**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND**

**TECHNICAL SCIENCES**

**CHENNAI-602105**

**BACHELOR OF ENGINEERING**

## IN

## COMPUTER SCIENCE ENGINEERING

## Submitted by

**Srinivas Reddy (192210136)**

**Under the supervision of**

**Mr. Arul Raj**

**September(2024)**

## 

## DECLARATION

I am **Srinivas Reddy** student of **‘**Bachelor of Engineering’, Department of Computer Science and Engineering, Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai. Hereby declare that the work presented **Develop the Resource management based on elastic cloud balancing and job shop scheduling** in this Capstone Project Work entitled is the outcome of our own bonafide work and is correct to the best of our knowledge and this work has been undertaken taking care of Engineering Ethics.

**Submitted by Srinivas Reddy (192210136)**

**Certificate**

This is to certify that the project entitled **“Develop the Resource management based on elastic cloud balancing and job shop scheduling”** submitted by G.Akhil has been carried out under our supervision. The project has been submitted as per the requirements in the current semester of B. Tech Information Technology.

Faulty-in-charge

Mr. Arul Raj

**TABLE OF CONTENT**

|  |  |
| --- | --- |
| **S.NO** | **TOPICS** |
| **1** | **Abstract** |
| **2** | **Introduction** |
| **3** | **Problem Statement** |
| **4** | **Proposed Design Work** |
|  | * Identifying Key Components |
|  | * Functionality |
|  | * Architectural Design |
| **5** | **GUI Design** |
|  | * Layout |
|  | * User - Friendly |
|  | * Color Selection |
| **6** | **Program / Coding** |
|  | * Language Selection |
|  | * Algorithm/Program |
|  | * Execution |
| **7** | **Implementation** |
|  | * Connecting the Components |
|  | * Cloud Deployment |
|  | * Project Testing |
| **8** | **Performance Evaluation And Code Implementation** |
| **9** | **Conclusion** |

**ABSTRACT**

Resource management in cloud computing involves efficient allocation of computational resources to tasks. The goal is to ensure optimal utilization, reducing costs and improving performance. This can be achieved using Elastic Cloud Balancing (ECB) and Job Shop Scheduling enhancing resilience against server failures. Cost optimization strategies, including spot instances and resource optimization, are implemented to maintain cost-effectiveness. Monitoring and logging services track performance, providing insights into system behavior under varying loads.

**INTRODUCTION**

Elastic Cloud Balancing allows cloud infrastructure to scale resources dynamically based on current demand. Resources are provisioned and de-provisioned in real-time as workloads fluctuate. The primary features of ECB include:

* **Auto-Scaling**: Dynamically increasing or reducing the number of compute instances.
* **Load Balancing**: Distributing incoming requests to ensure no server is overwhelmed.
* **Resource Monitoring**: Using tools to track resource utilization and adjust accordingly.

To achieve scalability, cloud features such as auto-scaling groups and load balancers are used, while cloud-based databases ensure data integrity and fast access.

**PROBLEM STATEMENT**

Traditional web applications often struggle to manage increased user traffic during peak times, leading to performance degradation or even service outages. The solution is to design a web application that can scale dynamically based on demand while maintaining performance and security.

**PROPOSED DESIGN WORK**

The proposed design for this **scalable web application** focuses on implementing a flexible and resilient architecture that can dynamically adjust to changing user traffic and workload requirements. The key design components include the following:

* **Three-Tier Architecture**:

The application is designed using a **three-tier architecture**, which is a common approach for scalability and modularity in web applications.

* **Frontend Tier**: Handles user interactions and serves static content (HTML, CSS, JavaScript). Typically, this layer is deployed on web servers or content delivery networks (CDNs) for faster load times.
* **Application Tier**: Implements the business logic and interacts with the database. This tier should be auto-scalable to accommodate fluctuating user traffic.
* **Database Tier**: Stores persistent data. In this architecture,databases are often replicated or partitioned across different regions for fault tolerance and availability.

