Cloud Application & Development Foundation

[BE SE Sixth Semester]

Nepal College of Information Technology
POKHARA UNIVERSITY

Unit III: Applications in the Cloud

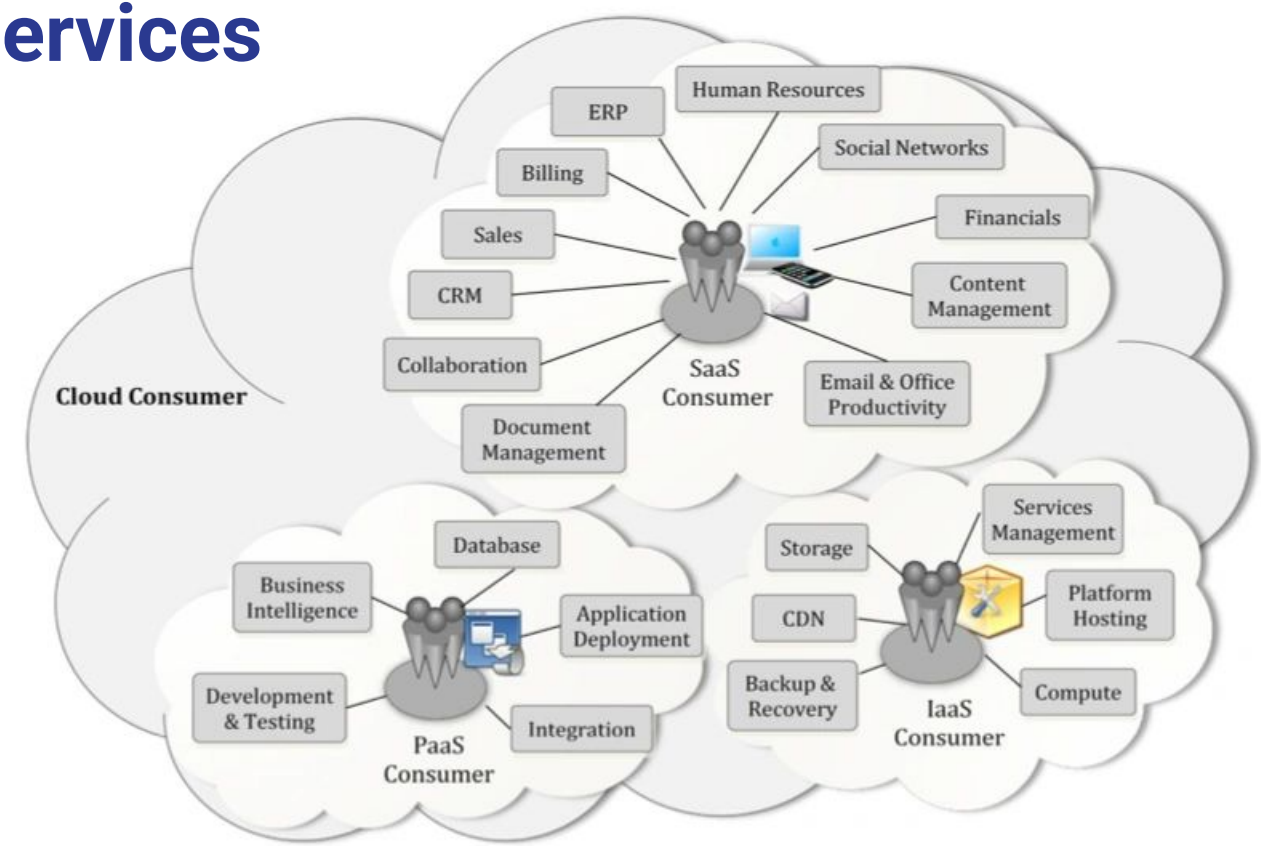
3.1 Web Services Deployment

- 3.1.1 Web Application Framework in Cloud
- 3.1.2 Web Hosting Services
- 3.1.3 APIs in Cloud

3.2 Cloud Continuous Delivery

- 3.2.1 DevOps
- 3.2.2 Git Basics
- 3.2.3 Containerization, Basics of Docker
- 3.2.4 Creating a Basic CI/CD Pipeline

Cloud Services



3.1.1 Web Application Frameworks in Cloud

- Web application frameworks simplify development by providing reusable components for routing, templating, and database integration.
- Cloud environments enhance scalability and deployment flexibility for frameworks.
- Popular frameworks:
  Django (Python), Ruby on Rails (Ruby), Spring Boot (Java), Express.js (Node.js).
- Frameworks reduce development time and ensure scalability in cloud deployments.
- Example: Deploying a Django app on AWS Elastic Beanstalk for auto-scaling.

