Field of computer Science & Engineering

Cloud Computing

جامعة الجلالة
GALALA UNIVERSITY

Project Title	Cloud file Uploader	
Team	Ahmed Hossam	221100101
	Ahmed Samir	221101415
	Kareem Tarek	221101524
	Amro Khaled	221100199
	Mazen Hossam	221100514
	Alaa Amr	221101154
Instructor	**Dr. Ahmed Elmasry**	

Abstract:

This project presents a web-based **Cloud File Uploader** system that enables users to securely register, log in, upload, and manage personal files via a browser interface. Built using **Node.js**, **Express**, and **Multer**, the system connects to a **MongoDB replica set** deployed across **Docker containers**, ensuring high availability and fault tolerance. The backend is designed to create isolated storage for each user, while the frontend provides an intuitive interface for interaction. Sessions are managed using Express middleware, and user-specific directories store uploaded files securely. The system was deployed across two VMs with shared storage, replicating cloud-based distributed environments. This project showcases key cloud computing principles such as scalability, availability, containerization, and service isolation.

Introduction:

In today's digital era, cloud storage solutions have become an essential part of personal and enterprise infrastructure. With the increasing demand for online file storage, sharing, and accessibility, this project introduces a simplified cloud-based file uploader system tailored for academic and practical deployment. Built using modern backend technologies like Node.js and MongoDB, and deployed within Docker containers, the application provides core functionalities such as user registration, authentication, file uploading, and retrieval. The system simulates a real-world distributed cloud environment using replica sets, shared volumes, and containerized services to demonstrate availability, data isolation, and horizontal scalability. The Cloud File Uploader offers an interactive frontend and showcases vital cloud computing concepts aligned with academic objectives.

Problem definition:

Cloud storage systems often face challenges related to data availability, consistency, user isolation, and secure access. Traditional single-server architectures are prone to failures and lack scalability. Moreover, storing user-uploaded content securely, managing concurrent access, and handling failovers are critical design concerns in cloud environments.

This project aims to solve these issues by:

- Designing a **multi-container architecture** with Docker.
- Implementing **MongoDB replication** for high availability.
- Developing a **session-based web app** that isolates user files.
- Ensuring **dynamic file storage per user**, hosted in a shared and replicated backend.

By addressing these issues, the project provides a cloud-native file uploader system demonstrating both technical skills and cloud design principles.

Objectives

The primary objectives of the Cloud File Uploader project are:

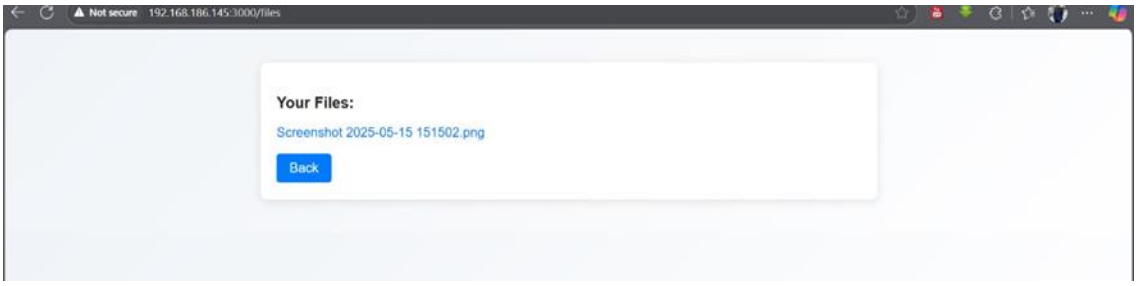
- To build a **secure cloud-based platform** for user file uploads and management.
- To enable **user registration and login** with individual file storage directories.
  To implement **session-based authentication** for secure access control.
- To containerize the backend and database services using **Docker** for portability and isolation.
- To deploy a **MongoDB Replica Set (Primary, Secondary, Arbiter)** for improved availability and fault tolerance.
- To mount and use **shared storage volumes** across virtual machines for persistent data access.
- To deliver a clean and intuitive **frontend interface** for user interaction.
- To simulate a real-world **distributed cloud system** using multiple virtual machines.

