Decision-Making Report

1. Backend Choice :

Firebase Functions

Firebase Functions was selected as the backend solution due to its seamless integration with Firebase Authentication, Firestore, and Firebase Storage — making it ideal for a serverless architecture in a Flutter application.

Key Factors Influencing the Decision:

- **Integration Speed:** Firebase Functions tightly integrate with other Firebase services, significantly reducing setup time and boilerplate code.
- **Cost Efficiency:** Firebase offers a generous free tier, and costs scale based on usage, which is optimal for early-stage and growing apps.
- **Auto Scaling:** Functions automatically scale with demand, eliminating the need for manual server provisioning.
- **Security & Maintenance:** With managed infrastructure, there’s no need to manage servers, patches, or scaling logic manually.

Comparison:

- **Laravel API:** Requires backend hosting and scaling setup; better for traditional web apps but slower to integrate with Firebase-native tools.
- **Supabase Edge Functions:** Good Postgres integration, but limited compared to Firebase's serverless maturity and real-time database support.
- **Python Flask/FastAPI:** Great flexibility, but requires full backend infrastructure, monitoring, and manual auth integration.

2. Database Choice :

<u>Firebase Firestore</u>

Firestore offers real-time data synchronization, offline support, and a flexible, schema-less structure — making it a strong choice for collaborative and mobile-first applications.

Firestore (Firebase) benefits:

- Real-time updates (ideal for chat, task collaboration).
- Native Flutter SDK support.
- Seamless Firebase Authentication integration.

Advantages Over Alternatives:

- **Supabase (PostgreSQL):** Structured and powerful for relational queries but adds complexity for highly dynamic, nested data like members, task assignments and task attachments.
- **MySQL:** Robust for strict schemas but lacks real-time capabilities and needs manual sync logic for Flutter.

3. Storage :

Firebase Storage

Firebase Storage is optimized for file uploads directly from mobile clients with built-in authentication support, making it a natural fit for a Flutter + Firebase stack.

Firebase Storage Advantages:

- Strong security via Firebase rules.
- Direct file uploads from client-side.
- Simple download URL management.

Comparison:

- **Amazon S3:** More configurable and scalable for enterprise-scale apps, but requires additional effort to integrate with Firebase Auth and manage access rules.
- **Supabase Storage:** Good for PostgreSQL integration but lacks the maturity and tooling Firebase offers for mobile-first development.

4. Implementation plan :

High-level overview of how database and storage are structured and connected to the app:

FocusFlow uses *Firebase* as the backend service to handle authentication, real-time data storage, and file management. The app is structured using a *Clean Architecture* approach, separating logic into Data, Domain, and Presentation layers.

- **Authentication**: Firebase Authentication manages user sign-in/sign-up, and user data is stored in **Cloud Firestore** under the /users collection.
- **Firestore Database**: Structured into collections for users, workspaces, boards, and tasks. Each entity has its own subcollections and is linked via IDs for nested relationships (e.g., boards under a workspace, tasks under a board).
- **Firebase Storage**: Planned for future use to handle file uploads like attachments, avatars, and profile pictures.
- **Data Layer**: Implements repository interfaces using Firebase as the source of truth. It handles all reads/writes from Firestore and Storage.
- **Presentation Layer**: Uses **Cubit** (Bloc) for reactive state management. UI reacts to Firebase data changes via streams or real-time listeners.

This setup allows modular, scalable interaction between the app and Firebase services, ensuring seamless real-time collaboration and secure data handling.