Cloud-Compatible Frameworks

- **Stateless design:** Supports horizontal scaling in cloud environments.
- **Modular architecture:** Facilitates microservices and containerization.
- **Support for cloud-native features:** Integration with cloud databases, caching, and APIs.

Examples:

➔ Django: ORM for cloud databases (e.g., AWS RDS, Azure SQL).
➔ Spring Boot: Built-in support for cloud configuration (e.g., Spring Cloud for AWS).
➔ Express.js: Lightweight and suitable for serverless (e.g., AWS Lambda).

Deploying Options on Cloud Platforms

1. **Platform as a Service (PaaS):**

   AWS Elastic Beanstalk, Azure App Service, Google App Engine.

2. **Infrastructure as a Service (IaaS):**

   AWS EC2, Azure VMs with manual setup.

3. **Serverless:**

   AWS Lambda, Azure Functions for lightweight frameworks like Express.js.

Steps for Deployment:

1. **Package the application**

   (ZIP file, Docker container or git repository using githost).

2. **Configure cloud environment**

   (environment variables, scaling policies).

3. **Deploy and monitor using cloud tools**

   (e.g., AWS CloudWatch, Azure Monitor).

Best Practices for Framework Deployment

- Use environment variables for configuration (database URLs, API keys).
- Implement health checks for auto-scaling and monitoring.
- Optimize for statelessness to support cloud scaling.
- Use cloud-native logging and monitoring (e.g., AWS CloudWatch Logs).
- Use Tools:
  - ➔ Dependency management: Maven (Java), Pip (Python), npm (Node.js).
  - ➔ Cloud-specific SDKs: AWS SDK, Azure SDK for integration.

Advanced Framework Features in Cloud

- **Microservices**:
  Break down apps into smaller services (e.g., Spring Boot with Kubernetes).
- **Serverless integration**:
  Run specific endpoints on AWS Lambda or Azure Functions.
- **API gateways**:
  Manage traffic with AWS API Gateway or Azure API Management.
- **Benefits**:
  - Improved scalability and fault tolerance.
  - Simplified deployment of individual components.

***Example**: Using AWS API Gateway with a Node.js microservice.*

3.1.2: Web Hosting Services

- Web hosting services provide infrastructure to serve web content (e.g., websites, APIs).
- Types of web hosting:
  Shared, VPS, dedicated, and cloud hosting
- Cloud hosting supports dynamic scaling and global content delivery.
- Cloud hosting: Scalable, reliable, and cost-effective compared to traditional hosting.
- **Example**:
  Hosting a static website on AWS S3 with CloudFront CDN.

Types of Cloud Web Hosting Services

- **Static Hosting:**
  - AWS S3, Azure Blob Storage, Google Cloud Storage for static websites.
  - Ideal for HTML, CSS, JavaScript, and media files.
- **Dynamic Hosting:**
  - AWS Elastic Beanstalk, Azure App Service, Google App Engine for dynamic apps.
  - Supports server-side logic (e.g., PHP, Python, Node.js).
- **Serverless Hosting:**
  - AWS Lambda, Azure Functions for event-driven apps.
  - Auto-scales with no server management.

Content Delivery Networks (CDNs)

Role of CDNs:

- Distribute content globally to reduce latency (e.g., AWS CloudFront, Azure CDN).
- Cache content at edge locations for faster delivery.

Features:

- SSL/TLS support for secure connections.
- DDoS protection and traffic optimization.

Example: Configuring CloudFront to serve a static website from S3.

Steps for Static Hosting:

1. Upload files to AWS S3 bucket or Azure Blob Storage.
2. Enable static website hosting in bucket settings.
3. Configure DNS (e.g., Route 53, Azure DNS) for custom domains.
4. Integrate with a CDN for performance

***Example**:*

Deploy a static website on AWS S3 and document the process.

