**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 fluctuating traffic loads, leading to underutilization or service failures.

Cloud platforms address these challenges by offering on-demand resource provisioning, automated scaling, and a suite of services to support modern hosting needs.

This project, "Scalable Web Hosting Using Cloud Platforms," focuses on designing a cloud-native infrastructure that automatically scales, balances loads, and ensures high availability using services from providers like AWS, Microsoft Azure, and Google Cloud Platform (GCP).

**2. Objective**

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

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

The goal is to create a robust, performance-oriented, and cost-optimized infrastructure that is 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
* Distribute traffic evenly using Load Balancers
* Use CDNs for global content delivery
* Manage infrastructure using Infrastructure as Code (IaC)
* Implement monitoring and alerting systems to maintain performance

**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**

1. Choose 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 with CPU/memory thresholds
6. Store static assets in S3 or Blob Storage
7. Set up a managed SQL Database (e.g., RDS, Cloud SQL)
8. Implement monitoring (e.g., CloudWatch, Azure Monitor)
9. Automate deployments using CI/CD (e.g., GitHub Actions, Jenkins)
10. Conduct 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 / Azure LB |
| DNS | Domain name resolution | Route 53 |
| CDN | Speed up content delivery | CloudFront |
| Storage | Store static files | S3 / Blob Storage |
| Database | Persistent data storage | RDS / Cloud SQL |
| Monitoring | Log and performance tracking | CloudWatch |

**3.5 Tools Used**

| Tool | Purpose |
| --- | --- |
| Terraform | Infrastructure as Code |
| Docker | Containerization |
| Git / GitHub | Version Control |
| GitHub Actions | CI/CD Pipeline |
| Postman | API Testing |
| JMeter / Locust | Load / Performance Testing |
| Visual Studio Code | Development IDE |

**3.6 Third-Party Components**

| Tool | Purpose |
| --- | --- |
| Cloudflare | DNS and DDoS Protection |
| NGINX / Apache | Web server / Reverse proxy |
| New Relic / Datadog | Application Performance Monitoring (APM) |
| Certbot | SSL Certificates (Let's Encrypt) |

**4. Phase 2: Design**

4.1 Project Design Goals

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

**4.2 Architecture Blueprint**

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

| DNS (Route 53) |

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

↓

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

| Load Balancer |

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

↓

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

| Auto Scaling Group |

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

↓ ↓ ↓

[Web App Instances...]

↓

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

| Database (RDS) |

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

**4.3 Flow Diagram (Execution Steps)**

1. User accesses website via browser
2. DNS resolves the domain name
3. Request hits Load Balancer
4. Traffic is routed to healthy EC2 instances
5. Application logic is executed
6. Database is queried if needed
7. Response is sent to client
8. Metrics and logs are recorded

**4.4 Justification for Services**

| Service | Reason for Use |
| --- | --- |
| EC2/VM | Core application hosting |
| Auto Scaling | Elastic resource management |
| ELB | Balanced and reliable traffic handling |
| Route 53 | Domain resolution and routing |
| S3 | Hosting static assets |
| RDS | Managed relational database |
| CloudFront | Fast content delivery worldwide |
| CloudWatch | Monitoring, logging, and alerts |

**4.5 Execution Plan**

| Step | Description |
| --- | --- |
| 1 | Clone the application repository |
| 2 | Build Docker image for the app |
| 3 | Write and deploy Terraform scripts |
| 4 | Set up DNS, Load Balancer, ASG, and Database |
| 5 | Deploy containers to EC2/ECS |
| 6 | Configure auto scaling policies |
| 7 | Set up monitoring and alerts |
| 8 | Integrate CI/CD pipeline (e.g., GitHub Actions) |
| 9 | Run API and load tests |
| 10 | Go live and monitor infrastructure continuously |

**5. Conclusion**

This project showcases a cloud-native, scalable, and resilient web hosting infrastructure that meets modern performance and reliability standards. It combines DevOps tools and cloud services to deliver a highly available and maintainable environment suitable for production-grade deployment.

The architecture outlined is a strong fit for industries such as e-commerce, ed-tech, health tech, and more—where uptime, scalability, and performance are mission-critical.