1. **Cloud Infrastructure Components**:

* The web application will be deployed on **Amazon Web Services (AWS)**, **Microsoft Azure**, or **Google Cloud Platform (GCP)** to leverage the inherent scalability, reliability, and performance of cloud platforms.
* **Compute Instances**: Virtual machines (VMs) such as **EC2 (AWS)** or **Azure Virtual Machines** will be used for the application tier.
* **Load Balancer**: **AWS Elastic Load Balancer**, **Azure Load Balancer**, or **GCP Load Balancer** distributes incoming traffic across multiple instances.
* **Auto-scaling**: Enables dynamic scaling by increasing or decreasing compute instances based on real-time traffic using **AWS Auto Scaling**, **Azure VM Scale Sets**, or **GCP Autoscaler**.
* **Managed Databases**: Use cloud-based databases like **Amazon RDS**, **Azure SQL Database**, or **Cloud SQL (GCP)** for persistent data storage with built-in backup and replication.

1. **Fault Tolerance and Redundancy**:
   * The system is designed to ensure **high availability** by distributing resources across **multiple availability zones** or regions. This ensures that the application remains functional even if one region or availability zone fails.
   * **Replication**: Database replication is enabled to ensure data redundancy, and services like **Amazon RDS** offer automated failover to standby instances.
   * **Backup**: Automated backup services are configured for databases to ensure disaster recovery in case of data loss.
2. **Security**:
   * Use cloud-native security services like **AWS Identity and Access Management (IAM)**, **Azure Active Directory (AD)**, or **GCP IAM** for managing user permissions and ensuring least-privilege access.
   * Implement **Web Application Firewalls (WAF)** and SSL/TLS encryption to secure the web application against cyber threats.
3. **Monitoring and Logging**:
   * Tools like **Amazon CloudWatch**, **Azure Monitor**, or **Google Cloud Operations Suite** are used to track resource utilization, application performance, and error rates.
   * Centralized logging helps with error detection and troubleshooting. Tools like **AWS CloudTrail**, **Azure Log Analytics**, or **GCP Cloud Logging** allow administrators to review logs and monitor application health.

**Functionality**

The **functionality** of the scalable web application revolves around ensuring it can handle **increased user traffic**, **dynamic workloads**, and **seamless performance**.

1. **User Authentication & Authorization**:
   * Use OAuth2 or services like **AWS Cognito**, **Azure AD B2C**, or **Firebase Authentication** to manage secure login for users.
   * Role-based access control (RBAC) can be implemented to restrict access to sensitive areas.
2. **Session Management**:
   * Implement **session tokens** or **cookies** for handling user sessions efficiently.
   * In cloud-based architectures, ensure that session data is stored in a **shared caching layer** (e.g., **Redis** via **ElastiCache** or **Azure Cache for Redis**) to maintain session persistence across scaled instances.
3. **Data Management**:
   * **CRUD operations** (Create, Read, Update, Delete) should be optimized through efficient querying, caching, and data partitioning.
   * Implement **API gateways** like **AWS API Gateway** or **Azure API Management** for secure and managed API calls to backend services.
4. **Auto-Scaling**:
   * Automatically adjust the number of instances running the application based on real-time traffic using **AWS Auto Scaling**, **Azure VM Scale Sets**, or **GCP Autoscaler**.
5. **Content Distribution**:
   * For media-heavy applications, leverage **Content Delivery Networks (CDNs)** to cache content closer to the end-user, improving response time.
   * Examples include **AWS CloudFront** or **Azure CDN**.

