CI/CD - CONTINUOUS INTEGRATION AND CONTINUOUS DELIVERY

WHAT IS CI/CD?

CI - CONTINUOUS INTEGRATION

Definition: CI or Continuous Integration is a process where you integrate a set of tools and processes that you follow before delivering your application to your customer.

What it includes:

- Code integration
- Automated testing
- Code quality checks
- Security scanning
- Build automation

CD - CONTINUOUS DELIVERY

Definition: CD or Continuous Delivery is a process where you deploy your application or deliver your application on a specific platform to your customer.

What it includes:

- Automated deployment
- Environment management
- Release automation
- Customer delivery

THE PROBLEM: DELIVERING APPLICATION TO CUSTOMER

Scenario:

You are an application developer and you have developed an application on your personal laptop.

Your customer is sitting in some other part of the world.

Question: How is your application delivered from your laptop to this customer?

Answer: Every organization goes through this process of CI/CD.

WHAT HAPPENS BEFORE DELIVERY

Before customer can use your application:

1. App has to be tested
2. Scanned for security vulnerabilities
3. Code quality checked
4. Performance tested
5. Finally app is deployed which can be accessed by customer

WHY AUTOMATION IS NEEDED

CI/CD is a process where we automate all of this.

Why we have to automate all steps:

- We want to deliver the product as soon as possible
- Save time
- Reduce manual errors
- Consistent process
- Faster releases

It is important to have all steps automated to save time and manual errors.

STANDARD CI/CD STEPS

These are the standard steps every organization has to follow:

1. Unit Testing
2. Static Code Analysis
3. Code Quality / Vulnerability Testing
4. Automation Testing

5. Reporting
6. Deploying Application

Let's understand each of these steps in detail.

STEP 1: UNIT TESTING

What is unit testing: Testing code in a specific block or specific functionality.

Example: Calculator App

You are writing calculator app and writing addition functionality in Python.

Code written:

python

```python
def add(a, b):
  c = a + b
  return c
```

Unit test: You will pass some arguments like 2 and 3 and see if output is 5 in your testing.

Test code:

python

```python
def test_add():
  result = add(2, 3)
  assert result == 5
  print("Test passed!")
```

What you are testing: You are testing code in that specific block or that specific functionality.

This is called unit testing.

Characteristics:

- Tests individual functions
- Isolated testing
- Fast execution
- Developer writes tests

- Catches bugs early

STEP 2: STATIC CODE ANALYSIS

What is static code analysis: Analyzing code without executing it to find issues.

Example: Calculator App

Same calculator app example.

Code written:

```python
var1 = 10
var2 = 20
var3 = 30
var4 = 40
var5 = 50
# ... total 20 variables declared

def add(a, b):
    result = a + b
    return result
```

Problem:

I have declared 20 variables but I am using just 2 variables.

Issue:

We are wasting a lot of memory.

What static code analysis does:

Checks if:

- You are syntactically correct
- You have well-formatted your application
- You have declared any unnecessary variables
- Code follows best practices
- Code is readable
- Code is maintainable

These kinds of things are called static code analysis.

What it checks:

- Unused variables
- Code formatting
- Naming conventions
- Code complexity
- Duplicate code
- Security issues
- Performance issues

Tools:

- Python: Pylint, Flake8
- Java: SonarQube, Checkstyle
- JavaScript: ESLint

STEP 3: CODE QUALITY AND VULNERABILITY TESTING

What is vulnerability testing:

Checking for security issues before delivery.

Example: Android Version

Scenario:

You got a new version of Android yesterday and you upgraded to it.

Today you got to know:

There is a security vulnerability. There can be a hacker who can attack your mobile.

Result:

This is a very bad customer experience.

Solution:

Before we deliver the product to the customer or before we promote the application from one stage to another stage, we do the code quality testing.

What is checked:

- Known security vulnerabilities
- Outdated dependencies
- Insecure code patterns
- Data exposure risks
- Authentication issues
- Authorization flaws

Tools:

- SonarQube
- OWASP Dependency Check

- Snyk
- Trivy

STEP 4: AUTOMATION TESTING

What is automation testing:

Verify application in end-to-end manner.

