Release Management Process

Multi-Process Java Application · GitHub + Gradle + GitHub Packages

Team Size	5 developers
Tech Stack	Java 21, Gradle, GitHub Actions, Linux (systemd)
Artifact Registry	GitHub Packages (Maven)
Environments	DEV → QA → PROD
Versioning	Semantic Versioning (SemVer 2.0)

Table of Contents

1. Versioning Strategy

Semantic Versioning (SemVer) with Pre-Release Tags

Each deployable JAR follows **SemVer 2.0**: `MAJOR.MINOR.PATCH[-prerelease]`. Version transitions encode the promotion stage across environments.

Component	Meaning	When to Bump
MAJOR	Breaking changes	API contract changes, config format changes, cross-process protocol breaks
MINOR	New features (backward-compatible)	New endpoint, new processing capability, new config option
PATCH	Bug fixes (backward-compatible)	Defect fix, performance tweak, dependency security patch

Pre-Release Tag Progression

Environment	Version Pattern	Example
DEV	MAJOR.MINOR.PATCH-dev.SHORT_SHA	1.4.0-dev.a3f9c2
QA	MAJOR.MINOR.PATCH-rc.N	1.4.0-rc.1
PROD	MAJOR.MINOR.PATCH	1.4.0

Why SemVer over CalVer

With multiple independent processes that communicate with each other, consumers and operators need to reason about **compatibility**. SemVer encodes compatibility directly in the version number. CalVer (e.g., 2026.02.1) is better suited for products with a single deployment cadence or where the date-of-release is more important than API compatibility signaling.

Version Source of Truth

The version lives in **one place only** — `gradle.properties` at each process's root directory:

```
# billing-engine/gradle.properties
version=1.4.0-SNAPSHOT
```

The CI pipeline strips `-SNAPSHOT` and appends the appropriate pre-release tag during build. Developers **never manually edit** version strings in feature branches. Version bumps occur exclusively on `main` via a dedicated commit after each production release.

Multi-Process Version Independence

Each process gets its **own version lifecycle**. A monorepo houses all processes, but each has its own `gradle.properties`. There is no requirement to keep versions in sync — `billing-engine` can be at 3.1.0 while `notification-service` is at 1.9.2. This prevents unnecessary releases.

2. Branching Model

Trunk-Based Development with Short-Lived Feature Branches

The team uses a **trunk-based** model where `main` is always releasable. Feature branches live 1–3 days maximum. Release branches are cut only when QA stabilization begins.

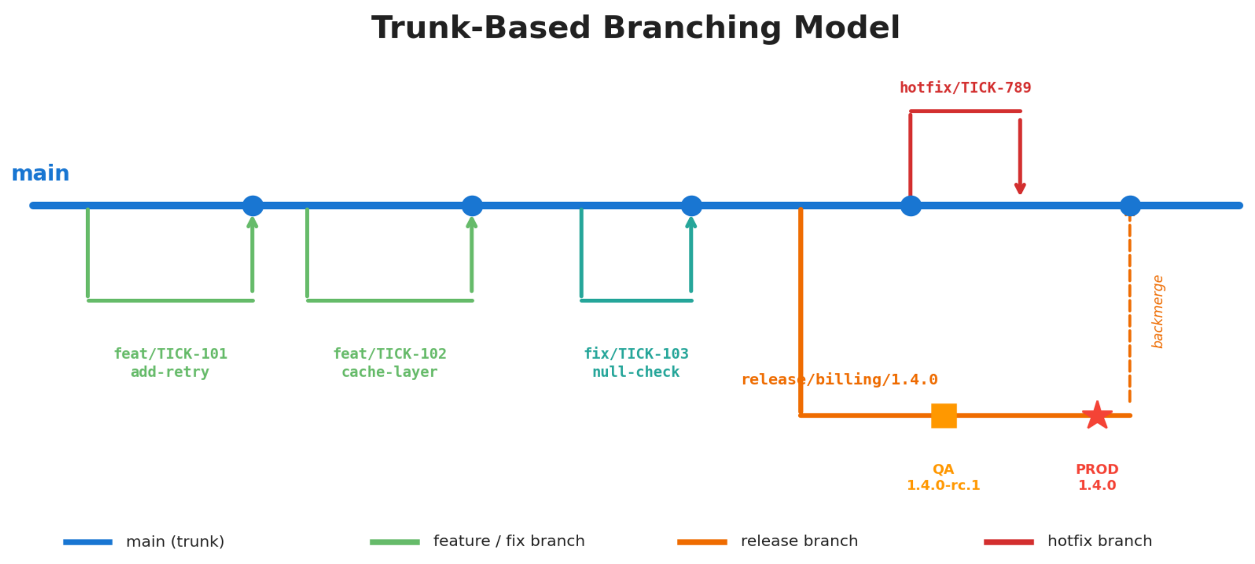


Figure 1: Trunk-Based Branching Model

Branch Pattern	Purpose	Lifetime	Merges Into
main	Integration trunk. Always builds green.	Permanent	—
feat/TICK-*	Developer work. One ticket, one branch.	1–3 days	main (via PR)
fix/TICK-*	Bug fixes	< 1 day	main (via PR)
release/{process}/{ver}	QA stabilization for a specific process	Days to ~1 week	main (backmerge)
hotfix/{process}/{tick}	Emergency production fix	Hours	release/* + main

Branch Protection Rules on main

Require **1 approving review** from any team peer. Require all CI status checks to pass (build + unit tests + static analysis). Require **linear history** via squash merge for a clean commit log. No direct pushes. No force pushes.

