**ApexalQ Day-05**

**Name- Ishan R. Gawande**
**Date-07/02/2024**

**Research on Following Topics**

1. **Technical debt management**
2. **Code optimization, code quality and maintenance**
3. **CI/CD deployment**
4. **Data privacy and compliance**
5. **Methodologies and best practices in Software dev**
6. **Networking ports and protocols**

## What is Technical debt management?

### Technical Debt Management

### What is Technical Debt?

Technical debt refers to the **hidden cost** of choosing **quick, suboptimal solutions** in software development instead of **long-term, scalable** approaches. It occurs when teams prioritize **speed over quality**, leading to **future maintenance, rework, and inefficiencies**.

💡 *Think of it like financial debt—if not repaid, the "interest" (extra effort needed to fix the problem) keeps growing over time!*

### How to Manage Technical Debt?

Effective **Technical Debt Management (TDM)** ensures that debt does not accumulate to a point where it slows down development or creates security risks.

#### 1. Identify & Track Technical Debt

✔ **Code Reviews & Audits**: Regularly assess code for outdated patterns, inefficiencies, or workarounds.

✔ **Automated Code Analysis**: Tools like **SonarQube, CodeClimate, or Snyk** detect bad code practices and vulnerabilities.

✔ **Documentation**: Maintain a **technical debt backlog** to track known issues and their impact.

#### 2. Categorize & Prioritize Debt

Not all technical debt is equal. Categorize it based on:

✅ **Intentional vs. Unintentional Debt**

- *Intentional*: “We’ll do this now and fix it later” (e.g., skipping unit tests for a release deadline).
- *Unintentional*: Poor architecture or legacy code.

✅ **Short-Term vs. Long-Term Impact**

- Does the debt slow down development?
- Does it cause security vulnerabilities?
- Does it increase system complexity?

🛠️ **Prioritization Frameworks**:

- **Eisenhower Matrix** (Urgent vs. Important)
- **Technical Debt Quadrant** (Fowler’s Model: Prudent, Reckless, Deliberate, Inadvertent)
- **Risk-Based Approach** (Security risks first)

**3. Regularly Allocate Time for Debt Reduction**

✔ **“Debt Sprints” or Refactoring Days**: Dedicate time each sprint to reduce technical debt.

✔ **Refactoring with Feature Development**: Improve the codebase while adding new features.

✔ **Use CI/CD & Automated Testing**: Helps catch and prevent new technical debt.

**4. Adopt Better Engineering Practices**

✔ **Follow Clean Code Principles**: Keep code readable and maintainable.

✔ **Use Modular Architecture**: Makes it easier to replace or upgrade components.

✔ **Invest in DevOps & Automation**: Automate testing, deployments, and monitoring.

✔ **Enforce Coding Standards**: Use linters, code reviews, and style guides.

**5. Communicate with Stakeholders**

✔ **Educate Management**: Show how reducing technical debt improves speed and lowers costs.

✔ **Balance Business Goals & Technical Needs**: Avoid technical perfectionism but also prevent excessive shortcuts.

✔ **Track Metrics**: Use **code churn, bug rates, and maintenance costs** to justify debt reduction efforts.

**Tools for Technical Debt Management**

- **SonarQube** - Automated code quality and security analysis
- **CodeClimate** - Technical debt monitoring & reporting
- **Snyk** - Security-focused vulnerability tracking
- **Jira / Trello** - Tracking technical debt items in backlog
- **ESLint, Prettier** - Enforcing coding best practices

**Why is Technical Debt Management Important?**

✔ **Reduces long-term costs** by preventing expensive rework.

✔ **Improves developer productivity** by keeping the codebase clean.

✔ **Enhances software security & stability** by eliminating hidden risks.

✔ **Ensures scalability** for future features and business growth.

## What is Code optimization, code quality and maintenance?

**Code Optimization, Code Quality, and Maintenance** 🚀

1️⃣ **Code Optimization (Performance & Efficiency)**

- *Goal*: Make code **run faster, consume less memory, and scale better**.
- *Focus Areas*: Algorithm efficiency, memory management, execution speed.