Hosting a Static Website

Objective: Host a static website on AWS S3 with CloudFront.

Steps:

1. Create an S3 bucket and enable static website hosting.
2. Upload HTML, CSS, and JavaScript files.
3. Configure CloudFront to distribute content.
4. Test the website using the CloudFront URL.

Tools Used: AWS S3, AWS CloudFront.

Steps for Dynamic Hosting:

1. Deploy app to AWS Elastic Beanstalk or Azure App Service.
2. Configure scaling and load balancing.
3. Set up monitoring and logging.

Assignments:

1. Hosting a WordPress site on Azure App Service
2. Case Study: High-Traffic e-commerce Website Handling Black Friday traffic spikes

Cloud Web Hosting: Best Practices

- Use HTTPS for secure communication.
- Optimize assets (e.g., compress images, minify CSS/JS).
- Enable caching for static content.
- Monitor performance with cloud tools (e.g., AWS CloudWatch).

Challenges:

➔ Managing costs for high-traffic websites.

➔ Ensuring high availability across regions.

APIs in Cloud Environments

- APIs (Application Programming Interfaces) enable communication between services.
- APIs enable scalable, modular architectures in the cloud.
- Cloud APIs: RESTful, GraphQL, or gRPC APIs hosted on cloud platforms.
- Use cases: Microservices, third-party integrations, mobile backends.

REST: Simple, widely adopted.

GraphQL: Efficient for complex queries.

gRPC: Low latency for internal services.

Types of Cloud APIs

- **REST APIs:** Stateless, use HTTP methods (GET, POST, PUT, DELETE).

  Example: AWS API Gateway with Lambda backend.

- **GraphQL APIs:** Query-based, flexible data retrieval.

  Example: Deploying a GraphQL server on Azure Functions.

- **gRPC APIs:** High-performance, used for microservices.

  Example: gRPC with Google Cloud Run.

APIs in Cloud Environments

- APIs (Application Programming Interfaces) enable communication between services.
- APIs enable scalable, modular architectures in the cloud.
- Cloud APIs: RESTful, GraphQL, or gRPC APIs hosted on cloud platforms.
- Use cases: Microservices, third-party integrations, mobile backends.

REST: Simple, widely adopted.

GraphQL: Efficient for complex queries.

gRPC: Low latency for internal services.

Unit III: Applications in the Cloud

3.1 Web Services Deployment

- 3.1.1 Web Application Framework in Cloud
- 3.1.2 Web Hosting Services
- 3.1.3 APIs in Cloud

3.2 Cloud Continuous Delivery

- 3.2.1 DevOps
- 3.2.2 Git Basics
- 3.2.3 Containerization, Basics of Docker
- 3.2.4 Creating a Basic CI/CD Pipeline

Introduction to DevOps

- DevOps is a set of practices combining software development (Dev) and IT operations (Ops) to shorten the development lifecycle and deliver high-quality software continuously.
- **Core Principles**

  Collaboration, automation, continuous integration, and continuous delivery (CI/CD).
- **Goals**

  Faster time-to-market, improved reliability, and enhanced team collaboration.

Key DevOps Practices

- **Continuous Integration (CI):**
  Developers merge code changes frequently, validated by automated tests to catch issues early.
- **Continuous Delivery/Deployment (CD):**
  Code is automatically built, tested, and deployed to production or staging environments.
- **Infrastructure as Code (IaC):**
  Manage infrastructure using code (e.g., Terraform, Ansible) for consistency and scalability.
- **Monitoring & Feedback:** Real-time monitoring (e.g., Prometheus, Grafana) ensures performance and quick issue resolution.

DevOps in Cloud

Benefits:

- Scalability: Cloud resources scale with DevOps pipelines.
- Cost Efficiency: Pay-as-you-go for CI/CD resources.
- Faster Time-to-Market: Automated pipelines reduce release cycles.

Challenges:

- Cultural resistance to DevOps adoption.
- Managing multi-cloud or hybrid environments.
- Security and compliance in automated workflows.

DevOps Tools in Cloud Environments

- **CI/CD:**

  AWS CodePipeline, Azure DevOps, Google Cloud Build.

