Module-1 Notes

DevOps is a **combination of cultural philosophies, practices, and tools** that enhances an organization's ability to **deliver applications and services at high velocity**. It aims to **bridge the gap between software development (Dev)** and **IT operations (Ops)** by fostering a culture of **collaboration, automation, and continuous improvement** across the entire software delivery lifecycle.

DevOps emphasizes:

- **Automation** of repetitive tasks like testing, integration, deployment, and infrastructure provisioning.
- **Continuous integration and continuous delivery (CI/CD)** pipelines that allow developers to merge code frequently and release updates rapidly.
- **Monitoring and feedback loops** to identify and resolve issues quickly.
- **Cross-functional teams** where developers, testers, operations, and sometimes even business stakeholders work together with shared responsibilities and goals.

At its core, DevOps transforms software delivery into a **collaborative, automated, and agile process**, resulting in faster releases, better quality, and improved user satisfaction.

🔐 DevSecOps – Detailed Definition

DevSecOps is an extension of DevOps that integrates **security practices into every stage of the software development and delivery process**. It ensures that **security is a shared responsibility** across development, operations, and security teams — rather than a siloed concern handled only at the end.

The primary goal of DevSecOps is to **"shift security left"** — meaning, security is considered **early and continuously** throughout the software lifecycle instead of waiting until after development is complete.

DevSecOps emphasizes:

- **Security automation** integrated into CI/CD pipelines — such as vulnerability scans, code analysis, secret detection, and policy enforcement.
- **Threat modeling** and secure design from the beginning.
- **Continuous compliance** and audit readiness.
- **Developer empowerment** to write secure code and fix vulnerabilities early.

In essence, DevSecOps embeds security as a foundational element of DevOps, allowing teams to deliver **secure, fast, and reliable software** without slowing down innovation.

✅ Benefits of DevOps

1. Faster Time to Market

Benefit: Faster development, testing, and deployment cycles.

How: CI/CD automates testing and deployment, reducing manual steps.

Example:

- A retail e-commerce company uses Jenkins for CI/CD. Developers push new features daily, and they go live the same day — speeding up innovation and competitive edge.

2. Continuous Delivery & Continuous Integration (CI/CD)

Benefit: Rapid, reliable software delivery with fewer bugs in production.

How: Automated testing and deployment pipelines catch bugs early and deploy frequently.

Example:

- A fintech app uses GitHub Actions for CI and ArgoCD for delivery to Kubernetes. New features are delivered twice a week without downtime.

3. Better Collaboration & Communication

Benefit: Breaks down silos between dev, QA, and ops teams.

How: Everyone shares responsibilities and uses shared tooling.

Example:

- In a media streaming platform, developers write Dockerfiles and Helm charts, and operations handle scaling — both using the same GitOps workflow.

4. Improved Automation

Benefit: Saves time, reduces human error, and increases reliability.

How: Automates build, test, deployment, and infra provisioning.

Example:

- An EdTech company uses Terraform to automate AWS infrastructure and Ansible for app config — scaling new environments in minutes.

5. Lower Failure Rates & Faster Recovery

Benefit: Bugs are detected and fixed earlier in smaller chunks.

How: Small, incremental changes are easier to test and rollback.

Example:

- An airline's booking system deploys 20+ small updates per week. When a bug is found, it's fixed and redeployed in less than 30 minutes using blue-green deployments.

6. Better Monitoring & Feedback Loops

Benefit: Increases visibility and speeds up issue detection/resolution.

How: Real-time metrics, logs, and alerts help identify issues before users do.

Example:

- A food delivery startup uses Prometheus + Grafana to track API latency and alert Slack if response time exceeds thresholds.

7. Reduced Costs Over Time

Benefit: Less downtime, fewer bugs, smaller team sizes due to automation.

How: Efficiency and predictability reduce the need for firefighting and rework.

Example:

- A SaaS startup reduced cloud costs by 30% after containerizing workloads, automating autoscaling, and eliminating manual deployment overhead.

✅ Benefits of DevSecOps

1. Security Built Into the Pipeline

Benefit: Security is not an afterthought; it's part of the development lifecycle.

How: Tools like Trivy, Snyk, and Gitleaks run automatically in the CI pipeline.

Example:

- A healthcare app scans Docker images and dependencies on every PR using GitLab CI — catching vulnerabilities **before** going live, meeting HIPAA compliance.

---

2. Shift Left: Catch Security Issues Early

Benefit: Early detection = less costly fixes and safer software.

How: Run static analysis, dependency checks, and IaC scans early.

Example:

- An insurance company integrates SonarQube and Checkov in its PR process — rejecting code that introduces insecure patterns or misconfigured cloud infrastructure.

