## **Reliability Patterns Applied in CI/CD**

Modern software delivery has shifted from monolithic applications to microservices architectures, where systems are split into smaller, independent services. This change has drastically impacted CI/CD (Continuous Integration/Continuous Deployment) pipelines, requiring patterns and best practices that ensure high availability, reliability, and faster delivery cycles.

This document outlines the key concepts and reliability patterns applied in CI/CD pipelines for microservices-based systems.

### 1. Bulkhead Pattern

* Description: Isolate different parts of a system (or pipeline) into separate compartments to contain failures.
* In CI/CD:
  + Separate pipelines for each microservice.
  + Dedicated agent pools for different services.
  + Isolate failures to individual services or environments.

### 2. Circuit Breaker Pattern

* Description: Stop calls to failing components to prevent cascading failures.
* In CI/CD:
  + Automatically halt deployment pipelines if repeated test failures occur.
  + Skip downstream stages if dependent service is unhealthy.

### 3. Retry Pattern

* Description: Automatically retry failed operations to handle transient failures.
* In CI/CD:
  + Retry steps in pipelines (e.g., deployment or test stages).
  + Configure exponential backoff for retries.

### 4. Shadow Deployment (Dark Launching)

* Description: Deploy new versions alongside existing ones, but don't expose to users.
* In CI/CD:
  + Pipeline deploys the new service but only sends mirrored traffic for testing.

### 5. Chaos Engineering (Resilience Testing)

* Description: Intentionally introduce failures to test system resilience.
* In CI/CD:
  + Introduce simulated failures as part of the deployment pipeline.
  + Validate service behavior under failure scenarios.

### 6. Timeout Pattern

* Description: Set a maximum time limit for an operation to prevent hanging processes.
* In CI/CD:
  + Apply timeouts to build, test, and deploy steps.
  + Automatically fail stages if they exceed the allowed time.
  + Prevents pipeline deadlocks and resource exhaustion.

**7. Failback Pattern**

* Description: Automatically revert to a previous stable state if a new deployment fails.
* In CI/CD:
  + Implement automatic rollback in pipelines.
  + Use health checks and monitoring to trigger failback.
  + Ensures minimal disruption during failures.

Monitoring and Observability Integration

Reliability and high availability depend heavily on observability. In microservices CI/CD pipelines, the following practices are applied:

* + Pipeline Monitoring: Track build times, failures, success rates.
  + Service Health Checks: Automate pre-deployment and post-deployment health checks.
  + Log Aggregation: Collect logs across services for centralized analysis.
  + Distributed Tracing: Trace requests across microservices to monitor performance.
  + Error Budgets: Track error rates and automate rollback if error budget exceeds the threshold.

Pipeline Architecture Example

Sample Microservice Pipeline

┌──────────────────────────┐

│ Build Stage │

├──────────────────────────┤

│ Unit Tests Run │

├──────────────────────────┤

│ Container Build & Push │

├──────────────────────────┤

│ Deploy to Staging Env │

├──────────────────────────┤

│ Run Integration Tests │

├──────────────────────────┤

│ Canary Deployment to Prod│

├──────────────────────────┤

│ Full Production Rollout │

└──────────────────────────┘

* Each microservice has an independent pipeline.
* Failures in Service A do not impact Service B.
* Automated testing, canary releases, and monitoring are built-in.

Conclusion

Microservices architectures require modern CI/CD pipelines that focus heavily on resilience and high availability. By applying patterns like Bulkhead, Circuit Breaker, Retry, Timeout, and Failback, organizations like Adobe, Microsoft, Nvidia, BY can build robust pipelines that:

* Minimize downtime.
* Isolate failures.
* Support fast, safe deployments.
* Enhance reliability through automated health checks, retries, and chaos testing.

By embedding these patterns into CI/CD pipelines, companies ensure that their microservices systems are reliable, resilient, and highly available, supporting continuous delivery of high-quality software.

## Bulkhead Pattern in CI/CD Pipelines

**Introduction**

The **Bulkhead Pattern** is a **resilience design pattern** that isolates different parts of a system into independent compartments. This ensures that if one part fails, the failure is contained and does not impact the rest of the system. The term comes from shipbuilding, where bulkheads (watertight walls) keep water from flooding the entire ship when a leak occurs.