💡 **Techniques for Code Optimization**

✅ **Choose the Right Data Structures** → Use **HashMaps** instead of **arrays** for fast lookups.

✅ **Reduce Redundant Computations** → Store frequently used values (**memoization**, caching).

✅ **Optimize Loops & Recursion** → Use **iterative approaches**, avoid deep recursion.

✅ **Parallel Processing** → Use **multi-threading** or **async programming** for performance.

✅ **Minimize API Calls & DB Queries** → Batch processing, caching, indexing.

✅ **Use Efficient Libraries** → Prefer optimized libraries over custom implementations.

🔧 **Example - Loop Optimization**

🚫 **Inefficient Code**:

```
for (let i = 0; i < arr.length; i++) {  // Recalculating length on each iteration
    console.log(arr[i]);
}
```

✅ **Optimized Code**:

```
const len = arr.length;
for (let i = 0; i < len; i++) {  // Store length once
    console.log(arr[i]);
}
```

📌 **Tools for Code Optimization**:

- **Profilers**: Chrome DevTools (JS), cProfile (Python), JProfiler (Java)
- **Load Testing**: JMeter, Locust
- **Benchmarking**: Google Benchmark (C++), Benchmark.js (JS)

2️⃣ **Code Quality (Readability, Maintainability, Security)**

🔹 *Goal*: Ensure code is **clean, readable, and easy to modify**.

🔹 *Focus Areas*: Readability, maintainability, security, best practices.

💡 **Best Practices for Code Quality**

✅ **Follow Clean Code Principles** → Keep functions **small and single-purpose**.

✅ **Write Meaningful Variable & Function Names** → Avoid temp, data, foo.

✅ **Avoid Code Duplication** → Use **DRY (Don't Repeat Yourself)** principle.

✅ **Follow Consistent Code Formatting** → Use **Prettier, ESLint, Black, PEP8**.

✅ **Write Unit Tests & Integration Tests** → Use **Jest, Mocha, JUnit, pytest**.

✅ **Document Your Code Properly** → Use **JSDoc, docstrings, Swagger for APIs**.

✅ **Perform Code Reviews** → Encourage peer reviews before merging code.

🔧 **Example - Bad vs. Good Naming**

🚫 **Bad Code**:

```
function cal(a, b) {
```

```
    return a + b;
}
```

✅ **Good Code**:

```
function calculateTotal(price, tax) {
    return price + tax;
}
```

📌 **Tools for Code Quality**:

- **Linters**: ESLint (JS), Pylint (Python), Checkstyle (Java)
- **Formatters**: Prettier, Black
- **Code Review Platforms**: GitHub, GitLab, Bitbucket

3️⃣ **Code Maintenance (Long-Term Stability & Evolution)**

🔹 *Goal*: Keep the **codebase scalable, modular, and easy to modify**.

🔹 *Focus Areas*: Version control, refactoring, dependency management.

💡 **Best Practices for Code Maintenance**

✅ **Modularize Code** → Follow **SOLID principles** to ensure scalability.

✅ **Use Version Control** → Use **Git (GitHub, GitLab, Bitbucket)** for tracking changes.

✅ **Refactor Code Regularly** → Improve structure **without changing behavior**.

✅ **Keep Dependencies Updated** → Use **npm, pip, Maven** for package management.

✅ **Automate Testing & CI/CD Pipelines** → Use **Jenkins, GitHub Actions, Travis CI**.

✅ **Document APIs & Architecture** → Maintain clear API contracts using **Swagger, OpenAPI**.