---

3. Continuous Compliance & Audit Readiness

Benefit: Stay always-ready for audits and regulations like GDPR, HIPAA, SOC2.

How: Automated checks for policies, encryption, access control, and logs.

Example:

- A fintech firm uses Open Policy Agent (OPA) and AWS Config rules to automatically validate IAM roles, encryption policies, and log retention — reducing audit time from weeks to hours.

---

4. Empowered Developers Who Write Secure Code

Benefit: Developers take ownership of security, reducing bottlenecks.

How: Developers get security feedback instantly in their IDE or CI.

Example:

- Developers at a logistics company use VSCode extensions (like CodeQL) to get real-time alerts about vulnerable patterns **before** committing the code.

---

5. Faster and Safer Releases

Benefit: Balance between speed and safety.

How: Security tests don't block delivery but run in parallel or pre-merge.

Example:

- A bank uses a security gate in their CI/CD: high/critical vulnerabilities must be fixed to proceed, while low/medium are logged for review — enabling weekly secure releases.

---

6. Real-Time Security Monitoring & Threat Detection

Benefit: Detect live threats and anomalous behavior in production.

How: Use runtime tools like Falco, Sysdig, or AWS GuardDuty.

Example:

- A ride-sharing platform uses Falco to detect if any container executes a shell command (possible intrusion) — and automatically quarantines the pod in Kubernetes.

---

7. Cost-Efficient Security Operations

Benefit: Security issues are fixed early when cheaper and simpler to resolve.

How: Proactive testing and education reduce emergency patches.

Example:

- A DevSecOps culture helped a B2B SaaS avoid a costly zero-day patching sprint, because the vulnerable library was already flagged and replaced before release.

🔄 Summary Table

Benefit	**DevOps**	**DevSecOps**
Faster Releases	✅ CI/CD pipelines	✅ Secure CI/CD pipelines
Automation	✅ Infra, deploy, tests	✅ Security scans, compliance checks
Collaboration	✅ Dev + Ops	✅ Dev + Sec + Ops
Feedback & Monitoring	✅ Logs, metrics, alerts	✅ Security alerts, runtime protection
Security Integration	❌ Usually late in process	✅ Early and continuous
Compliance	❌ Manual effort	✅ Automated and continuous
Risk Management	❌ Reactive	✅ Proactive and preventive

Key Features of DevOps

1. Continuous Integration (CI)

✅ **What It Is:**

A practice where developers frequently push small code changes to a shared repository. Each push triggers an automated build and test pipeline.

💡 **Why It Matters:**

- Reduces integration issues
- Encourages smaller, frequent commits
- Detects bugs early

🛠 **Tools:**

Jenkins, GitHub Actions, GitLab CI, CircleCI

📌 **Example:**

An online payment gateway uses **GitLab CI** to run unit and integration tests every time a developer pushes code. If a test fails, the developer is notified instantly, reducing bug detection time from days to minutes.

---

2. Continuous Delivery (CD)

✅ **What It Is:**

An extension of CI where code changes are automatically delivered to staging or production environments after passing tests.

💡 **Why It Matters:**

- Frequent and safe software releases
- Reduced manual intervention
- Enables quick feature delivery

🛠 **Tools:**

Spinnaker, ArgoCD, Jenkins, GitLab CD

📌 **Example:**

A SaaS startup uses **ArgoCD** to sync code updates from Git to their Kubernetes cluster automatically, enabling multiple daily production releases without downtime.

---

3. Infrastructure as Code (IaC)

✅ **What It Is:**

Managing and provisioning infrastructure using declarative configuration files (e.g., YAML, HCL) that are version-controlled and reproducible.

💡 **Why It Matters:**

- Eliminates manual setup errors
- Allows quick replication of environments
- Supports GitOps workflows

🛠 **Tools:**

Terraform, AWS CloudFormation, Pulumi, Ansible

📌 **Example:**

A multi-region ecommerce platform uses **Terraform** to provision identical AWS environments (VPCs, EC2s, Load Balancers) across regions with a single command.

---

4. Automated Testing

✅ **What It Is:**

Executing tests automatically after every code change to ensure application quality and prevent regressions.

💡 **Why It Matters:**

- Catches issues early
- Increases confidence in deployments
- Supports test-driven development (TDD)

🛠 **Tools:**

JUnit, TestNG, Selenium, Cypress, PyTest

📌 **Example:**

An insurance company uses **JUnit + Selenium** in a Jenkins pipeline. Every code push triggers a suite of over 1,000 tests covering business logic and UI flows.

---

5. Monitoring & Logging

✅ **What It Is:**

Tracking application health and infrastructure performance using metrics, logs, and alerts.

