1. Scalability Triggers and Strategies

a. Triggers for Scaling Components

- **Backend Microservices:**
  - **CPU utilization:** Scale up or down based on CPU usage.
  - **Memory utilization:** Scale up or down based on memory consumption.
  - **Request rate:** Scale up or down based on the number of incoming requests.
- **Database Performance:**
  - **Query latency:** Scale up or down based on the average query response time.
  - **I/O utilization:** Scale up or down based on I/O operations.
  - **Connection pool usage:** Scale up or down based on connection pool saturation.
- **Messaging Systems:**
  - **Message backlog:** Scale up or down based on the number of unprocessed messages.
  - **Throughput:** Scale up or down based on the message processing rate.
  - **Latency:** Scale up or down based on message delivery latency.
- **Containerized Services on Kubernetes:**
  - **CPU and memory utilization:** Scale up or down based on resource usage.
  - **Pod restarts:** Scale up or down based on frequent pod restarts.
  - **Network traffic:** Scale up or down based on network traffic.

b. Scaling Strategies

- **Horizontal scaling:** Add more instances of a component to distribute the load.
- **Vertical scaling:** Increase the resources (CPU, memory) of existing instances, better in DB
- **Auto-scaling:** automatically adjust the number of instances based on predefined metrics.
- **Predictive scaling (would require some more research):** Use historical data and machine learning to anticipate future load and scale proactively.

2. Logging and Monitoring Strategy

a. Log Aggregation

- **Frontend services:** Collect logs from web servers , proxy servers and application servers.
- **Backend microservices:** Collect logs from application servers and frameworks.
- **Databases:** Collect logs from database servers and query engines.
- **Message brokers:** Collect logs from message brokers and producers/consumers.

b. Types of Logs

- **Application logs:** Logs generated by the application code.
- **System logs:** Logs generated by the operating system and infrastructure components.
- **Error logs:** Logs related to errors and exceptions.
- **Audit logs:** Logs of security-related events.

c. Tools

- **Log aggregation:** Fluentd, Logstash, prometheus
- **Storage:** Elasticsearch
- **Querying:** Kibana, Grafana

d. Monitoring Setup

- **Key metrics:** CPU utilization, memory usage, network traffic, response time, error rates, database query performance, message queue backlog.
- **Alerting mechanism:** Use Prometheus, Alertmanager, or cloud platform features to create alerts based on predefined thresholds.

3. Pipeline Stages and Description

1. Build and Test

This stage involves compiling the code, running unit tests, and performing code quality checks.

- **Steps:**
    - **Code Compilation:** The code is compiled to ensure there are no syntax errors or build issues.
    - **Unit Testing:** Run unit tests to verify individual components of the application.
    - **Code Quality Checks:** Use tools to perform static analysis (e.g., SonarQube) to ensure code quality and adherence to standards.
- **Tools:**
    - **Azure DevOps, Jenkins** or **GitHub Actions** for orchestrating the build pipeline.
    - **Unit Testing Frameworks:**
        - For .NET: NUnit or xUnit
        - For Python: pytest
    - **Static Analysis:** SonarQube

2. Containerization

In this stage, the application is packaged into a Docker container to ensure consistent deployments across environments.

- **Steps:**

    - Create a Dockerfile that specifies the application's environment, dependencies, and how to run the application.
    - Build the Docker image using the Dockerfile.
- **Tools:**
    - **Docker** for creating and managing container images.
    - **Docker Hub** or **Azure Container Registry** for storing built images.

3. Continuous Integration

This stage involves running integration tests to ensure that different components of the application work together correctly.

- **Steps:**
    - Set up a test environment with mock services or databases.
    - Deploy the application to the test environment (could be a separate Kubernetes namespace).
    - Execute integration tests against the deployed application.
- **Tools:**
    - **Azure DevOps** , **Github Actions** or **Jenkins** for orchestrating the CI pipeline.
    - **Postman** or **REST Assured** for API testing.

4. Continuous Delivery

In this stage, the application is deployed to a staging environment where acceptance tests are run to verify functionality before production deployment.

- **Steps:**
    - Deploy the application to a staging Kubernetes cluster, or the corresponding environment. (normal 3 tier architecture), or an ECS
    - Execute acceptance tests (e.g., end-to-end tests) to ensure the application meets business requirements.
- **Tools:**
    - **Helm** for managing Kubernetes deployments, making it easier to deploy applications consistently.
    - **Kubernetes** (AKS, EKS, GKE)
    - **Selenium** or **Cypress** for automated acceptance testing.

5. Continuous Deployment

In this final stage, the application is automatically deployed to the production environment once it passes all previous tests.

- **Process:**
    - Monitor the staging environment results.
    - Automatically promote the build to production upon successful acceptance testing.