Example: Calculator App

Check whether:

- Addition works
- Subtraction works
- Multiplication works
- Division works
- All features work together correctly

Types of automation testing:

Integration testing:

Test how different parts work together

End-to-end testing:

Test complete user flows from start to finish

Regression testing:

Ensure new changes didn't break existing features

Performance testing:

Test application under load

Tools:

- Selenium (web applications)
- Cypress (web applications)
- JMeter (performance testing)
- Postman (API testing)

STEP 5: REPORTING

What is reporting:

All the reports will be generated which has logs and other information.

What reports include:

- Test results
- Code coverage
- Security scan results
- Performance metrics
- Build status
- Deployment status

Why important:

- Track quality over time

- Identify trends
- Share with team
- Audit trail

Tools:

- Jenkins (built-in reports)
- Allure Reports
- SonarQube dashboards

STEP 6: DEPLOYMENT

What is deployment:

App is deployed which is then accessible by customers.

Where deployed:

- Production servers
- Cloud platforms
- Container orchestration platforms
- CDN (for static content)

Deployment methods:

- Blue-Green deployment
- Canary deployment
- Rolling deployment
- Recreate deployment

WHY AUTOMATION IS CRITICAL

If we do all of this manually:

It will take a lot of time to deliver the features to the customer.

Manual process timeline:

```
Day 1: Developer writes code

Day 2: Manual testing

Day 3: Code review

Day 4: Security scan

Day 5: Deploy to staging

Day 6: More testing

Day 7: Deploy to production


Total: 7 days for one feature
```

Automated process timeline:

```
Hour 1: Developer pushes code

Hour 2: Automated tests run

Hour 3: Code scanned, deployed to staging

Hour 4: Deployed to production

Total: 4 hours for one feature
```

` ` `

So we choose automation.

HOW DEVELOPMENT ACTUALLY WORKS

What usually happens:

Whenever we create an application, we do not create it in one go or we do not write all the code in one go.

What we do:

We break the application into chunks.

Process:

1. You create Jira stories for each feature or functionality
2. You keep submitting them to your Git repository
3. You store everything in a Git repo

VERSION CONTROL WORKFLOW

Every time you are confident about a change:

Example: Addition functionality

Process:

You are writing addition functionality and you keep on writing optimized code till Version 1 to Version 15.

On Version 15:

You become confident that this is the perfect code and it has to go to the customer.

Before that:

Whenever you are writing Version 1, Version 2 up to Version 14, all of them have to be stored in some place.

This is called VCS (Version Control System).

Tools for VCS:

- GitHub
- GitLab
- Bitbucket
- Azure Repos

Your CI/CD gets executed when you push your changes to one of these tools.

CI/CD TRIGGER

When CI/CD starts:

Whenever this app developer is confident about Version 1, he will push this Version 1 to the VCS (GitHub).

From there, your CI/CD pipeline or process will take care of further things.

CI/CD PIPELINE WITH JENKINS

DEPLOYING JENKINS

What we do:

Step 1:

App developer is confident about Version 1 of the code and he pushes the changes to GitHub.

Step 2:

We will deploy a CI/CD tool (Jenkins) in our organization.

Step 3:

We will tell Jenkins: "Always look at the GitHub. If a new commit is made to the Git repository on a specific branch of a specific repo, just tell me and then I will run a set of actions like unit testing and other stuff."

JENKINS AS ORCHESTRATOR

What Jenkins does:

Jenkins will act as a pipe or orchestrator or a tunnel.

As part of this pipeline:

Jenkins will orchestrate a lot of tools in it.

TOOLS INTEGRATED WITH JENKINS

Example: Java application

Tool 1: Maven

- Used for building Java application
- Compiles code
- Creates JAR/WAR files

Tool 2: JUnit

- Running unit tests
- Validates code functionality

Tool 3: SonarQube

- Code quality analysis
- Code testing
- Security scanning

Tool 4: ALM or other reporting tools

- Generate reports

- Track metrics

And many more tools can be integrated.

DEVOPS ENGINEER RESPONSIBILITY

What DevOps engineer does:

Configure or install all of these tools within Jenkins so that all of these get run whenever your code gets committed to the GitHub repository.

How it works:

We can integrate these tools in Jenkins and many other tools as well.

Jenkins is called orchestrator:

Jenkins will integrate all of the other tools by using Jenkins pipeline.

What is Jenkins pipeline:

Jenkins pipeline is something that takes care of automating all of these tools and executing the actions that these tools do.