🔧 **Example - Refactoring Code**

🚫 **Before Refactoring**:

```
function getUserData(id) {
    let user = fetchUserFromDB(id);
    if (user) {
        return user;
    } else {
        return null;
    }
}
```

✅ **After Refactoring** (Simplified Return Statement):

```
function getUserData(id) {
   return fetchUserFromDB(id) || null;
}
```

📌 **Tools for Code Maintenance**:

- **CI/CD**: Jenkins, GitHub Actions, CircleCI
- **Code Refactoring**: SonarQube, CodeClimate
- **Dependency Management**: npm (JS), pip (Python), Maven (Java)

**Summary Table**

| Aspect | Goal | Key Strategies | Tools |
|---|---|---|---|
| **Code Optimization** | Improve Performance | Use efficient algorithms, caching, parallel processing | Chrome DevTools, JProfiler, Benchmark.js |
| **Code Quality** | Maintain Readability & Security | Follow Clean Code, use Linters, write Unit Tests | ESLint, Prettier, SonarQube |
| **Code Maintenance** | Ensure Long-Term Stability | Modularize, Refactor, Automate CI/CD | Git, Jenkins, Swagger |

**What is CI/CD deployment?**

**CI/CD Deployment: Step-by-Step Guide 🚀**

🔹 **CI/CD (Continuous Integration & Continuous Deployment)** automates **code integration, testing, and deployment**, ensuring smooth and reliable software delivery.

## 1️⃣ CI/CD Workflow Overview

💡 **Continuous Integration (CI)** → Automatically tests & integrates new code.

💡 **Continuous Deployment (CD)** → Automatically deploys tested code to production.

### CI/CD Pipeline Stages

1. **Code Commit** → Developer pushes code to a Git repository.
2. **Build** → The application is built (compilation, dependency installation).
3. **Test** → Unit tests, integration tests, security checks run automatically.
4. **Deploy** → Code is deployed to staging or production.
5. **Monitor** → Logs and performance metrics are monitored.

📌 **Popular CI/CD Tools**:

- **GitHub Actions, GitLab CI/CD, Jenkins, CircleCI, Travis CI, ArgoCD**

## 2️⃣ Setting Up a CI/CD Pipeline (Example: GitHub Actions)

### 👨‍💻 Example: Deploy a Node.js App to AWS Using GitHub Actions

#### Step 1: Create a .github/workflows/deploy.yml File

📌 *This file defines the CI/CD pipeline.*

```
name: Deploy Node.js App

on:
  push:
    branches:
      - main  # Runs when code is pushed to the main branch

jobs:
  build-test-deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout Code
        uses: actions/checkout@v3

      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: 18
```

```
    - name: Install Dependencies
      run: npm install

    - name: Run Tests
      run: npm test

    - name: Build App
      run: npm run build

    - name: Deploy to AWS
      env:
        AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
        AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
      run: |
        aws s3 sync ./build s3://your-bucket-name --delete
```

📌 **How It Works:**

✅ Pulls latest code from GitHub.

✅ Sets up **Node.js** & installs dependencies.

✅ Runs **tests** before deployment.

✅ Deploys the built project to **AWS S3**.

### 3️⃣ CI/CD Deployment to Docker + Kubernetes

👨‍💻 *Example: Deploy a Dockerized App to Kubernetes using Jenkins*

**Step 1: Create a Jenkinsfile**

📌 *Defines CI/CD stages in Jenkins Pipeline.*

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                sh 'docker build -t myapp:latest .'
            }
        }
        stage('Test') {
            steps {
                sh 'docker run --rm myapp:latest npm test'
```

