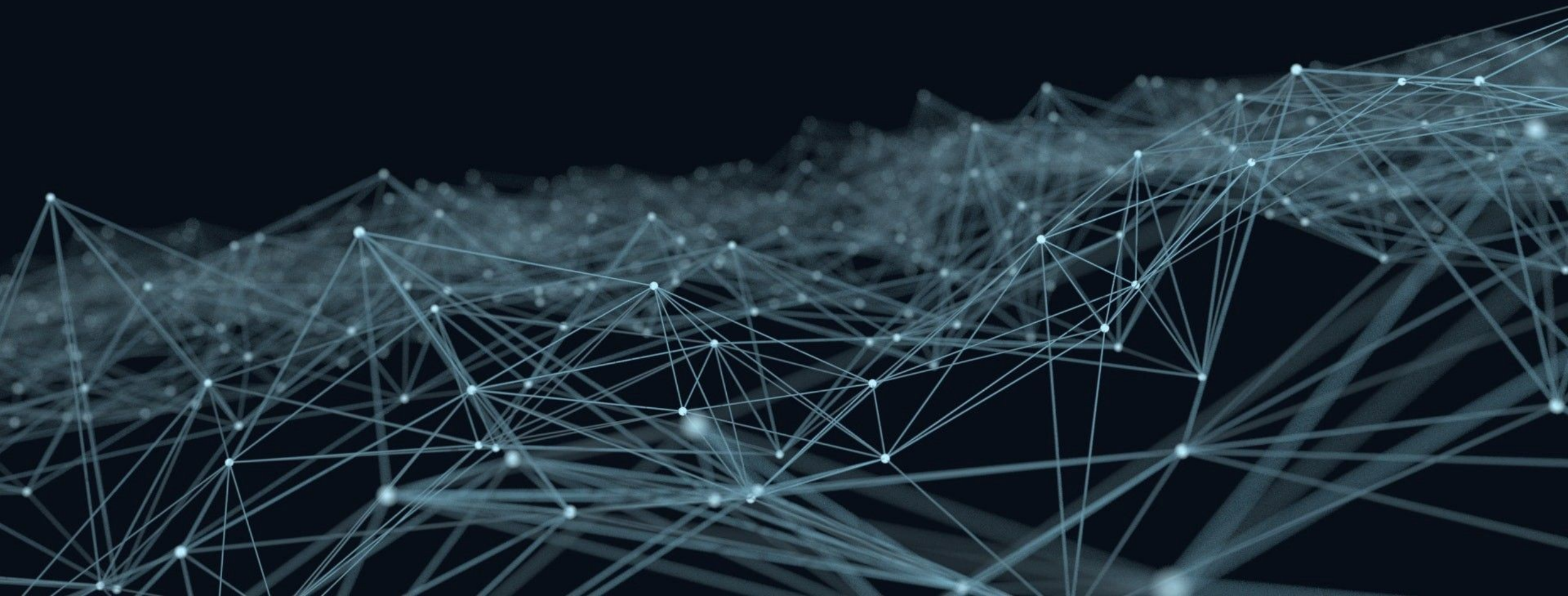
Introduction to CI/CD

Module 5: Jenkins



Origins

- In the early 2000s, most software teams
  - Integrated code infrequently
  - Ran builds manually
  - Discovered integration problems late in the cycle
- In 2004 by Kohsuke Kawaguchi at Sun Microsystems created the Hudson project
  - Open-source written in Java
  - Designed to automate builds and tests
  - Integrated well with Java build tools like Ant and Maven
  - Easy to set up and xxtensible
  - Focused on automation rather than process enforcement

Hudson to Jenkins (2011)

- In 2010–2011, Oracle acquired Sun Microsystems
  - This led to disagreements between the Hudson community and Oracle regarding:
    - *Project governance*
    - *Trademark ownership*
    - *Contribution control*
- As a result
  - The community forked Hudson
  - The new project was named Jenkins
  - Most contributors and users moved to Jenkins
- From this point on
  - Jenkins became the community-driven project
  - Hudson gradually declined in relevance

Job-Centric CI (2011–2014)