**Architectural Design**

The architectural design of a **scalable web application** involves breaking down the application into **tiers** or **layers** that allow for better management, scalability, and performance. The general approach for the architecture consists of three main layers:

1. **Front-End Layer**:
   * **Responsibility**: Manages user interaction and presents the user interface (UI).
   * **Technology**: This could be implemented using static HTML, CSS, and JavaScript, or dynamic technologies like **React.js**, **Vue.js**, or **Angular**.
   * **Cloud Integration**: Deployed on **Amazon S3** (AWS), **Azure Blob Storage**, or a virtual server instance.
   * **Scaling**: **CDNs** (Content Delivery Networks) such as **AWS CloudFront** or **Azure CDN** are used to cache static content and reduce load on web servers, ensuring fast delivery across the globe.
2. **Application Layer**:
   * **Responsibility**: Houses the core business logic of the application.
   * **Technology**: Built using **Node.js**, **Python (Django/Flask)**, **Ruby on Rails**, or **Java (Spring Boot)**.
   * **Cloud Integration**: Hosted on **EC2 instances (AWS)**, **App Service (Azure)**, or **Google App Engine (GCP)**.
   * **Auto-scaling Groups**: Automatically increases or decreases the number of application servers based on demand. For example, **AWS Auto Scaling** can dynamically adjust the number of EC2 instances depending on the workload.
3. **Database Layer**:
   * **Responsibility**: Manages data storage and ensures data persistence.
   * **Technology**: Can include **SQL databases** (e.g., **MySQL**, **PostgreSQL**) or **NoSQL databases** (e.g., **MongoDB**, **DynamoDB**).
   * **Cloud Integration**: Managed database services like **Amazon RDS**, **Azure SQL Database**, or **Google Cloud SQL** are commonly used.
   * **Fault Tolerance**: Implement **replication** and **automated backups** to ensure high availability.

**Additional Architectural Components:**

* **Load Balancers**: Distributes incoming user traffic to ensure no single server is overwhelmed. **AWS Elastic Load Balancer (ELB)**, **Azure Load Balancer**, or **GCP Load Balancer** can be used.
* **Monitoring & Logging**: Services like **Amazon CloudWatch**, **Azure Monitor**, or **Google Cloud Operations Suite** (formerly Stackdriver) track real-time metrics for CPU, memory, and traffic.
* **Security**: Ensure end-to-end encryption (SSL/TLS), use **IAM roles**, and implement **WAF (Web Application Firewall)** for protection against common attacks like DDoS and SQL injections.

**GUI DESIGN**

When designing the **Graphical User Interface (GUI)** for a scalable web application, the focus is on creating a responsive, user-friendly, and intuitive interface. The key aspects of the GUI design for this project include:

1. **Responsive Design**:
   * The UI should adapt to various screen sizes (desktops, tablets, and smartphones).
   * Use front-end frameworks like **Bootstrap** or **Tailwind CSS** to ensure a responsive layout.
   * Technologies such as **React.js** or **Vue.js** are ideal for building dynamic user interfaces, where content is updated asynchronously without reloading the page.
2. **Interactive Components**:
   * **Navigation Menus**: Easy-to-access navigation that allows users to move through the application seamlessly.
   * **Forms and Input Fields**: For tasks such as login, signup, or data submission, designed with proper validation and feedback.
   * **Graphs and Charts**: Data visualization components for displaying metrics or user data. Libraries like **Chart.js** or **D3.js** can be integrated to display real-time data.
3. **UX/UI Best Practices**:
   * **Minimalism**: Keep the design clean and avoid clutter.
   * **Accessibility**: Ensure the design follows web accessibility guidelines (WCAG 2.0), allowing users with disabilities to navigate the application easily.
   * **Consistency**: Maintain consistent typography, colors, and button styles throughout the app to enhance the user experience.
4. **Loading Indicators and Notifications**:
   * Provide feedback to the users with loading spinners or progress bars when the application is performing background tasks such as fetching data or processing requests.
   * Implement **notifications** for successful actions, errors, or important updates (for example, using libraries like **Toastify** for toast notifications).
5. **Theme and Branding**:
   * Ensure the application’s GUI aligns with the company's branding. Use corporate colors, logos, and design patterns to create a visually appealing and professional look.
   * Offer users **dark mode** options for better accessibility and comfort during nighttime use.

