Pre-Coding Preparation for Creating a Cloud Storage System in C++

#### 1. Requirement Gathering and Analysis

- \*\*Define the Scope\*\*: Identify the key features and functionalities your cloud storage system will include, such as:

- \*\*File Upload and Download\*\*: Ensure that users can upload and download files of varying sizes. Include functionality for resumable uploads, chunk-based uploads for large files, and error handling for network interruptions.

- \*\*User Authentication and Account Management\*\*: Integrate multi-factor authentication (MFA) for enhanced security. Implement OAuth2 for third-party logins if needed.

- \*\*Data Encryption\*\*: Use symmetric encryption (e.g., AES-256) for data at rest and asymmetric encryption (e.g., RSA) for key exchange. Ensure TLS (Transport Layer Security) for data in transit.

- \*\*Metadata Handling\*\*: Design metadata management that includes detailed information such as file versions, creation/modification timestamps, file ownership, and access control lists (ACLs).

- \*\*User Stories\*\*: Create detailed user stories to capture potential user interactions. Examples include:

- "As a user, I want to upload files to the cloud so that I can access them from anywhere."

- "As an admin, I want to manage user accounts to ensure system security and compliance."

- "As a user, I want to share files with other users with customizable access permissions."

- \*\*Identify Constraints\*\*: List and detail technical limitations and requirements such as:

- \*\*Storage Capacity\*\*: Estimate space requirements based on expected user growth.

- \*\*Bandwidth Usage\*\*: Analyze bandwidth requirements, considering peak times and load balancing.

- \*\*Budget\*\*: Account for costs related to server maintenance, software licenses, and potential cloud service fees.

- \*\*Compliance\*\*: Ensure the system meets data protection regulations like GDPR or CCPA if applicable.

#### 2. System Design and Architecture

- \*\*High-Level Architecture\*\*: Sketch diagrams to represent the structure, showing interactions between key components such as:

- \*\*Client-Side Application\*\*: Handles user interactions, UI/UX, and communicates with the backend via RESTful or gRPC APIs.

- \*\*Web Server\*\*: Processes incoming HTTP or HTTPS requests, manages sessions, and routes API calls.

- \*\*File Storage System\*\*: Utilizes local disk storage, network-attached storage (NAS), or cloud-based solutions like AWS S3.

- \*\*Database\*\*: Stores user credentials, file metadata, and logs. Ensure database scalability by implementing sharding and replication.

- \*\*Authentication Service\*\*: Manages user login, session tokens, and MFA.

- \*\*Encryption Service\*\*: Integrates OpenSSL for C++ to handle encryption and decryption of files and data.

- \*\*Component Breakdown\*\*:

- \*\*Networking Module\*\*:

- Use libraries like Boost.Asio for managing TCP/HTTP communication.

- Implement rate-limiting and request throttling to prevent DDoS attacks.

- \*\*File Handling Module\*\*:

- Develop functions for reading, writing, and deleting files using POSIX or C++17 filesystem library.

- Include a background job system for tasks like file compression or virus scanning.

- \*\*Authentication Module\*\*:

- Use password hashing libraries (e.g., Argon2 or bcrypt) to securely store user credentials.

- Implement JWT (JSON Web Token) for stateless session management.

- \*\*Encryption Service\*\*:

- Integrate OpenSSL or Botan for symmetric and asymmetric encryption.

- Use HMAC (Hash-based Message Authentication Code) for verifying data integrity.

- \*\*Data Models\*\*:

- \*\*User Table\*\*:

- Fields: `user\_id` (primary key, UUID), `username`, `hashed\_password`, `email`, `created\_at`, `last\_login`, `MFA\_enabled`.

- \*\*File Metadata Table\*\*:

- Fields: `file\_id` (primary key, UUID), `user\_id` (foreign key), `file\_path`, `file\_size`, `upload\_date`, `last\_modified`, `file\_version`, `permissions`.

- \*\*Access Log Table\*\*:

- Fields: `log\_id`, `user\_id`, `action`, `timestamp`, `IP\_address`, `device\_info`.

#### 3. Technology Stack Selection

- \*\*Programming Language and Frameworks\*\*:

- Primary Language: \*\*C++ (C++17 or later)\*\*

- \*\*Networking Library\*\*: Boost.Beast for HTTP handling or cpp-httplib for simpler implementations.