💡 **Why It Matters:**

- Detects and resolves issues in real-time
- Improves visibility and observability
- Aids in capacity planning

🛠 **Tools:**

Prometheus, Grafana, ELK Stack, DataDog, New Relic

📌 **Example:**

A logistics startup uses **Prometheus + Grafana** to monitor API latency, memory usage, and database connections. Alerts are sent to Slack if response times exceed 500ms.

---

6. Collaboration & Shared Responsibility

✅ **What It Is:**

Breaking down silos between Dev, QA, and Ops teams to collaborate and share responsibility for the software lifecycle.

💡 **Why It Matters:**

- Encourages accountability and ownership
- Promotes faster resolution of issues
- Reduces miscommunication

🛠 **Tools:**

Slack, Microsoft Teams, Jira, Confluence, GitHub

📌 **Example:**

In a media streaming company, Devs, Ops, and QA use the same GitHub repo and Kanban board. Everyone sees the CI/CD status and logs, which enables joint debugging sessions during incidents.

---

7. Automated Rollbacks & Deployment Strategies

✅ **What It Is:**

Ability to quickly rollback to a previous version of an application in case of failure.

💡 **Why It Matters:**

- Reduces downtime
- Enhances confidence in frequent deployments

🛠 **Deployment Techniques:**

Blue-Green, Canary, Rolling, Feature Flags

📌 **Example:**

A banking application deploys new releases via **blue-green deployment**. If issues arise in the new version, traffic is redirected to the previous version instantly.

Key Features of DevSecOps

1. Security as Code

✅ **What It Is:**

Security policies, access controls, and compliance rules are written and version-controlled as code.

💡 **Why It Matters:**

- Enforces consistency and repeatability
- Enables peer reviews for security policies

🛠 **Tools:**

OPA (Open Policy Agent), HashiCorp Sentinel, Rego

📌 **Example:**

A healthcare platform uses **OPA** to enforce security policies like encrypted S3 buckets and restricted IAM roles as code within Terraform, ensuring HIPAA compliance.

---

2. Shift-Left Security

✅ **What It Is:**

Integrating security checks early in the development process (coding and build stages), rather than waiting for post-deployment scans.

💡 **Why It Matters:**

- Reduces the cost and effort of fixing vulnerabilities
- Makes security part of the Dev workflow

🛠 **Tools:**

Snyk, SonarQube, Trivy, Gitleaks

📌 **Example:**

A fintech company integrates **Trivy** and **Gitleaks** into GitHub Actions to scan every PR for vulnerable packages and hardcoded secrets before the code reaches staging.

3. Automated Security Testing

✅ **What It Is:**

Security scans are embedded into the CI/CD pipeline and run automatically with every code change.

💡 **Why It Matters:**

- Detects vulnerabilities instantly
- Enables continuous compliance and risk management

🛠 **Tools:**

CodeQL, Brakeman, OWASP ZAP, Fortify, Checkov

📌 **Example:**

A B2B SaaS firm uses **SonarQube** to run static code analysis on every PR, enforcing clean code and OWASP Top 10 security rules.

---

4. Dependency & Container Scanning

✅ **What It Is:**

Automatic scanning of application dependencies and Docker images to identify known vulnerabilities (CVEs).

💡 **Why It Matters:**

- Ensures no vulnerable libraries or images are shipped
- Helps organizations stay compliant with security standards

🛠 **Tools:**

Trivy, Clair, Anchore, Snyk

📌 **Example:**

A delivery app uses **Trivy** to scan Docker images for every microservice. If a critical CVE is found, the build fails and alerts are sent to the security team.

---

5. Secrets Management & Detection

✅ **What It Is:**

Detecting hardcoded secrets (API keys, tokens, passwords) and managing them securely using secret managers.

💡 **Why It Matters:**

- Prevents secret leaks and breaches

- Enforces secure access control

🛠 Tools:

Gitleaks, Talisman, HashiCorp Vault, AWS Secrets Manager

📌 Example:

Before every PR merge, a media app uses **Gitleaks** in GitHub Actions to prevent credentials from being committed accidentally. All secrets are injected at runtime using **Vault**.

---

6. Runtime Threat Detection

✅ What It Is:

Monitoring live applications and infrastructure for suspicious behavior or breaches.

💡 Why It Matters:

- Helps identify attacks in real-time
- Adds a second layer of defense after code review

🛠 Tools:

Falco, Sysdig Secure, AWS GuardDuty

📌 Example:

An IoT company uses **Falco** in its Kubernetes clusters to detect events like shell access inside containers or changes to binary files — indicators of compromise.