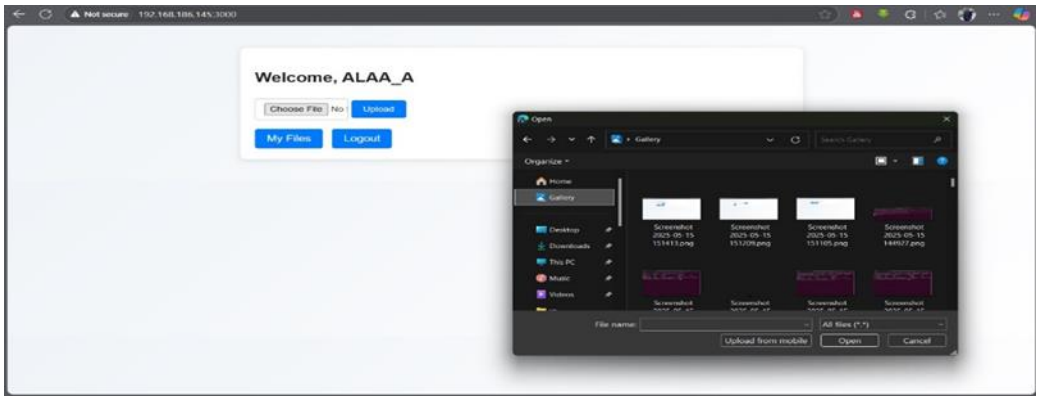
System Architecture

The system architecture is distributed across two virtual machines and includes multiple containers managed using Docker. The key components are:

Frontend

- Delivered via the Node.js app.
- Renders responsive HTML pages using server-side rendering.
- Users interact with forms for registration, login, file upload, and download.





Backend

- Built on Node.js with Express and Multer.
- Handles API routing, file storage, and session control.
- Creates user-specific directories for file isolation.
- Utilizes Express-session for stateful user sessions.

Open index.js ~/app

```
require('dotenv').config();
const express   = require('express');
const session   = require('express-session');
const mongoose  = require('mongoose');
const multer    = require('multer');
const fs        = require('fs-extra');
const path      = require('path');

const app  = express();
const PORT = process.env.PORT || 3000;

// ensure host upload folder
const UPLOAD_BASE = path.join(__dirname, 'data', 'uploads');
fs.ensureDirSync(UPLOAD_BASE);

// connect to MongoDB
console.log('→ connecting to:', process.env.MONGO_URI);
mongoose
  .connect(process.env.MONGO_URI, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log('✅ MongoDB connected'))
  .catch(err => console.error('❌ MongoDB connection error:', err));

// user schema (plain-text password)
const userSchema = new mongoose.Schema({
  username: { type: String, unique: true, required: true },
  password: { type: String, required: true }
```

Open index.js ~/app

```
});
const User = mongoose.model('User', userSchema);

// middleware
app.use(express.urlencoded({ extended: true }));
app.use(express.json());
app.use(session({
  secret: 'cloud-secret',
  resave: false,
  saveUninitialized: false
}));

// multer storage → data/uploads/<username>
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    if (!req.session.username) return cb(new Error('Not logged in'), null);
    const userFolder = path.join(UPLOAD_BASE, req.session.username);
    fs.ensureDirSync(userFolder);
    cb(null, userFolder);
  },
  filename: (req, file, cb) => cb(null, file.originalname)
});
const upload = multer({ storage });

// helper to render full HTML with inline CSS
const renderPage = (title, bodyContent) => `<!DOCTYPE html>
<html lang="en">
<head>
```

```
Open                    index.js
                        ~/app
// list user files
app.get('/files', (req, res) => {
  if (!req.session.username) return res.status(403).send(renderPage('Error', `<div class="container">Not logged
in.<br><a class="button" href="/login">Login</a></div>`));
  const dir = path.join(UPLOAD_BASE, req.session.username);

  fs.readdir(dir, (err, files) => {
    if (err) return res.status(500).send(renderPage('Error', `<div class="container">Error reading files.<br><a
class="button" href="/">Back</a></div>`));
    const list = files.map(f => `<li><a href="/uploads/${encodeURIComponent(req.session.username)}/$
{encodeURIComponent(f)}" target="_blank">${f}</a></li>`).join('');
    res.send(renderPage('My Files', `
      <div class="container">
        <h3>Your Files:</h3>
        <ul>${list}</ul>
        <a class="button" href="/">Back</a>
      </div>
    `));
  });
});

// logout
app.get('/logout', (req, res) => {
  req.session.destroy(() => res.redirect('/'));
});

// start server
app.listen(PORT, () => console.log(`🚀 Server running at http://localhost:${PORT}`));
```