- Early versions of Jenkins focused on:
  - Freestyle jobs
  - UI-driven configuration
  - Scripted build steps
- Characteristics of early Jenkins
  - Configuration stored on the server
  - Pipelines defined through the web UI
  - Heavy reliance on plugins
  - Less reproducible builds
- This reflected the state of CI at the time
  - Automation existed, but IaC was not common

Job-Centric CI (2011–2014)

- Early versions of Jenkins focused on:
  - Freestyle jobs
  - UI-driven configuration
  - Scripted build steps
- Characteristics of early Jenkins
  - Configuration stored on the server
  - Pipelines defined through the web UI
  - Heavy reliance on plugins
  - Less reproducible builds
- This reflected the state of CI at the time
  - Automation existed, but IaC was not common

Pipeline as Code (2014–2016)

- As DevOps practices matured, teams demanded
  - Version-controlled build definitions
  - Reproducibility
  - Better support for complex workflows
- This led to the introduction of the Jenkins Pipeline and Jenkinsfile
- Innovations
  - Pipelines defined in code
  - Stored in Git repositories
  - Reviewed like application code
  - Support for stages, parallelism, and conditions
- This marked a major evolution in Jenkins’ design philosophy

DevOps Era (2016–Present)

- Jenkins evolved to support
  - Microservices
  - Containers and Docker
  - Kubernetes-based build agents
  - Cloud-native workflows
- Notable Trends
  - Declarative pipelines simplified syntax
  - Dynamic agents replaced static build servers
  - Integration with modern SCM platforms (GitHub, GitLab)
- Jenkins shifted from being “a CI tool” to a general-purpose automation engine

Jenkins Today

- Why Jenkins is still relevant
  - Extreme flexibility
  - Large plugin ecosystem
  - Strong support for legacy and complex systems
- Challenges
  - Requires operational effort
  - Plugin compatibility management
  - Competition from managed CI/CD platforms
- Despite newer tools, Jenkins remains widely used in:
  - Enterprises
  - Regulated environments
  - Complex, heterogeneous systems

GitLab and GitHub

- GitHub Actions and GitLab CI/CD are modern alternatives to Jenkin' pipelines
  - These are integrated pipeline platforms built directly into their Git hosting services
- These repository tools treat CI/CD as a native feature of the repository
  - Jenkins = external automation engine
  - GitHub/GitLab = pipelines embedded in the SCM platform
- The same basic pipeline concepts apply to these the same as Jenkins
  - The syntax and format of the configuration files
- These turn GitHub and GitLab into orchestration engines

GitHub Actions

- GitHub Actions is GitHub’s built-in automation and CI/CD platform
  - Pipelines are defined using YAML workflow files stored in the repository.
- Key characteristics
  - Tight integration with GitHub repositories
  - Event-driven (push, pull request, tag, issue events)
  - Pipelines live in .github/workflows
  - Uses hosted runners or self-hosted runners
- Conceptual flow
  - GitHub event → Action workflow → Jobs → Steps
- Strengths
  - Zero infrastructure to manage
  - Simple setup for GitHub users
  - Strong marketplace of reusable actions

GitLab CI/CD

- GitLab CI/CD is a first-class feature of GitLab, not an add-on.
  - Pipelines are defined in a single file: “.gitlab-ci.yml”
- Key characteristics
  - CI/CD tightly integrated with GitLab repos
  - Built-in artifact storage and environment tracking
  - Native support for Docker and Kubernetes
  - GitLab Runners execute jobs
- Conceptual flow
  - Git commit → GitLab pipeline → Stages → Jobs
- Strengths
  - Very cohesive developer experience
  - Strong DevOps lifecycle integration
  - Minimal configuration overhead

Comparison

- Configuration style contrast
- Jenkins
  - Highly flexible
  - Scriptable pipelines (Groovy)
  - Powerful but verbose
  - Requires plugin management
- GitHub / GitLab
  - Opinionated YAML syntax
  - Simpler mental model
  - Less customization, faster onboarding
  - Platform constraints apply