Example:

Once you integrate Maven, your pipeline should also execute it. Similarly for other tools.

This is Step 1 of Jenkins setup.

STEP 2: MULTI-ENVIRONMENT DEPLOYMENT

What Jenkins can do:

Your Jenkins can promote your application to different environments like Dev, Staging, and Production.

UNDERSTANDING ENVIRONMENTS

What is this process:

Usually organizations divide their environments into multiple environments.

ENVIRONMENT 1: PRODUCTION

What it is:

Production environment is the environment which your customer is directly using.

Characteristics:

- Real users access this
- Must be highly available
- Cannot have downtime
- Most resources allocated

Setup details:

Even though we perform different steps which we saw earlier (unit testing, static code analysis, etc.), there is a difference between environment to environment.

In production environment:

- Will have a lot of servers
- Replicates the exact setup where you are deploying your app to your customer

Example production setup:

- Maybe 10 EC2 instances with auto-scaling
- Or Kubernetes platform with 3 masters and 30 worker nodes
- Load balancers
- Database replicas
- Full infrastructure

ENVIRONMENT 2: STAGING

What it is:

Testing environment before production.

Characteristics:

- Less resources than production
- Used for final testing before production
- QA team tests here

Setup details:

In staging, you might be having less amount of RAM or less amount of CPU.

Here you deploy your app before deploying it to production environment.

Example staging setup:

- Maybe 2 EC2 instances using auto-scaling
- Or Kubernetes platform with 3 masters and 5 worker nodes
- Single load balancer
- Smaller database

ENVIRONMENT 3: DEVELOPMENT (DEV)

What it is:

First deployment environment for developers and basic QA.

Relationship to staging:

Dev is like -1 for your staging (one level below staging).

Characteristics:

- Minimal resources
- Simple setup
- Quick testing
- Lowest cost

Example dev setup:

- Single EC2 instance

- Or single node Kubernetes cluster

- No load balancer

- Basic setup

WHY DEPLOY TO DEV FIRST

Reason:

Your QA might want to execute tests on a simple environment like a simple EC2 instance.

Process:

Step 1: Deploy to Dev

- Test on simple EC2 instance

- Or single node Kubernetes cluster

- Quick validation

Step 2: Once everything is good

Your Jenkins can automatically promote using some manual approval or automatic approval.

Step 3: Promote to Staging

Now the app will be promoted to staging environment.

Staging setup:

Instead of simple EC2 instance, you might have a cluster here.

- Maybe using auto-scaling you created 2 EC2 instances

- Or maybe you are using Kubernetes platform where you might have 3 masters and 5 nodes

Step 4: Promote to Production

After staging testing completes.

Production setup:

Maybe you have 3 masters and 30 worker nodes.

PROGRESSIVE TESTING STRATEGY

Every time before you deploy your application to production:

Step 1:

Test it on a single node Kubernetes cluster in your development environment

Step 2:

Go to staging environment with 3 masters and 5 nodes

Step 3:

Finally to production environment with 3 masters and 30 worker nodes

---

WHY NOT PRODUCTION-LIKE STAGING?

Question:

Why do we not have production-like environment in staging?

Answer:

Because it is a very costly setup.

Example:

If production has 30 worker nodes, creating another 30 worker nodes for staging would double the infrastructure cost but staging is only used for testing, not serving customers.

Solution:

Use smaller staging environment that still validates the application but costs much less.

THE JENKINS SCALABILITY PROBLEM

MODERN ARCHITECTURE: MICROSERVICES

Current situation:

Nowadays big companies like Twitter, Flipkart, Amazon have microservices architecture.

Scale:

To deploy these 1000s of services, Jenkins is used as the platform.

WHAT IS JENKINS

Jenkins is:

A binary you will install it on one host (laptop or EC2 instance).

Problem:

You keep adding machines to it because there are 1000s of developers in the organization and one Jenkins node cannot take all the load.

JENKINS MASTER-SLAVE ARCHITECTURE

What you do:

Step 1:

Deploy your Jenkins master on one of the EC2 instances.

Step 2:

Keep connecting multiple instances to it (worker nodes/agents/slaves).