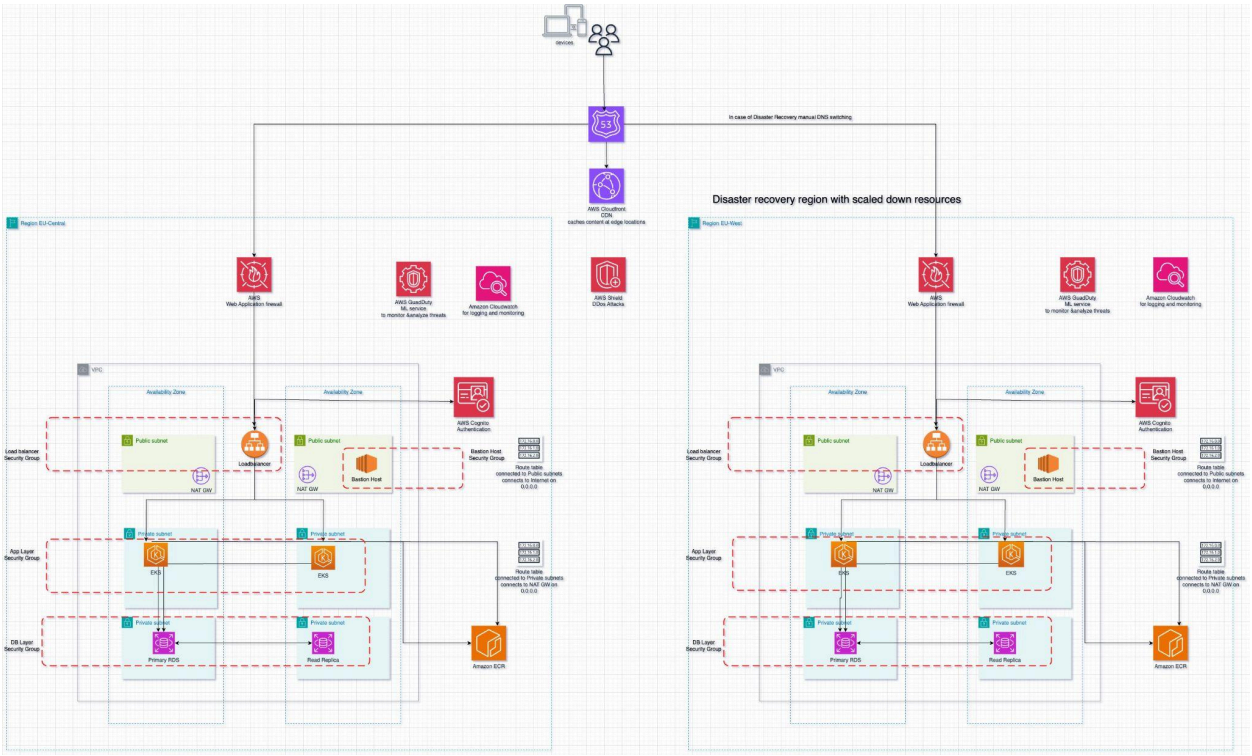
    - Deploy the application to the production AKS cluster.
- **Tools:**
    - **Kubernetes** (AKS, EKS, GKE) for hosting the production environment.
    - **Helm** for deploying application charts to production.
    - **GitOps tools** like **Argo CD** or **Flux** can be used to manage deployment states and facilitate continuous deployment practices.

b. Tools Overview

Here's a summary of the tools used at each stage:

Stage	***Tools***
Build and Test	Azure DevOps, GitHub Actions, NUnit/xUnit, pytest, SonarQube
Containerization	Docker, Azure Container Registry, Docker Hub
Continuous Integration	Azure DevOps, GitHub Actions, Jenkins, Postman
Continuous Delivery	Helm, Kubernetes , Selenium/Cypress
Continuous Deployment	Kubernetes , Helm, Argo CD or Flux

4. Architecture Diagram:



Overview:

