Implement Global Trusted Pipeline Library

📌 Summary

Implemented Global 'Trusted' Jenkins Shared Library to improve structure, testability, and maintainability of pipeline logic.

❗ Problem Background

Before this implementation, there was **no centralized Jenkins shared library**, and each pipeline was built independently with similar logic hardcoded in multiple places. This led to several architectural and operational issues:

- 🔁 **High Code Duplication**: Core stages (e.g., `Prepare Workspace`, `Lint`, `Test`, `Build`) were manually defined across multiple pipelines.
- ⚠️ **Tight Coupling & Low Abstraction**: Business logic was directly tied to shell commands and specific implementations, reducing testability.
- 📉 **Poor Maintainability**: Fixes, enhancements, or configuration changes had to be applied in every individual pipeline, increasing maintenance overhead and the risk of inconsistency.
- 🚫 **Lack of Reusability**: Pipelines could not easily share or extend common functionality, making it difficult to scale Jenkins usage across projects or teams.
- 🧱 **Violation of Architectural Principles**: Patterns like SRP (Single Responsibility Principle), DRY (Don't Repeat Yourself), and separation of concerns were routinely broken.

---

1. Problem Examples and Architectural Issues

🔍 Stage Level: Repeated Stage Logic Across Pipelines

🔹 **Issue**: Identical stage structures were repeated across multiple pipelines.

🔹 The stages **bolded and Strikethroughed in the table below** represent logic that was highly duplicated — the same implementation was copy-pasted across different pipelines.

PipelineForJenkins	DLX-Pull Request	DLX-Deployment	JS-Pull Request	JS-Deployment
~~**Load Shared Library**~~	~~**Load Shared Library**~~	~~**Load Shared Library**~~	~~**Load Shared Library**~~	~~**Load Shared Library**~~

~~Prepare Workspace~~	~~Prepare WORKSPACE~~	~~Delete Merged Branch~~	~~Prepare WORKSPACE~~	~~Delete Merged Branch~~
Lint Groovy Code	~~Linting~~	~~Prepare WORKSPACE~~	~~Install Dependencies~~	~~Prepare WORKSPACE~~
Generate Groovydoc	~~EditMode Tests~~	~~Linting~~	~~Linting~~	~~Install Dependencies~~
Run Unit Tests	~~PlayMode Tests in Editor~~	~~EditMode Tests~~	~~Unit Testing~~	~~Linting~~
Publish Test Results	Code Coverage Send Reports	~~PlayMode Tests In Editor~~	~~Static Analysis~~	~~Unit Testing~~
Static Analysis	~~Build Project~~	~~Build Project~~		~~Static Analysis~~
		Deploy Build		Check Build and Deploy Condition
				Server Build and Deploy
				Client Build and Deploy

🔍 **Step Level - example (1):** `mergeBranchIfNeeded()`

- Main Issue: **Tightly Coupled Controller-Service Logic with Hard-Coded Shell Dependencies**

❌ **SRP Violation (Single Responsibility Principle)**

`mergeBranchIfNeeded()` performs too many responsibilities in a single function:

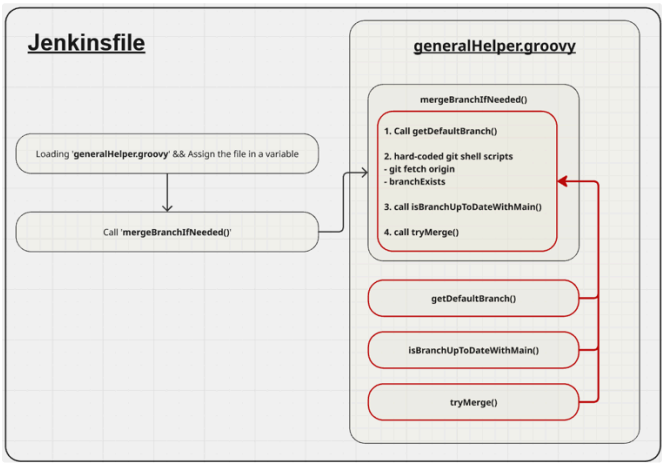
① Determining the default branch

② Checking branch status

③ Attempting a merge

④ Handling errors

→ All combined into one function, violating separation of concerns



❌ **Hard-Coded Git Commands**

Commands like `sh 'git fetch origin'`, `git show-ref`, `git merge`, and `git merge --abort`

→ Shell commands are deeply embedded in the logic, making the function hard to test and difficult to maintain