TEAM ALLOCATION EXAMPLE

Scenario:

- 100 developers total
- 10 teams

Configuration:

Team 1:

The job that is running, run the pipeline on Node 1

Team 2:

Run the pipeline on Node 2

Team 3:

Run the pipeline on Node 3

And so on for all 10 teams.

THE PROBLEM

Issue:

If there are 10 teams, you have to create 10 Jenkins worker machines.

Scalability concern:

This might scale up if the teams increase or other requirements change.

Result:

You will have to create a lot of compute.

What is compute:

Compute is RAM, CPU, and hardware.

Consequences:

Problem 1:

The setup becomes very costly.

Problem 2:

The maintenance also becomes very huge.

What you need:

You should look for some setup which can automatically scale up and scale down.

AUTO-SCALING QUESTION

Someone might ask:

"I can integrate my Jenkins with auto-scaling groups and we can scale up and scale down automatically."

Your response:

We are talking about real-time requirements.

Real requirement:

I want to scale my Jenkins to 0 (zero). I don't even want my Jenkins master node to exist.

Jenkins limitation:

This is not possible with Jenkins. The master node must always be running.

THE WASTE PROBLEM

When waste happens:

Times like weekends:

There are 0 code changes that are made and 0 pipelines that have to be executed.

In case of microservices:

You have 20-30 Jenkins instances and 20-30 Jenkins setups.

If not configured well:

There are 100s of VMs which are created but not used.

Your requirement:

I want 0 instances when I am not making any changes.

Conclusion:

In such cases, Jenkins is not a tool you should recommend.

THE SOLUTION: LEARNING FROM KUBERNETES PROJECT

HOW KUBERNETES MANAGES CI/CD

Let's see how Kubernetes project is managing these things.

Kubernetes facts:

- 1000s of developers
- Developers in different parts of the globe
- They usually communicate using GitHub

Question:

How is Kubernetes handling CI/CD?

SCENARIO: DEVELOPER IN INDIA

Example:

Whenever a developer is making code change in India and pushing it to GitHub, how is Kubernetes dealing with the code change?

Important question:

How is it not wasting the VMs that we just explained above?

CHECKING KUBERNETES GITHUB PAGE

If we go to Kubernetes GitHub page:

We can see how Kubernetes is handling any pull request or new commits happening.

Observation:

Let's say the last pull request was made 4 hours ago and there is no new pull request or new commit made to the repo.

Requirement:

There should be 0 compute instances that Kubernetes project is wasting.

If not:

You are losing a lot of money for your organization.

KUBERNETES SOLUTION: GITHUB ACTIONS

If we see this last pull request:

What usually happens is they have configured it with GitHub Actions.

What is GitHub Actions:

GitHub Actions is another way of doing CI/CD (other than Jenkins).

Here they are running the entire CI/CD using GitHub Actions.

HOW GITHUB ACTIONS WORKS

Step 1: Code change made

Whenever a code change is made to GitHub.

Step 2: Spin up container

GitHub Actions will spin up a Kubernetes pod or a Docker container for you.

Step 3: Execute in container

Everything gets executed on a Docker container.

Step 4: Container terminates

Once execution completes, container is deleted.

Step 5: Shared resources

If you are not using it, the server or the worker which was used to run this Docker container will be used by some other project.

This is shared resources model.

KUBERNETES ORGANIZATION ON GITHUB

If we go to Kubernetes organization on GitHub:

There are many repositories which you can see (77+ repositories).

GITHUB-HOSTED RUNNERS (OPTION 1)

Setup:

If you are using GitHub Actions and you are using runner or worker node from GitHub itself (Microsoft).

What Microsoft does:

Microsoft will create containers for you or Kubernetes pods for you on the servers itself.

Where servers are:

You don't know where these servers are getting created.

Cost:

If it is an open source or public project, you will get it for free.

How it works:

Developer commits to any repository:

1. GitHub Actions triggers
2. Microsoft creates container on their servers
3. CI/CD runs in container
4. Container deleted after completion
5. No cost if public/open source project

SELF-HOSTED RUNNERS (OPTION 2)

Scenario:

You don't want to use Microsoft and your code is secure (private/confidential).

What you can do:

For all of these repositories (77 repos), you can create one common server.

Where to host:

You can host this server on any cloud platform (AWS, Azure, GCP).

HOW SELF-HOSTED RUNNER WORKS

Setup:

Create one Kubernetes cluster on your cloud platform.