- **Version Control:**

  GitHub, GitLab, Bitbucket.

- **Monitoring:**

  AWS CloudWatch, Azure Monitor, Prometheus.

- **IaC:**

  Terraform, AWS CloudFormation, Azure Resource Manager.

Exercise: Setting Up a DevOps Workflow

Objective:

Create a basic DevOps pipeline using AWS CodePipeline.

Steps:

1. Create a GitHub repository with a sample Node.js app.
2. Configure AWS CodePipeline to pull code from GitHub.
3. Set up CodeBuild for automated testing.
4. Deploy to AWS Elastic Beanstalk.

Tools Used:

AWS CodePipeline, CodeBuild, Elastic Beanstalk, GitHub.

Git Basics

- Distributed version control system for tracking code changes.
- Core features: Branching, merging, commits, repositories.
- Essential for collaborative development and CI/CD pipelines.
- Cloud integration:
  Hosted Git services (e.g., GitHub, GitLab, Bitbucket).
- Example:
  Using GitHub for code storage and triggering AWS CodePipeline.

Why is version control critical for cloud-based teams?

Git Core Commands

Basic Commands:

- **git init:** Initialize a repository.
- **git add:** Stage changes for commit.
- **git commit**:
  Save changes with a message.
- **git push**: Upload changes to a remote repository.
- **git pull**: Fetch and merge changes from a remote repository.

Branching Commands:

- **git branch**:
  Create or list branches.
- **git checkout**:
  Switch branches.
- **git merge**:
  Combine branches.

General Practices: Creating a feature branch and merging it into main.

Git in Cloud Platforms

Cloud Git Services / GitHosts:

- **GitHub:** Collaboration, PRs, and Actions for CI/CD.
- **GitLab:** Built-in CI/CD pipelines and DevOps tools.
- **Bitbucket:** Integration with AWS CodePipeline and Azure DevOps.

Features

- Webhooks for pipeline triggers.
- Access control for repositories.
- Integration with cloud IDEs (e.g., AWS Cloud9).

Common Challenges in Using Git

- **Merge Conflicts:**
  Occur when multiple developers edit the same code, requiring manual resolution, which can be time-consuming and error-prone.
- **Complex Workflows:**
  Branching strategies (e.g., Gitflow) can overwhelm teams, especially with large projects or inexperienced users.
- **Large Repositories:**
  Managing repos with large files or long histories slows performance (e.g., cloning, fetching) and increases complexity.

Overcoming Git Challenges

- **Conflict Prevention:** Use clear communication, smaller commits, and tools like Git hooks to catch issues early.
- **Simplified Workflows**: Adopt straightforward branching models (e.g., trunk-based development) to reduce complexity for teams.
- **Optimizing Repos:** Use Git LFS (Large File Storage) for binaries and shallow clones to manage large repositories efficiently.
- **Training & Tools:** Provide team training and use GUI tools (e.g., Sourcetree, GitKraken) to ease Git’s learning curve.

Hands-On Example: Using Git with GitHub

Course Project

<< Group Division >>

Introduction to Containerization

- **Containers:** Lightweight, portable units for running applications.
- Ensures consistency across development, testing, and production.
- Difference from VMs: Containers share the host OS, reducing overhead.
- Docker: Leading containerization platform.
- Podman: Daemonless container engine, alternative to Docker
- Example: Running a Node.js app in a Docker container on AWS ECS.

Docker Basics

- Open-source platform for containerization
- Packages applications and dependencies into containers
- Lightweight, portable, and consistent across environments
- Enables developers to build, ship, and run applications anywhere
- Cross-platform support (Linux, Windows, macOS)
- Facilitate the same environment from development to production

Docker Core Concepts:

- **Container:**
  Runnable instance of an image, isolated with its own filesystem and processes
- **Image:**
  Read-only template with application, libraries, and dependencies
- **Volume:**
  Persistent storage for containers, independent of container lifecycle
- **Network:**
  Enables communication between containers and external systems

Basic Docker Workflow

- Write a **Dockerfile** to define the application environment
- Build an image using **docker build**
- Run a container from the image with **docker run**
- Share images via **Docker Hub** or private registries
- Manage containers with commands like **docker stop**, **docker rm**

