Technical Questions

1. Backend Architecture: [Scaling to Support 1 Million Workspaces]

To support scalability as the user base and number of workspaces grow to 1 million or more, the backend should be designed with scalability, resilience, and modularity in mind. The following architectural strategies are recommended:

- ***Microservices Architecture***

Decompose the monolithic application into independent, loosely coupled services—such as AuthService, WorkspaceService, BoardService, and TaskService. This modular design allows:

  - Independent deployment and scaling of services.
  - Better fault isolation and maintainability.
  - Language and technology flexibility across services.

- ***Database Sharding***

Implement **horizontal partitioning** (sharding) of the database based on user ID or workspace ID. This helps:

  - Distribute workload across multiple database instances.
  - Reduce query latency under high load.
  - Prevent performance bottlenecks in single-node architectures.

- ***Caching Layer***

Incorporate a high-performance caching layer using **Redis** to:

  - Reduce database reads for frequently accessed data (e.g., workspace metadata, task summaries).
  - Improve latency and throughput of read-heavy operations.

- ***Asynchronous Processing***

Integrate message brokers like **RabbitMQ** or **Apache Kafka** to handle non-blocking, long-running tasks such as:

  - Email notifications.
  - Activity logging.

This decouples the user experience from backend processing and increases system responsiveness.

- ***API Gateway***

Utilize an API Gateway (**AWS API Gateway**, or **NGINX,** etc.) to:

  - Route and load balance traffic across microservices.
  - Enforce security policies (authentication, authorization).
  - Implement rate limiting, logging, and request transformation.

---

2. Authentication Strategy: [Securing Authentication across Mobile and Web Applications and Handle token expiration and refresh flows]

A robust authentication strategy is critical to ensuring secure access across mobile and web platforms. The implementation should combine industry standards with secure storage and proper token management. The following components form a comprehensive approach:

- ***Authentication Protocols***

Leverage modern authentication protocols such as:

- **OAuth 2.0 / OpenID Connect**: For federated login using identity providers like Google, Apple, or traditional email/password authentication.
- **JWT (JSON Web Tokens)**: Used for stateless authentication and role-based access control (RBAC), where tokens encode user roles, permissions, and metadata.

- ***Token Types and Storage***

Implement a dual-token system to balance usability and security:

- **Access Token (short-lived)**: Valid for 15 minutes to reduce exposure risk.
- **Refresh Token (long-lived)**: Valid for 7–30 days depending on security requirements.

Secure Token Storage:

- **Mobile**:
  - **iOS:** Store tokens in the Keychain.
  - **Android:** Use EncryptedSharedPreferences or Android Keystore.
- **Web**:
  - Use **HttpOnly**, Secure Cookies to prevent XSS attacks.
  - Avoid **localStorage** or **sessionStorage** for sensitive tokens.

- ***Token Expiration and Refresh Flows***

Ensure seamless user experience while maintaining security:

1. **Token Lifecycle**:
    - The access token is included in each API request.
    - When it expires, the client silently uses the refresh token to obtain a new access token.
2. **Refresh Flow Handling**:
    - On receiving a 401 Unauthorized response, a refresh token request is triggered.
    - Replace the old tokens on success and retry the original request.
3. **Failure Handling**:
    - If the refresh token is invalid or expired (e.g., revoked, tampered), clear stored credentials.
    - Force user logout and redirect to the login screen.

This mechanism ensures continued secure access without frequent interruptions while guarding against unauthorized access through compromised tokens.

---

3. Database Modeling: Structuring Users, Workspaces, Boards, and Tasks in Firestore

When designing a scalable and maintainable data model in Firestore, it's essential to embrace NoSQL principles like data denormalization, subcollections, and document references. Below is a robust schema tailored for collaboration-based applications like task managers or project boards.

- ***Core Entities and Their Attributes***

Each entity is represented as a collection of documents:

- **Users**

  users/{userId} → { name, email, photoUrl, etc. }

- **Workspaces**

  workspaces/{workspaceId} → { name, createdBY (ref: users/{userId}), createdAt, etc. }

- **Boards (within Workspaces)**

  workspaces/{workspaceId}/boards/{boardId} → { name, description, createdAt, etc. }

- **Tasks (within Boards)**

  workspaces/{workspaceId}/boards/{boardId}/tasks/{taskId} →
  { title, description, dueDate, assignedTo: [ref: users/{userId}], createdAt, etc. }

- ***Modeling Relationships***

While Firestore lacks support for native joins, relationships can still be represented using document references and subcollections and this how to model them:

- **One-to-Many**
  - **User → Workspaces (as owner)**
    - workspaces/{workspaceId} has an createdBy field referencing users/{userId}.
  - **Workspace → Boards**
    - Boards are modeled as subcollections under their parent workspace.
    - workspaces/{workspaceId}/boards/{boardId}
  - **Board → Tasks**
    - Tasks are stored in nested subcollection.
    - workspaces/{workspaceId}/boards/{boardId}/tasks/{taskId}
- **Many-to-Many**
  - **Users ↔ Workspaces (Members)**
    - Store an array of user references as member objects in each workspace document under a \`members\` field.
  - **Tasks ↔ Users (Assignees)**
    - Use an array of user references as member objects in the \`assignedTo\` field within each task document.

