### **Final Project Report**

#### **1) PROJECT TITLE**

STUDY GROUP MANAGER - STUDIOUS

#### **2) TEAM MEMBERS**

* Ruban Chakaravarthi
* Darshan Vijayaraghavan
* Vignesh Kumar Karthikeyan

### **3) PROJECT GOALS**

The **Study Group Manager-Studious** project aspires to help students connect, collaborate, and learn by offering a comprehensive, user-friendly platform that facilitates seamless study group formation and management. It addresses critical challenges such as identifying compatible group members, coordinating schedule which often hinder effective collaboration among students. The platform matches users based on shared academic interests, goals, and availability, ensuring optimal group dynamics and compatibility.

To enhance convenience, the platform integrates essential third-party service like **Zoom**, enabling users to plan, schedule, and participate in virtual study sessions effortlessly. A dedicated study room booking feature ensures optimal utilization of available resources while minimizing conflicts, thus catering to both virtual and in-person collaboration needs.

The platform's **scalable cloud-based architecture**, built on containerized apps, ensures it can dynamically handle varying user loads while maintaining high performance and reliability. Additional features such as real-time chat, email notifications, friend management, and group event planning create a great ecosystem for collaborative learning.

By streamlining manual processes and promoting peer engagement, the Study Group Manager-Studious project empowers students to focus on their academic goals while fostering an efficient, and highly interactive learning environment. This makes it an indispensable tool for modern education, bridging the gap between traditional study methods and the growing demand for digital solutions in collaborative learning.

### **4) LIST OF SOFTWARE AND HARDWARE COMPONENTS**

#### **Software Components:**

1. **Frontend Development:**
   1. **React.js:**

Used to build the platform’s user interface, React.js delivers a dynamic, responsive experience tailored for the seamless formation and management of study groups. It supports features such as real-time chat, friend management, and event planning, ensuring smooth interaction between users.

1. **Backend Infrastructure:**
   1. **Node.js:**

The backend uses Node.js to develop REST APIs for secure and efficient communication between the frontend and backend services. It powers essential features like study group creation, user profile management, and schedule coordination, chatting with friends, friend request.

* 1. **RabbitMQ:**

RabbitMQ acts as a message broker to handle asynchronous tasks like email notifications and session updates. This ensures reliable task processing for features like automated group notifications in email and large file handling.

* 1. **Redis**:  
      Employed as a caching layer, Redis accelerates real-time data retrieval for tasks such as study room availability, and chat updates.

1. **Database Technologies:**
2. **Fire store:**

A NoSQL database used to store dynamic data like user profiles, study group information, chat messages, and session schedules. This database ensures that the platform supports real-time updates and concurrent user operations.

1. **Container Orchestration:**
   1. **Kubernetes:**

Facilitates the deployment and management of microservices. Kubernetes ensures high availability and fault tolerance for key services like chat functionality

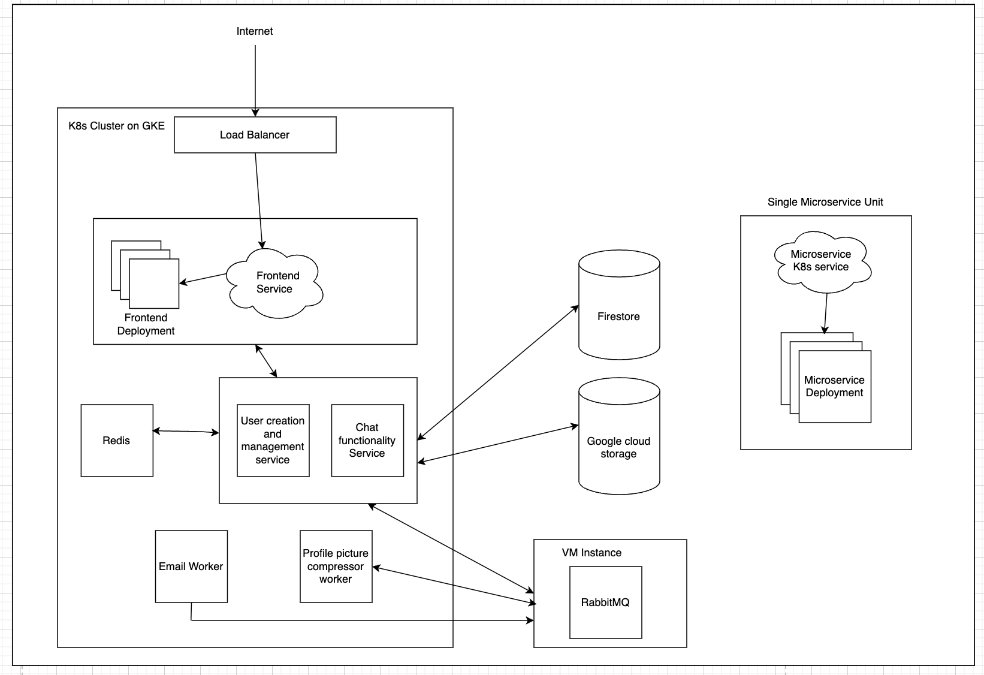
1. **Infrastructure Automation:**
   1. **Terraform:**