3. Environment Promotion Pipeline

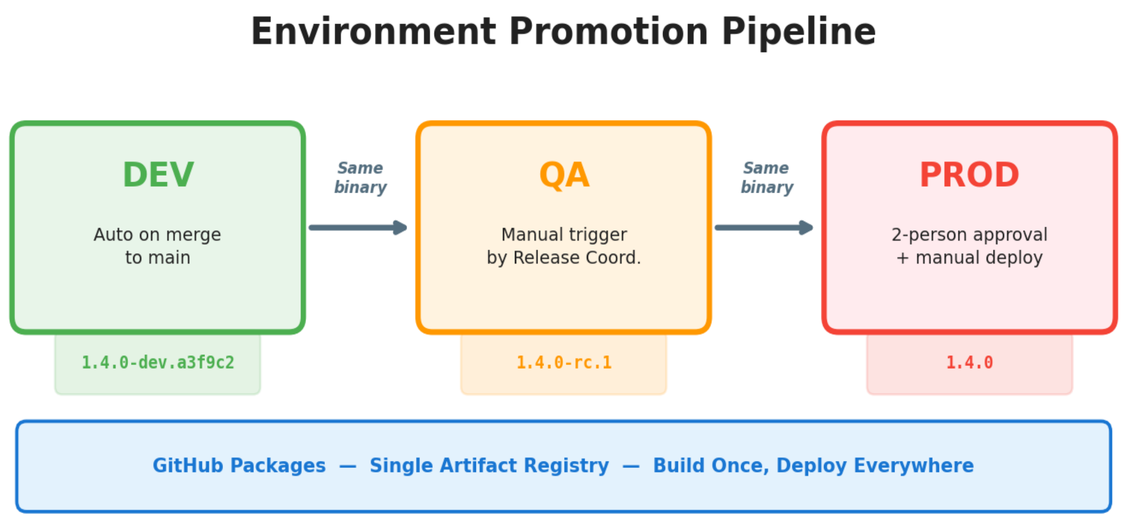

Environment Promotion Pipeline
DEV
Auto on merge to main
1.4.0-dev.a3f9c2
Same binary
QA
Manual trigger by Release Coord.
1.4.0-rc.1
Same binary
PROD
2-person approval + manual deploy
1.4.0
GitHub Packages — Single Artifact Registry — Build Once, Deploy Everywhere

Figure 2: Environment Promotion Pipeline

Environment	Trigger	Artifact Version	Who Deploys	Purpose
DEV	Auto on merge to main	1.4.0-dev.a3f9c2	CI (automatic)	Integration testing
QA	Manual workflow dispatch	1.4.0-rc.1	Release Coordinator	Regression + acceptance
PROD	Manual + 2 approvals	1.4.0	Release Coordinator + approvers	Live traffic

Artifact Immutability Principle: The same JAR binary that passes QA is the exact binary deployed to production. No rebuilding. The artifact is published to GitHub Packages once during the DEV build, tagged with its version, and pulled by each subsequent environment. This guarantees what you tested is what you ship.

4. CI/CD Workflow Architecture

Clarification: Why Two Workflows That Both Compile

A common question arises: if we follow a "build once" principle, why do both `ci-validate.yml` and `deploy-dev.yml` compile the code? The answer is that these serve **fundamentally different purposes**:

Workflow	Trigger	Purpose	Produces Artifact?
ci-validate.yml	Pull Request opened/updated	Fast feedback gate: compile, test, and analyze code BEFORE merge. Throwaway build — validates correctness only.	No — build output discarded
deploy-dev.yml	Merge to main	The OFFICIAL build: produces the versioned, immutable JAR that will travel through all environments unchanged.	Yes — published to GitHub Packages

The "build once" principle applies from `deploy-dev.yml` onward. The PR validation build is a disposable quality gate — it exists to give developers fast feedback ("does my code compile and pass tests?") before their changes land on `main`. The artifact it produces is never deployed anywhere.