Process:

Step 1: Developer makes changes

Whenever a developer is making changes in any of these 77 repos.

Step 2: Create pod

You can create a Kubernetes pod on this Kubernetes cluster.

Step 3: Execute CI/CD

Pipeline runs inside the pod.

Step 4: Delete pod

Once the execution is done, delete these pods.

Step 5: Cluster free again

Cluster will be free one more time.

Step 6: Reuse for other project

Any other project can use that Kubernetes cluster.

RESOURCE OPTIMIZATION

What we have achieved:

Instead of:

- Wasting resources for each and every project
- Creating Jenkins instances for each and every project

We created:

One common compute infrastructure (Kubernetes cluster) which can be used across:

- Multiple people in your organization
- Multiple projects in your organization

Result:

You can save your resources or save your compute.

Key benefit:

Since it is shared across all projects, we are literally using 0 compute instances when there are no code changes.

This is about modern-day CI/CD setup.

JENKINS vs KUBERNETES: SCALABILITY COMPARISON

JENKINS APPROACH

Adding more capacity:

What you do:

If we are using Jenkins and we want to add one more server, we keep adding one more worker node or you make an additional worker node and attach them to your existing Jenkins setup.

Process:

1. Create new EC2 instance
2. Install Jenkins agent
3. Configure connection to master
4. Add to Jenkins
5. Assign jobs to node

Limitations:

- Manual process
- Takes time
- Always running (costs money)
- Cannot scale to zero
- Fixed capacity

KUBERNETES APPROACH

Adding more capacity:

What you can do:

In terms of Kubernetes, you can easily increase your worker nodes and scale it up to 100 or 1000 or whatever you like.

Example with managed Kubernetes (EKS):

If it is a managed Kubernetes cluster on EKS, you can directly:

- Add nodes
- Scale up nodes
- Scale down nodes
- Even scale to zero (with proper configuration)

Commands for EKS:

```
aws eks update-nodegroup-config \

  --cluster-name my-cluster \
  --nodegroup-name my-nodes \
  --scaling-config minSize=0,maxSize=100,desiredSize=10
```

Process:

1. Update desired node count
2. EKS creates/removes nodes automatically
3. Nodes join/leave cluster automatically
4. Start accepting workloads immediately

Benefits:

- Automatic scaling
- Fast scaling (minutes)
- Can scale to zero
- Unlimited capacity
- Cost-effective

DETAILED COMPARISON

JENKINS SETUP

Architecture:

```
Jenkins Master (EC2 instance - always running)
```

```
  |
  +--- Worker Node 1 (Team 1) - always running
  +--- Worker Node 2 (Team 2) - always running
  +--- Worker Node 3 (Team 3) - always running
  +--- Worker Node 4 (Team 4) - always running
  +--- Worker Node 5 (Team 5) - always running
  +--- Worker Node 6 (Team 6) - always running
  +--- Worker Node 7 (Team 7) - always running
  +--- Worker Node 8 (Team 8) - always running
  +--- Worker Node 9 (Team 9) - always running
  +--- Worker Node 10 (Team 10) - always running
```

Resource usage:

- 11 EC2 instances always running (1 master + 10 workers)
- Each instance: t2.medium (2 CPU, 4GB RAM)
- Total: 22 CPUs, 44GB RAM
- Running 24/7

Cost calculation:

- 11 instances × $30/month = $330/month
- Weekend usage: 0% (but still paying full cost)
- Waste: Significant

Problems:

- Cannot scale to zero
- Master node must run always
- Worker nodes stay running even if idle

- High fixed cost
- Manual scaling

GITHUB ACTIONS + KUBERNETES SETUP

Architecture:

```
Kubernetes Cluster (shared)
  |
  +--- Pod 1 (created when Project 1 commits)
  +--- Pod 2 (created when Project 2 commits)
  +--- Pod 3 (created when Project 3 commits)
  ...
  +--- Pod N (created when Project N commits)

When no commits: 0 pods running
```

Resource usage:

- Pods created only when needed
- Each pod gets resources as required
- Pods deleted after job completes
- Cluster can scale down to minimal nodes (or even 0)

Cost calculation:

During work hours (Monday-Friday, 9 AM - 6 PM):