```
            }
        }
        stage('Push to Docker Hub') {
            steps {
                withDockerRegistry([credentialsId: 'docker-hub', url: '']) {
                    sh 'docker tag myapp:latest mydockerhubusername/myapp:latest'
                    sh 'docker push mydockerhubusername/myapp:latest'
                }
            }
        }
        stage('Deploy to Kubernetes') {
            steps {
                sh 'kubectl apply -f k8s-deployment.yaml'
            }
        }
    }
}
```

📌 **How It Works:**

✅ Builds a **Docker image**.

✅ Runs tests in a **container**.

✅ Pushes the image to **Docker Hub**.

✅ Deploys the image to **Kubernetes**.

4️⃣ **Best Practices for CI/CD Deployment**

✅ **Use Infrastructure as Code (IaC)** → Terraform, Helm Charts for managing infrastructure.

✅ **Automate Rollbacks** → If a deployment fails, revert to the previous version.

✅ **Use Canary Deployments** → Deploy to a small % of users before full rollout.

✅ **Monitor & Log Everything** → Use **Prometheus, Grafana, ELK Stack**.

✅ **Secure Secrets** → Store API keys in **GitHub Secrets, AWS Secrets Manager**.

**What is Data privacy and compliance?**

**Data Privacy and Compliance: A Comprehensive Guide** 🔒

In today's digital world, **data privacy** and **compliance** are essential for securing sensitive information and adhering to regulations. Organizations must implement robust **security, governance, and auditing measures** to ensure compliance with legal frameworks like **GDPR, HIPAA, CCPA, and ISO 27001**.

## 1️⃣ What is Data Privacy?

📌 **Data privacy** refers to the **protection of personal, financial, or confidential data** from unauthorized access, breaches, or misuse.

**Key Principles of Data Privacy:**

✅ **Data Minimization** → Collect only necessary data.

✅ **Purpose Limitation** → Use data only for its intended purpose.

✅ **User Consent & Control** → Allow users to control their data.

✅ **Encryption & Security** → Protect data with strong encryption.

✅ **Retention & Deletion Policies** → Don't store data longer than needed.

## 2️⃣ Key Data Privacy Laws & Compliance Frameworks

| Regulation | Region | Key Focus Areas |
|---|---|---|
| **GDPR** (General Data Protection Regulation) | 🇪🇺 EU | Consent, Right to be Forgotten, Data Breach Notification |
| **CCPA** (California Consumer Privacy Act) | 🇺🇸 California, USA | Consumer Data Control, Opt-Out of Sale |

| | | |
|---|---|---|
| **HIPAA** (Health Insurance Portability & Accountability Act) | 🇺🇸 USA | Protects Healthcare Data (PHI & ePHI) |
| **ISO 27001** | 🌎 Global | Information Security Management System (ISMS) |
| **SOC 2** | 🌎 Global | Data Security for SaaS Companies |
| **PIPEDA** (Personal Information Protection and Electronic Documents Act) | 🇨🇦 Canada | Consumer Privacy & Data Handling |

### 3️⃣ How to Implement Data Privacy & Compliance in Your System

📌 *To ensure compliance, businesses must integrate security best practices into their software development and IT infrastructure.*

🔹 **Step 1: Data Classification & Access Control**

✅ **Identify Sensitive Data** → Personal, Financial, Medical, Business IP.

✅ **Classify Data** → Public, Internal, Confidential, Restricted.

✅ **Implement Role-Based Access Control (RBAC)** → Limit who can access data.

🔧 *Example: Restricting Access in a Web App (Node.js + MongoDB)*

```
if (user.role !== "admin") {
  return res.status(403).json({ error: "Access Denied" });
}
```

🔹 **Step 2: Data Encryption & Secure Storage**

✅ **Use End-to-End Encryption (E2EE)** → AES-256, RSA, TLS 1.3.

✅ **Encrypt Data at Rest & In Transit** → AWS KMS, Hashicorp Vault, OpenSSL.

✅ **Tokenization & Hashing for PII** → Hash passwords using **bcrypt, Argon2**.

🔧 *Example: Secure Password Hashing (Node.js + bcrypt)*

```
const bcrypt = require("bcrypt");
const saltRounds = 10;

async function hashPassword(password) {
  const hashed = await bcrypt.hash(password, saltRounds);
  return hashed;
}
```

🔹 **Step 3: User Consent Management & Data Deletion**

✅ **Provide Transparent Consent Forms** → Opt-in & opt-out options.

✅ **Implement 'Right to be Forgotten'** → Allow users to delete data.

✅ **Use Cookie Consent Banners** → GDPR & CCPA compliance.

🔧 *Example: GDPR-Compliant Cookie Banner (JavaScript)*