---

7. Continuous Compliance

✅ What It Is:

Ensure applications and infrastructure stay compliant with industry standards (e.g., PCI-DSS, GDPR, HIPAA) at all times.

💡 Why It Matters:

- Avoids fines and legal issues
- Builds trust with customers and stakeholders

🛠 Tools:

Cloud Custodian, OpenSCAP, AWS Config, Aqua Security

📌 Example:

A fintech firm uses **AWS Config** with custom rules to monitor encryption, logging, and access control — ensuring PCI-DSS compliance across all AWS accounts.

DevOps vs DevSecOps — Features Comparison Table

Feature	DevOps	DevSecOps
CI/CD Pipelines	✅ Yes	✅ Yes (with security steps)
Infrastructure as Code	✅ Yes	✅ Yes (with policy validation)
Security Integration	❌ Optional / late-stage	✅ Early and continuous
Automated Testing	✅ Functional testing	✅ Functional + Security testing
Monitoring	✅ Performance & errors	✅ + Security threat detection
Secrets Management	❌ Sometimes overlooked	✅ Integrated and enforced
Compliance	❌ Manual or ad hoc	✅ Continuous and automated
Culture	Dev + Ops	Dev + Sec + Ops

Deployment Strategies

🔍 What is a Deployment Strategy?

A **deployment strategy** defines **how you release new versions of software** into production — while minimizing downtime, reducing risk, and ensuring a smooth experience for users. The strategy affects how traffic is routed, how rollbacks are handled, and how testing is done in live environments.

Types of Deployment Strategies

1. Recreate Deployment (Traditional Approach)

✅ Definition:

Stops the old version completely and then deploys the new one. There is **complete downtime** during the transition.

🧠 When to Use:

- For small internal apps
- When downtime is acceptable
- Simple apps without live traffic

📌 Example:

An internal HR portal is updated after working hours. The old app is stopped, and the new version is deployed manually.

🛠️ Steps to Implement:

1. Stop the current running app/service.
2. Deploy the new version.
3. Start the service again.
4. Test functionality manually or with scripts.

---

2. 🟢🔵 Blue-Green Deployment

✅ Definition:

Two identical environments (Blue & Green) run in parallel. One is live (e.g., Blue), and the new version is deployed to the other (Green). After testing, traffic is switched.

💡 Benefits:

- Near zero-downtime
- Easy rollback (switch back to old environment)

- Enables pre-production validation

🧠 **When to Use:**

- For stable apps with predictable releases
- When infrastructure can be duplicated

📌 **Example:**

A banking app maintains a “Blue” environment in production. A new release is deployed to “Green”. After validation, the load balancer reroutes traffic to Green. If errors occur, switch back to Blue.

🛠️ **Steps to Implement:**

1. Set up two identical environments (Blue & Green).
2. Route traffic to Blue (current prod).
3. Deploy new version to Green.
4. Test Green environment thoroughly.
5. Switch traffic to Green using DNS or load balancer.
6. Monitor, then decommission Blue.

---

3. Canary Deployment

✅ **Definition:**

Gradually roll out the new version to a **subset of users**, monitor performance, then increase rollout in phases.

💡 **Benefits:**

- Low-risk release
- Real-user feedback
- Quick rollback if issues arise

🧠 **When to Use:**

- High-traffic, customer-facing apps
- Data-driven decision making

📌 **Example:**

A music app releases a new player feature to 5% of users. After tracking engagement and crash reports for 2 days, the deployment is scaled to 100%.

🛠️ **Steps to Implement:**

1. Deploy new version to a small set (e.g., 5%) of servers or pods.

2. Use feature flags, load balancer rules, or service mesh (like Istio) to route specific users.
3. Monitor metrics (latency, errors, CPU).
4. If all is good, increase rollout to 25%, 50%, etc.
5. Fully roll out to 100%.
6. Roll back if issues are detected.

---

4. Rolling Deployment

✅ **Definition:**

Gradually replace the old version with the new one **in batches**, one server or pod at a time.

💡 **Benefits:**

- No downtime
- Resource efficient (no duplicate environment)

🧠 **When to Use:**

- Microservices or container-based apps
- Large-scale distributed systems

📌 **Example:**

An ecommerce site with 10 backend pods upgrades 1 pod at a time. Kubernetes replaces old pods with new ones using rolling updates.

🛠️ **Steps to Implement (Kubernetes):**

1. Define deployment object with updated container image.
2. Set maxUnavailable and maxSurge values in the rollout strategy.
3. Apply deployment.
4. Kubernetes updates pods in controlled batches.
5. Monitor each pod before proceeding to the next.

---

5. A/B Testing Deployment

✅ **Definition:**

Run **two or more versions (A and B)** of your application simultaneously to compare user behavior and performance.