---

❌ **Tight Coupling**

`mergeBranchIfNeeded()` directly calls `getDefaultBranch()`, `isBranchUpToDateWithMain()`, and `tryMerge()`

→ Internal flow is tightly controlled within one method, increasing coupling

🔍 **Step Level - example (2)** `initializeEnvironment()`

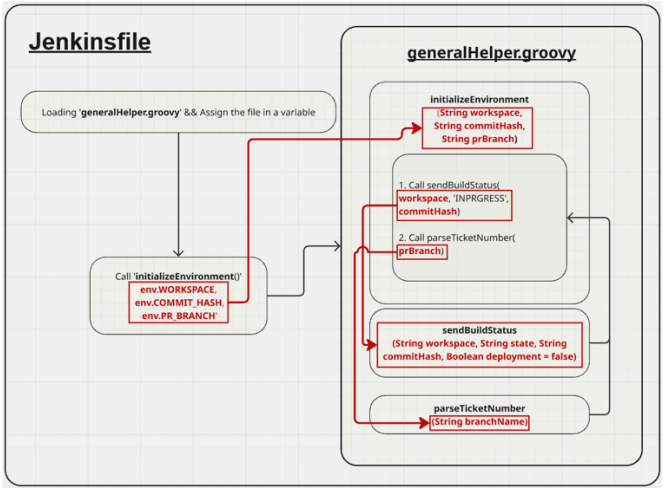
- Main Issue: **Parameter Propagation**

❌ **Parameter Propagation**

The function accepts multiple parameters (`workspace` , `commitHash` , `prBranch`) only to forward them unchanged to other functions (`sendBuildStatus()` , `parseTicketNumber()`).

→ This adds unnecessary coupling and bloats the function signature.

→ The function behaves more like a relay than an abstraction.



❌ **SRP Violation (Single Responsibility Principle)**

`initializeEnvironment()` performs multiple distinct actions in a single function:

① Sending build status to Bitbucket

② Extracting ticket number from branch

③ Setting global environment variables

→ All of these should ideally be separated into distinct responsibilities.

❌ **Hidden Side Effects**

This function sets environment variables (`env.TICKET_NUMBER` , `env.FOLDER_NAME`) implicitly, without returning any values

→ Introduces **invisible global state** that complicates debugging and testing

---

❌ **External System Coupling**

The function directly invokes `sendBuildStatus()` , which calls an external Python script

→ Tightly coupled with Bitbucket and the Python runtime

→ Difficult to replace, simulate, or unit test

🔍 **Step Level - example (3)** `cloneOrUpdateRepo()`

- Main Issue: **SRP Violation + Hard-Coded**

❌ **SRP Violation (Single Responsibility Principle)**

`cloneOrUpdateRepo()` and `checkoutBranch()` handle too many unrelated concerns:

① Input parameter validation

② Directory existence check via shell `find`

③ Git repository validation by checking `.git`

④ Git clone operation

⑤ Git fetch operation

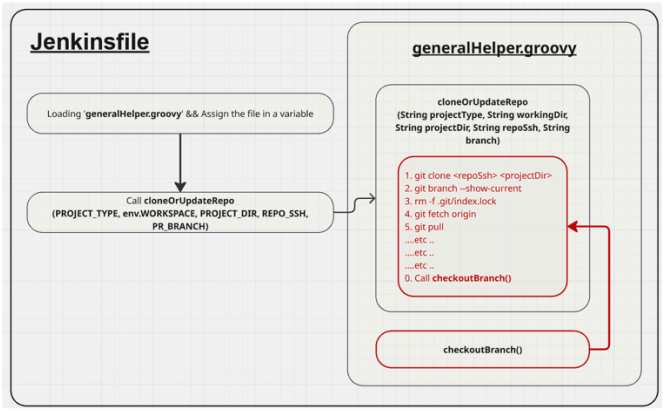
⑥ Git pull operation

⑦ Branch existence verification (local & remote)