- Active development happening
- Maybe 5 nodes running
- Cost: $150/month (for work hours only)

During nights and weekends:

- No active development
- Cluster scales down to 1 node (or 0 with Karpenter)
- Cost: $10/month (minimal)

Average cost: $80-100/month

Savings: 70% compared to Jenkins

Benefits:

- Scales to zero possible
- Pay only for usage
- Automatic scaling
- Shared resources
- No waste

WEEKEND SCENARIO COMPARISON

Saturday and Sunday (48 hours):

Jenkins setup:

- Master node: Running

- 10 worker nodes: Running

- Code changes: 0

- Pipelines executed: 0

- CPU usage: 0%

- Cost for weekend: $20 (wasted)

- Resources utilized: 0%

GitHub Actions + Kubernetes setup:

- Kubernetes cluster: Scaled to 0-1 nodes

- Pods running: 0

- Code changes: 0

- Pipelines executed: 0

- Cost for weekend: $0-2

- Resources utilized: 0% (but not paying for idle)

Difference: $18-20 saved every weekend

DETAILED GITHUB ACTIONS WORKFLOW

UNDERSTANDING GITHUB ACTIONS

What is GitHub Actions:

Built-in CI/CD service provided by GitHub (owned by Microsoft).

Where it runs:

- On GitHub's infrastructure (Microsoft's servers)

- Or on your own self-hosted runners

GITHUB-HOSTED RUNNERS

What they are:

Virtual machines provided by GitHub/Microsoft for running CI/CD pipelines.

Available runners:

- Ubuntu Linux

- Windows Server

- macOS

How it works:

Step 1: Developer pushes code to GitHub

```
git push origin main
```

Step 2: GitHub detects push GitHub sees new commit on main branch.

Step 3: GitHub reads workflow file File location: .github/workflows/ci.yml in your repository

Example workflow file:

yaml

name: CI Pipeline

```yaml
on:
  push:
    branches: [ main ]

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

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

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.10'

      - name: Install dependencies
        run: pip install -r requirements.txt

      - name: Run unit tests
        run: pytest tests/

      - name: Run security scan
        run: bandit -r .
```

Step 4: GitHub creates runner (container/VM)

- Fresh Ubuntu virtual machine created
- Clean environment every time
- All tools pre-installed

Step 5: Execute workflow steps

Each step runs in order:

1. Checkout code from repository
2. Set up Python 3.10
3. Install dependencies
4. Run unit tests
5. Run security scan

Step 6: Job completes

All steps finished.

Step 7: Runner terminates

Virtual machine or container is deleted.

Step 8: Resources freed

Those resources can now be used by another project.

Time taken: Maybe 5 minutes

Cost:

- Public repo: Free
- Private repo: 2000 free minutes/month, then pay per minute

Resource usage:

5 minutes of compute, then zero.

SELF-HOSTED RUNNERS WITH KUBERNETES

For secure/private code where you don't want to use GitHub's servers:

SETUP

Step 1: Create Kubernetes cluster

Create one common Kubernetes cluster on any cloud platform:

- AWS EKS
- Azure AKS
- GCP GKE
- On-premises Kubernetes

Example:

Create EKS cluster with 10 worker nodes.

Step 2: Install GitHub Actions Runner Controller

This is a Kubernetes operator that manages GitHub Actions runners.

Install command:

```
```

kubectl apply -f actions-runner-controller.yaml

Step 3: Configure repositories Connect your 77 repositories to use this self-hosted runner.

In each repository's workflow file:

yaml

```
jobs:
  build:
    runs-on: self-hosted
```

HOW IT WORKS

Scenario: 77 repositories using one Kubernetes cluster

Repository 1: Developer commits code

Step 1: Code pushed to GitHub Developer pushes code to Repository 1.

Step 2: GitHub Actions triggers GitHub detects commit.

Step 3: Request runner GitHub Actions requests a runner from your Kubernetes cluster.

Step 4: Create pod Kubernetes creates a pod on your cluster.

Step 5: Execute pipeline Pod runs all CI/CD steps:

- Build
- Test
- Scan
- Deploy

Step 6: Job completes All steps finished successfully.

Step 7: Pod deleted Kubernetes deletes the pod.

Step 8: Resources freed Those resources (CPU, RAM) are now available.