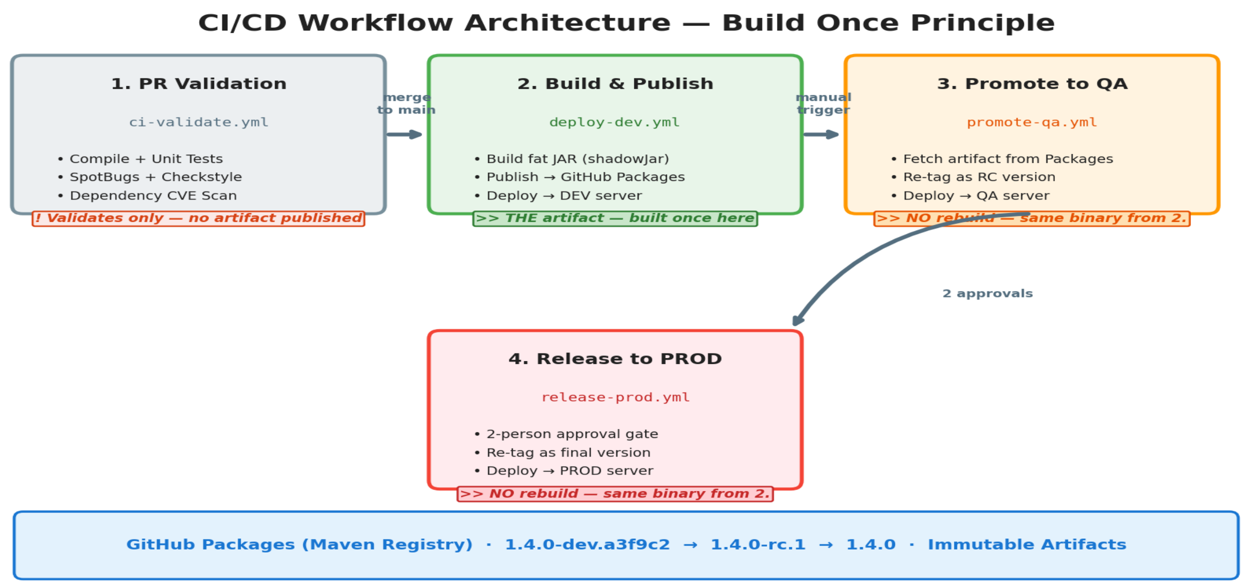

CI/CD Workflow Architecture — Build Once Principle
1. PR Validation
ci-validate.yml
• Compile + Unit Tests
• SpotBugs + Checkstyle
• Dependency CVE Scan
! Validates only — no artifact published
merge to main
2. Build & Publish
deploy-dev.yml
• Build fat JAR (shadowJar)
• Publish → GitHub Packages
• Deploy → DEV server
>> THE artifact — built once here
manual trigger
3. Promote to QA
promote-qa.yml
• Fetch artifact from Packages
• Re-tag as RC version
• Deploy → QA server
>> NO rebuild — same binary from 2.
2 approvals
4. Release to PROD
release-prod.yml
• 2-person approval gate
• Re-tag as final version
• Deploy → PROD server
>> NO rebuild — same binary from 2.
GitHub Packages (Maven Registry) · 1.4.0-dev.a3f9c2 → 1.4.0-rc.1 → 1.4.0 · Immutable Artifacts

Figure 3: CI/CD Workflow Architecture — Build Once Principle

4.1 PR Validation — ci-validate.yml

Triggers on every pull request targeting `main`. Uses a change-detection matrix to only validate the processes whose files changed (monorepo efficiency).

```
# .github/workflows/ci-validate.yml
name: PR Validation
on:
  pull_request:
    branches: [main]
```

```
jobs:
  detect-changes:
    runs-on: ubuntu-latest
    outputs:
      matrix: ${{ steps.set-matrix.outputs.matrix }}
    steps:
      - uses: actions/checkout@v4
      - id: set-matrix
        run: |
          # Detect which process subdirectories changed
          # Outputs JSON matrix of affected processes

  validate:
    needs: detect-changes
    runs-on: ubuntu-latest
    strategy:
      matrix: ${{ fromJson(needs.detect-changes.outputs.matrix) }}
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-java@v4
        with:
          distribution: temurin
          java-version: '21'
          cache: gradle
      - name: Build &amp; Unit Test
        run: ./gradlew :${{ matrix.process }}:build
      - name: Static Analysis (SpotBugs + Checkstyle)
        run: ./gradlew :${{ matrix.process }}:check
      - name: Dependency Vulnerability Scan
        run: ./gradlew :${{ matrix.process }}:dependencyCheckAnalyze
```

4.2 Build & Deploy to DEV — deploy-dev.yml

Triggers automatically on merge to `main`. This is the **single build** that produces the immutable artifact. The JAR is published to GitHub Packages, then deployed to DEV.

```
# .github/workflows/deploy-dev.yml
name: Build &amp; Deploy to DEV
on:
  push:
    branches: [main]

jobs:
  build-publish-deploy:
    runs-on: ubuntu-latest
    strategy:
      matrix: ${{ fromJson(needs.detect-changes.outputs.matrix) }}
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-java@v4
        with: { distribution: temurin, java-version: '21', cache: gradle }

      - name: Compute Dev Version
        run: |
          BASE=$(grep '^version=' ${{ matrix.process }}/gradle.properties \
                | cut -d= -f2 | sed 's/-SNAPSHOT//')
          echo "VERSION=${BASE}-dev.${GITHUB_SHA::7}" &gt;&gt; $GITHUB_ENV

      - name: Build Fat JAR
        run: ./gradlew :${{ matrix.process }}:shadowJar -Pversion=${{ env.VERSION }}
      - name: Publish to GitHub Packages
        run: ./gradlew :${{ matrix.process }}:publish -Pversion=${{ env.VERSION }}
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}

      - name: Deploy to DEV Server
        uses: appleboy/ssh-action@v1
        with:
          host: ${{ vars.DEV_SERVER_HOST }}
          username: ${{ vars.DEPLOY_USER }}
          key: ${{ secrets.DEV_SSH_KEY }}
          script: /opt/deploy/deploy.sh ${{ matrix.process }} ${{ env.VERSION }}
```