⑧ Branch checkout

⑨ Local branch hard reset to origin

⑩ Untracked files cleanup via `git clean`

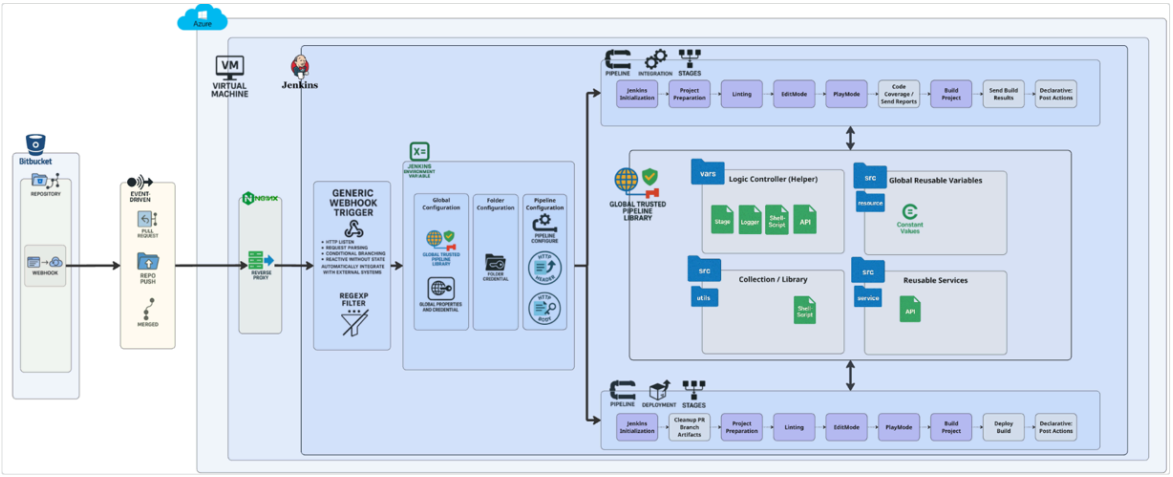


⑪ Git lock file cleanup (`.git/index.lock`)
⑫ Project directory cleanup (`rm -rf`) for recovery
⑬ Error handling and exception throwing
⑭ Logging of operation steps and statuses
→ These responsibilities span validation, filesystem access, Git interaction, recovery logic, and control flow, and should be delegated to separate components or helpers for maintainability and testability.

---

2. What's Been Updated in This PR

To address the above architectural problems and operational inefficiencies, this PR introduces a **Global Trusted Jenkins Shared Library**, which consolidates repeated logic, improves abstraction, and promotes reusability across multiple pipelines.



📐 **Overview of New Jenkins Architecture Diagram**

- The diagram below illustrates the new event-driven Jenkins architecture, powered by a Global Trusted Pipeline Library:

a. **Trigger Flow (Left to Right)**
   - Bitbucket Webhook events (e.g., PR created, pushed, or merged) trigger Jenkins jobs through a reverse proxy and the **Generic Webhook Trigger** plugin.
   - The webhook payload is parsed and filtered via RegExp, enabling conditional logic without polling.

b. **Centralized Configuration**
   - Jenkins now leverages **Global Configuration**, **Folder-level credentials**, and **Per-pipeline HTTP parameters**, reducing pipeline-local setup duplication.

c. **Global Trusted Pipeline Library**
   i. The core of the architecture is the Shared Library

d. **Pipeline Stage Reusability**
   - Pipelines (both integration and deployment) reuse the same shared stage building blocks, including `Jenkins Initialization` , `Project Preparation` , `Linting` , `Testing` , and `Build/Deploy` .
   - The separation between control (vars) and logic (src) layers promotes testability, modularity, and clean architecture.

📁 **Shared Library Structure**

The shared library is organized under the `sharedLibraries` directory and follows a layered architecture:

- `src/service/`:
  - `BitbucketApiService`: Handles HTTP request construction, token headers, error handling, and communication with Bitbucket.
    - Exceptions handled:
      - DNS failure → `UnknownHostException`
      - TCP failure → `ConnectException`, `SocketTimeoutException`
      - TLS failure → `SSLHandshakeException`
      - I/O issues → `IOException`
- `src/utils/`: Each library exposes **Closures that return standardized Groovy Maps**, describing shell commands (`script`, `label`, etc.). This enables consistent and reusable script execution via a centralized executor (e.g., `shellScriptHelper`).
  - `ShellLibrary`: Generic shell command utilities
  - `GitLibrary`: Git operations wrapped as shell commands
  - `SSHShellLibrary`: SSH-based operations
- `vars/`: exposes Groovy-based pipeline steps callable directly from Jenkinsfiles:
  - Complex or repetitive stage logic has been modularized (e.g., `stageLintUnity`, `stageProjectPrepare`).
  - Introduces `logger` for consistent output across both stage logs and console logs.
  - Helper files (e.g., `shellScriptHelper`, `bitbucketHelper`) route script execution and API interaction through central logic and error handling layers.