Automates the setup of infrastructure components, such as Kubernetes clusters and virtual machines on Google Cloud, streamlining deployment and scaling for dynamic user needs.

#### **Hardware Components:**

1. **Google Kubernetes Engine (GKE):**  
    GKE hosts Kubernetes clusters, supporting the deployment of the platform’s microservices. This ensures that features like user matching, real-time messaging, and Zoom integration operate smoothly under varying user loads.
2. **VM Instances:**  
    A dedicated VM instance runs RabbitMQ, enabling reliable message queue management for asynchronous tasks, such as booking notifications and study group reminders
3. **Google Cloud Storage:**  
    Designed for storing media assets, such as study materials and profile pictures, Google Cloud Storage provides fast and secure access to resources critical for collaboration.

### **5) ARCHITECTURAL DIAGRAM**



**6) INTERACTION OF SOFTWARE AND HARDWARE COMPONENTS**

The Study Group Manager-Studious platform is architected to deliver a seamless and efficient user experience by integrating various software and hardware components that work cohesively. The architecture is designed to handle user interactions, real-time communications, asynchronous tasks, and data management with high scalability and reliability.

1. **Client Interaction and Load Balancing**

Users access the platform through their web browsers initiating connections to the Frontend Service. These requests are first directed to a Load Balancer, which intelligently distributes incoming traffic across multiple instances of the Frontend Service deployed on Google Kubernetes Engine (GKE). This distribution ensures optimal resource utilization, high availability, and fault tolerance, preventing any single frontend instance from becoming a bottleneck.

**b. Frontend Service Communication**

The Frontend Service, built with React.js, provides a dynamic and responsive user interface, allowing students to create and manage study groups, schedule sessions, and engage in real-time chats. Once a user interacts with the frontend, such as creating a study group or sending a message, the frontend communicates directly with the Backend Services to process these requests.

**c. Backend Services Overview**

The backend comprises two primary services:

\* **REST API Service:** Developed using Node.js, this service handles standard HTTP requests for operations like user authentication, study group creation, profile management, and data retrieval. It serves as the backbone for CRUD (Create, Read, Update, Delete) operations within the platform.

\* **Chat Functionality Service:** Also built with Node.js, this service leverages Socket.io to facilitate real-time, bidirectional communication between users. It enables instant messaging, notifications, and live collaboration within study groups.

**d. Real-Time Communication and Session Management**

To maintain persistent and reliable chat sessions, the architecture employs session affinity (also known as sticky sessions). This ensures that once a user establishes a connection with a specific backend pod hosting the Chat Functionality Service, all subsequent interactions during that session are routed to the same pod. This consistency is crucial for maintaining an open communication channel and delivering real-time updates without interruption.

**e. Distributed Messaging with Redis**

The Chat Functionality Service integrates Redis as an in-memory data store to manage socket information and facilitate message routing. When a user sends a message, the service first attempts to retrieve the recipient's socket information from its in-memory connections array. If the recipient is not connected to the same pod, the message is published to Redis, allowing other pods to subscribe and forward the message to the appropriate recipient if their socket information is available. This mechanism ensures efficient and reliable message delivery across a distributed system.

**f. Asynchronous Task Processing with RabbitMQ**

For handling tasks that do not require immediate processing, such as compressing profile pictures or sending email notifications, the architecture utilizes RabbitMQ as a message broker. Two specialized workers—the Profile Picture Worker and the Email Worker—listen to their respective queues in RabbitMQ:

\* **Profile Picture Worker:** Processes image uploads by compressing and optimizing profile pictures before storing them in Google Cloud Storage.

\* **Email Worker:** Manages the sending of transactional emails, such as account verification, password resets, and notifications about study group activities.