This data model ensures logical organization, high scalability, and efficient querying for collaborative applications in Firestore.

---

4. State Management (Frontend):

In Flutter development, choosing the right state management solution is critical for creating scalable, maintainable, and performant applications. Among the most widely adopted tools are **Provider**, **Riverpod**, and **BLoC (Business Logic Component)**. Each approach offers unique benefits and trade-offs, making it essential to evaluate them against the specific needs of the project. However, For this application, **BLoC** was the most suitable choice due to its:

- **Robustness in managing complex state flows**, such as transitions between Workspaces, Boards, and Tasks.
- **Strict separation of concerns**, promoting a clear distinction between UI and business logic.
- **Testability**, as business logic is encapsulated within blocs and can be independently tested.
- **Support for event-driven architecture**, ideal for real-time updates and multi-user collaboration.

- **Comprehensive Comparison Table :**

Feature/Criteria	Provider	Riverpod	BLoC
Boilerplate	Minimal setup; uses ChangeNotifier and Consumer widgets	Low boilerplate; relies on declarative providers	High; needs definition of events, states, blocs, and stream mapping
Architecture Fit	Loose structure; can grow messy in large apps	Promotes clean separation and immutability	Strong architectural enforcement; aligns with Clean Architecture principles

Feature/Criteria	Provider	Riverpod	BLoC
Scalability	Suitable for small to medium projects	Highly scalable; automatic disposal and modular provider hierarchy	Excellent for enterprise-scale apps with well-defined business rules
Reactivity Model	Basic; manual notification with notifyListeners()	Fine-grained reactivity with built-in caching and auto-dispose	Stream-based reactivity; handles complex async workflows elegantly
Performance	Good, but updates entire widget trees if not optimized	Excellent; only rebuilds affected widgets	Good, but higher memory overhead due to stream usage
Testing Support	Moderate; tightly coupled to UI in some cases	Strong; logic remains separate from widgets	Excellent; logic and state are fully decoupled, ideal for unit testing
Type Safety	Limited type safety and possible runtime issues	Full type safety with compile-time checks	Strong type safety with enforced state transitions
Code Maintainability	Degrades with scale; harder to manage complex logic	Clean, modular, and easy to refactor	High maintainability; each bloc encapsulates specific domain logic
IDE Tooling & Support	Excellent support with Flutter tools	Excellent; strong integration and auto-complete support	Excellent; full support in major IDEs like VS Code and Android Studio

5. Offline Handling

Ensuring robust offline support is critical for enhancing user experience, particularly in environments with intermittent connectivity. To enable users to continue working seamlessly offline and synchronize their changes once connectivity is restored, the following strategies can be implemented:

- ***Local Data Persistence :***

Implementing reliable local storage is the foundation of offline-first functionality:

  - **Hive / SQLite**: Ideal for storing structured data such as workspaces, tasks, and user metadata locally.
  - **Shared Preferences**: Suitable for persisting lightweight configuration or frequently accessed preferences.
  - **Caching**: Cache backend responses (e.g., boards and task lists) locally to reduce network dependency and enhance load performance.

- ***Change Tracking Mechanism :***

To support deferred synchronization, track and store user interactions performed while offline:

  - Maintain an **operation queue** locally that records CRUD operations with timestamps.
  - Classify operations (create, update, delete) and preserve metadata required for eventual synchronization.
  - Implement conflict resolution policies (e.g., last-write-wins or manual resolution prompts) on sync.

- ***Synchronization Strategy :***

Once the device regains internet connectivity, synchronize the local changes with the backend:

  - **Batched Sync**: Group multiple queued operations into a single network request to optimize bandwidth and ensure transactional integrity.
  - **Conflict Handling**: Detect discrepancies during sync and resolve using rules or user intervention.

- ***Firebase Firestore Integration :***

Firestore provides native offline support for both mobile and web:

    - Automatically caches documents for offline use.
    - Queues and syncs write operations when the network is restored.
    - However, manual control is needed for:
        - Fine-grained caching strategies.
        - Custom conflict resolution (especially for collaborative multi-user updates).

- ***Connectivity Monitoring :***

Efficient synchronization depends on accurate network status monitoring:

    - Use packages such as ***connectivity_plus*** or ***internet_connection_checker*** to:
        - Monitor network status changes in real-time.
        - Trigger queued sync operations immediately upon reconnecting.
        - Notify users of offline status and pending actions.