🚀 **Webhook Event Configuration Update**

Previous configuration:	What Changed:
• `PullRequest:Created` • `PullRequest:Updated` • `PullRequest:Merged`	• `PullRequest:Updated` was **removed** because it triggers on metadata changes such as title edits, description updates, or reviewer additions — not actual code changes. • Jenkins pipelines were unintentionally triggered by these events, leading to **wasted executions and user confusion**. • Generic Webhook Trigger **does not offer sufficient filtering** at the payload level to avoid this.

✅ **Improvement:**

- Replaced with `Repo:Push` event to more accurately reflect actual source code changes.
- Instead of extracting individual fields from the webhook payload using complex filter chains, the entire payload is now passed as a **single environment variable**.
- In the **Initialization Stage**, the payload is parsed and mapped to Jenkins environment variables as needed, enabling centralized and context-aware variable assignment.

🔧 **Jenkinsfile(s) Refactoring Highlights**

All major pipelines were refactored to use the shared library:

- **DLX** and **Jenkins** pipelines now invoke common logic via `stageXXX()` calls. ***(No implemented in JS Pipeline yet)***

---

3. Before vs After

🛠️ Before: Fragmented, Script-Centric Pipelines	🧱 After: Modular, Architecture-Driven Shared Library
• Each pipeline independently defined core logic (e.g., checkout, linting, build, deploy). • Business logic tightly coupled with shell commands (`sh 'git fetch'` , `sh 'rm -rf'`), scattered across Jenkinsfiles. • Shared behavior (like branch checks or merge validation) was **copy-pasted** across 5+ pipelines. • No centralized error handling, inconsistent logging formats, and no testable structure. • Webhook triggers fired pipelines **even for non-code changes**, causing noise and confusion.	• Pipelines now **invoke centralized stage modules** like `stageProjectPrepare()` and `stageLintUnity()` from the `vars/` layer. • Core business logic is abstracted into **service classes (`src/service/`)** with domain-specific responsibility. • Shell commands are encapsulated into **utility closures** (`src/utils/`) for consistency and reusability. • **Logger system unified** across all pipelines — both stage logs and console outputs now follow the same pattern. • Webhook events are filtered and processed via a **centralized payload handling strategy**, reducing pipeline noise.

🛠️ Before vs 🧱 After: Log Output Structure Changes

🔍 Before (Legacy Log Style)

- Only raw `sh` command results were printed
- No clear separation between messages; lacked visual flow
- Commands like `git fetch` , `git show-ref` , `git clean` , `echo` were all output at the same level
- No logging conventions for error messages, status updates, or step transitions
- Hard to trace which Service or Stage was responsible for each log line

✨ After (Shared Logger + Stage Message System Introduced)

- Each log message now includes emojis and prefixes like `Step Starting` , `Step Info` , `Step Completed`
- Output now follows a **structured, multi-level format** with clear context at every step
- Logging conventions such as `▶ Step Starting` , `✔ Step Executed` , `➡ Step Processing` , `🔁 ShellScript` , and `ℹ Print Message` are used consistently
- Log blocks are grouped by semantic purpose — e.g., Bitbucket status updates, Git operations, environment setup
- Stage boundaries are visually obvious
  → Easier debugging
  → Faster collaboration and troubleshooting during reviews

Stage Logs (Jenkins Initialization)

Use a tool from a predefined Tool Installation -- dotnet-8 (self time 33ms)

Fetches the environment variables for a given tool in a list of 'FOO=bar' strings suitable for the withEnv step. (self time 39ms)

Print Message -- 🚀 Step Starting: ['env' ‘[ShellScript] Print all Jenkins environment variables’ (Execute)]... (self time 27ms)

[ShellScript] Print all Jenkins environment variables -- env (self time 289ms)

Print Message (self time 29ms)

Print Message (self time 31ms)

Print Message -- ===== STAGE Starting: [Jenkins Initialization] ===== (self time 26ms)

Print Message -- ---- STEPS Starting Group: [Set Up Environment Variables] ---- (self time 27ms)

Print Message -- 💬 Step Info: [X-Event-Key: pullrequest:created] (self time 29ms)

Print Message -- 🚀 Step Starting: ['git rev-parse origin/TEST-111-merge' ‘[ShellScript] Get remote branch 'TEST-111-merge' hash from origin’ (Execute)]... (self time 25ms)

[ShellScript] Get remote branch 'TEST-111-merge' hash from origin -- git rev-parse origin/TEST-111-merge (self time 290ms)