**PROGRAM / CODING**

When developing the scalable web application, the **coding** needs to be optimized for both performance and scalability. Here are the major points:

1. **Front-End Development**:
   * Use frameworks like **React.js**, **Vue.js**, or **Angular** to create highly interactive and responsive UIs.
   * Implement **lazy loading** to defer loading of non-essential elements until they are required.
   * Minify JavaScript and CSS files and use image compression tools to optimize load time.
2. **Back-End Development**:
   * **Languages**: Use high-performance languages like **Node.js**, **Python (Flask/Django)**, or **Java (Spring Boot)**.
   * **API Design**: Design RESTful or GraphQL APIs to allow communication between the front-end and back-end.
   * **Error Handling**: Implement robust error handling mechanisms, ensuring errors are logged and monitored.
3. **Database Development**:
   * **SQL Databases** (e.g., MySQL, PostgreSQL): Use **normalized** schema designs, with proper indexing and query optimization to improve database performance.
   * **NoSQL Databases** (e.g., MongoDB, DynamoDB): Focus on **denormalization** where necessary for performance improvement, especially when scaling horizontally.
4. **Microservices Approach**:
   * Use a **microservices architecture** where each service is loosely coupled and can be scaled independently. For example, services like user management, payments, and notifications can be developed and scaled individually.
   * Containerization tools like **Docker** allow services to be packaged with their dependencies and deployed consistently across environments.
5. **Deployment Automation**:
   * Use Infrastructure as Code (IaC) tools like **Terraform** or **AWS CloudFormation** to automate infrastructure setup and scaling.
   * Implement a **CI/CD pipeline** using tools like **Jenkins**, **GitHub Actions**, or **GitLab CI** to automate deployment processes, reducing the chance of errors during manual deployments.
6. **Caching and Performance Optimization**:
   * Cache frequently accessed data using solutions like **Redis** or **Memcached**.
   * Implement **database query optimization** to reduce response time.

By carefully considering architectural design, functionality, and optimized code development, the web application will be **scalable**, **resilient**, and able to perform well under varying load conditions.

**Implementation**

**1. Infrastructure Setup:**

* **Virtual Machines (VMs)**: The first step in the implementation is provisioning VMs (e.g., EC2 in AWS) for the **application tier**. These instances are configured for auto-scaling based on CPU or memory utilization.
* **Load Balancer**: A **load balancer** is deployed to distribute user requests across multiple application servers, ensuring that no single server becomes a bottleneck.
* **Auto-Scaling Configuration**: Auto-scaling groups are configured to spin up new instances based on predefined thresholds (such as CPU utilization exceeding 70% or incoming traffic increasing beyond a certain limit).

**2. Database Setup:**

* **Managed Databases**: A **relational database** (e.g., MySQL on Amazon RDS or Azure SQL Database) is set up for handling persistent data. **Read replicas** are configured to offload read-heavy operations from the primary database.
* **Backup and Failover**: Automated backup policies and failover mechanisms are enabled to ensure data integrity and disaster recovery. Multi-AZ replication in RDS ensures that database services are highly available.

**3. CI/CD Pipeline:**

* The **Continuous Integration/Continuous Deployment (CI/CD)** pipeline automates the testing and deployment of code updates. Tools like **Jenkins**, **GitLab CI**, or **AWS CodePipeline** can be used to implement a streamlined development and deployment process.
* **Docker** containers are employed to package the application, ensuring consistent environments across development, testing, and production stages. Container orchestration can be handled using **Kubernetes** or **AWS ECS/EKS**.

**4. Security and Access Control:**

* Implement **IAM roles** to ensure proper access controls. Each component of the system (frontend, application, database) is assigned appropriate permissions based on the principle of least privilege.
* **SSL Certificates** are used for securing data transmission between the client and the server.
* **Firewalls and Security Groups**: Security groups are configured to allow only trusted traffic to the application and database tiers.

**5. Caching:**

* **Caching layers** like **AWS ElastiCache (Redis)** or **Azure Cache for Redis** are implemented to reduce the load on the database by caching frequently requested data.
* This ensures that the application responds quickly even under heavy load.