```
document.getElementById("acceptCookies").addEventListener("click", function() {
    document.cookie = "userConsent=true; path=/; expires=365d";
});
```

🔹 **Step 4: Data Auditing & Compliance Monitoring**

✅ **Log User Activity & Access** → Use audit logs (e.g., AWS CloudTrail, ELK Stack).

✅ **Detect & Respond to Breaches** → Implement **SIEM** tools like **Splunk, Wazuh**.

✅ **Perform Regular Security Audits** → Use **SOC 2 Type II**, **ISO 27001** frameworks.

🔧 *Example: Logging User Activity in MongoDB*

```
const logActivity = (user, action) => {
  db.activityLogs.insertOne({
    user: user.id,
    action: action,
    timestamp: new Date(),
  });
};
```

🔹 **Step 5: Secure Third-Party Integrations & API Security**

✅ **Validate & Sanitize API Inputs** → Prevent **SQL Injection & XSS**.

✅ **Implement OAuth 2.0 & JWT for Authentication**.

✅ **Monitor Third-Party APIs for Risks** → Use API Gateways (e.g., Kong, Apigee).

🔧 *Example: API Rate Limiting in Express.js (Prevent DDoS)*

```
const rateLimit = require("express-rate-limit");

const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // Limit each IP to 100 requests per window
});

