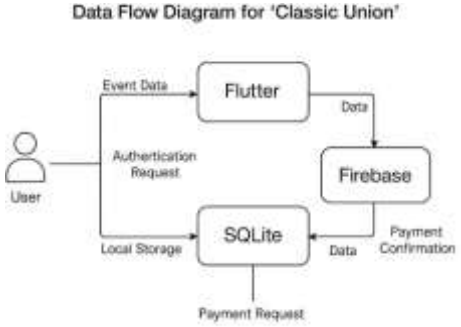
The high-level architecture comprises three interacting domains: the **Flutter Client**, the **Firebase Ecosystem**, and the **External Services** (Stripe). Data flow is rigorously controlled:

1. **Authentication and RBAC Flow:** User authentication occurs via Firebase Auth, feeding into Firestore Security Rules which act as the access control gate for all database operations.

2. **Application Data Flow:** Managed through the layered Flutter architecture (Presentation Domain Data), with Repositories mediating access between the local SQLite cache and the remote Firestore source of truth.

3. **Critical Logic Flow:** Client requests for resource allocation or payment initiation are routed exclusively via HTTPS calls to secure Cloud Functions, which execute high- integrity, server-side business logic.

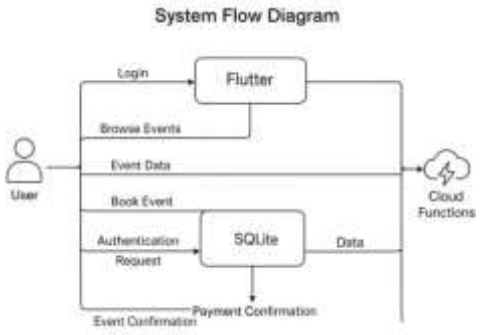
6.2 Detailed Data Flow Diagram (DFD) and



Sequence Diagram

The most critical flow within the system, the High-Integrity Event Reservation process, is best illustrated using a sequence diagram to demonstrate the necessary delegation to the server environment for atomic operations. This visualization confirms that the mechanism to mitigate high-contention race conditions is structurally enforced.

Sequence Diagram: High-Integrity Event Reservation Flow



7. Discussion and Evaluation

7.1 Architectural Strengths and Trade-offs

The proposed architecture achieves its core design goals by leveraging the strengths of its chosen components. Firebase inherently provides horizontal scalability, allowing the system to handle increasing load without manual server provisioning. The BLoC architecture ensures that the local UI rendering remains fast and responsive, efficiently managing complex, incoming data streams without freezing the user interface.

The design's primary strength is the delegation of concurrency control to the Cloud Functions. By executing the booking logic in a server-side atomic transaction, the architecture guarantees data integrity and prevents resource over- allocation, successfully resolving the risks associated with client-side optimistic concurrency failures during peak event demand.

A notable trade-off involves the complexity introduced by the hybrid persistence model (Firestore and SQLite). Integrating and synchronizing two distinct database technologies requires careful maintenance and rigorous Repository implementation. However, this complexity is justified by the functional necessity of optimizing read performance, supporting advanced offline querying, and achieving significant cost reductions by minimizing routine Firebase read operations, making the overall system more economical and robust for long-term operation.

7.2 Evaluation Against Non-Functional Requirements (NFRs)

- **Security:** High security standards are met through the multi-layered enforcement of RBAC via Firestore Security Rules and the secure, decoupled payment flow utilizing Cloud Functions and Stripe Webhooks. Sensitive financial processes are entirely backend-centric, preventing exposure of secret keys.
- **Data Integrity:** Achieved through mandatory server-side atomic transactions for all resource-modifying operations (booking, capacity updates).
- **Maintainability:** The adherence to Clean Architecture principles, coupled with the modularity of BLoC, enforces distinct, testable layers, which is crucial for reducing technical debt and simplifying feature development in a large-scale application.