**6. Monitoring and Alerts:**

* Real-time monitoring solutions like **Amazon CloudWatch**, **Azure Monitor**, or **GCP Operations Suite** are configured to track resource utilization, latency, and error rates.
* **Alerts** are set up to notify the development team if certain thresholds are crossed (e.g., high CPU usage, slow query response times, or database connection issues).

**7. Load Testing:**

* **Load testing** tools such as **Apache JMeter** or **Blazemeter** are used to simulate high traffic conditions and monitor the performance of the application.
* The system’s ability to handle varying traffic levels is tested by scaling the number of incoming requests to identify any bottlenecks or performance degradation.

**PERFORMANCE EVALUATION**

Performance evaluation in a scalable web application ensures that the system can handle varying levels of traffic, maintains responsiveness, and optimally utilizes resources. Here are the key methods and metrics to assess performance:

1. **Scalability Testing**:
   * **Load Testing**: This simulates large numbers of concurrent users to observe how the application scales under load. Tools like **Apache JMeter**, **Gatling**, or **Blazemeter** can be used for this.
   * **Auto-Scaling Evaluation**: Verify that the auto-scaling groups correctly increase or decrease resources based on load conditions, ensuring minimal response times and no downtime.
2. **Latency and Response Time**:
   * Measure the time taken for the application to respond to user inputs, database queries, or API requests.
   * **Monitoring Tools**: Cloud platforms such as **AWS CloudWatch**, **Azure Monitor**, or **Google Cloud Monitoring** can be used to monitor metrics such as CPU usage, memory usage, and network throughput.
   * **Average Response Time**: This should be as low as possible. Aim for sub-100ms for user interactions.
3. **Error Rate and Fault Tolerance**:
   * Ensure the application continues to function smoothly even when certain components fail.
   * **Error Monitoring**: Track and log errors using services like **Sentry** or **Rollbar**. These services provide insights into which parts of the application are prone to failure and require improvement.
   * **Disaster Recovery**: Test fault tolerance by simulating failures (e.g., a server going down) and ensuring the system can recover automatically without user impact.
4. **Resource Utilization**:
   * **CPU and Memory Usage**: Continuously monitor resource usage to ensure that no server is over-utilized or under-utilized. Cloud platforms often provide dashboards for visualizing usage metrics.
   * **Database Performance**: Evaluate the database’s read/write performance and optimize queries to avoid bottlenecks. Tools like **AWS RDS Performance Insights** or **Azure SQL Analytics** can help.
5. **Real-World Performance Metrics**:
   * **Throughput**: Measure how much data the application can process in a given time frame (requests per second, transactions per minute, etc.).
   * **Capacity Planning**: By understanding peak loads and average traffic, adjustments to infrastructure (like adding more nodes or upgrading instances) can be planned accordingly.
6. **Caching Mechanism**:
   * Use services like **AWS ElastiCache** (Redis) or **Azure Cache for Redis** to cache frequently accessed data, reducing the load on the database and speeding up response times.

**CODE IMPLEMENTATION:**

import random

import time

class ElasticCloudBalancer:

def \_\_init\_\_(self, max\_servers, min\_servers=1):

self.max\_servers = max\_servers

self.min\_servers = min\_servers

self.active\_servers = min\_servers

def scale\_up(self):

if self.active\_servers < self.max\_servers:

self.active\_servers += 1

print(f"Scaling up: Active servers = {self.active\_servers}")

def scale\_down(self):

if self.active\_servers > self.min\_servers:

self.active\_servers -= 1

print(f"Scaling down: Active servers = {self.active\_servers}")

def manage\_load(self, current\_load):

# Simple rule: 1 server per 10 tasks

optimal\_servers = max(1, current\_load // 10)

if optimal\_servers > self.active\_servers:

self.scale\_up()

elif optimal\_servers < self.active\_servers:

self.scale\_down()

**CONCLUSION**

Combining Elastic Cloud Balancing with Job Shop Scheduling improves cloud resource management by optimizing both resource allocation and task execution. This approach ensures the system can handle varying workloads efficiently while minimizing costs.