app.use("/api", limiter);
```

### 4️⃣ Best Tools for Data Privacy & Compliance

📌 *These tools help in securing and monitoring compliance requirements.*

| Category | Tools |
|---|---|
| **Encryption** | OpenSSL, Hashicorp Vault, AWS KMS |
| **Identity & Access Management** | Okta, Auth0, Azure AD |
| **Data Loss Prevention (DLP)** | Symantec DLP, McAfee DLP |
| **Audit Logging & SIEM** | Splunk, ELK Stack, AWS CloudTrail |
| **Compliance Monitoring** | Vanta, Drata, OneTrust |
| **Penetration Testing** | Burp Suite, OWASP ZAP, Metasploit |

## 5️⃣ Industry Best Practices for Data Privacy

✅ **Follow the Principle of Least Privilege (PoLP)** → Restrict access to only necessary users.

✅ **Use Zero Trust Architecture** → Authenticate & verify all connections.

✅ **Regularly Update & Patch Systems** → Protect against known vulnerabilities.

✅ **Train Employees on Cybersecurity** → Conduct phishing attack simulations.

✅ **Implement a Data Breach Response Plan** → Have a clear incident response strategy.

## 6️⃣ Case Study: GDPR Compliance in a Healthcare System 🏥

🔹 *Scenario*: A telemedicine platform storing **Electronic Health Records (EHR)** wants to comply with **GDPR & HIPAA**.

**Solution Implemented:**

✅ **Encrypted Health Records (AES-256)** → Data stored securely in a blockchain-based EHR system.

✅ **RBAC for Doctors & Patients** → Patients control **who can access their records**.

✅ **Consent-Based Access via Smart Contracts** → Patients provide **on-chain consent** before sharing data.

✅ **Automatic Log Monitoring** → Monitored access requests with **SIEM & ELK Stack**.

## 7️⃣ Summary & Key Takeaways

| Aspect | Key Action Items |
|---|---|
| **Data Encryption** | Use AES-256, RSA, TLS 1.3 |
| **Access Control** | Implement RBAC, OAuth 2.0, JWT |

| | |
|---|---|
| **User Consent** | GDPR-compliant banners, delete user data |
| **Audit Logging** | Log all access requests, SIEM monitoring |
| **Compliance Frameworks** | Follow GDPR, HIPAA, ISO 27001 |

## What are Methodologies and best practices in Software dev?

### Methodologies & Best Practices in Software Development 🚀

Software development involves structured approaches to **designing, building, testing, and deploying** applications efficiently. Choosing the right **methodology** and following **best practices** ensures high-quality, scalable, and maintainable software.

### 1️⃣ Software Development Methodologies

📌 *Different methodologies exist based on project requirements, team size, and goals.*

🔹 **Agile Development**

✅ **Iterative & Incremental** – Divides work into small cycles (*sprints*).

✅ **Collaboration & Feedback** – Frequent client interactions & adaptability.

✅ **Popular Frameworks → Scrum, Kanban, SAFe, XP**

🔧 *Best For:* Startups, evolving requirements, rapid development.

🔹 **Waterfall Model**

✅ **Linear & Sequential** – Follows fixed **phases (Requirement → Design → Development → Testing → Deployment → Maintenance)**.

✅ **Clear Documentation** – Best for projects with well-defined requirements.

🔧 *Best For:* Enterprise software, government projects, banking systems.

🔹 **DevOps (CI/CD)**

✅ **Automates Development & Deployment** – Combines Dev & Ops teams.

✅ **Uses CI/CD Pipelines** – Continuous **Integration, Testing, and Deployment**.

✅ **Popular Tools** → Docker, Kubernetes, Jenkins, GitHub Actions.

🔧 *Best For:* Cloud applications, microservices, scalable architectures.

🔹 **Test-Driven Development (TDD)**

✅ **Write Tests Before Code** – Ensures robust, bug-free software.

✅ **Red-Green-Refactor Cycle** – Write a failing test → Code → Pass test → Refactor.

✅ **Popular Frameworks** → JUnit (Java), PyTest (Python), Mocha (Node.js).

🔧 *Best For:* High-quality, bug-free, and maintainable software.

🔹 **Lean Software Development**

✅ **Eliminates Waste** – Reduces unnecessary steps in development.

✅ **Faster Delivery** – Focuses on customer value.

✅ **Principles** → *Build only what's needed, deliver fast, improve continuously*.

🔧 *Best For:* Startups, small teams, MVP (Minimum Viable Product).

🔹 **Extreme Programming (XP)**

✅ **Frequent Releases** – Small, frequent iterations.

✅ **Pair Programming** – Two developers work on the same code.

✅ **Continuous Feedback** – Customer-driven improvements.

🔧 *Best For:* High-risk projects requiring constant updates.

2️⃣ **Software Development Best Practices**

🔹 **Code Quality & Maintainability**

✅ **Follow SOLID Principles** – Improves maintainability.

✅ **Use Design Patterns** – Factory, Singleton, Observer, etc.

✅ **Write Clean & Readable Code** – Proper naming, comments, and structure.

✅ **Refactor Regularly** – Improve existing code without changing behavior.

🔧 *Example: Clean Code (JavaScript)*

```
// ❌ Bad Practice
function x(a, b) {
  return a * b;
}

// ✅ Good Practice
function calculateArea(width, height) {
  return width * height;
}
```

🔹 **Version Control & Collaboration**

✅ **Use Git for Version Control** – GitHub, GitLab, Bitbucket.

✅ **Follow Git Branching Strategy** – Main, Develop, Feature branches.

✅ **Code Reviews & Pull Requests** – Ensure quality and security.

🔧 *Example: Git Branching Strategy*

```
 main (Stable Release)
   ├── develop (Ongoing Development)
   │    ├── feature-1
   │    ├── feature-2
   │    ├── hotfix-1
```

🔹 **Security Best Practices**

✅ **Use Secure Authentication** – OAuth 2.0, JWT, Multi-Factor Authentication.

✅ **Encrypt Sensitive Data** – AES-256, Hashing with bcrypt.

✅ **Perform Regular Security Audits** – Use tools like OWASP ZAP, Burp Suite.

✅ **Implement Role-Based Access Control (RBAC)**.

🔧 *Example: Hashing Passwords in Python*

