**Nimbus Architectural Design Document**

## 1. High-Level Overview

Nimbus is a cloud-based API built with .NET 8 to handle file and folder management. It provides endpoints for file uploads, retrieval, and folder structure management. The system is designed to be flexible and scalable, serving as a backend for a cloud storage solution, easily integrating into different environments.

## 2. Architecture Overview

Nimbus follows a multi-layered architecture that separates responsibilities into distinct layers to ensure flexibility, scalability, and maintainability:

- **API Layer**: This layer handles incoming HTTP requests, performs input validation, authentication, and routes requests to the business logic layer. It's where all the user-facing interactions with the system happen.
- **Business Layer**: Contains the core logic for file management, such as validating folder actions (e.g., preventing the creation of a folder on the same level as the root folder) and orchestrating operations that need to be performed on files and folders such as recursively delete a folder and its subfolders.
- **Persistence Layer**: Manages interactions with the database. Currently, both metadata and file content are stored in a single relational table. The database supports atomic operations to ensure consistency across file and metadata storage.

## 3. Core Components

1. **API Layer**:
    - Built with ASP.NET Core.
    - API endpoints follow REST principles to manage files and folders, including operations like upload, retrieve, and delete.
    - Implements middleware for authentication (via API key) and global error handling.
    - Future improvements: use of specific user facing DTOs, increase consistency in endpoints definitions.
2. **Business Logic Layer**:
    - Services responsible for validating business rules and orchestrating the file operations.
    - Future improvements include better layer decoupling through patterns like MediatR.
3. **Persistence Layer**:
    - Uses Entity Framework Core with the Unit of Work and Repository patterns to handle file metadata.
    - Currently, file content is stored directly in the database, but future improvements will split this into a dedicated file content repository, potentially using Blob Storage or a filesystem for better scalability. UOW will guarantee atomicity of transactions.

## 4. Architectural Decisions

- **Database Structure**: The metadata is stored in a relational database (SQL Server), which is well-suited for structured data and allows for complex queries on file attributes. The file content itself is stored alongside metadata in the database for simplicity, but this could be split into another storage solution (such as blob storage) in future versions.
- **API Gateway**: While not implemented yet, in a full-scale deployment, the API could sit behind an API Gateway to handle routing, rate limiting, and centralized monitoring. This would also add a layer of security and reliability in production environments.
- **Horizontal Scaling**: Currently, the API is designed for vertical scaling, but it's structured in a way that it could be horizontally scaled if needed, especially once services are decoupled more effectively.

**5. Future Points of Improvement**

1. **Layer Separation and Decoupling**:
    - Introduce dedicated DTOs for each layer (API, Business, Persistence) to ensure a clear separation of concerns.
    - Use the MediatR package to further decouple the API controllers from the business logic, making the system more maintainable.
2. **Splitting File Persistence**:
    - Refactor the file persistence layer:
        - Metadata Repository: Continue to store metadata in a relational database.
        - File Content Repository: Transition to a more scalable solution (e.g., Blob Storage or a dedicated file system).
        - Use the Unit of Work pattern to ensure both persistence actions happen atomically.
3. **API Cleanup:**
    - Move the GetFileListByFolderId method from the FileController to the FolderController, as it makes more sense semantically for folder management.
    - Define consistently named endpoint for each controller.
    - Implement missing endpoints (e.g. patch for updating existing items)
4. **Authentication Enhancements**:
    - Transition from API key to JWT authentication to support user roles and more fine-grained access control.
    - Implement a claims-based access control mechanism to enforce specific file access based on user roles.
5. **Testing and CI/CD**:
    - Expand unit tests to cover more edge cases and validate business rules.
    - Implement automated integration tests to ensure the entire system works as expected.
    - Set up CI/CD pipelines (e.g., with GitHub Actions or Azure DevOps) to automate builds, run tests, and deploy to cloud environments like Azure or AWS.
6. **Documentation**:
    - Provide more detailed documentation covering endpoint usage, expected input/output, and sample API requests/responses.