Time: 10 minutes

Resources used: 1 pod for 10 minutes

Repository 2: Another developer commits

Step 1: Code pushed to GitHub Different developer pushes code to Repository 2.

Step 2: Create pod Kubernetes creates another pod on same cluster.

Step 3: Execute pipeline Runs CI/CD for Repository 2.

Step 4: Pod deleted After completion, pod is removed.

This can happen simultaneously:

- Repository 1 pod running
- Repository 2 pod running
- Repository 3 pod running
- All on same Kubernetes cluster
- All using shared resources

NO COMMITS SCENARIO

When no one is committing code:

Current state:

- All 77 repositories: No new commits
- Kubernetes cluster: No pods running
- Nodes: Can scale down to minimum (or zero)

Resource usage:

- 0 active pods
- Maybe 1-2 nodes running (or 0 with advanced setup)
- Minimal cost

Weekend scenario:

- Saturday and Sunday: No code changes
- Cluster scaled to 0-1 nodes
- Cost: $0-10 for entire weekend
- Massive savings compared to Jenkins

SHARED RESOURCES BENEFIT

What we have done:

Instead of creating separate Jenkins instances for each project:

- Project 1: Jenkins instance 1
- Project 2: Jenkins instance 2
- Project 3: Jenkins instance 3
- ... 77 projects = 77 Jenkins instances

We created: One common Kubernetes cluster shared across all 77 projects.

How sharing works:

Time 10:00 AM:

- Project 5 commits code
- Pod created
- Uses 2 CPU, 4GB RAM
- Runs for 8 minutes
- Pod deleted

Time 10:05 AM:

- Project 12 commits code
- Pod created on SAME cluster
- Uses 1 CPU, 2GB RAM
- Runs for 5 minutes
- Pod deleted

Time 10:08 AM:

- Project 23 commits code
- Pod created on SAME cluster
- Uses 4 CPU, 8GB RAM
- Runs for 15 minutes
- Pod deleted

Same cluster, different projects, maximum efficiency.

Result:

- Save resources
- Save compute
- Save money

Since it is shared across all projects: We are literally using 0 compute instances when there are no code changes.

This is modern-day CI/CD setup.

SCALING COMPARISON

JENKINS SCALING

Adding more capacity:

What you usually do: Keep adding one more worker node or you make an additional worker node and attach them to your existing Jenkins setup.

Process:

1. Provision new EC2 instance
2. SSH into instance
3. Install Java
4. Install Jenkins agent
5. Configure connection to master
6. Test connection
7. Add node to Jenkins UI
8. Configure node properties
9. Assign labels
10. Start using

Time required: 30-60 minutes per node

Scaling down:

1. Stop assigning jobs to node
2. Wait for running jobs to complete

3. Disconnect node from master
4. Terminate EC2 instance

Manual process, takes time.

KUBERNETES SCALING

Adding more capacity:

What you can do: In terms of Kubernetes, you can easily increase your worker nodes and scale it up to 100 or 1000 or whatever you like.

Example with managed Kubernetes (EKS): If it is a managed Kubernetes cluster on EKS, you can directly add nodes, scale up or scale down these nodes.

Process:

1. Update node group desired capacity
2. EKS automatically provisions nodes
3. Nodes join cluster automatically
4. Ready to use

Time required: 2-5 minutes

Scaling down:

1. Reduce desired capacity
2. EKS automatically drains and terminates nodes
3. Workloads moved to remaining nodes

Automatic process, very fast.

AUTO-SCALING IN KUBERNETES

Even better with Kubernetes Cluster Autoscaler or Karpenter:

How it works:

1. Pods waiting to be scheduled
2. Not enough resources on existing nodes
3. Cluster Autoscaler detects this
4. Automatically adds more nodes

5. Pods get scheduled
6. When pods complete and delete
7. Nodes become idle
8. Cluster Autoscaler removes idle nodes

Completely automatic, no human intervention.

EXAMPLE SCALING SCENARIO

Morning 9 AM (work starts):

- Developers start committing code
- Multiple pipelines triggered
- Kubernetes cluster: 2 nodes
- More pods need to run
- Cluster auto-scales to 10 nodes
- All pipelines run simultaneously

Afternoon 2 PM (lunch time):

- Fewer commits
- Fewer pipelines
- Some nodes idle
- Cluster scales down to 5 nodes

Evening 6 PM (work ends):

- No more commits
- Last pipelines finishing
- Cluster scales down to 1 node

Night 10 PM:

- No activity
- Cluster scales to 0 nodes (with Karpenter)
- Zero cost

Next morning 9 AM:

- First commit arrives
- Cluster scales up to 2 nodes
- Cycle repeats

Completely automatic, completely efficient.