4.3 Promote to QA — promote-qa.yml

Manually triggered. Pulls an **existing artifact** from GitHub Packages — no rebuild. Re-tags the binary with a release candidate version and deploys it to the QA server.

```
# .github/workflows/promote-qa.yml
name: Promote to QA
on:
  workflow_dispatch:
    inputs:
      process:
        description: 'Process to promote'
        required: true
        type: choice
        options: [billing-engine, notification-service, data-processor]
      version:
        description: 'Dev version to promote (e.g., 1.4.0-dev.a3f9c2)'
        required: true
        type: string

jobs:
  promote:
    runs-on: ubuntu-latest
    environment: qa    # GitHub environment protection rules
    steps:
      - name: Compute RC Version
        run: |
          BASE=$(echo "${{ inputs.version }}" | sed 's/-dev\..*//')
          echo "RC_VERSION=${BASE}-rc.1" &gt;&gt; $GITHUB_ENV

      - name: Pull Artifact from GitHub Packages
        run: |
          # Download the exact dev JAR — NO rebuild
          mvn dependency:copy -Dartifact=com.company:${{ inputs.process }}:${{ inputs.version }}

      - name: Re-tag &amp; Republish as RC
        run: |
          # Publish same binary under the RC version tag

      - name: Deploy to QA Server
        uses: appleboy/ssh-action@v1
        with:
          host: ${{ vars.QA_SERVER_HOST }}
          key: ${{ secrets.QA_SSH_KEY }}
          script: /opt/deploy/deploy.sh ${{ inputs.process }} ${{ env.RC_VERSION }}

      - name: Create GitHub Pre-Release
        uses: softprops/action-gh-release@v2
        with:
          tag_name: ${{ inputs.process }}/v${{ env.RC_VERSION }}
          prerelease: true
          generate_release_notes: true
```

4.4 Release to Production — release-prod.yml

Manually triggered with a **2-person approval gate** enforced via GitHub environment protection rules. Pulls the RC artifact — no rebuild — re-tags as final version, deploys, and bumps the version on `main`.

```
# .github/workflows/release-prod.yml
```

```
name: Release to Production
on:
  workflow_dispatch:
    inputs:
      process:
        description: 'Process to release'
        required: true
        type: choice
        options: [billing-engine, notification-service, data-processor]
      rc_version:
        description: 'RC version to release (e.g., 1.4.0-rc.1)'
        required: true
        type: string

jobs:
  release:
    runs-on: ubuntu-latest
    environment: production   # Requires 2 approvals
    steps:
      - name: Compute Release Version
        run: |
          RELEASE=$(echo "${{ inputs.rc_version }}" | sed 's/-rc\..*//')
          echo "RELEASE_VERSION=${RELEASE}" &gt;&gt; $GITHUB_ENV

      - name: Promote Artifact
        run: |
          # Re-tag RC artifact as final release in GitHub Packages

      - name: Deploy to Production
        uses: appleboy/ssh-action@v1
        with:
          host: ${{ vars.PROD_SERVER_HOST }}
          key: ${{ secrets.PROD_SSH_KEY }}
          script: /opt/deploy/deploy.sh ${{ inputs.process }} ${{ env.RELEASE_VERSION }}

      - name: Create GitHub Release
        uses: softprops/action-gh-release@v2
        with:
          tag_name: ${{ inputs.process }}/v${{ env.RELEASE_VERSION }}
          prerelease: false
          generate_release_notes: true

      - name: Bump Version on main
        run: |
          # Checkout main, increment PATCH: 1.4.0 -&gt; 1.5.0-SNAPSHOT, commit
```

5. GitHub Packages — Artifact Organization

GitHub Packages uses the standard Maven coordinate system to organize artifacts. Each process is a separate `artifactId`, and the version string encodes the promotion stage. This creates a natural hierarchy without any extra tooling.