8. Sample Output

Figure 8.1 User Data Management

Figure 8.2 Admin Data Management

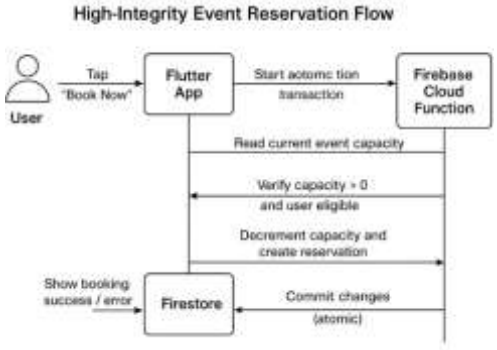


Figure 8.3 Over View Data Management

Figure 8.4 Sample login page

9. Conclusion

This architectural plan successfully establishes a resilient, scalable, and secure framework for the 'Classic Union' event organizing application. The technical contribution lies in the strategic formulation of a hybrid data persistence model that maximizes cost efficiency (SQLite caching) without compromising distributed integrity (Firestore source of truth). Crucially, the plan implements a high-integrity server-side concurrency model using Firebase Cloud Functions to govern resource allocation, thereby circumventing the inherent limitations of client-side optimistic locking for high-contention booking flows. The overall framework provides a dependable foundation for the application's immediate development phase.

8.1 Future Work and Enhancements

Future development efforts should focus on enhancing the attendee and organizer experience, leveraging the architecture's modularity to integrate high-engagement features:

- **Advanced Attendee Experience:** The modular design supports integrating high-touch features such as **Contactless Check-In**. This can be implemented using NFC or QR code scanning functionalities built into the branded event app, utilizing the Flutter embedder's native capabilities for seamless access. Real-time check-in status updates can be managed efficiently using the low-latency Firebase Realtime Database layer, triggered via the check-in event.
- **Gamification and Personalization:** Leveraging the local SQLite storage, which holds cached event metadata and user preferences, can enable the development of personalization tools. This includes implementing features like AI-based personalized event recommendations or gamification elements, such as leaderboards and virtual scavenger hunts, thereby enhancing overall attendee engagement.
- **Enhanced Analytics and Reporting:** Extending the data layer to integrate granular tracking of sales, attendance, and session performance will allow organizers and sponsors to gain deeper, more precise insights. This granular data is essential for accurate calculation of Return on Investment (ROI) and facilitating robust financial planning for subsequent events.

9.REFERENCES

- [1] P. Kirubhakar et al., *"Mobile Application for College Event Management,"* IRJMETS, vol. V, pp. 1–6, 2024.

[2] T. Väänänen, *"Flutter in Cross- Platform Development: Tools and Performance Analysis,"* MSc Thesis, Univ. of Oulu, 2025.

[3] Google Developers, *Flutter Documentation,* Google LLC, 2025.

[4] O. Obradović and M. Keleč, *"Performance Analysis on Android SQLite Database,"* Semantic Scholar,2019.

[5] IJSRET, *"A Smart Event Management App Using Flutter and Firebase (Evecurate),"* vol. 6, no. 3, pp. 122–127, 2021.

[6] Google Firebase, *"Cloud Firestore Documentation,"* Firebase Docs, 2025.

[7] Google Firebase, *"Realtime Database Overview,"* Firebase Docs, 2025.

[8] Google Firebase, *"Choose a Database: Cloud Firestore or Realtime Database,"* Firebase Blog, 2025.

[9] S. K. Verma et al., *"A Performance Comparison of SQLite and Firebase Databases from a Practical Perspective,"* ResearchGate, 2024.

[10]K. Anand and P. Nandhini, *"Hybrid Mobile App for Event Management using Flutter and*

Firebase," IJERT, vol. 13, no. 2, pp. 45–50, 2024.
[11]Google Firebase, *"Firebase Authentication and Security Rules,"* Docs, 2025.
[12]Firebase Developers Blog, *"Optimizing Firestore Performance and Cost,"* 2025.

[13]J. Lee et al., *"Synchronization and Conflict Resolution in Offline- First Mobile Apps,"* ACM Mobile Computing, vol. 23, no. 4, pp. 112–120,2023.
[14]ResearchGate, *"Literature Review on the Development of Mobile Applications for Academic Events,"* 2024.