This separation of concerns ensures that resource-intensive tasks do not impede the responsiveness of the main application services.

**g. Data Storage and Management**

The platform employs Firestore, a scalable NoSQL database, to store dynamic and relational data, including user profiles, study group details, chat logs, and scheduling information. Firestore provides real-time synchronization, enabling instant updates across all connected clients.

For storing media assets like profile pictures, session, Google Cloud Storage is utilized. This service offers secure, scalable, and high-performance storage solutions, ensuring that users can reliably access their files whenever needed.

**h. Infrastructure Automation with Terraform**

To streamline the deployment and management of cloud resources, the architecture integrates Terraform. This Infrastructure as Code (IaC) tool automates the provisioning of resources on Google Cloud, ensuring consistent and repeatable deployments across different environments. Terraform manages components such as Kubernetes cluster, virtual machine for RabbitMQ, and storage services, reducing manual configuration errors and enhancing scalability.

**i. Container Orchestration with Kubernetes**

All microservices, including the frontend, backend services, and workers, are containerized and orchestrated using Kubernetes on GKE. Kubernetes handles the deployment, scaling, and management of these containers, ensuring that the platform can dynamically adjust to varying user loads. Features like automatic scaling, self-healing, and rolling updates contribute to the platform's robustness and uptime.

### **7) DESCRIPTION OF THE COMPONENTS AND ITS ADVANTAGE AND DISADVANTAGE**

1. **Frontend Service (React.js) Purpose:** The frontend handles all user interactions, delivering a seamless and interactive experience. It is responsible for rendering pages, handling user input, and communicating with backend APIs for data retrieval and updates.

* **Why we Chose This Component:** React.js is highly flexible and allows for rapid development of dynamic, responsive user interfaces. It’s a popular choice for building scalable web applications.
* **Advantages:** React’s component-based architecture allows for code reuse, making development faster and more organized. It also ensures excellent performance due to the virtual DOM.
* **Disadvantages:** For newcomers, React’s ecosystem can be overwhelming, especially with frequent updates.

1. **Backend Services (Node.js) Purpose:** The backend manages the business logic, processes user requests, and handles database interactions. It also facilitates real-time features like chat using Socket.io.

* **Why we Chose This Component:** Node.js is particularly effective for building lightweight, fast, and scalable backend services. Its asynchronous nature makes it ideal for handling multiple simultaneous requests.
* **Advantages:** It’s efficient for I/O-heavy operations and has a vast package ecosystem. Also, it uses JavaScript, which allows for a unified language across the stack.
* **Disadvantages:** It’s single-threaded, which makes it less effective for CPU-intensive operations. Additionally, if not designed properly, it can lead to callback hell or complex asynchronous flows.

1. **Redis Purpose:** Redis is used to cache frequently accessed data and manage real-time tasks like storing session data or message queues for chat.

* **Why we Chose This Component:** Its in-memory design makes it perfect for tasks requiring high-speed read and write operations. Redis is also ideal for implementing Pub/Sub systems, which are crucial for real-time communication.
* **Advantages:** Its performance is unmatched for real-time operations. Redis also simplifies complex operations like Pub/Sub.
* **Disadvantages:** Since it’s in-memory, data persistence can be an issue unless configured correctly. It also has a limited storage capacity, tied to available memory.

1. **RabbitMQ Purpose:** RabbitMQ acts as the messaging backbone of the application. It ensures asynchronous communication between different microservices, enabling features like task queues, notifications, and background job processing.

* **Why we Chose This Component:** I needed a reliable and proven message broker to handle inter-service communication, and RabbitMQ offers features like message acknowledgments and retries, ensuring no data is lost.
* **Advantages:** It’s robust and supports advanced routing, making it easy to implement complex workflows. It ensures reliable message delivery, even in case of service failures.
* **Disadvantages:** RabbitMQ can be resource-intensive, requiring proper configuration for high throughput. Additionally, it may introduce latency if not optimized for the application’s needs.

1. **Firestore Purpose:** Firestore stores all dynamic data, such as user information, group details, and chat logs. Its real-time synchronization ensures that all users see updated data immediately.

* **Why we Chose This Component:** Firestore’s real-time features are ideal for collaborative applications like this one, where users need instant updates.
* **Advantages:** Its integration with Firebase makes it easy to use for both mobile and web apps. It also handles scaling automatically, reducing operational overhead.
* **Disadvantages:** Firestore is a NoSQL database, which may require additional effort for complex queries. There can also be cost implications for heavy usage.