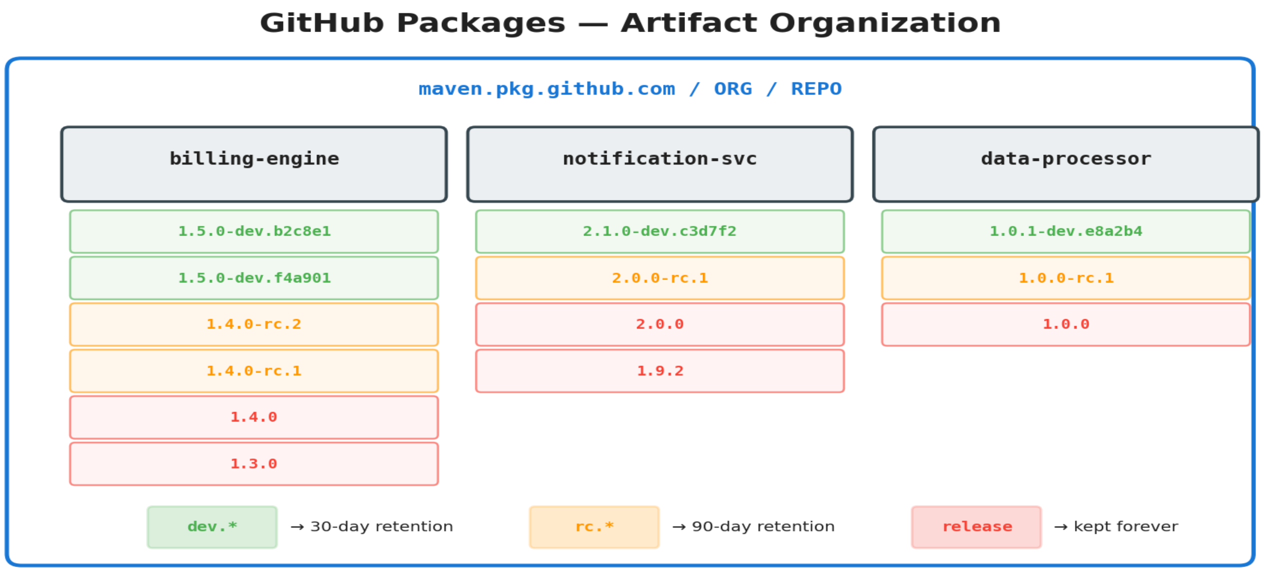


Figure 4: GitHub Packages — Artifact Organization by Process and Stage

Maven Coordinates

```
# All processes share the same groupId and repository
groupId:    com.company
artifactId: &lt;process-name&gt;    # billing-engine | notification-service | data-processor
version:    &lt;semver-tag&gt;      # 1.4.0-dev.a3f9c2 | 1.4.0-rc.1 | 1.4.0

# Registry URL (single repo for all processes)
https://maven.pkg.github.com/ORG/REPO
```

Because GitHub Packages scopes packages to a repository, all process artifacts live under a **single repository's package registry**. They are distinguished by their `artifactId` (the process name). Each version tag is unique and immutable once published.

Retention Policy

Version Pattern	Example	Retention	Cleanup Method
-dev.*	1.4.0-dev.a3f9c2	30 days	Scheduled GH Actions workflow (weekly)
-rc.*	1.4.0-rc.1	90 days	Scheduled GH Actions workflow (monthly)
Release (no suffix)	1.4.0	Indefinite	Manual deletion only

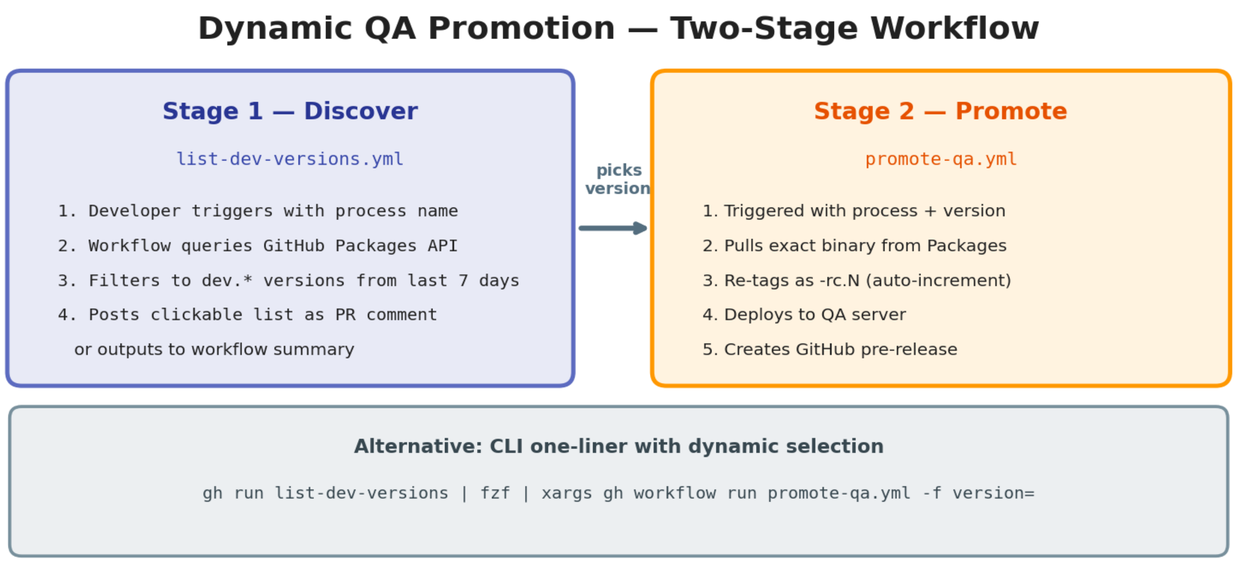
Gradle Publishing Configuration

```
// build.gradle (each process)
publishing {
    publications {
        maven(MavenPublication) {
            from components.java
            groupId = 'com.company'
            artifactId = project.name      // e.g., 'billing-engine'
            version = project.version      // set by CI via -Pversion=
        }
    }
    repositories {
        maven {
            name = "GitHubPackages"
            url = uri("https://maven.pkg.github.com/ORG/REPO")
            credentials {
                username = System.getenv("GITHUB_ACTOR")
                password = System.getenv("GITHUB_TOKEN")
            }
        }
    }
}
```