COST COMPARISON EXAMPLE

Scenario: 10 teams, 50 developers, microservices architecture

---

JENKINS SETUP COSTS

Infrastructure needed:

- 1 Jenkins master: t2.large (2 CPU, 8GB) - $70/month
- 10 worker nodes: t2.medium (2 CPU, 4GB each) - $300/month

Total: $370/month

Running time: 24/7 (720 hours/month)

Actual usage: Maybe 8 hours/day on weekdays = 160 hours/month

Utilization: 160/720 = 22%

Waste: 78% of resources

Wasted money: $288/month

GITHUB ACTIONS + KUBERNETES SETUP COSTS

Infrastructure needed:

- Kubernetes cluster: Scales 0-10 nodes based on demand
- Node type: t2.medium (2 CPU, 4GB)

Usage pattern:

Weekdays (9 AM - 6 PM): 5 nodes average Weekday nights: 1 node Weekends: 0-1 nodes

Average nodes: 3 nodes

Cost: 3 nodes × $30/month = $90/month

GitHub Actions (private repos): ~$50/month for compute minutes

Total: $140/month

Savings: $370 - $140 = $230/month (62% savings)

Plus:

- Better resource utilization
- Faster scaling
- More flexibility

SUMMARY

CI/CD DEFINITION

CI (Continuous Integration): Integrate tools and processes before delivering application

CD (Continuous Delivery): Deploy application to platform accessible by customer

---

STANDARD CI/CD STEPS

1. Unit Testing: Test individual functions
2. Static Code Analysis: Check code quality without running
3. Code Quality/Vulnerability Testing: Security and quality checks
4. Automation Testing: End-to-end testing
5. Reporting: Generate logs and reports
6. Deployment: Deploy to customer-facing environment

ENVIRONMENTS

Development:

- Minimal resources
- Simple setup (1 EC2 or 1-node Kubernetes)
- First testing

Staging:

- Medium resources

- Production-like setup (smaller scale)
- Final testing before production
- Example: 3 masters, 5 worker nodes

Production:

- Full resources
- Customer-facing
- High availability required
- Example: 3 masters, 30 worker nodes

Why not same as production in staging: Very costly setup

JENKINS LIMITATIONS

Problems:

- Cannot scale to zero
- Master must always run
- Manual worker node management
- High fixed costs
- Resource waste during idle times
- 10 teams = 10 worker machines minimum

Cost issue:

- Weekends: 0 code changes, full cost
- Nights: Low activity, full cost
- Utilization: 20-30% typically

MODERN CI/CD SOLUTION

GitHub Actions + Kubernetes:

How it works:

1. Code committed to GitHub
2. GitHub Actions triggers

3. Kubernetes pod created
4. CI/CD runs in pod
5. Pod deleted after completion
6. Resources freed for other projects

Benefits:

- Shared resources across all projects
- Scale to zero possible
- Pay only for usage
- Automatic scaling
- No waste

Options:

GitHub-hosted runners:

- Microsoft provides servers
- Free for public repos
- You don't manage infrastructure

Self-hosted runners:

- Create one Kubernetes cluster
- Share across all 77 repositories
- You control infrastructure
- Secure for private code

Resource efficiency:

- No commits = 0 pods = minimal cost
- Active development = pods created dynamically = pay for usage only
- Literally using 0 compute when no code changes

SCALING COMPARISON

Jenkins:

- Add worker: Manual, 30-60 minutes

- Remove worker: Manual, wait for jobs to finish
- Cannot scale to zero
- Fixed capacity

Kubernetes:

- Add nodes: Automatic, 2-5 minutes
- Remove nodes: Automatic, graceful drain
- Can scale to zero
- Unlimited capacity

EKS scaling: Directly add nodes, scale up or down with simple API calls or auto-scaling policies.

This is the evolution from traditional Jenkins-based CI/CD to modern container-based CI/CD with better efficiency, lower costs, and automatic scaling.