💡 **Benefits:**

- Test features with real users

- Optimize UX and business decisions

🧠 **When to Use:**

- UI/UX testing
- Data-driven product decisions

📌 **Example:**

A social network tests two different layouts of the profile page. 50% of users get Version A, 50% get Version B. Analytics determine which performs better.

🛠️ **Steps to Implement:**

1. Deploy both versions (A & B) simultaneously.
2. Use feature flag service or traffic router to segment users.
3. Collect metrics (bounce rate, session time, etc.).
4. Analyze results and make decisions.
5. Promote the winning version.

---

6. Shadow Deployment

✅ **Definition:**

Deploy the new version alongside the old one, but route **a copy of live traffic** to the new version **without exposing it to users**.

💡 **Benefits:**

- Real-world testing without user risk
- Detect runtime issues before launch

🧠 **When to Use:**

- Backend services
- New architecture or performance testing

📌 **Example:**

An API gateway shadows live traffic to a new recommendation engine backend. Errors are logged but not shown to users. Engineers refine the new service based on results.

🛠️ **Steps to Implement:**

1. Deploy the new service version alongside production.
2. Configure proxy or gateway (e.g., Envoy, Istio) to duplicate requests to new version.
3. Do not return shadow responses to the user.

4. Log and analyze outputs.
5. Tune and finalize the new version.

Choosing the Right Deployment Strategy

Strategy	Downtime	Rollback Speed	User Impact	Complexity	Ideal For
Recreate	High	Medium	All users	Low	Small internal apps
Blue-Green	Low	Fast	None	Medium	Critical apps needing stability
Canary	Low	Fast	Few users	Medium	Gradual rollouts with monitoring
Rolling	Low	Medium	Some users	Medium	Container-based infra (e.g., K8s)
A/B Testing	None	N/A	Segments	High	UI/UX and performance comparisons
Shadow	None	N/A	None	High	Risk-free backend validation

Tools That Support These Strategies

Tool/Platform	Supports
Kubernetes	Rolling, Canary, Blue-Green, Shadow
ArgoCD	Blue-Green, Canary
Spinnaker	All strategies
LaunchDarkly	A/B testing
Istio / Envoy	Canary, A/B, Shadow, Traffic splitting
GitHub Actions / GitLab CI	Pipeline-based rollouts
Terraform + Ansible	Infra automation for Blue/Green

DevOps Environments

A **DevOps environment** is a **dedicated setup of infrastructure, software, and configuration** that supports specific phases of the software delivery lifecycle — from development and testing to production deployment.

Each environment is isolated but **replicates production in different degrees**, allowing the team to build, test, and deploy confidently.

💡 Why Multiple Environments?

Using multiple environments allows:

- Safe development without affecting live users
- Thorough testing in near-production conditions
- Controlled and phased rollout of features
- Easier debugging and rollback
- Enforcement of quality gates (like in CI/CD)

🧱 Common Types of DevOps Environments

🔹 1. Local Development Environment

Where developers write and test code on their own machines.

🔧 **Tools:**

- **IDE**: VS Code, IntelliJ
- **Containerization**: Docker / Docker Compose
- **Minikube / Kind** for local Kubernetes
- **Mock APIs** and **SQLite/local DBs**

✅ **Used For:**

- Writing and debugging code
- Running unit tests
- Experimenting with new features

🧠 **Example:**

A Java developer runs mvn spring-boot:run to locally test a new login API before pushing code.

🔹 2. Development Environment (DEV)

First shared environment where features are deployed after passing CI.

🔧 **Tools:**

- Kubernetes Cluster (Namespace: dev)
- Jenkins CI/CD or GitHub Actions
- Docker Registry for images
- Prometheus/Grafana for monitoring

✅ **Used For:**

- Integrating new features
- Team collaboration
- Smoke testing
- Basic integration testing

🧠 **Characteristics:**

- Often unstable
- Frequent redeployments
- Uses mock data / dev secrets
- Debug logs enabled
- Feature flags for in-progress code

📌 **Example:**

After CI pipeline completes, the Docker image is deployed to the dev namespace using:

kubectl apply -f dev-deployment.yaml

🔹 3. QA / Testing Environment

Used by testers for validating application quality.

🔧 **Tools:**

- Selenium, JMeter, Postman for test automation
- TestNG/JUnit results in Jenkins
- Test data populated by scripts or synthetic generation

✅ **Used For:**

- Functional testing
- Regression testing
- API testing
- Performance/load testing

🧠 **Characteristics:**

- Test users and test DBs
- More stable than Dev
- Periodically reset

📌 **Example:**

Testers verify that a bug fix doesn't reintroduce a login failure by rerunning automated regression tests.