Database Layer

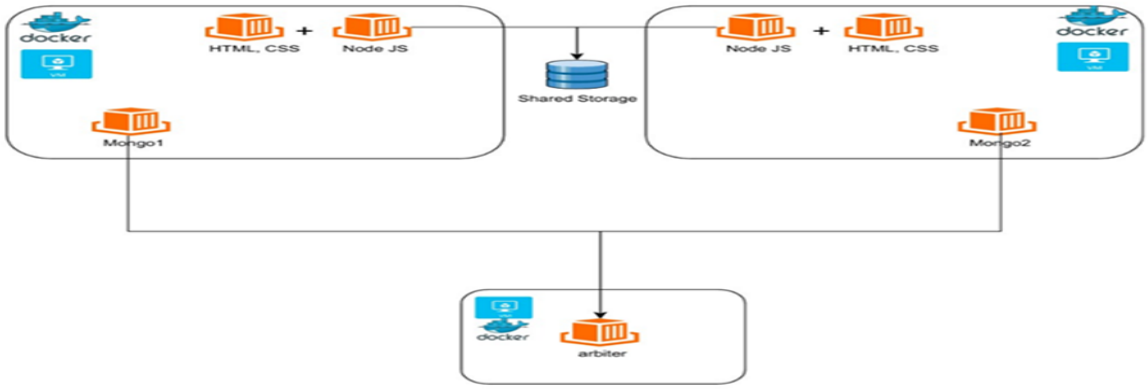
- A **MongoDB replica set** with:
  - **mongo1** (Primary)
  - **mongo2** (Secondary)
  - **arbiter** (Votes in case of primary failure)
- All containers communicate over a virtual network.
- Provides redundancy and automatic failover.

Shared Storage

- User file directories stored on a shared volume accessible by both VMs.
- Mounted using vmhgfs-fuse to simulate cloud storage persistence.

Deployment

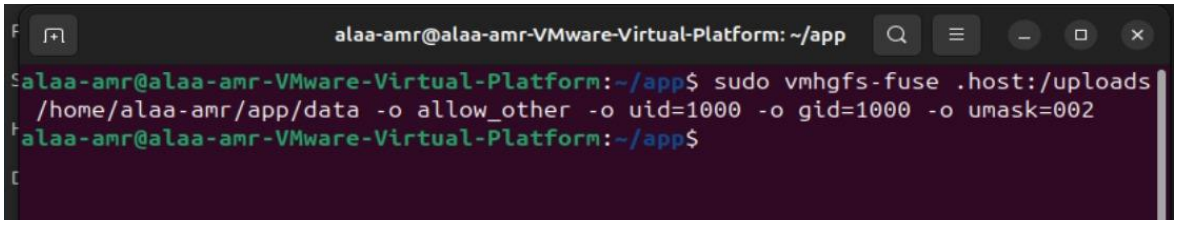
- Each VM runs Node.js and MongoDB containers.
- The Node.js container connects to the database through the replica set URI.