- \*\*Database Library\*\*: SOCI or libpqxx for interfacing with SQL databases.

- \*\*Encryption Libraries\*\*: OpenSSL for cryptographic operations.

- \*\*JSON Parsing\*\*: Use nlohmann/json for handling JSON data in API responses.

- \*\*Database Options\*\*:

- \*\*PostgreSQL\*\* for robust relational database management with support for advanced features like JSONB.

- \*\*SQLite\*\* for lightweight, embedded use cases.

- \*\*Security Tools\*\*:

- Ensure OpenSSL is compiled with TLS 1.2+ support.

- Set up DDoS protection using tools like fail2ban or cloud-based services (e.g., Cloudflare).

#### 4. Proof of Concept (PoC)

- \*\*Prototype Development\*\*:

- Create a minimal HTTP server using Boost.Beast.

- Implement a basic file upload endpoint that accepts multipart form data.

- Test file upload and download with curl or Postman.

- \*\*Benchmarking\*\*:

- Measure latency, file transfer speeds, and response times.

- Run basic stress tests using Apache JMeter or locust.io to simulate multiple concurrent users.

#### 5. Project Planning

- \*\*Timeline and Milestones\*\*:

- \*\*Phase 1\*\*: Build the server framework, establish basic file handling functionality (2-3 weeks).

- \*\*Phase 2\*\*: Develop and test the user authentication module, integrate JWT (2 weeks).

- \*\*Phase 3\*\*: Implement encryption and secure storage protocols (3 weeks).

- \*\*Phase 4\*\*: Develop the client application (e.g., CLI, desktop app) and connect with the server (3 weeks).

- \*\*Phase 5\*\*: Conduct integration testing, load testing, and optimize the system for performance (1-2 weeks).

- \*\*Task Assignments\*\*:

- \*\*Networking\*\*: Assign a team member with expertise in socket programming.

- \*\*Security and Encryption\*\*: Delegate to someone familiar with cryptography libraries.

- \*\*Database Management\*\*: Assign to a member experienced in SQL schema design and optimization.

- \*\*Version Control\*\*: Use Git for tracking code changes. Implement branching strategies (e.g., Git Flow) for efficient development.

#### 6. Detailed Algorithm Planning

- \*\*Pseudo Code\*\*:

- \*\*File Upload Function\*\*:

void handleUpload(Request req) {

authenticateUser(req);

validateFileSize(req.file);

saveFileToDisk(req.file);

storeMetadataToDatabase(req.fileMetadata);

respondSuccess();

}

```

- \*\*Encryption Process\*\*:

``

void encryptFile(std::string filePath) {

std::ifstream inFile(filePath, std::ios::binary);

std::ofstream outFile(filePath + ".enc", std::ios::binary);

AES\_encrypt(inFile, outFile, encryptionKey);

inFile.close();

outFile.close();

}

```

- \*\*Edge Case Identification\*\*:

- \*\*Large File Uploads\*\*: Implement chunked file transfer with resumable upload capability.

- \*\*Concurrent User Access\*\*: Use file locking mechanisms to prevent race conditions.

- \*\*File Corruption Handling\*\*: Add checksums or hash verification (e.g., SHA-256) to detect file integrity issues.

- \*\*Timeouts and Retries\*\*: Implement exponential backoff for retrying failed connections.

#### 7. Testing Plan

- \*\*Testing Strategy\*\*:

- \*\*Unit Tests\*\*: Use Google Test (gtest) for C++ to test individual functions like `authenticateUser()`.

- \*\*Integration Tests\*\*: Simulate interactions between components using tools like CTest.

- \*\*Load Tests\*\*: Stress test the system with simulated user loads using Apache JMeter.

- \*\*Security Testing\*\*:

- \*\*Penetration Tests\*\*: Run basic security tests to identify vulnerabilities (e.g., SQL injection, XSS).

- \*\*Code Analysis\*\*: Use tools like cppcheck and clang-tidy for static code analysis.

- \*\*Environment Setup\*\*:

- \*\*Local Server Development\*\*: Use Docker to set up containers for local testing.

- \*\*Virtual Machines\*\*: Test distributed deployment on VM instances to ensure compatibility and load handling.

- \*\*Deployment\*\*: Plan for cloud-based deployment with providers like AWS EC2 or DigitalOcean, using CI/CD pipelines for continuous integration.