🔹 **4. Pre-Production / PPD / UAT Environment**

Mimics production closely — a **mirror of the production stack**.

🔧 **Tools:**

- Kubernetes Cluster (Namespace: ppd)
- Real secrets from Vault/SealedSecrets
- DNS & SSL setup identical to prod
- Canary deployment support

✅ **Used For:**

- Final validation before prod
- Business user approval (UAT)
- Load/stress testing with production-like data

🧠 **Characteristics:**

- Ingress + HTTPS configured
- Connected to staging backend systems
- Metrics + Logging fully enabled

📌 **Example:**

Before releasing a new invoice system, business users test it in PPD with real workflows and sign off.

🔹 **5. Production (PROD) Environment**

This is the **live environment** accessed by real users.

🔧 **Tools:**

- Kubernetes/EC2/VMs for infrastructure
- Helm/ArgoCD/Flux for GitOps deployments
- cert-manager for TLS
- Prometheus/Grafana for live metrics
- Loki or ELK for logging
- Sentry/NewRelic for error tracking

✅ **Used For:**

- Running the actual app for customers
- Ensuring high availability and performance
- Capturing real-time logs and metrics

🧠 **Characteristics:**

- Immutable deployments (tagged versions)
- Load-balanced, auto-scaled
- Restricted access

- 24/7 Monitoring & alerting
- Rollback enabled
- Strong RBAC & security policies

📌 **Example:**

When a developer merges code to the main branch, Jenkins deploys the new Docker image using Helm:

helm upgrade --install app-prod ./charts --values values-prod.yaml

---

🔹 **6. Disaster Recovery (DR) Environment**

A backup clone of PROD, used during critical outages.

🔧 **Tools:**

- Same as PROD but on different region/cloud
- Replicated DB (cold/hot)
- Automated failover mechanism

✅ **Used For:**

- Business continuity during failures
- Simulated DR drills

🧠 **Additional Types**

Environment	Purpose
Sandbox	Experimental playground
Demo	Demos for clients or internal
Canary	Release new features to subset
Blue/Green	Parallel deployment strategy

🌍 **Example Environment Flow in a Real DevOps Lifecycle**

1. **Dev pushes code → DEV**
2. QA runs tests in **QA**
3. Business reviews in **PPD**
4. Code gets promoted to **PROD**
5. If PROD fails, rollback or shift traffic to **Blue-Green/DR**

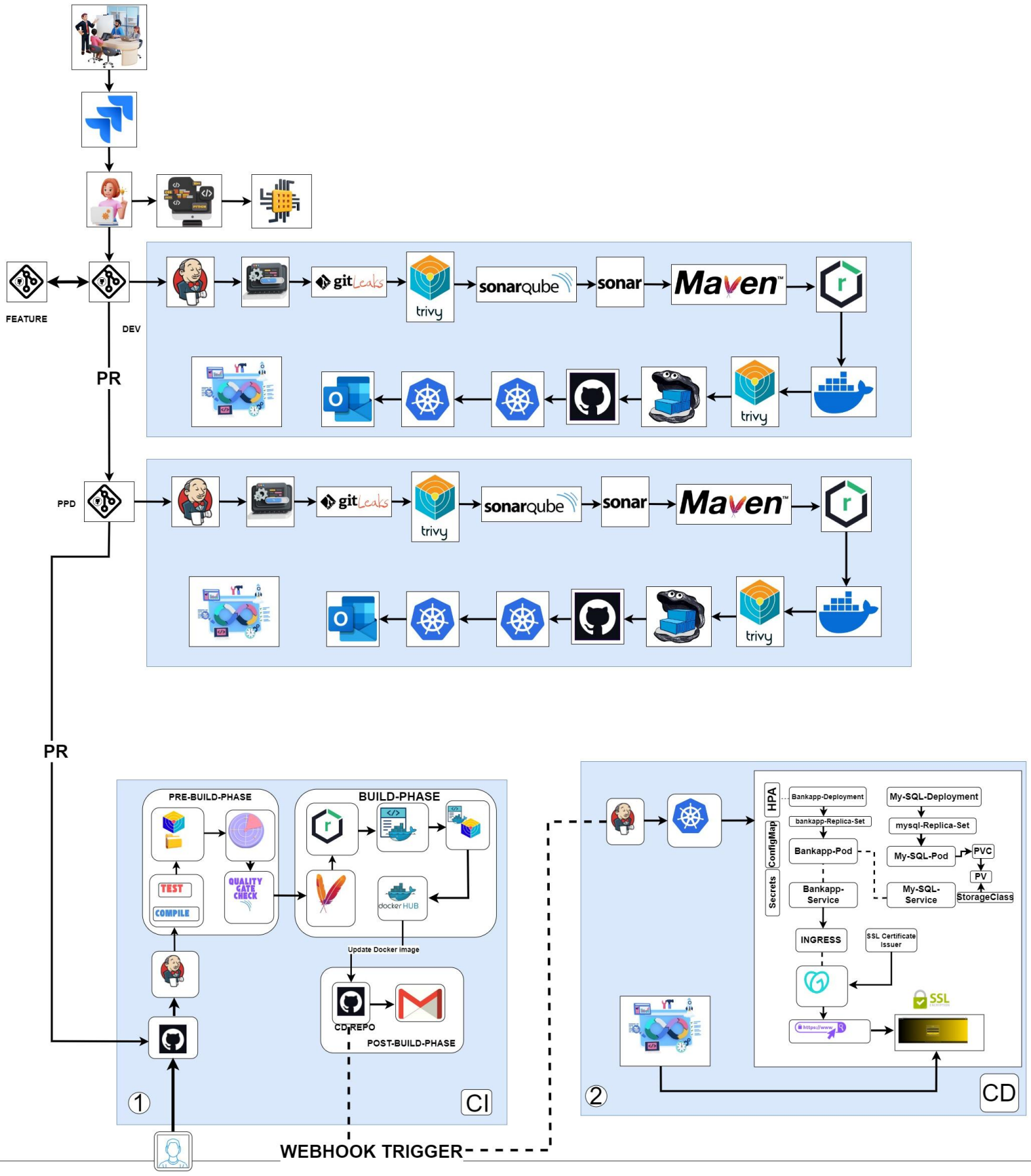
✅ Best Practices for Managing Environments