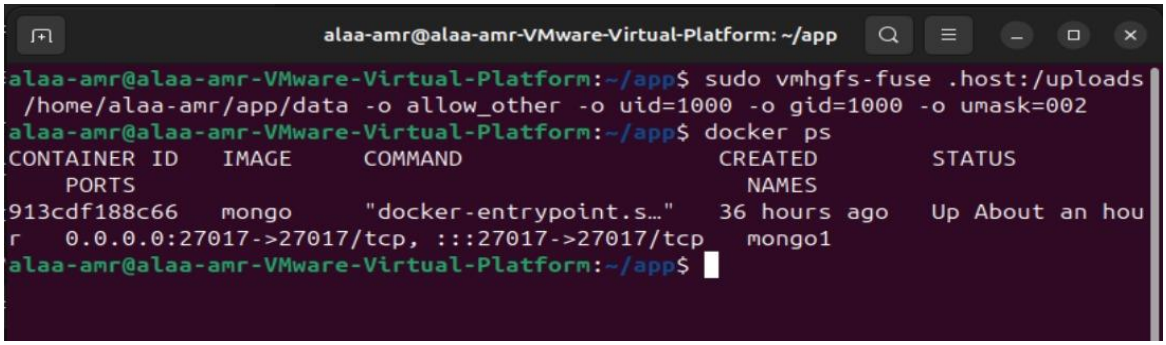


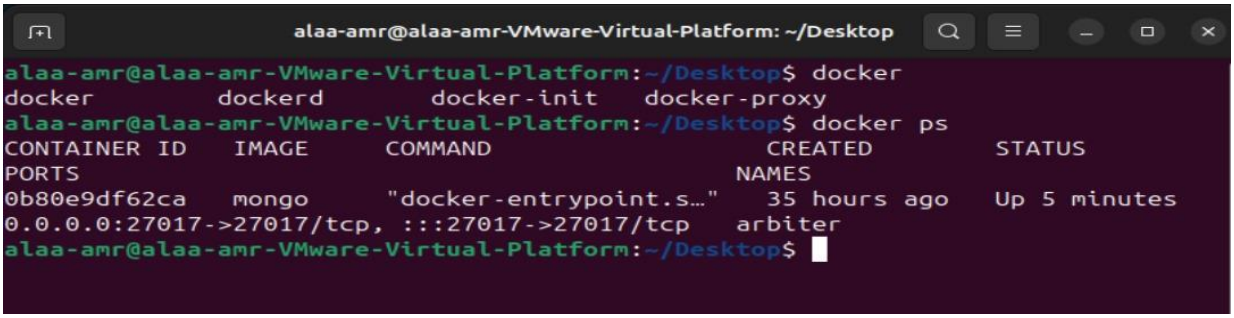
Working Principle

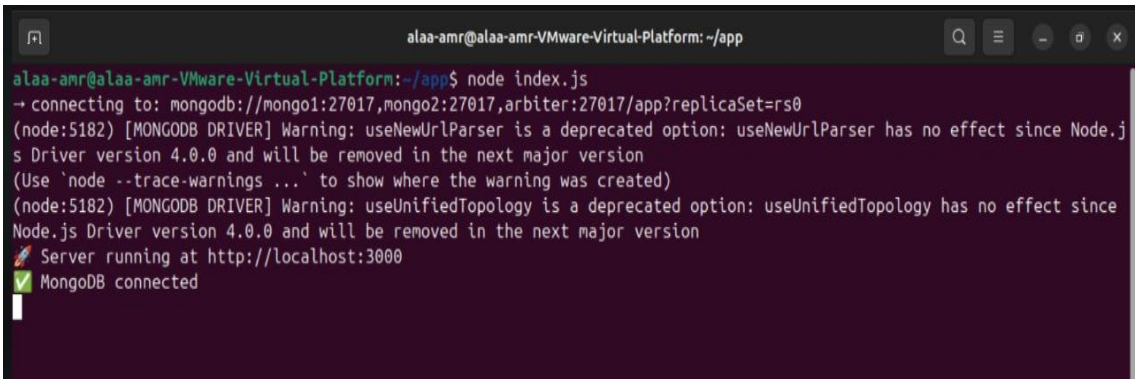
The system follows a clear user-driven flow from registration to logout:

1. **User Registration**
   When a new user signs up, their credentials are stored in the MongoDB replica set. A dedicated directory is automatically created for their future uploads.
2. **User Login**
   Returning users can log in using their credentials. Once authenticated, a session is created, giving access to upload and file management features.
3. **File Uploading**
   Authenticated users upload files through a form. Each file is saved in the user's personal directory, ensuring isolation between users.
4. **File Listing & Download**
   Users can view all files they've uploaded, presented in a list. Files are accessible for download directly through the web interface.
5. **Logout**
   Logging out ends the session and prevents unauthorized access to user files or upload features.
6. **Database Availability**
   MongoDB's replica set ensures continuous availability. If the primary node fails, a secondary is promoted automatically with help from an arbiter node.


