Here’s a concise, copy-pasteable checklist you can use to review and approve **bot-opened PRs** (strict GitFlow: release/x.y → main, release/x.y → develop, hotfix back-merges, and label-driven backports).

**Bot PR Review Checklist**

**1) Preconditions (quick triage)**

* **Author is the bot** (e.g., github-actions[bot] or your bot account).
* **PR source/target match the flow**
  + Release publish: release/x.y → main
  + Release sync: release/x.y → develop
  + Hotfix sync: hotfix/x.y.z → develop and/or hotfix/x.y.z → release/\*
  + Backport: backport/\* → release/x.y
* **No human edits to bot branch** (branch should be auto-generated; changes should be purely merges/cherry-picks).

**2) Labels & metadata**

* **automerge** label is present (or automation has enabled auto-merge).
* **No do-not-automerge** label (if present, stop and ask owner).
* **Backport PRs** have backport label (and the originating PR has auto-backport:release/x.y).
* **Hotfix PRs** mention the **tag** (e.g., v2.3.1) in title/body.

**3) Diff sanity (keep it minimal & expected)**

* **Release → main**: Diff should match what was **tagged** on release/x.y (no extra features).
* **Release → develop**: Only stabilization fixes; **no new features**.
* **Hotfix → develop/release**: Only the hotfix commits (or cherry-picks); check for accidental unrelated changes.
* **Backport → release/x.y**: Diff equals the single cherry-picked commit(s) from the source PR (with -x note in commit message).

**4) Required checks (branch-specific)**

| **Target branch** | **Must pass checks (example)** |
| --- | --- |
| main | unit tests, integration/smoke, linter, SAST/Secret scan, image scan, SBOM/provenance, policy gate |
| release/\* | unit tests, targeted smoke/regression, linter, SAST/Secret scan |
| develop | unit tests, linter, SAST/Secret scan (integration optional if slow) |

* All **required checks are green**.
* **No skipped/required-but-missing** checks (check branch protection settings).

**5) Security & compliance**

* **No new secrets** in diff; secret scan clean.
* **Dependency changes** (if any) are expected and scanned (SCA).
* **SBOM** and **provenance** artifacts are attached/generated for main releases.

**6) Merge safety**

* **No conflicts**; if the bot opened a **Draft** due to conflicts, do **not** flip to ready until resolved correctly.
* **Merge method** is correct (usually **squash**) and **auto-merge** is enabled (unless intentionally paused).
* **Merge queue** eligible (if enabled) — no blocking conditions.

**7) Release targeting rules**

* **Release → main** PR references the **exact tag** (e.g., v2.4.0) being promoted.
* **Backport** targets the **right release line** (e.g., release/2.4) that is still supported/open.
* **Hotfix** PR points at currently active release lines (if release/\* still open).

**8) Observability & rollout notes (for main)**

* PR or tag has **release notes/changelog** generated.
* **Rollout/rollback plan** noted (flags, canary, or previous tag).
* If migrations are included, confirm **reversibility** or **safe forward-only** plan acknowledged.

**9) Final actions**

* Approve the PR (CODEOWNERS).
* Confirm **auto-merge** is toggled on (or manually merge if policy requires).
* After merge, verify **artifact promotion** (no rebuilds) and **deployment** (staging/prod as applicable).

**Quick decision guide**

* **Approve now** if: diff is expected/minimal, checks green, labels correct, and target branch rules are met.
* **Hold** if: wrong target branch, missing label/notes, or failing/insufficient checks.
* **Reject/close** if: empty diff (already merged by other path), unrelated changes, or security policy violations.

**PR comment templates**

**Approve (no issues)**

✅ Looks good. Diff matches expected scope. All required checks green. Proceeding with auto-merge.

**Hold (needs fix)**

⏸️ Holding auto-merge:  
• Missing label backport (or incorrect target branch).  
• Please ensure required checks listlistlist are enabled for this branch.  
• Update title/body to include tag/version context.

**Reject/close (obsolete or wrong)**

❌ Closing: Changes already present in target branch (empty diff) / incorrect target. Please re-open with corrected base.