**Purpose**

The goal of the Bulkhead Pattern is to **increase reliability and resilience** by **limiting the blast radius** of failures. By isolating services, pipelines, or stages, the system can continue to operate partially, even when one part encounters issues.

**Benefits**

* **Failure Containment:** One failing service/pipeline does not bring down others.
* **Faster Feedback:** Parallel execution ensures failures do not block unrelated services.
* **Improved Reliability:** Issues in one area are isolated from the rest of the system.
* **Scalability:** New services can be added independently without affecting existing services.
* **Controlled Resource Usage:** Dedicated resources per service avoid resource contention.

**Applying Bulkhead Pattern in CI/CD**

**Microservices Pipelines**

Each microservice has its own CI/CD pipeline, ensuring that a failure in one service’s build, test, or deployment does not block other services.

**Environment Isolation**

Separate pipelines are used for:

* Development
* QA
* Staging
* Production

A failure in the **Development** environment does not impact the **Production** pipeline.

**Parallel Execution**

Pipelines for different services, environments, or stages (build, test, deploy) can run in parallel, avoiding bottlenecks and enabling faster feedback.

**Isolated Agent Pools**

Jenkins, GitHub Actions, or similar tools can use **dedicated agent pools** for specific services or teams, ensuring:

* Resource isolation
* Failure isolation

**Example Jenkinsfile with Bulkhead Implementation**

pipeline {

agent none

stages {

stage('Build Service A') {

agent { label 'serviceA-agent' }

steps {

sh 'build-service-a.sh'

}

}

stage('Build Service B') {

agent { label 'serviceB-agent' }

steps {

sh 'build-service-b.sh'

}

}

stage('Test Service A') {

agent { label 'serviceA-agent' }

steps {

sh 'test-service-a.sh'

}

}

stage('Deploy Service A') {

agent { label 'serviceA-agent' }

steps {

sh 'deploy-service-a.sh'

}

}

}

}

**Explanation**

* Each service uses a dedicated agent (bulkhead).
* Service A’s pipeline does not affect Service B’s pipeline.
* Failures are contained within individual stages and services**.**

**Bulkhead vs Circuit Breaker**

|  |  |  |
| --- | --- | --- |
| **Pattern** | **Purpose** | **Example in CI/CD** |
| Bulkhead | Isolate components to limit failure spread | Separate agents for different microservices pipelines |
| Circuit Breaker | Detect and stop calls to failing services | Stop deployment if repeated test failures occur |

**Benefits Summary**

* Isolation of failures per service or stage
* Reduced blast radius for failures
* Parallel execution and faster feedback loops
* Improved resource allocation and monitoring

**Real-World Usage in Adobe CI/CD**

Adobe and other large organizations apply the Bulkhead Pattern to:

* Run isolated pipelines per microservice
* Use dedicated agent pools for different environments and teams
* Ensure environment failures do not cascade into production pipelines
* Run independent builds, tests, and deployments in parallel

**Conclusion**

The Bulkhead Pattern is essential in modern CI/CD pipelines, especially for organizations using **microservices architectures** and **multi-environment deployments**. It improves the overall **resilience, reliability, and speed** of software delivery pipelines.

## Circuit Breaker Pattern in Jenkins CI/CD

In Jenkins CI/CD, the Circuit Breaker Pattern ensures that the pipeline continues to function gracefully even when external dependencies, such as APIs or databases, become unavailable or unreliable. This pattern helps in preventing wasted efforts and time by automatically halting failed service calls and retrying them in a controlled manner, or skipping certain steps while continuing other processes.

Steps to Implement Circuit Breaker in Jenkins Pipeline:

1. Basic Jenkins Pipeline Setup: Create a simple Jenkins pipeline with multiple stages. One of the stages will be responsible for making calls to external services.