alaa-amr@alaa-amr-VMware-Virtual-Platform: ~/app
alaa-amr@alaa-amr-VMware-Virtual-Platform:~/app$ sudo vmhgfs-fuse .host:/uploads
/home/alaa-amr/app/data -o allow_other -o uid=1000 -o gid=1000 -o umask=002
alaa-amr@alaa-amr-VMware-Virtual-Platform:~/app$


alaa-amr@alaa-amr-VMware-Virtual-Platform: ~/app
alaa-amr@alaa-amr-VMware-Virtual-Platform:~/app$ sudo vmhgfs-fuse .host:/uploads
/home/alaa-amr/app/data -o allow_other -o uid=1000 -o gid=1000 -o umask=002
alaa-amr@alaa-amr-VMware-Virtual-Platform:~/app$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
913cdf188c66 mongo "docker-entrypoint.s…" 36 hours ago Up About an hou
r 0.0.0.0:27017->27017/tcp, :::27017->27017/tcp mongo1
alaa-amr@alaa-amr-VMware-Virtual-Platform:~/app$


alaa-amr@alaa-amr-VMware-Virtual-Platform: ~/Desktop
alaa-amr@alaa-amr-VMware-Virtual-Platform:~/Desktop$ docker
docker dockerd docker-init docker-proxy
alaa-amr@alaa-amr-VMware-Virtual-Platform:~/Desktop$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
0b80e9df62ca mongo "docker-entrypoint.s…" 35 hours ago Up 5 minutes
0.0.0.0:27017->27017/tcp, :::27017->27017/tcp arbiter
alaa-amr@alaa-amr-VMware-Virtual-Platform:~/Desktop$


alaa-amr@alaa-amr-VMware-Virtual-Platform: ~/app
alaa-amr@alaa-amr-VMware-Virtual-Platform:~/app$ node index.js
→ connecting to: mongodb://mongo1:27017,mongo2:27017,arbiter:27017/app?replicaSet=rs0
(node:5182) [MONGODB DRIVER] Warning: useNewUrlParser is a deprecated option: useNewUrlParser has no effect since Node.j
s Driver version 4.0.0 and will be removed in the next major version
(Use `node --trace-warnings ...` to show where the warning was created)
(node:5182) [MONGODB DRIVER] Warning: useUnifiedTopology is a deprecated option: useUnifiedTopology has no effect since
Node.js Driver version 4.0.0 and will be removed in the next major version
Server running at http://localhost:3000
MongoDB connected

Implementation

Backend

The backend handles user authentication, file handling, and session control. It ensures that all uploaded files are stored in user-specific folders and that unauthorized access is prevented through active session checks.

Database

The application uses a three-node MongoDB replica set to ensure data replication and fault tolerance. This guarantees availability even if one database node becomes unavailable.

File Storage

Uploaded files are stored in isolated directories for each user. The file system is managed dynamically to ensure each user's data remains separated and organized.

Dockerized Deployment

All services — backend, database, and supporting components — are containerized using Docker. The setup runs across two virtual machines, sharing volumes and enabling a realistic simulation of a distributed cloud system.

Shared Storage

To allow both VMs to access uploaded files, shared volumes are mounted across them. This ensures consistency and availability of user files regardless of which backend instance is serving the request.

Methodology

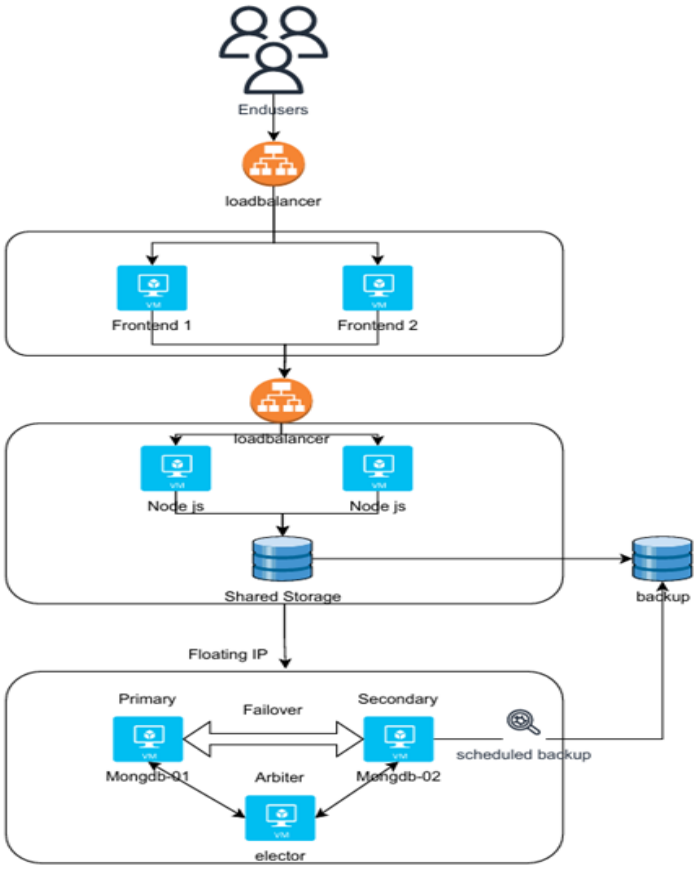
The development of the Cloud File Uploader project followed a modular and layered methodology, combining containerized deployment, scalable backend services, and user-focused frontend design. The implementation was structured in the following phases:

1. System Planning and Requirement Analysis

The team identified the core functional requirements: user authentication, secure file uploads, session management, and high-availability storage. Based on these, a microservice-like structure was proposed using Docker and MongoDB replica sets.

2. Backend Development

A backend service was developed using Node.js and Express to handle HTTP requests and user sessions. RESTful routes were defined for user registration, login, file upload, and listing. Multer was configured to manage file uploads into dynamically created directories based on user sessions. Middleware was used to handle session persistence and restrict unauthorized access.



3. Database Configuration

MongoDB was deployed in a three-node replica set within Docker containers — including one primary, one secondary, and one arbiter node. The replica set ensured that user data remained available and consistent even during node failure. This setup simulated a high-availability production-grade database system.

4. Frontend Integration

Simple yet effective HTML pages were rendered server-side to enable registration, login, upload, and file browsing functionality. Each page was styled using inline CSS to ensure a uniform and responsive user experience. The frontend was tightly integrated with backend logic for seamless flow.

5. Dockerization and VM Deployment

All services — MongoDB, Node.js backend, and shared volumes — were containerized using Docker. The containers were deployed across two virtual machines to reflect a distributed cloud setup. Shared directories were mounted using a virtual file system interface to replicate persistent cloud storage across hosts.

6. Testing and Validation

The complete system was tested under multiple scenarios, including successful and failed login attempts, file upload consistency, session validation, and node failover behavior. Docker logs and MongoDB replica set status were regularly monitored to ensure system integrity and uptime.

Features

The Cloud File Uploader project includes a range of core and supportive features designed to ensure a smooth user experience and strong backend reliability.

User Authentication

- Allows users to register with a unique username and password.
- Login mechanism ensures only authenticated users can access file-related functionalities.
- Session management is implemented to maintain user state securely during navigation.

User-Specific File Storage

- Each user has a dedicated directory where uploaded files are stored.
- Ensures data isolation between users and prevents unauthorized file access.

File Upload & Retrieval

- Users can upload multiple types of files through a simple interface.
- Uploaded files are listed in a personal dashboard with links to access or download them.

Web-Based Interface

- Minimalistic HTML pages rendered from the backend.
- Styled with inline CSS for clarity and responsiveness.

- Pages include welcome, registration, login, dashboard, and file listing views.

Dockerized Deployment

- Entire system (Node.js backend, MongoDB replica set) runs in Docker containers.
- Allows for consistent, portable, and easily replicable deployments.

MongoDB Replica Set

- Deployed across three containers: primary, secondary, and arbiter.
- Provides high availability, redundancy, and automatic failover capability.

Shared Persistent Storage

- User data is stored on shared volumes mounted across virtual machines.
- Ensures consistent access to files regardless of the VM serving the request.

Scalable Cloud Simulation

- Multi-VM architecture reflects real-world distributed systems.
- Can be extended easily to support more instances or services.

Challenges Faced

Throughout the development and deployment of the Cloud File Uploader system, the team encountered several technical and architectural challenges, which were addressed through iterative problem-solving and testing.

1. MongoDB Replica Set Configuration in Docker

Setting up a MongoDB replica set across three Docker containers (primary, secondary, arbiter) required precise coordination. Ensuring all nodes could discover each other and elect a primary was challenging due to network configurations and container linking. This required additional work to configure Docker networks and replica initiation scripts properly.

2. Docker-VM Shared Volume Integration

One major hurdle was enabling shared access to uploaded files between multiple VMs. Mounting shared directories (/uploads) using vmhgfs-fuse introduced permission issues and performance limitations, which had to be resolved using specific mount options and consistent folder structures across hosts.