**Optional: one-screen “lite” checklist (for busy approvers)**

* Source → Target valid (release→main, release→develop, hotfix/backport OK)
* Labels: automerge present; **no** do-not-automerge
* Diff minimal/expected (no features on release lines)
* Checks: **all required green** (per branch policy)
* Security scans passed (secrets/SAST/SCA)
* Tag/version referenced (for release/hotfix)
* Approve + confirm auto-merge on

Got it—you can do the same “autobot” flows in **Jenkins** using a few lightweight jobs plus a GitHub token. Below is a practical setup you can copy. I’ll show:

1. prerequisites
2. job wiring (triggers)
3. Jenkinsfiles for each flow (using gh CLI; curl works too)
4. optional shared-library helpers (to keep Jenkinsfiles tiny)

**1) Prerequisites**

* **Jenkins plugins**
  + *Pipeline: Declarative*, *Pipeline Utility Steps* (for readJSON), *Credentials Binding*, *Generic Webhook Trigger* (to catch GitHub events), *Git*.
* **Bot credentials**
  + Create a **GitHub App** or a **PAT** for a bot account with repo scope.
  + Store in Jenkins as **Secret Text**: GH\_TOKEN\_BOT.
* **Agents**
  + Install **Git** and **GitHub CLI (gh)** on the agent image.
* **Env**
  + Set repo env in jobs or a global:  
    GITHUB\_OWNER=your-org  
    GITHUB\_REPO=your-repo

**2) Job wiring (triggers)**

* **Job A: auto-pr-on-release-tag**  
  Trigger: GitHub **Release → published** webhook → Generic Webhook Trigger.
* **Job B: auto-pr-on-hotfix-tag**  
  Trigger: GitHub **Release → published** webhook (same endpoint as A, routed by logic), or create a second job—your call.
* **Job C: backport-on-label**  
  Trigger: GitHub **Pull request → closed** webhook (we only act when merged==true and a label matches auto-backport:\*).
* **Job D: autobot-pr-sweeper**  
  Trigger: Jenkins **cron** nightly.

In GitHub repo → Settings → Webhooks → add Jenkins URL for **Generic Webhook Trigger** (content type JSON; events: Releases + Pull Requests).

**3) Jenkinsfiles**

**A) Release tag → PR release/x.y → main and release/x.y → develop**

**Job**: auto-pr-on-release-tag (Declarative Pipeline)

pipeline {

agent any

environment {

GH\_TOKEN = credentials('GH\_TOKEN\_BOT')

GITHUB\_OWNER = 'your-org'

GITHUB\_REPO = 'your-repo'

}

triggers {

GenericTrigger(

genericVariables: [

[key: 'payload', value: '$']

],

causeString: 'GitHub release event',

// Use a token/secret per your security policy

printContributedVariables: false,

printPostContent: false

)

}

stages {

stage('Parse payload & guard') {

steps {

script {

def json = readJSON text: env.payload

if (json.action != 'published') {

currentBuild.result = 'NOT\_BUILT'; error "Not a published release"

}

env.TAG = json.release.tag\_name // e.g. v2.3.0

// We expect tags from release/x.y

env.VERSION\_NO\_V = env.TAG.startsWith('v') ? env.TAG.substring(1) : env.TAG

env.LINE = env.VERSION\_NO\_V.split('\\.')[0..1].join('.') // 2.3

env.REL\_BRANCH = "release/${env.LINE}" // release/2.3

}

}

}

stage('Open/ensure PRs') {

steps {

sh '''

set -euo pipefail

export GH\_TOKEN="${GH\_TOKEN}"

# Ensure release branch exists remotely

git ls-remote --exit-code --heads https://github.com/${GITHUB\_OWNER}/${GITHUB\_REPO}.git "${REL\_BRANCH}" >/dev/null

# Open PR: release -> main (publish)

OPEN=$(gh pr list -R ${GITHUB\_OWNER}/${GITHUB\_REPO} --base main --head "${REL\_BRANCH}" --state open --json number -q 'length(.)')

if [ "$OPEN" -eq 0 ]; then

gh pr create -R ${GITHUB\_OWNER}/${GITHUB\_REPO} --base main --head "${REL\_BRANCH}" \

--title "Release → main (${REL\_BRANCH})" \

--body "Automated PR to publish tag ${TAG} from \`${REL\_BRANCH}\`."

fi

PR\_MAIN=$(gh pr list -R ${GITHUB\_OWNER}/${GITHUB\_REPO} --base main --head "${REL\_BRANCH}" --state open --json number -q '.[0].number' || true)

[ -n "${PR\_MAIN}" ] && gh pr merge -R ${GITHUB\_OWNER}/${GITHUB\_REPO} "${PR\_MAIN}" --auto --squash || true