6. Dynamic QA Promotion

A key usability question: **can we dynamically populate the version input** when triggering the QA promotion workflow? GitHub Actions' `workflow_dispatch choice` inputs are statically defined in YAML — they cannot query an API at render time. However, there are excellent workarounds:

Recommended: Two-Stage Workflow Pattern


Dynamic QA Promotion — Two-Stage Workflow
Stage 1 — Discover
list-dev-versions.yml
1. Developer triggers with process name
2. Workflow queries GitHub Packages API
3. Filters to dev.* versions from last 7 days
4. Posts clickable list as PR comment
or outputs to workflow summary
picks version
Stage 2 — Promote
promote-qa.yml
1. Triggered with process + version
2. Pulls exact binary from Packages
3. Re-tags as -rc.N (auto-increment)
4. Deploys to QA server
5. Creates GitHub pre-release
Alternative: CLI one-liner with dynamic selection

```
gh run list-dev-versions | fzf | xargs gh workflow run promote-qa.yml -f version=
```

Figure 5: Two-Stage Dynamic QA Promotion

The first workflow (`list-dev-versions.yml`) queries the GitHub Packages API, filters to recent `-dev.*` versions for the selected process, and presents them in the workflow run summary. The Release Coordinator picks one, then triggers the second workflow with that exact version string.

Discovery Workflow

```
# .github/workflows/list-dev-versions.yml
name: List Available Dev Versions
on:
  workflow_dispatch:
    inputs:
      process:
        description: 'Process to query'
        required: true
        type: choice
        options: [billing-engine, notification-service, data-processor]

jobs:
  list:
    runs-on: ubuntu-latest
    steps:
      - name: Query GitHub Packages API
        env:
          GH_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        run: |
          VERSIONS=$(gh api \
            "/orgs/ORG/packages/maven/com.company.${{ inputs.process }}/versions" \
            --jq '.[].name | select(contains("dev"))' \
            | head -10)

          echo "## Available Dev Versions for ${{ inputs.process }}" &gt;&gt; $GITHUB_STEP_SUMMARY
          echo "" &gt;&gt; $GITHUB_STEP_SUMMARY
          echo "| Version | Published |" &gt;&gt; $GITHUB_STEP_SUMMARY
          echo "|---------|-----------|" &gt;&gt; $GITHUB_STEP_SUMMARY
          # ... format each version as a table row

      - name: Output for downstream
        run: echo "$VERSIONS"  # Also usable by gh CLI for chaining
```

Alternative: gh CLI One-Liner

For developers comfortable with the terminal, a single gh command can query, select, and trigger in one flow:

```
# Interactive: list versions, pick with fzf, trigger promotion
gh api "/orgs/ORG/packages/maven/com.company.billing-engine/versions" \
  --jq '.[].name | select(contains("dev"))' \
  | fzf --prompt="Select version&gt; " \
  | xargs -I{} gh workflow run promote-qa.yml \
      -f process=billing-engine -f version={}
```

7. Deployment Mechanics on Linux Servers

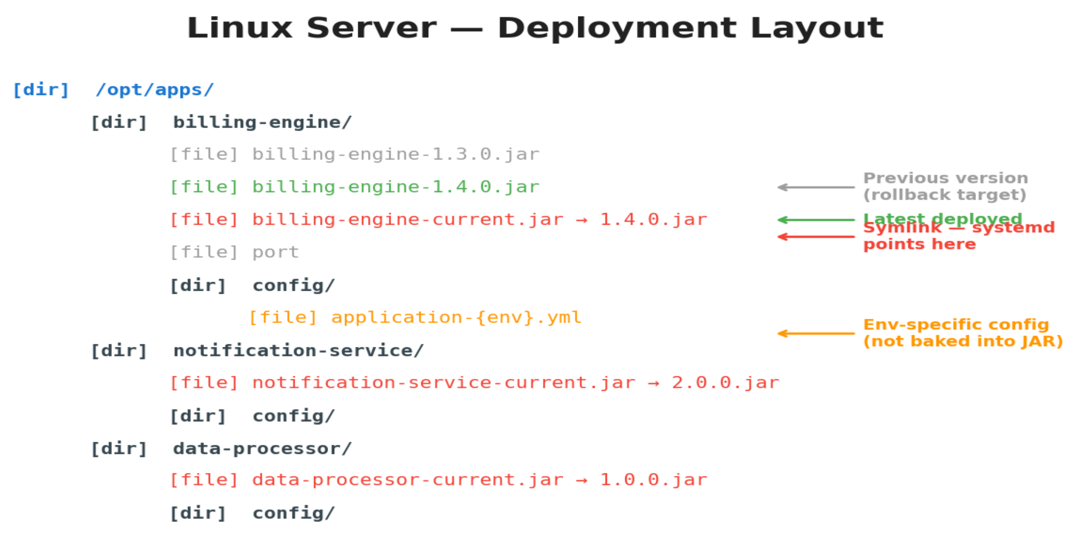

Linux Server — Deployment Layout
[dir] /opt/apps/
[dir] billing-engine/
[file] billing-engine-1.3.0.jar
[file] billing-engine-1.4.0.jar
[file] billing-engine-current.jar → 1.4.0.jar
[file] port
[dir] config/
[file] application-{env}.yml
[dir] notification-service/
[file] notification-service-current.jar → 2.0.0.jar
[dir] config/
[dir] data-processor/
[file] data-processor-current.jar → 1.0.0.jar
[dir] config/
Previous version (rollback target)
Latest deployed
Symlink — systemd points here
Env-specific config (not baked into JAR)