3. Session Management and Route Protection

Ensuring that users could not bypass authentication and access other users' files required strict session validation across all routes. Improper middleware usage initially led to unauthorized access paths, which were patched through route restructuring and session checks.

4. Frontend Responsiveness and Styling

Since HTML pages were rendered directly from the backend without modern frontend frameworks, styling and layout had to be carefully handled using inline CSS. Creating a clean and responsive layout without external libraries was time-consuming and limited in flexibility.

5. Password Handling (Security Considerations)

While the system functioned as expected, passwords were stored in plaintext due to simplicity. This posed a security risk and highlighted the need for hashing mechanisms in real-world systems — something noted for future improvements.

6. Testing in a Distributed Environment

Testing the full flow (login → upload → view → logout) across two virtual machines required syncing container states, MongoDB replica status, and user session behavior. Downtime simulation and failover validation were difficult to observe without full monitoring tools.

Future Work

While the current implementation of the Cloud File Uploader achieves its core objectives, several enhancements can be made to improve its **security**, **usability**, and **scalability** for real-world deployment.

1. Secure Password Storage

Currently, passwords are stored in plaintext, which is a critical security risk. Future iterations should implement **password hashing** using algorithms like **bcrypt** to protect user credentials.

2. File Metadata and Search

Adding a database layer to store **file metadata** (upload date, size, type) would enable advanced file management features, including **searching**, **sorting**, and **filtering** uploaded files.

3. Multiple File Uploads and Drag-and-Drop Support

The frontend could be improved to allow **bulk uploads**, drag-and-drop functionality, and real-time progress indicators to enhance user experience.

4. Cloud Hosting and Autoscaling

The system can be deployed on cloud platforms like **AWS**, **Azure**, or **Google Cloud**, using services like **Kubernetes** to scale containers based on user demand. Load balancers and health checks can improve uptime and traffic handling.

5. Monitoring and Logging

Integrating tools such as **Prometheus** for monitoring and **ELK stack (Elasticsearch, Logstash, Kibana)** for logging can help track usage, performance, and error analysis across services.

6. Mobile Compatibility

Improving frontend responsiveness and implementing a mobile-friendly interface would make the platform accessible from smartphones and tablets.

Conclusion

The Cloud File Uploader project successfully demonstrates the design and implementation of a scalable, containerized file management system built using modern web technologies and cloud computing principles. By combining **Node.js**, **MongoDB replica sets**, and **Docker**, the system provides users with a secure and isolated environment for uploading and managing personal files.

Through this project, the team gained practical experience in deploying distributed services, managing user sessions, configuring Dockerized database clusters, and simulating real-world cloud architecture across multiple virtual machines. Despite challenges related to container orchestration, shared volume handling, and session control, the final system met its intended goals.

The project reflects a solid foundation for cloud-based application development and sets the stage for future improvements such as security hardening, cloud platform hosting, enhanced UI/UX features, and system monitoring. Overall, the Cloud File Uploader stands as a valuable prototype for academic learning and real-world cloud deployment practices.

References

- Node.js Documentation. (2024). Node.js v20.x Docs.

Retrieved from: https://nodejs.org/en/docs

- Express.js. (2024). Express - Node.js web application framework.

Retrieved from: https://expressjs.com/

- MongoDB Inc. (2024). MongoDB Manual — The Developer Data Platform.

Retrieved from: https://www.mongodb.com/docs/manual/

- Multer Middleware. (2024). Multer Documentation.

GitHub Repository: https://github.com/expressjs/multer

- VMware Workstation Pro. (2024). Run virtual machines on your desktop.

Retrieved from: https://www.vmware.com/products/workstation-pro.html

- Mozilla Developer Network. (2024). HTTP Cookies — Using HTTP cookies in web apps.

Retrieved from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

- W3Schools. (2024). HTML Forms and Input Elements.

Retrieved from: https://www.w3schools.com/html/html_forms.asp

- Stack Overflow. (2024). How to manage user sessions in Node.js using express-session.

Retrieved from: https://stackoverflow.com/questions/40381401/

- GeeksforGeeks. (2023). Introduction to Cloud Computing.

Retrieved from: https://www.geeksforgeeks.org/introduction-to-cloud-computing/