```
import bcrypt

password = "securepassword".encode("utf-8")
hashed = bcrypt.hashpw(password, bcrypt.gensalt())

print(hashed)
```

🔹 **Performance Optimization**

✅ **Reduce Database Calls** – Use indexing, caching (Redis, Memcached).

✅ **Optimize Code Execution** – Avoid unnecessary loops and computations.

✅ **Use Load Balancing** – Distribute traffic across multiple servers.

✅ **Enable Lazy Loading** – Load images/data only when needed.

🔧 *Example: Caching API Responses in Node.js*

```
const redis = require("redis");
const client = redis.createClient();

async function getCachedData(key) {
  const data = await client.get(key);
  if (data) return JSON.parse(data);
}
```

🔹 **Testing & Automation**

✅ **Write Unit Tests** – Test individual components.

✅ **Integration & End-to-End (E2E) Testing** – Ensure the system works as expected.

✅ **Automate Testing in CI/CD Pipelines**.

✅ **Popular Tools** → Jest, Mocha (JavaScript), JUnit (Java), PyTest (Python).

🔧 *Example: Unit Test with Jest (JavaScript)*

```
test("adds 2 + 3 to equal 5", () => {
  expect(2 + 3).toBe(5);
});
```

🔹 **CI/CD Deployment Best Practices**

✅ **Automate Build & Deployment Pipelines** – Use Jenkins, GitHub Actions, GitLab CI/CD.

✅ **Implement Canary Releases** – Deploy changes to a small subset before full rollout.

✅ **Monitor Deployments & Rollbacks** – Use Prometheus, Grafana.

🔧 *Example: GitHub Actions for CI/CD*