1. **Google Cloud Storage Purpose:** This is the storage solution we used for all user-uploaded content, such as profile pictures.

* **Why we Chose This Component:** Google Cloud Storage provides durable, secure, and scalable object storage, with the ability to handle large files efficiently.
* **Advantages**: It offers automatic replication for data reliability and is easy to integrate with other Google Cloud services.
* **Disadvantages:** Latency may be noticeable when retrieving large files, and the cost can become significant for frequent uploads and downloads.

1. **Kubernetes (GKE) Purpose:** Kubernetes is used to deploy, scale, and manage containerized applications. It ensures high availability and efficient resource utilization for the microservices.

* **Why we Chose This Component:** Kubernetes is the standard for container orchestration, offering features like self-healing, auto-scaling, and rolling updates, which are critical for maintaining the system.
* **Advantages:** It simplifies the management of complex, distributed systems and provides built-in tools for monitoring and scaling.
* **Disadvantages:** The learning curve is steep, and managing Kubernetes clusters requires significant expertise. It can also be resource-intensive.

1. **Terraform Purpose**: Terraform automates the provisioning and management of the application’s infrastructure, ensuring consistency and repeatability.

* **Why we Chose This Component:** Terraform’s ability to manage infrastructure as code makes it easy to track changes and collaborate across teams.
* **Advantages:** It simplifies multi-cloud management and provides clear version control for infrastructure. Rollbacks and updates are straightforward.
* **Disadvantages:** Debugging can be tricky for complex configurations, and understanding the syntax can take time for new users.

### **7) TESTING AND DEBUGGING**

* **Logging**: Added detailed logs in backend and RabbitMQ workers which help to trace execution.
* **Monitoring:** Utilized Google Cloud’s Log monitoring for real-time error tracking and system monitoring. Any containers that failed due to an issue can be identified. It also provides an easy way to look at the logs of each pod, the CPU utilization etc.
* **Testing:** Used integration tests to catch bugs early in development. Integration tests and regression tests were performed at each and every stage of the project development, so that we don’t run into huge app breaking problems at any point.
* Used Minikube cluster to deploy the clusters locally and tested the integration in a containerized environment.

### **8) Working System, Capabilities and limits of the final system.**

When a user visits the application, they can sign up by filling out a registration form with details such as their name, email, password, and preferences. Upon submitting the form, the backend validates the input and securely stores the user’s data in the database with hashed passwords. A welcome email is sent to confirm their registration, and the user is redirected to the login page. Once logged in using their credentials, they gain access to the dashboard where they can create or explore study groups.

To create a study group, the user specifies details such as the group name, description, and whether the group is online or offline. If offline, a location is required; if online, a Zoom link is automatically generated. The created group becomes visible to all users, who can join it with a single click. Upon joining, members receive an email notification, and the group appears in their "Upcoming Events" section with detailed information, including the location or Zoom link. Users can then engage in group chats or private messaging, facilitating seamless communication within the platform.

The final solution offers a scalable, real-time, and fault-tolerant system that delivers a smooth user experience through microservices, Kubernetes, and Google Cloud components. It efficiently handles large-scale workloads with features like load balancing, automated scaling, and real-time updates using Redis and Firestore.

The modular architecture ensures flexibility for future enhancements, while tools like Terraform streamline deployment and infrastructure management. However, the solution comes with some limitations, including a steep learning curve for advanced tools, potential cost challenges for cloud resources, and complexities in debugging and monitoring distributed systems. Additionally, while real-time features are powerful, they rely on stable network performance, and Node.js may struggle with CPU-intensive tasks. Overall, the architecture balances performance and scalability but requires careful cost monitoring and skilled management to avoid operational bottlenecks.

### **Conclusion**

The **Study Group Manager** project successfully addresses the challenges of creating and managing study groups by providing a scalable, user-friendly platform. By integrating modern technologies like React.js, Node.js, RabbitMQ, and Kubernetes, the system ensures seamless collaboration, efficient scheduling, and real-time communication.

The platform’s robust architecture and comprehensive testing mechanisms position it as an invaluable tool for students, enhancing their academic experience and fostering collaborative learning. Future enhancements, such as AI-based friend recommendations and multilingual support, can further elevate the platform’s utility and appeal.

This project demonstrates the effective use of cloud technologies and modern development practices to build a scalable, flexible, and high-performing solution tailored to the needs of today’s learners.