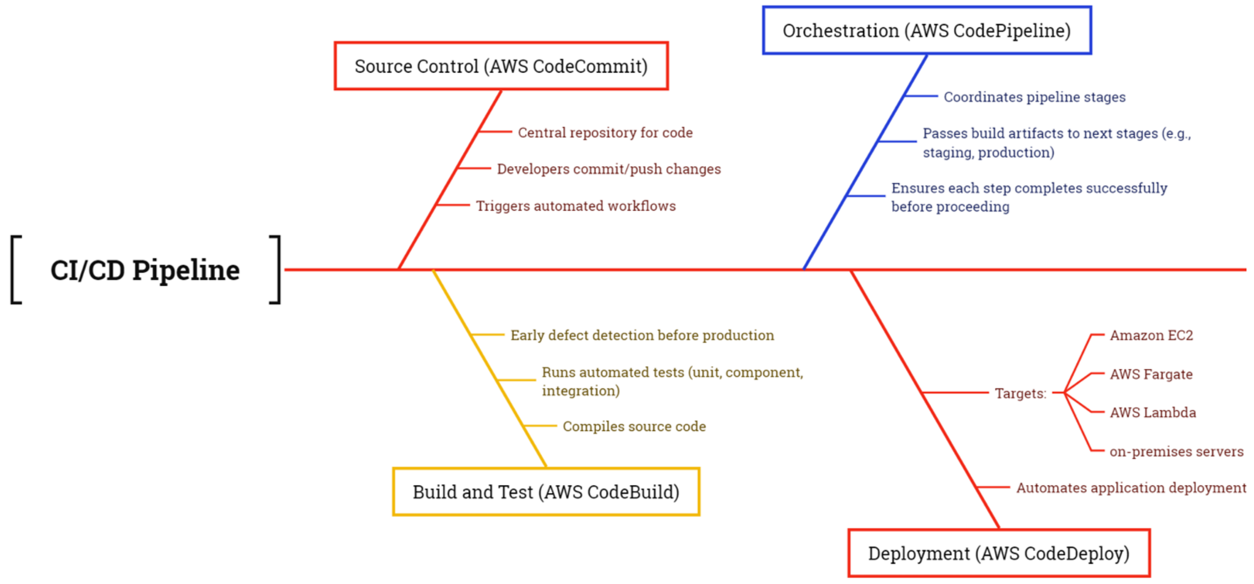
Week 02: DevOps 2 Part 2 Notes

CI/CD Pipelines

- **Source Control (AWS CodeCommit)**
  Developers **commit or push their code changes** to a central repository, **which triggers** automated workflows.
- **Build and Test (AWS CodeBuild)**
  CodeBuild **compiles the source code** and runs **automated unit**, **component**, and **integration tests**. This **early testing** helps catch defects **before** they **reach production**.
- **Orchestration (AWS CodePipeline)**
  CodePipeline **coordinates** each pipeline stage, **passing build artifacts** to **subsequent steps** such as **staging** or **production deployment**.
- **Deployment (AWS CodeDeploy or Other Services)**
  CodeDeploy automates the **deployment of new application versions** to various platforms, such as Amazon **EC2**, AWS **Fargate**, AWS **Lambda**, or **on-premises servers**.


CI/CD Pipeline
Source Control (AWS CodeCommit)
Central repository for code
Developers commit/push changes
Triggers automated workflows
Orchestration (AWS CodePipeline)
Coordinates pipeline stages
Passes build artifacts to next stages (e.g., staging, production)
Ensures each step completes successfully before proceeding
Early defect detection before production
Runs automated tests (unit, component, integration)
Compiles source code
Build and Test (AWS CodeBuild)
Targets:
Amazon EC2
AWS Fargate
AWS Lambda
on-premises servers
Automates application deployment
Deployment (AWS CodeDeploy)

Testing and Quality Assurance

Note: **across the pipeline**, multiple layers of testing ensure quality:

- **Unit Testing**

Tests individual methods or classes
Verifies smallest testable parts of code

- **Component Testing**

Tests integrated functions within a microservice
Ensures service-level functionality

- **Integration Testing**

Tests interactions between multiple services
Validates system connectivity

- **Functional (End-to-End) Testing**

Simulates real user workflows
Tests complete business scenarios

- **Performance Testing**

Measures application behavior under load
Validates system scalability and response times

These tests are **embedded across pipeline stage**s to ensure only stable, reliable code reaches production.

Deployment Strategies

- **In-Place (Rip and Replace)**

Updates existing instances directly
Fastest, however requires downtime
Best for: Development environments, internal apps
Risk Level: High
Rollback: Difficult