Comparison

- Jenkins is often chosen when
  - You need extreme customization
  - You support many SCM systems
  - You have complex legacy workflows
  - You control infrastructure
- GitHub / GitLab pipelines are often chosen when
  - Repositories are already hosted on the platform
  - Teams want fast setup
  - Minimal operational overhead is desired
  - Standard CI/CD patterns are sufficient

Jenkins Distributed Architecture

- The main node is the default node
  - Runs the Jenkins instance
  - Manages access to Jenkins Interface
  - Should be the only node accessible via web interface
  - The master/main node is the controller
  - Best practice is that CICD jobs are delegated to other nodes
- Agent/Slave nodes
  - Tasked with running parts of the CICD jobs
  - Agents are tagged so they can be used selectively
  - Jenkins does not have to be installed on an agent
  - Just Java to run the Jenkins agent jar file
  - Communication takes place via ssh

Jenkins Pipelines

- A pipeline defines a CICD process for that project
- Stages
  - Sequential series of steps to be executed, for example
  - Build > Test > Package > Deploy
- JenkinsFile (infrastructure as code)
  - A script that defines the stages of a pipeline
  - Older form is written in the Groovy scripting language
  - New form is a declarative form of scripting
- Three basic configurations
  - The JenkinsFile is kept in Jenkins
  - The JenkinsFile is kept in a SCM repository
  - The JenkinsFile is kept in the project itself
    - *This allows for multi branch builds*

Post Build Steps

- In addition to stages, Jenkins has a post build stage
- Contains any of a number of clause types
  - Always – always executes
  - Failure – executes only when the pipeline fails
  - Success – executes only when the build succeeds
  - Cleanup – always runs after all the other clauses run
  - Changed – runs if the pipeline or a stage completion status is different from a previous run
- There are more clause types at:
  - https://jenkins.io/doc/book/pipeline/syntax/#post

