**Scalable Web Hosting Using Cloud Platforms**

**Project Documentation**

**1. Introduction**

The rapid growth of internet usage and web-based applications has created a pressing demand for scalable, reliable, and cost-efficient hosting solutions. Traditional hosting environments often fall short when handling varying traffic loads, leading to either underutilization or service failures. Cloud platforms provide a robust solution by offering on-demand resource provisioning, automated scaling, and a host of services to support modern web hosting needs.

This project, titled **"Scalable Web Hosting Using Cloud Platforms,"** focuses on building a cloud-native infrastructure that can automatically scale, balance loads, and ensure high availability using services offered by providers like AWS, Azure, or Google Cloud Platform.

**2. Objective**

To design and implement a scalable web hosting architecture leveraging cloud-native services such as:

* Auto Scaling
* Load Balancing
* CDN (Content Delivery Network)
* DNS Management
* Monitoring & Alerting
* CI/CD Pipelines

The aim is to build an infrastructure that is robust, performance-oriented, and cost-optimized while being easy to deploy and manage.

**3. Phase 1: Research**

**3.1 Detailed Project Explanation**

This project aims to:

* Host a dynamic web application
* Automatically scale resources based on user demand
* Ensure even traffic distribution using Load Balancers
* Accelerate global content delivery through CDNs
* Use Infrastructure as Code for repeatability and version control
* Implement monitoring and alerting systems to detect and respond to performance anomalies

**3.2 Requirement Flow Diagram (Text-Based)**

Client Browser

↓

DNS Resolution (e.g., Route 53)

↓

Cloud Load Balancer (e.g., AWS ELB)

↓

Auto-Scaling Group

↓

Web App Servers (EC2/Containers)

↓

Application Logic/API Layer

↓

Database (RDS/Cloud SQL)

↓

Response Back to Client

**3.3 Implementation Steps (Expanded)**

1. Select a cloud provider (AWS/Azure/GCP)
2. Register a domain and configure DNS (e.g., Route 53)
3. Set up a Load Balancer (e.g., ELB)
4. Launch Web App on EC2/VM instances
5. Configure Auto Scaling Group with defined CPU/memory thresholds
6. Store static assets using S3 or Blob Storage
7. Configure a managed SQL Database (RDS or Cloud SQL)
8. Implement monitoring (CloudWatch/Azure Monitor)
9. Automate deployments using CI/CD (GitHub Actions, Jenkins)
10. Perform functional and load testing

**3.4 Cloud Services Used**

| **Service** | **Purpose** | **Example** |
| --- | --- | --- |
| Compute | Host web application | EC2/VM/GCE |
| Auto Scaling | Dynamic resource provisioning | ASG/VMSS |
| Load Balancer | Distribute incoming traffic | ELB/LB |
| DNS | Resolve domain to IP address | Route 53 |
| CDN | Speed up content delivery | CloudFront |
| Storage | Store static files | S3/Blob Storage |
| Database | Persistent data storage | RDS/Cloud SQL |
| Monitoring | Logs and metrics | CloudWatch |

**3.5 Tools Used**

| **Tool** | **Purpose** |
| --- | --- |
| Terraform | Infrastructure as Code |
| Docker | Containerization of applications |
| Git/GitHub | Version control |
| GitHub Actions | CI/CD pipeline |
| Postman | API Testing |
| JMeter/Locust | Load/Performance Testing |
| VS Code | Development IDE |

**3.6 Third-Party Components**

| **Tool** | **Purpose** |
| --- | --- |
| Cloudflare | DDoS Protection & DNS |
| NGINX/Apache | Web server, reverse proxy |
| New Relic/Datadog | Application performance monitoring |
| Certbot | SSL certificate (Let's Encrypt) |

**4. Phase 2: Design**

**4.1 Project Design Goals**

* Handle high user traffic with minimal latency
* Ensure high availability and fault tolerance
* Automate deployments and updates
* Provide real-time monitoring and feedback

**4.2 Blueprint of Architecture**

+------------------------------+

| DNS (Route 53) |

+--------------+---------------+

↓

+-------------------+

| Load Balancer |

+--------+----------+

↓

+----------------------+

| Auto Scaling Group |

+---------+------------+

↓ ↓ ↓

[Web App 1] [Web App 2] ... [Web App n]

↓

+-------------------+

| Database (RDS) |

+-------------------+

**4.3 Flow Diagram (Execution Steps)**

1. User accesses the website via browser
2. DNS resolves the domain name
3. Request routed to ELB
4. Load Balancer directs traffic to a healthy EC2 instance
5. Application logic processes the request
6. If necessary, DB is queried
7. Response is returned to the client
8. Metrics and logs are recorded

**4.4 Justification for Services Used**

| **Service** | **Reason for Use** |
| --- | --- |
| EC2/VM | Runs the core web application |
| Auto Scaling | Ensures elasticity in resource usage |
| ELB | Handles traffic spikes evenly |
| Route 53 | Resolves domain names efficiently |
| S3 | Stores static content |
| RDS | Manages structured data |
| CloudFront | Global delivery of assets |
| CloudWatch | Enables visibility into performance |

**4.5 Execution Plan**

| **Step** | **Description** |
| --- | --- |
| 1 | Clone the application repository |
| 2 | Build the application Docker image |
| 3 | Write Terraform scripts for provisioning infrastructure |
| 4 | Deploy DNS, Load Balancer, Auto Scaling Group, and RDS |
| 5 | Launch application containers to EC2 or ECS |
| 6 | Configure Auto Scaling rules |
| 7 | Set up alerts and metrics in CloudWatch |
| 8 | Link CI/CD tools for continuous deployment |
| 9 | Run API and load tests |
| 10 | Go live and monitor continuously |

**5. Conclusion**

This project demonstrates the practical implementation of scalable, resilient, and cloud-native web hosting infrastructure. By combining modern DevOps tools and cloud-native services, the system ensures minimal downtime, quick deployment, and robust handling of dynamic workloads.

Such architectures are essential for real-world applications in e-commerce, education, health tech, and beyond. The practices and technologies outlined serve as a foundational blueprint for production-grade deployments in today’s cloud-first era.