```
name: CI/CD Pipeline
```

```
on: [push]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Install Dependencies
        run: npm install
      - name: Run Tests
        run: npm test
      - name: Deploy to Production
        run: npm run deploy
```

### 3️⃣ Choosing the Right Methodology & Best Practices

- **For Startups & MVPs** → Agile + Lean + CI/CD.
- **For Enterprise Applications** → DevOps + Waterfall for well-defined processes.
- **For High-Security Applications** → TDD + Security Audits + Secure Authentication.
- **For AI & Data-Driven Projects** → Agile + DevOps + Data Security Compliance.

### 4️⃣ Summary & Key Takeaways

| Aspect | Best Practices |
|---|---|
| **Code Quality** | SOLID, Clean Code, Design Patterns |
| **Version Control** | Git, Branching Strategy, Code Reviews |
| **Security** | Encryption, OAuth 2.0, Security Audits |

| | |
|---|---|
| **Performance** | Caching, Lazy Loading, Database Optimization |
| **Testing** | Unit, Integration, End-to-End, Automated Tests |
| **CI/CD** | GitHub Actions, Jenkins, Canary Releases |
| **Methodology** | Agile, DevOps, Lean, TDD |

## What are Networking ports and protocols?

**Networking Ports & Protocols – A Comprehensive Guide** 🌐🔌

In computer networking, **ports and protocols** define how data is sent and received over the internet or a local network. Understanding these is crucial for **security, performance, and troubleshooting**.

### 1️⃣ What Are Networking Ports?

A **port** is a logical communication endpoint used by protocols to identify specific processes or services on a device.

- Ports are represented by **numbers (0–65535)**.
- Commonly used in **TCP/IP-based communication**.
- Managed by **IANA (Internet Assigned Numbers Authority)**.

| Port Range | Usage |
|---|---|
| **0 – 1023** | Well-known/system ports (assigned to standard services). |

| | |
|---|---|
| **1024 – 49151** | Registered ports (assigned for proprietary applications). |
| **49152 – 65535** | Dynamic/private ports (used for temporary client connections). |

## 2️⃣ Important Networking Protocols & Their Ports

- **Application Layer Protocols (Layer 7)**

| Protocol | Port | Description |
|---|---|---|
| **HTTP** | 80 | Web browsing (unencrypted). |
| **HTTPS** | 443 | Secure web browsing (TLS/SSL encryption). |
| **FTP** | 21 | File Transfer Protocol (unencrypted). |
| **SFTP** | 22 | Secure FTP (over SSH). |
| **SMTP** | 25 | Sending emails (unencrypted). |
| **SMTPS** | 465 | Secure SMTP (TLS/SSL encryption). |

| | | |
|---|---|---|
| **POP3** | 110 | Retrieving emails (unencrypted). |
| **IMAP** | 143 | Email retrieval (modern alternative to POP3). |
| **IMAPS** | 993 | Secure IMAP. |
| **DNS** | 53 | Resolving domain names to IPs |
| **DHCP** | 67/68 | Automatic IP assignment. |
| **RDP** | 3389 | Remote Desktop Protocol (Windows remote access). |
| **SSH** | 22 | Secure remote login. |
| **Telnet** | 23 | Unsecure remote access (deprecated). |

- **Transport Layer Protocols (Layer 4)**

| Protocol | Port | Description |
|---|---|---|

| | | |
|---|---|---|
| **TCP (Transmission Control Protocol)** | N/A | Connection-oriented, reliable communication. |
| **UDP (User Datagram Protocol)** | N/A | Connectionless, faster but less reliable. |

- **Network Layer Protocols (Layer 3)**

| Protocol | Function |
|---|---|
| **IP (Internet Protocol)** | Routing data between devices. |
| **ICMP (Internet Control Message Protocol)** | Error reporting (e.g., ping). |

- **Data Link Layer Protocols (Layer 2)**

| Protocol | Function |
|---|---|
| **Ethernet (IEEE 802.3)** | **Wi-Fi (IEEE 802.11)** |
| **Wired network communication.** | Wireless network communication. |

| | |
|---|---|
| **ARP (Address Resolution Protocol)** | Resolves IP addresses to MAC addresses. |

### 3️⃣ TCP vs. UDP – Key Differences

Both **TCP** and **UDP** operate at the **Transport Layer (Layer 4)** but differ in functionality.

| Feature | TCP (Reliable) | UDP (Faster) |
|---|---|---|
| **Connection** | Connection-oriented | Connectionless |
| **Reliability** | Ensures all data is received | No guarantee of data delivery |
| **Use Cases** | Web browsing, file transfers, emails | Video streaming, VoIP, gaming |
| **Error Checking** | Yes (Retransmission on failure) | No |
| **Speed** | Slower (acknowledgments required) | Faster (no acknowledgments) |

✅ **Use TCP when reliability is important (e.g., banking, emails, file transfers).**

✅ **Use UDP when speed is critical (e.g., online gaming, VoIP calls, live streaming).**

### 4️⃣ Common Networking Scenarios & Best Practices

🔹 **Secure Network Communication**

✔ **Use HTTPS instead of HTTP** – Encrypts data with TLS.

✔ **Use SFTP instead of FTP** – Secure file transfers.

✔ **Disable Telnet & use SSH** – Telnet is insecure.

✔ **Restrict unnecessary open ports** – Prevents attacks.

🔹 **Troubleshooting & Diagnostics**

✔ **Use ping (ICMP)** – Check network connectivity.

✔ **Use traceroute** – Identify routing issues.

✔ **Use netstat or ss** – Check open ports and active connections.

✔ **Use nmap** – Scan for open ports and vulnerabilities.

**Example:** Check which ports are open on a server using nmap

nmap -p 1-65535 <server-ip>

5️⃣ **Summary Table – Important Ports & Protocols**

| Service | Protocol | Port |
|---|---|---|
| Web Traffic | HTTP | 80 |
| Secure Web Traffic | HTTPS | 443 |
| Secure Shell | SSH | 22 |
| File Transfer | FTP | 21 |
| Secure File Transfer | SFTP | 22 |

| | | |
|---|---|---|
| Email Sending | SMTP | 25, 465 |
| Email Retrieval | POP3 | 110 |
| Secure Email | IMAPS | 993 |
| Remote Desktop | RDP | 3389 |
| Domain Name Resolution | DNS | 53 |
| DHCP | DHCP | 67/68 |