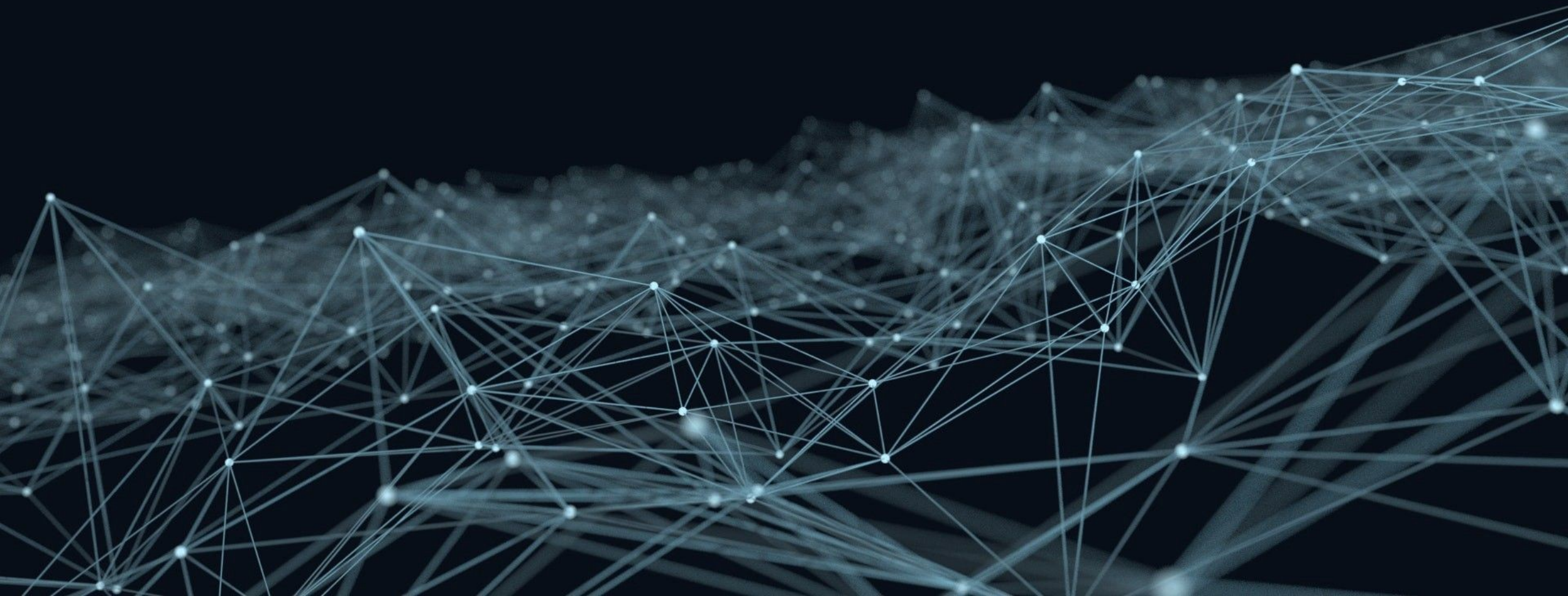
Introduction to Jenkins

Module 5: Shared Libraries



Topics

- Why Shared Libraries matter
- Library structure and best practices
- Global variables and helper utilities
- Versioning and dependency strategy

Shared Libraries

- As Jenkins usage increases in an organization
  - Pipelines become long and repetitive
  - Teams copy and paste logic
  - The same fixes and updates have to be applied in multiple locations
  - Coding and development standards drift over time
  - Security practices become inconsistent
- This leads to
  - Maintenance pain from a fragmented pipeline code base
  - Increased risk of errors in pipeline code
  - Hard-to-govern pipelines since they don’t follow standards
  - Increasing number of “snowflake” pipelines that are idiosyncratic