Figure 6: Linux Server Deployment Layout

Deployment Script — /opt/deploy/deploy.sh

Each server has a standardized deployment script. It downloads the artifact from GitHub Packages, updates the `current` symlink, restarts the systemd service, and verifies health. If the health check fails, it **automatically rolls back** to the previous version.

```
#!/usr/bin/env bash
set -euo pipefail

PROCESS=$1
VERSION=$2
DEPLOY_ROOT="/opt/apps"
ARTIFACT_URL="https://maven.pkg.github.com/ORG/REPO/\
  com/company/${PROCESS}/${VERSION}/${PROCESS}-${VERSION}.jar"

echo "[deploy] Downloading ${PROCESS} v${VERSION}..."
curl -sL -H "Authorization: token ${GITHUB_TOKEN}" \
     -o "${DEPLOY_ROOT}/${PROCESS}/${PROCESS}-${VERSION}.jar" \
     "${ARTIFACT_URL}"

# Symlink to new version
ln -sfn "${DEPLOY_ROOT}/${PROCESS}/${PROCESS}-${VERSION}.jar" \
        "${DEPLOY_ROOT}/${PROCESS}/${PROCESS}-current.jar"

echo "[deploy] Restarting ${PROCESS}..."
sudo systemctl restart "${PROCESS}"

echo "[deploy] Verifying health..."
for i in $(seq 1 30); do
  if curl -sf "http://localhost:$(cat /opt/apps/${PROCESS}/port)/health"; then
    echo "[deploy] ${PROCESS} v${VERSION} is healthy."
    exit 0
  fi
  sleep 2
done

echo "[deploy] FAILED health check - rolling back."
PREVIOUS=$(ls -t "${DEPLOY_ROOT}/${PROCESS}/"*.jar | grep -v "${VERSION}" | head -1)
ln -sfn "${PREVIOUS}" "${DEPLOY_ROOT}/${PROCESS}/${PROCESS}-current.jar"
sudo systemctl restart "${PROCESS}"
exit 1
```

Systemd Service Unit (per process)

```
# /etc/systemd/system/billing-engine.service
[Unit]
Description=Billing Engine
After=network.target

[Service]
User=appuser
Group=appuser
ExecStart=/usr/bin/java -jar /opt/apps/billing-engine/billing-engine-current.jar \
          --spring.profiles.active=${ENVIRONMENT}
Restart=on-failure
RestartSec=10
SuccessExitStatus=143
StandardOutput=journal
StandardError=journal

[Install]
WantedBy=multi-user.target
```

8. Team Roles & Responsibilities

With a team of 5, the process avoids heavyweight ceremony. Key roles **rotate monthly** to spread knowledge and prevent bottlenecks.

Role	Count	Responsibility
Release Coordinator (rotating)	1	Triggers QA/Prod promotions, writes release notes, coordinates timing
PR Reviewer	Any 1 peer	Every PR gets exactly 1 review — fast, no bottleneck
On-Call / Hotfix Owner (rotating)	1	First responder for prod issues, owns hotfix branches
All Developers	5	Write code, write tests, own features through to prod

Approval Gates

Gate	Required Approvers	Mechanism
PR to main	1 team member	GitHub PR review
DEV to QA	Release Coordinator	Workflow dispatch trigger (no approval gate)
QA to PROD	2 team members (not the author)	GitHub Environment protection rule
Hotfix to PROD	1 peer + Release Coordinator	Expedited review + deployment

9. Release Cadence

Weekly Release Train (Suggested)

Day	Activity
Monday EOD	Feature freeze: all features for this train must be merged to main
Tuesday AM	Release Coordinator cuts release branch, promotes RC to QA
Tuesday-Wednesday	QA team runs regression and acceptance tests
Wednesday PM	QA sign-off; Release Coordinator triggers PROD deployment

This cadence is a **guideline, not an obligation**. If nothing meaningful changed since the last release, skip it. If something urgent lands on Thursday, do an off-cycle release. The tooling supports any cadence — the process is designed to flex.

10. Hotfix Workflow

Hotfixes bypass the weekly train. They are for production-critical issues that cannot wait.