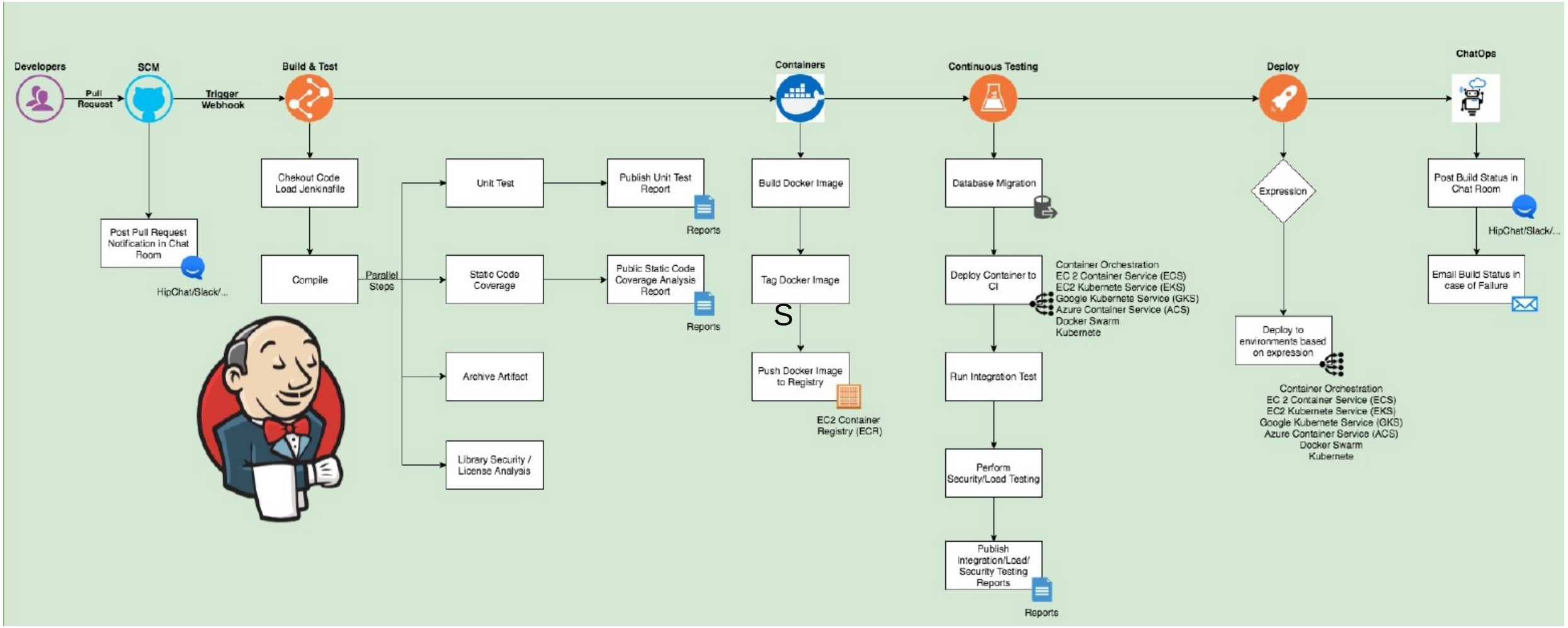
Environment Variables

- There are a number of predefined environment variables
  - Accessed via the global variable “env”
  - Value is accessed using Groovy syntax
    - *“Build ID is ${BUILD_ID}”*
  - We can define environment variables in either the whole pipeline or a given stage using the environment block

Pipeline Features

- Can support complex, real-world, CD Pipeline requirements
  - Pipelines can fork/join, loop, parallel, and more
- Resilient:
  - Pipeline executions can survive master restarts
- Can be paused:
  - Pipelines can pause and wait for human input/approval.
- Pipeline configuration is treated as as code in source control

Pipeline


Developers
SCM
Pull Request
Trigger Webhook
Build & Test
Containers
Continuous Testing
Deploy
ChatOps
Post Pull Request Notification in Chat Room
HipChat/Slack/...
Chekout Code Load Jenkinsfile
Compile
Parallel Steps
Unit Test
Static Code Coverage
Archive Artifact
Library Security / License Analysis
Publish Unit Test Report
Reports
Public Static Code Coverage Analysis Report
Reports
Build Docker Image
Tag Docker Image
Push Docker Image to Registry
EC2 Container Registry (ECR)
Database Migration
Deploy Container to CI
Container Orchestration
EC 2 Container Service (ECS)
EC2 Kubernete Service (EKS)
Google Kubernete Service (GKS)
Azure Container Service (ACS)
Docker Swarm
Kubernete
Run Integration Test
Perform Security/Load Testing
Publish Integration/Load/ Security Testing Reports
Reports
Expression
Deploy to environments based on expression
Container Orchestration
EC 2 Container Service (ECS)
EC2 Kubernete Service (EKS)
Google Kubernete Service (GKS)
Azure Container Service (ACS)
Docker Swarm
Kubernete
Post Build Status in Chat Room
HipChat/Slack/...
Email Build Status in case of Failure

Pipeline Stage View

billing-rest - Stage View

	Declarative: Checkout SCM	Initialize	Checkout	Build	Publish Reports	SonarQube analysis	ArchiveArtifact	Docker Tag & Push	Deploy - CI	Deploy - QA	Deploy - UAT	Deploy - Production	Declarative: Post Actions
Average stage times: (Average full run time: ~2min 59s)	768ms	1s	799ms	57s	5s	14s	125ms	23s	37ms	32ms	32ms	32ms	792ms
#118 Jul 17 20:58 — 1 commit	817ms	1s	690ms	1min 36s	10s	24s	198ms	38s	38ms				1s
#117 Jul 16 15:28 — 1 commit	792ms	1s	708ms	1min 36s	8s	23s	179ms	38s	42ms				2s
#116 Jul 15 21:13 — No Changes	672ms	869ms	694ms	1min 33s	10s	24s	183ms	37s	40ms				75ms

Pipeline Features

- Can support complex, real-world, CD Pipeline requirements
  - Pipelines can fork/join, loop, parallel, and more
- Resilient:
  - Pipeline executions can survive master restarts
- Can be paused:
  - Pipelines can pause and wait for human input/approval.
- Pipeline configuration is treated as as code in source control

Questions