Example Jenkins Pipeline Code (Basic):

pipeline {  
 agent any  
 stages {  
 stage('Build') {  
 steps {  
 echo 'Building application...'  
 }  
 }  
 stage('Test') {  
 steps {  
 echo 'Running Tests...'  
 }  
 }  
 stage('External Service Call') {  
 steps {  
 script {  
 try {  
 // Call to external service (API, database, etc.)  
 externalServiceCall()  
 } catch (Exception e) {  
 echo "Service call failed: ${e.message}"  
 currentBuild.result = 'FAILURE'   
 }  
 }  
 }  
 }  
 }  
}

2. Simulate Circuit Breaker Logic with Retry and Timeout: Implement retry and timeout logic to simulate the open, closed, and half-open states of the circuit breaker. The pipeline will automatically retry the failed service calls and mark the build as failed or skipped if the maximum retries are exceeded.

Example Jenkins Pipeline Code (Retry and Timeout):

pipeline {  
 agent any  
 stages {  
 stage('Build') {  
 steps {  
 echo 'Building application...'  
 }  
 }  
 stage('Test') {  
 steps {  
 echo 'Running Tests...'  
 }  
 }  
 stage('External Service Call') {  
 steps {  
 script {  
 def maxRetries = 3  
 def success = false  
  
 for (int i = 1; i <= maxRetries; i++) {  
 try {  
 echo "Attempt ${i} to call external service"  
 // Call external service  
 externalServiceCall()  
 success = true  
 break   
 } catch (Exception e) {  
 echo "Attempt ${i} failed: ${e.message}"  
 if (i == maxRetries) {  
 echo "Circuit breaker opened after ${maxRetries} failed attempts"  
 currentBuild.result = 'FAILURE'  
 } else {  
 echo "Retrying in 10 seconds..."  
 sleep 10   
 }  
 }  
 }  
  
 if (!success) {  
 error "External service call failed after ${maxRetries} attempts"  
 }  
 }  
 }  
 }  
 }  
}

3. External Service Call with Timeout and Circuit Breaker Logic: This step ensures that the pipeline doesn't wait indefinitely for external service calls. It uses timeouts to simulate the opening of the circuit breaker.

Example Jenkins Pipeline Code (External Service Call with Timeout):

pipeline {  
 agent any  
 stages {  
 stage('Build') {  
 steps {  
 echo 'Building application...'  
 }  
 }  
 stage('Test') {  
 steps {  
 echo 'Running Tests...'  
 }  
 }  
 stage('External Service Call') {  
 steps {  
 script {  
 def maxRetries = 3  
 def timeoutSeconds = 30  
 def success = false  
  
 for (int i = 1; i <= maxRetries; i++) {  
 try {  
 echo "Attempt ${i} to call external service"  
 timeout(time: timeoutSeconds, unit: 'SECONDS') {  
 // Simulating an external service call (replace with actual call)  
 externalServiceCall()  
 }  
 success = true  
 break   
 } catch (TimeoutException e) {  
 echo "Timeout occurred on attempt ${i}: ${e.message}"  
 } catch (Exception e) {  
 echo "Service call failed: ${e.message}"  
 }  
  
 if (i == maxRetries) {  
 echo "Circuit breaker opened after ${maxRetries} failed attempts"  
 currentBuild.result = 'FAILURE'  
 } else {  
 echo "Retrying in 10 seconds..."  
 sleep 10  
 }  
 }  
  
 if (!success) {  
 error "External service call failed after ${maxRetries} attempts"  
 }  
 }  
 }  
 }  
 }  
}

4. Fallback Mechanism (Optional): In case of failure, the pipeline can implement a fallback mechanism to handle temporary unavailability of external services. The fallback logic allows the pipeline to continue with mock data or skip the affected service call.

Example Jenkins Pipeline Code (Fallback):

stage('External Service Call') {  
 steps {  
 script {  
 def maxRetries = 3  
 def success = false  
 def fallback = false  
  
 for (int i = 1; i <= maxRetries; i++) {  
 try {  
 echo "Attempt ${i} to call external service"  
 externalServiceCall()  
 success = true  
 break  
 } catch (Exception e) {  
 echo "Attempt ${i} failed: ${e.message}"  
 if (i == maxRetries) {  
 echo "Opening circuit breaker"  
 fallback = true  
 currentBuild.result = 'UNSTABLE'  
 break  
 } else {  
 echo "Retrying in 10 seconds..."  
 sleep 10  
 }  
 }  
 }  
  
 if (fallback) {  
 echo "Fallback: Using mock data or skipping service call"  
 }  
 }  
 }  
}