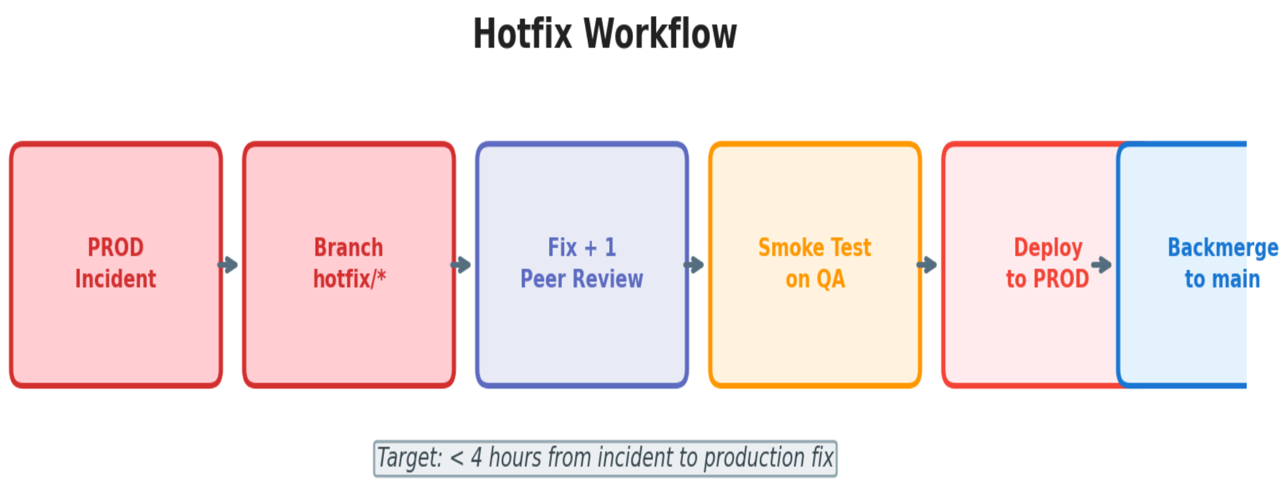


Figure 7: Hotfix Workflow — Target: Under 4 Hours to Production

Step	Action	Details
1	Branch from main	Create hotfix/{process}/TICK-NNN from the release tag
2	Fix + expedited review	1-person peer review (not the full PR process)
3	QA smoke test	Minimal regression on QA environment (no full test suite)
4	Deploy to PROD	Release Coordinator triggers with 1 approval (expedited)
5	Backmerge to main	Ensure the fix lands on trunk for future releases

11. Rollback Procedure

Automatic Rollback

The deployment script (Section 7) includes a built-in health check loop. If the new version does not respond healthy within 60 seconds, the script **automatically reverts** the symlink to the previous JAR and restarts the service.

Manual Rollback

If an issue is discovered after deployment passes the health check:

```
# On the production server:
```

```
/opt/deploy/rollback.sh billing-engine

# The script:
# 1. Reverts the symlink to the previous JAR on disk
# 2. Restarts the systemd service
# 3. Verifies the health endpoint returns 200
# 4. Logs the rollback event
```

Any team member can initiate a rollback in an emergency. Post-rollback, the team conducts a brief async retrospective via a GitHub Issue labeled `post-mortem`.

12. Observability & Release Validation

Deployment Health Endpoint

Every deployed process exposes a health endpoint that includes version and deployment metadata:

```
GET /health
{
  "status": "UP",
  "version": "1.4.0",
  "deployedAt": "2026-02-25T14:30:00Z",
  "commitSha": "a3f9c2d"
}
```

Post-Deploy Checklist (Automated in CI)

After each deployment, the CI workflow automatically validates: health endpoint returns 200 with the correct version; application logs show no ERROR-level entries in the first 5 minutes; and if applicable, a smoke test suite passes against the deployed environment.

Release Tracking

Every production release creates a **GitHub Release** with auto-generated release notes (from PR titles since last release). This serves as the audit trail, changelog, and communication vehicle for stakeholders.

13. Security Practices

Practice	Implementation
Secrets management	GitHub Actions Secrets + Environment Secrets. Never in code or config files.
SSH keys	Per-environment keys, rotated quarterly. Deploy user has minimal sudo for systemctl only.
Dependency scanning	OWASP Dependency Check on every PR. Critical CVEs block merge.
SBOM	Software Bill of Materials generated with each release, attached to the GitHub Release.
Signed commits	Encouraged (GPG or SSH signing) but not enforced to keep a 5-person team moving fast.
Network access	Prod servers accept SSH only from GitHub Actions runner IPs (IP allowlisting).

14. Quick Reference Cheat Sheet

Developer Daily Flow

```
branch from main  --&gt;  code  --&gt;  PR  --&gt;  1 review  --&gt;  squash merge  --&gt;  auto-deploy DE
```

Promotion Flow

```
DEV (auto)  --&gt;  QA (manual trigger, same binary)  --&gt;  PROD (2 approvals, same binary)
```

Version Lifecycle

```
1.4.0-dev.a3f9c2  --&gt;  1.4.0-rc.1  --&gt;  1.4.0
```

Key Principles

> Build once, deploy everywhere (artifact immutability)

> Version per process, not per repository

> Automate the boring stuff, gate the risky stuff

> Any team member can rollback production in an emergency

> Weekly cadence suggestion, not a weekly obligation

> Same binary travels DEV to QA to PROD — never rebuilt

> Rotate Release Coordinator and On-Call roles monthly

This document should be reviewed and updated quarterly, or whenever the team's processes change materially.