# Open PR: release -> develop (sync)

OPEN=$(gh pr list -R ${GITHUB\_OWNER}/${GITHUB\_REPO} --base develop --head "${REL\_BRANCH}" --state open --json number -q 'length(.)')

if [ "$OPEN" -eq 0 ]; then

gh pr create -R ${GITHUB\_OWNER}/${GITHUB\_REPO} --base develop --head "${REL\_BRANCH}" \

--title "Release back-merge → develop (${REL\_BRANCH})" \

--body "Automated PR to sync ${TAG} fixes back into \`develop\`."

fi

PR\_DEV=$(gh pr list -R ${GITHUB\_OWNER}/${GITHUB\_REPO} --base develop --head "${REL\_BRANCH}" --state open --json number -q '.[0].number' || true)

[ -n "${PR\_DEV}" ] && gh pr merge -R ${GITHUB\_OWNER}/${GITHUB\_REPO} "${PR\_DEV}" --auto --squash || true

'''

}

}

}

}

**B) Hotfix tag → PR(s) back to develop and any open release/\***

**Job**: auto-pr-on-hotfix-tag

pipeline {

agent any

environment {

GH\_TOKEN = credentials('GH\_TOKEN\_BOT')

GITHUB\_OWNER = 'your-org'

GITHUB\_REPO = 'your-repo'

}

triggers {

GenericTrigger(

genericVariables: [[key: 'payload', value: '$']],

causeString: 'GitHub release event'

)

}

stages {

stage('Parse & guard') {

steps {

script {

def json = readJSON text: env.payload

if (json.action != 'published') { currentBuild.result='NOT\_BUILT'; error "Not published" }

env.TAG = json.release.tag\_name // v2.2.5

env.VERSION = env.TAG.startsWith('v') ? env.TAG.substring(1) : env.TAG // 2.2.5

def parts = env.VERSION.tokenize('.')

env.IS\_HOTFIX = (parts.size()==3 && parts[2].isInteger() && parts[2].toInteger() >= 1) ? 'true':'false'

if (env.IS\_HOTFIX != 'true') { currentBuild.result='NOT\_BUILT'; error "Not a hotfix tag" }

env.HOTFIX\_BRANCH = "hotfix/${env.VERSION}"

}

}

}

stage('Back-merge PRs') {

steps {

sh '''

set -euo pipefail

export GH\_TOKEN="${GH\_TOKEN}"

# Use hotfix branch if exists, else main as source

SRC="${HOTFIX\_BRANCH}"

if ! gh api repos/${GITHUB\_OWNER}/${GITHUB\_REPO}/branches/${HOTFIX\_BRANCH} >/dev/null 2>&1; then

SRC="main"

fi

# PR to develop

OPEN=$(gh pr list -R ${GITHUB\_OWNER}/${GITHUB\_REPO} --base develop --head "$SRC" --state open --json number -q 'length(.)')

if [ "$OPEN" -eq 0 ]; then

gh pr create -R ${GITHUB\_OWNER}/${GITHUB\_REPO} --base develop --head "$SRC" \

--title "Hotfix ${TAG} → develop" \

--body "Back-merge of hotfix ${TAG} into develop."

fi

PRD=$(gh pr list -R ${GITHUB\_OWNER}/${GITHUB\_REPO} --base develop --head "$SRC" --state open --json number -q '.[0].number' || true)

[ -n "$PRD" ] && gh pr merge -R ${GITHUB\_OWNER}/${GITHUB\_REPO} "$PRD" --auto --squash || true

# PRs to open release/\*

gh api repos/${GITHUB\_OWNER}/${GITHUB\_REPO}/branches --paginate \

-q '.[] | select(.name | startswith("release/")) | .name' > rels.txt || true

while read -r BR; do

[ -z "$BR" ] && continue

OPEN=$(gh pr list -R ${GITHUB\_OWNER}/${GITHUB\_REPO} --base "$BR" --head "$SRC" --state open --json number -q 'length(.)')

if [ "$OPEN" -eq 0 ]; then

gh pr create -R ${GITHUB\_OWNER}/${GITHUB\_REPO} --base "$BR" --head "$SRC" \

--title "Hotfix ${TAG} → ${BR}" \

--body "Back-merge of hotfix ${TAG} into ${BR}."

fi

PRN=$(gh pr list -R ${GITHUB\_OWNER}/${GITHUB\_REPO} --base "$BR" --head "$SRC" --state open --json number -q '.[0].number' || true)