Shared Libraries

- Shared libraries support
  - Centralized reusable pipeline logic
  - Best practices for coding and code quality
  - Reduced duplication of code in pipelines
  - Improved security and maintainability
- Logic is encapsulated in library calls
  - Explicit execution logic in a step
    - *sh 'mvn clean verify'*
  - Replaced by execution logic encapsulated in library calls
    - *RunMavenBuild()*
  - Pipelines describe intent; shared libraries implement behaviors

Library Structure

- A Jenkins shared library is a Git repository
  - It has a standard well-defined layout
  - Not all of the directories are required
  - This structure is considered a standard best practice
- vars/
  - Global pipeline steps
  - Contains shared steps exposed directly to pipelines
  - One file = one callable step
  - File name becomes the step name
  - Best practice
    - *Keep logic here simple*
    - *Delegate complexity to src/*

```
(shared-library-repo)
├── vars/
├── src/
├── resources/
└── README.md
```

```
runTests()
deployToEnv('prod')
```

Library Structure

- src/
  - Helper utilities and classes
  - Contains Groovy classes
  - Used internally by shared steps
  - Not exposed directly to pipeline code
- Used for
  - Validation logic
  - Parsing
  - Reusable utilities
  - Complex workflows
- Analogy
  - vars/ is the API
  - src/ is the implementation

```
shared-library/
├── vars/
│   └── runMavenBuild.groovy
└── src/
    └── com/example/build/
        └── MavenHelper.groovy
```

```groovy
package com.example.build

class MavenHelper {

    static void runBuild(script, String goals) {
        script.echo "Running Maven goals: ${goals}"
        script.sh "mvn ${goals}"
    }

    static void validateGoals(String goals) {
        if (!goals) {
            throw new IllegalArgumentException("Maven goals must not be empty")
        }
    }
}
```

Library Structure

- resources/
  - Templates and other assets
  - Stores non-code artifacts
    - *YAML, JSON, XML, shell scripts, templates*
  - Loaded at runtime
- Common uses
  - Kubernetes manifests
  - Deployment templates
  - Configuration files

```
shared-library/
├── resources/
│   ├── docker/
│   │   └── Dockerfile.template
│   ├── kubernetes/
│   │   ├── deployment.yaml
│   │   └── service.yaml
│   ├── scripts/
│   │   └── deploy.sh
│   ├── config/
│   │   └── sonar-project.properties
│   └── README-template.md
```

Library Structure

- How resources are loaded
  - The libraryResource step loads files from resources/
  - In the example, the configuration files needed are created from the corresponding resource
  - Avoids hardcoding files or strings
- Best practices
  - Avoid putting complex logic directly in pipelines
  - Don’t expose too many global steps
  - Don’t duplicate logic across libraries
  - Don’t let different teams modify shared libraries directly

```
def deploymentYaml = libraryResource('kubernetes/deployment.yaml')

writeFile file: 'deployment.yaml', text: deploymentYaml
sh 'kubectl apply -f deployment.yaml'
```

```
def script = libraryResource('scripts/deploy.sh')

writeFile file: 'deploy.sh', text: script
sh 'chmod +x deploy.sh && ./deploy.sh'
```

Versioning Strategies

- Shared libraries should be
  - Versioned
  - Predictable
  - Reproducible
- Without versioning
  - Library changes can break pipelines
  - Debugging becomes difficult
  - Rollbacks are painful

Versioning Strategy

- Common approaches
  - Git tags (v1.0.0)
  - Branches (main, stable)
  - Semantic versioning
- Pipelines can
  - Lock to a version
  - Upgrade intentionally
  - Test changes safely

Managing Library Dependencies

- Best practices
  - Minimize library dependencies
  - Avoid circular dependencies
  - Keep libraries focused on a single area of functionality
  - Prefer one “core” shared library with additional specialized libraries
  - Shared libraries should evolve slowly and carefully
- Governance and control in enterprise setups
  - Library changes require review for potential risks and impact on pipelines
  - CI tests validate libraries to ensure they don’t regress
  - Releases are documented
  - Breaking changes are communicated to ensure dependencies can be modified
  - Shared libraries become part of the enterprise Jenkins environment

Questions