- Use **separate namespaces** or **clusters** for each environment
- **Parameterize deployments** using Helm or Kustomize
- Automate deployments using **GitOps** tools like ArgoCD
- Inject secrets securely via **Vault/SealedSecrets**
- Monitor and alert every environment (not just prod)
- Log retention policies should vary by environment
- Lock PROD deployments with **manual approvals**
- Automate **environment creation for each PR**

📦 Tool Stack per Environment (Sample Matrix)

Tool	Dev	QA	PPD	Prod
Jenkins	✅	✅	✅	✅
Docker	✅	✅	✅	✅
Kubernetes	✅	✅	✅	✅
SonarQube	✅	✅	✅	❌
Trivy	✅	✅	✅	✅
Prometheus/Grafana	✅	✅	✅	✅
HashiCorp Vault	❌	✅	✅	✅
cert-manager	❌	✅	✅	✅
ArgoCD	✅	✅	✅	✅

Jenkins-Based End-to-End CI/CD Pipeline for Kubernetes



🧭 1. Project Planning & Development Kickoff

The pipeline begins even before any code is written. It starts from **project management and task planning**, where business requirements are broken down into small deliverables using tools like **JIRA**.

Tools:

- **JIRA** – Sprint planning, task tracking, and linking code to features
- **Git (GitHub/GitLab/Bitbucket)** – Source code repository with branching strategy

Flow:

- A new **user story** is created in JIRA:
  "As a user, I want to login securely with email and password."
- The story is broken down into:
  - Create login API
  - Add validation
  - Connect to DB
  - Write unit tests
- A developer is assigned the task and begins coding.

👨‍💻 2. Local Development & Feature Branching

The developer pulls the latest code and creates a **feature branch** using Git. This ensures isolated development without affecting the main codebase.

Example:

```
git checkout -b feature/login-api
```

Tools:

- **VS Code, IntelliJ IDEA** – Code editors
- **Docker Compose / Minikube** – Run the full app stack locally
- **Postman** – API testing
- **Prettier/ESLint** – Code formatting

During development:

- Business logic is implemented.
- Environment variables are defined.
- Unit tests are written.
- .env or config.yml is prepared.

🔁 3. Git Push & Pull Request Lifecycle

After completing development, the code is pushed to the remote repository and a **Pull Request (PR)** is raised from the feature branch to dev.

Example:

```
git push origin feature/login-api
```

A webhook is triggered to Jenkins when the PR is raised, which kicks off the **automated CI pipeline**.

⚙️ 4. Jenkins CI Pipeline Triggered

The Jenkins master receives the webhook and triggers the pipeline defined in the Jenkinsfile. Jenkins checks out the code, sets up the environment, and runs all configured stages.