Print Message -- 💡 Step Shell Executed Result: [Script: git rev-parse origin/TEST-111-merge Label: [ShellScript] Get remote branch 'TEST-111-merge' hash from origin ReturnStdout: 899bb193568ba512c6ab1dade8a35460c37c32a56] (self

Print Message -- 🚀 Step Starting: [Configure Report Directory Environment Variable]... (self time 35ms)

Print Message -- ✅ Step Completed: [REPORT_DIR: /var/lib/jenkins/workspace/Test-Pipeline-for-DevOps/PR-Test-Pipeline/PRJobs/TEST-111-merge] (self time 23ms)

Print Message -- 🚀 Step Starting: ['git remote show origin | \ grep 'HEAD branch' | \ awk '{print $NF}' ‘[ShellScript] Get default branch name for 'origin' remote’ (Execute)]... (self time 33ms)

[ShellScript] Get default branch name for 'origin' remote -- git remote show origin | \ grep 'HEAD branch' | \ awk '{print $NF}' (self time 531ms)

Print Message -- 💡 Step Shell Executed Result: [Script: git remote show origin | \ grep 'HEAD branch' | \ awk '{print $NF}' Label: [ShellScript] Get default branch name for 'origin' remote ReturnStdout: main] (self time 23ms)

Print Message -- 💬 Step Info: [DESTINATION_BRANCH: main] (self time 25ms)

Print Message -- 🚀 Step Starting: [Extract ticket number from PR branch: TEST-111-merge]... (self time 33ms)

Print Message -- 💬 Step Info: [Extract ticket number: TEST-111] (self time 26ms)

Print Message -- 🚀 Step Starting: [Get folder name: Test-Pipeline-for-DevOps/PR-Test-Pipeline]... (self time 32ms)

Print Message -- 💬 Step Info: [Get folder name: Test-Pipeline-for-DevOps] (self time 25ms)

Print Message -- 🚀 Step Starting: ['git fetch origin' ‘[ShellScript] Fetch all remote branches from origin’ (Execute)]... (self time 27ms)

[ShellScript] Fetch all remote branches from origin -- git fetch origin (self time 530ms)

Print Message -- 💡 Step Shell Executed Result: [Successful Executed: git fetch origin] (self time 31ms)

Print Message -- ---- STEPS Completed Group: [Set Up Environment Variables] ---- (self time 25ms)

Print Message -- ---- STEPS Starting Group: [Send Build Status to Bitbucket] ---- (self time 59ms)

Print Message -- 🚀 Step Starting: [bitbucketApiLibrary: Starting createBuildStatusForCommit execution]... (self time 32ms)

Print Message -- 🏃 Step Processing: [bitbucketApiLibrary: Target API URL: https://api.bitbucket.org/2.0/repositories/VARLab/pipeline-test/commit/f236517c1b4df058f26dc8ba4ecc74690232160 2/statuses/build] (self time 26ms)

Print Message -- 🏃 Step Processing: [bitbucketApiLibrary: Request Body: {"key":"438-Pull-Request","state":"INPROGRESS","description":"\u483c\u4ffc3 In Progress","url":"https://jenkins.vconestoga.com/job/Test-Pipeline-for-DevOps/job/P

Print Message -- 🚀 Step Starting: [bitbucketApiHelper: bitbucketApiHelper execution started]... (self time 24ms)

Print Message -- 🏃 Step Processing: [BitbucketApiService: Executing POST request to https://api.bitbucket.org/2.0/repositories/VARLab/pipeline-test/commit/f236517c1b4df058f26dc8ba4ecc74690232160 2/statuses/build] (self time 3s)

Print Message -- 🏃 Step Processing: [BitbucketApiService: Response Status Code: 201] (self time 27ms)

Print Message -- 🏃 Step Processing: [BitbucketApiService: Request successful. Status: 201] (self time 27ms)

Print Message -- 🏃 Step Processing: [BitbucketApiService: HttpResponse closed.] (self time 35ms)

Print Message -- 🏃 Step Processing: [BitbucketApiService: HttpClient closed.] (self time 14ms)

Print Message -- ✅ Step Completed: [bitbucketApiHelper: bitbucketApiHelper execution completed] (self time 29ms)

Print Message -- ---- STEPS Completed Group: [Send Build Status to Bitbucket] ---- (self time 35ms)

Print Message -- ===== STAGE Completed: [Jenkins Initialization] ===== (self time 16ms)