Common Use Cases of Docker

- **Development**: Consistent dev environments across teams
- **CI/CD**: Streamlined build and deployment pipelines
- **Microservices**: Run multiple services in isolated containers
- **Testing**: Create disposable environments for testing
- **Cloud Deployment:** Deploy applications to cloud platforms

Benefits of Docker

- Consistency: Identical environments across dev, test, and production
- Efficiency: Lower resource usage than traditional VMs
- Scalability: Seamless integration with orchestration tools like Kubernetes
- Modularity: Supports microservices with isolated services
- Fast Deployment: Rapid container startup and deployment
- Community: Access to thousands of pre-built images on Docker Hub
- Security: Isolated containers reduce attack surface

Basic Docker Commands

- `docker --version` – Check installed Docker version
- `docker pull <image>` – Download an image from Docker Hub
- `docker images` – List all local images
- `docker run <image>` – Run a container from an image
- `docker ps` / `docker ps -a` – Show running / all containers

Managing Containers

- `docker stop <container>` – Stop a running container
- `docker rm <container>` – Remove a stopped container
- `docker rmi <image>` – Delete an image
- `docker exec -it <container> bash` – Access container shell
- `docker build -t <name> .` – Build image from Dockerfile

Docker in Cloud Environments

Cloud Docker Services:

- AWS Elastic Container Service (ECS): Managed container orchestration.
- Azure Container Instances (ACI): Serverless containers.
- Google Cloud Run: Serverless container deployment.

Orchestration:

- Kubernetes: Advanced orchestration (e.g., AWS EKS, Azure AKS).
- Simplifies scaling and management of containers.

Example: Deploying a Dockerized app on Google Cloud Run.

Introduction to Containerization

Dockerfile Structure:

- FROM: Base image (e.g., python:3.9).
- WORKDIR: Set working directory.
- COPY: Copy files to the container.
- RUN: Execute commands during build.
- CMD: Specify default command at runtime.

Example Dockerfile:

```
FROM node:16
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
CMD ["npm", "start"]
```

Best Practices for Docker

- Minimize layers for smaller images.
- Use multi-stage builds for optimization.
- Use official, minimal base images.
- Avoid running containers as root.
- Scan images for vulnerabilities (e.g., Docker Scan, AWS ECR).
- Optimize image size with .dockerignore.

Automating Development & Deployment

- Understand CI/CD pipelines for cloud applications.
- CI: Automate code integration and testing.
- CD: Automate deployment to production or staging.
- Pipelines: Workflow of build, test, and deploy stages.
- Reduces manual errors and speeds up releases.
- Example: GitHub Actions deploying to AWS Elastic Beanstalk.

Automating Development & Deployment

Continuous Integration (CI):

- Developers frequently merge code changes into a central repository.
- Each commit triggers automated builds and tests.
- Aim: Detect and fix bugs early, improve code quality.

Continuous Delivery / Deployment (CD):

- Ensures that code is always in a deployable state.
- After CI, software is automatically pushed to staging or pre-production environments.
- Extends CD by deploying every change automatically to production after passing tests.
- Requires high test coverage and confidence in automation.

Components of a CI/CD Pipeline

Pipeline Stages:

- **Build**:
  Compile code and create artifacts.
- **Test**:
  Run unit, integration, and
  end-to-end tests.
- **Deploy**:
  Push artifacts to cloud
  environments.

Tools:

- **GitHub Actions**:
  YAML-based workflows.
- **AWS CodePipeline**:
  Fully managed CI/CD.
- **Azure Pipelines:**
  Multi-cloud support.
- **Jenkins**:
  Open-source with cloud plugins.

Setting Up a CI/CD Pipeline

```
name: CI/CD Pipeline
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - name: Set up Node.js
      uses: actions/setup-node@v3
      with: { node-version: '16' }
    - run: npm install
    - run: npm test
    - run: npm build
```

Best Practices for CI/CD Pipelines

- Run tests in isolated environments.
- Use secrets management for sensitive data.
- Implement rollback strategies for failed deployments.
- Monitor pipeline performance and failures.
- Pipeline maintenance for large projects.
- Ensuring security in public runners.

Thank you