Pipeline Configuration:

```
pipeline {
  agent any
  stages {
    stage('Pre-Build') { ... }
    stage('Build') { ... }
    stage('Test') { ... }
    stage('Quality Gate') { ... }
    stage('Security Scan') { ... }
    stage('Docker Build & Push') { ... }
    stage('Deploy to Dev') { ... }
  }
}
```

---

5. Pre-Build Phase

This phase validates the code structure and checks for early errors.

Tasks:

- Code formatting checks (Prettier, Black)
- Linting (ESLint, Checkstyle)
- Secrets scanning (Gitleaks)
- Dependency resolution (npm install, mvn clean install)

Example:

If Gitleaks detects a hardcoded secret (like an AWS access key), it halts the pipeline before any further action, preventing exposure.

---

6. Build Phase

The code is compiled and packaged.

Tool: Maven

- Clean build: mvn clean install
- Creates .jar or .war file in /target

Jenkins Output:

[INFO] BUILD SUCCESS

[INFO] Final artifact: target/myapp-1.0.0-SNAPSHOT.jar

---

7. Test Phase

All unit tests are executed and test reports are generated.

Tools:

- **JUnit**, **TestNG** – Unit test framework
- **Jacoco** – Code coverage
- **Jenkins JUnit plugin** – Displays results in dashboard

Example Output:

Total Tests: 45 | Passed: 43 | Failed: 2 | Coverage: 88%

If test coverage falls below 80%, Jenkins can be configured to fail the pipeline.

🧠 8. Code Quality Analysis (SonarQube)

Static analysis is conducted using **SonarQube**. Jenkins integrates with a running SonarQube server to analyze the code.

Metrics:

Bugs
Code Smells
Vulnerabilities
Duplications
Maintainability Index

Quality Gate:

- Fails build if:
    - Coverage < 80%
    - Any new critical issue
    - High technical debt

Jenkins Console:

SonarQube Quality Gate: FAILED

Reason: 3 Critical Bugs, 2 Vulnerabilities

🛡️ 9. Security Scanning (Trivy)

The .jar or Docker image is scanned for CVEs using **Trivy**.

Example Output:

```
trivy fs ./target/myapp.jar
```

- Finds issues in dependencies (e.g., log4j, spring-core)
- Can break pipeline if severity = HIGH or CRITICAL

---

10. Docker Image Build & Push

Jenkins builds a Docker image of the app and pushes it to a container registry.

Dockerfile:

```
FROM openjdk:17
COPY target/myapp.jar app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Commands:

```
docker build -t registry.com/myapp:dev .
docker push registry.com/myapp:dev
```

Output:

Successfully pushed myapp:dev to Docker Hub

---

11. Notifications

The build result is communicated through various channels:

Examples:

- **Slack**: "Build #105 Passed ✅"
- **Email**: "Subject: Jenkins Pipeline Failed ❌"
- **GitHub Status Check**: PRs show CI status inline

---

12. Deploy to Kubernetes (Dev)

Jenkins connects to the **Kubernetes DEV cluster** and applies manifests.

Process:

- Set KUBECONFIG in Jenkins agent
- Use kubectl apply or Helm chart

YAML Sample:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 2
  template:
    spec:
      containers:
        - image: registry.com/myapp:dev
```

- DEV environment has mock services or sandbox data
- Monitoring enabled (Prometheus/Grafana)

---

13. Merge to QA & PPD

Once verified in DEV:

- PR from dev → QA
- PR from QA → ppd
- Jenkins runs same CI pipeline
- Deploys to PPD cluster with real integrations

PPD Key Characteristics:

- Real secrets fetched from **HashiCorp Vault**
- Feature flags turned ON
- Load testing / staging users perform UAT

---

14. Merge to Main & PROD Deployment

The final merge triggers the **PROD release pipeline**.

Features:

- Blue/Green or Canary deployment via Helm
- TLS with **cert-manager**

- Live traffic routing via **Ingress**

Helm Command:

helm upgrade --install myapp ./charts/myapp --values prod-values.yaml

---

🔍 15. Kubernetes Resources Breakdown

Resource	Role
Deployment	Rollouts, rolling updates
ReplicaSet	Maintains desired pod count
Pod	Hosts containers
Service	Exposes pods inside cluster
Ingress	Routes traffic to services
PVC	Persistent storage for DB or state
Secret/ConfigMap	Injects runtime config and credentials

---

16. Observability and Post-Deploy

- **Prometheus**: Monitors metrics (CPU, memory, HTTP errors)
- **Grafana**: Dashboards for health and business KPIs
- **Loki/ELK**: Centralized logs
- **Sentry**: Real-time error tracking

---

17. Security Best Practices

- Enforce non-root users in Docker
- Regular base image scans
- Vault-injected secrets (never hardcoded)
- RBAC for Jenkins and cluster access
- Image Signing with Cosign (optional)