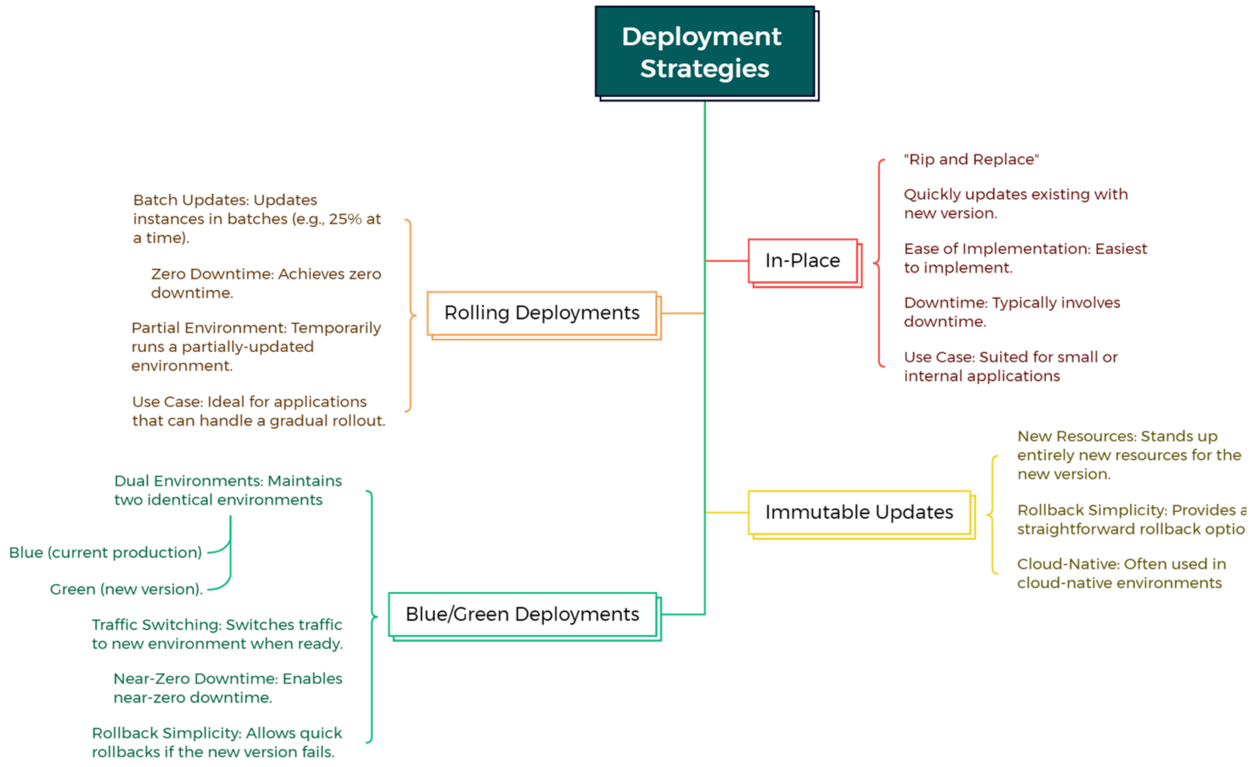
- **Rolling Deployments**

Updates instances gradually in batches
Zero downtime, mixed versions temporarily
Best for: Stateless apps, high-availability needs
Risk Level: Medium
Rollback: Moderate complexity

- **Immutable Updates**

Creates new instances, destroys old ones
No modification of existing instances
Best for: Cloud-native apps, containerized systems
Risk Level: Low
Rollback: Simple

- **Blue/Green Deployments**
- Maintains two identical environments
- Switches traffic when new version is ready
- **Best for:** Critical production applications
- Risk **Level:** Low
- **Rollback:** Very simple (switch back to blue)


Deployment Strategies
In-Place
"Rip and Replace"
Quickly updates existing with new version.
Ease of Implementation: Easiest to implement.
Downtime: Typically involves downtime.
Use Case: Suited for small or internal applications
Rolling Deployments
Batch Updates: Updates instances in batches (e.g., 25% at a time).
Zero Downtime: Achieves zero downtime.
Partial Environment: Temporarily runs a partially-updated environment.
Use Case: Ideal for applications that can handle a gradual rollout.
Immutable Updates
New Resources: Stands up entirely new resources for the new version.
Rollback Simplicity: Provides a straightforward rollback optio
Cloud-Native: Often used in cloud-native environments
Blue/Green Deployments
Dual Environments: Maintains two identical environments
Blue (current production)
Green (new version).
Traffic Switching: Switches traffic to new environment when ready.
Near-Zero Downtime: Enables near-zero downtime.
Rollback Simplicity: Allows quick rollbacks if the new version fails.

Organizations increasingly need to balance standardization with developer autonomy.

Two AWS services address this challenge:

AWS Proton

Purpose: Manages and automates infrastructure deployment for serverless and containerized applications using templates.
Key Components:

- **Environment Template**: Defines shared infrastructure (VPCs, databases, clusters)
- **Service Template**: Specifies application infrastructure and CI/CD resources
- **Service Instance**: Deployed service template within a specific environment

AWS Service Catalog

Purpose: Enables centralized management of approved IT resources as products, ensuring compliance while simplifying deployment.
Key Components:

- **Portfolio**: Collection of approved products (CloudFormation stacks)
- **Product**: Deployable IT resource (e.g., EC2 instance, web application)
- **End Users**: Teams who deploy pre-approved products

Integration

AWS Service Catalog and Proton serve different yet complementary roles in CI/CD pipelines:

- Service Catalog provides governance and a catalog of pre-approved IT services and resources
- Proton automates the deployment and ongoing management of containerized and serverless applications

AWS Tools in Action

Putting it all together:

- **CodeCommit** hosts the source code.
- **CodeBuild** compiles and tests the application.
- **CodePipeline** orchestrates the sequence of stages, from code commit to production.
- **CodeDeploy** or alternative methods (Elastic Beanstalk, manual processes) handle deployments to various compute targets.
- **AWS Proton** provisions and updates entire environments for modern container and serverless applications.
- **AWS Service Catalog** provides a portfolio of preapproved infrastructure components for broader compliance and governance.
- **Amazon CloudWatch** provides real-time logging and metrics, for faster response to incidents.