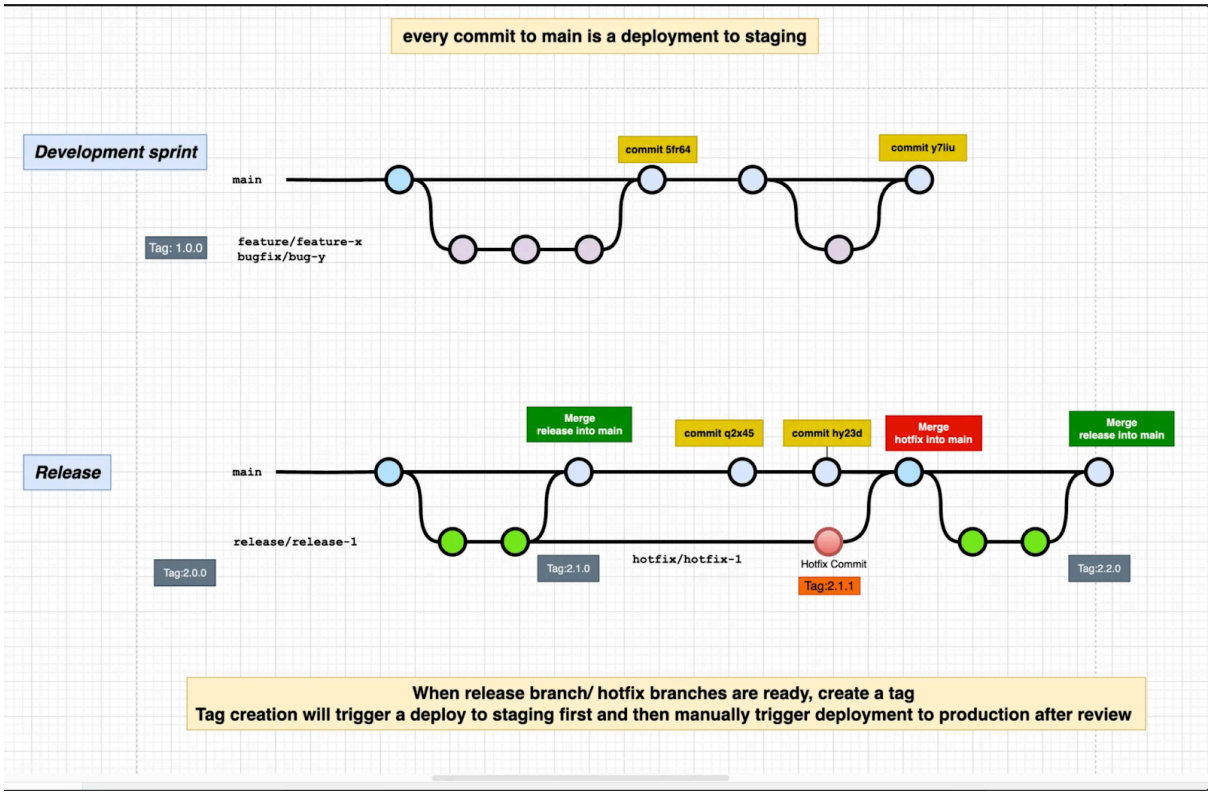
Our application follows a robust three-tier architecture designed to ensure scalability, reliability, and security. Below are the key components and features of our architecture:

- **Application**: Hosted on Amazon EKS with CDN integration for caching to improve performance and scalability.
- **Backend&DB**: Deployed on Amazon EKS for container orchestration, allowing for scalability and flexibility. It interacts with a relational database managed by Amazon RDS for data persistence. The database is replicated across multiple availability zones for high availability and fault tolerance.
- **Database Replication**: Amazon RDS provides built-in replication features such as Multi-AZ deployments and Read Replicas. Multi-AZ deployments automatically replicate data synchronously across multiple Availability Zones to ensure high availability and durability. Read Replicas can be used to offload read traffic from the primary database instance, improving scalability and performance.
- **Authentication**: Handled by AWS Cognito, providing secure user authentication and authorization.

- **Security**: Utilizes various AWS security services including AWS Shield, AWS GuardDuty, and AWS WAF to protect against DDoS attacks, threats, and web application firewall.
- **Network Infrastructure**: Configured with security groups, NACLs, and route tables to control traffic flow and ensure secure communication within the network. NAT Gateways are employed to provide internet access for private subnets.
- **Logging and Monitoring**: Leveraging AWS CloudWatch for logging and monitoring to gain insights into application performance and troubleshoot issues proactively.
- **High Availability**: The application is deployed across multiple availability zones within each region for fault tolerance and high availability. Additionally, a Disaster Recovery (DR) application is set up to provide failover capabilities, connected to Route 53 for traffic routing.

This architecture is designed to meet the demands of modern applications, providing scalability, security, and resilience to handle varying workloads and maintain high performance.
For more details on the architecture components, refer to the diagram above.

GitFlow Diagram:



Flow of the pipeline:

1. A developer commits code changes to the repository.
2. The build server is triggered.

3. The code is built and tested.
4. If tests pass, the artifact is deployed to the staging environment.
5. Acceptance tests are run in the staging environment.
6. If tests pass, the artifact is deployed to the production environment.
7. The application is monitored for issues.
8. If issues are detected, a rollback to a previous version may be necessary.

5. Chaos Engineering Plan Summary

Objectives

- **Test Resilience:** Validate system performance under unexpected failures.
- **Measure Recovery:** Assess the speed and effectiveness of recovery.
- **Identify Weaknesses:** Discover bottlenecks in the architecture.

Types of Failures to Simulate

1. **Network Latency:** Introduce delays between microservices using tools like **Chaos Mesh** or **Gremlin**.
2. **Service Failure:** Randomly terminate microservices with **Chaos Monkey** or **LitmusChaos**.
3. **Database Failures:** Simulate database outages by blocking access or shutting down databases (using **Pumba**).
4. **Resource Constraints:** Limit CPU or memory for specific services, simulating stress conditions (also with **Pumba**).
5. **Dependency Latency/Failure:** Simulate external API delays or failures using service virtualization tools.

Tools for Chaos Engineering

- **Chaos Monkey:** Random instance termination.
- **Gremlin:** Comprehensive failure simulation.
- **Chaos Mesh:** Cloud-native chaos experiments in Kubernetes.
- **LitmusChaos:** Kubernetes chaos framework.
- **Pumba:** Chaos testing for Docker.

Measuring Recovery

1. **Availability Metrics:** Monitor uptime and health checks.
2. **Performance Metrics:** Track response times and error rates.
3. **Recovery Time:** Measure Mean Time to Recovery (MTTR) and validate automatic failovers.