[ -n "$PRN" ] && gh pr merge -R ${GITHUB\_OWNER}/${GITHUB\_REPO} "$PRN" --auto --squash || true

done < rels.txt

'''

}

}

}

}

**C) Label-driven backport → cherry-pick to release/x.y then PR**

**Job**: backport-on-label

pipeline {

agent any

environment {

GH\_TOKEN = credentials('GH\_TOKEN\_BOT')

GITHUB\_OWNER='your-org'; GITHUB\_REPO='your-repo'

}

triggers {

GenericTrigger(

genericVariables: [[key: 'payload', value: '$']],

causeString: 'GitHub PR closed'

)

}

stages {

stage('Parse & guard') {

steps {

script {

def ev = readJSON text: env.payload

if (ev.action != 'closed' || !ev.pull\_request.merged) {

currentBuild.result='NOT\_BUILT'; error "Not merged PR"

}

// find label "auto-backport:release/x.y"

def labels = ev.pull\_request.labels\*.name

def target = labels.find { it?.startsWith('auto-backport:') }

if (!target) { currentBuild.result='NOT\_BUILT'; error "No auto-backport label" }

env.TARGET\_BRANCH = target.split(':',2)[1] // release/x.y

env.BASE\_BRANCH = ev.pull\_request.base.ref // branch receiving the PR

env.MERGE\_SHA = ev.pull\_request.merge\_commit\_sha ?: ''

}

}

}

stage('Cherry-pick & PR') {

steps {

sh '''

set -euo pipefail

export GH\_TOKEN="${GH\_TOKEN}"

# Identify commit to cherry-pick: prefer merge\_commit\_sha; fallback to latest on base

SHA="${MERGE\_SHA}"

if [ -z "$SHA" ] || [ "$SHA" = "null" ]; then

git clone --quiet https://github.com/${GITHUB\_OWNER}/${GITHUB\_REPO}.git repo

cd repo

git fetch origin "${BASE\_BRANCH}"

SHA=$(git rev-parse "origin/${BASE\_BRANCH}")

else

git clone --quiet https://github.com/${GITHUB\_OWNER}/${GITHUB\_REPO}.git repo

cd repo

fi

# Ensure target exists

if ! git ls-remote --exit-code --heads origin "${TARGET\_BRANCH}" >/dev/null 2>&1; then

echo "Target ${TARGET\_BRANCH} not found; exiting."; exit 0

fi

git fetch origin "${TARGET\_BRANCH}"

git checkout -b "backport/${TARGET\_BRANCH#release/}-${SHA:0:7}" "origin/${TARGET\_BRANCH}"

set +e

git cherry-pick -x "${SHA}"

STATUS=$?

set -e

git push --set-upstream origin HEAD

cd ..

TITLE="Backport ${SHA:0:7} → ${TARGET\_BRANCH}"

BODY="Automated cherry-pick of ${SHA}."

PR\_URL=$(gh pr create -R ${GITHUB\_OWNER}/${GITHUB\_REPO} --base "${TARGET\_BRANCH}" --head "$(git -C repo rev-parse --abbrev-ref HEAD)" --title "${TITLE}" --body "${BODY}" || true)

# Try auto-merge if no conflicts

if [ $STATUS -eq 0 ]; then

PRN=$(gh pr list -R ${GITHUB\_OWNER}/${GITHUB\_REPO} --base "${TARGET\_BRANCH}" --head "$(git -C repo rev-parse --abbrev-ref HEAD)" --state open --json number -q '.[0].number' || true)

[ -n "$PRN" ] && gh pr merge -R ${GITHUB\_OWNER}/${GITHUB\_REPO} "$PRN" --auto --squash || true

fi

'''

}

}

}

}

**D) Nightly sweeper (rebase/close/auto-merge bot PRs)**

**Job**: autobot-pr-sweeper

pipeline {

agent any

environment {

GH\_TOKEN = credentials('GH\_TOKEN\_BOT')

GITHUB\_OWNER='your-org'; GITHUB\_REPO='your-repo'

}

triggers { cron('H 2 \* \* \*') } // nightly

stages {

stage('Sweep') {

steps {

sh '''

set -euo pipefail

export GH\_TOKEN="${GH\_TOKEN}"

gh pr list -R ${GITHUB\_OWNER}/${GITHUB\_REPO} --state open --json number,author,baseRefName,headRefName \

-q '.[] | select(.author.login | test("^(github-actions\\[bot\\]|auto-)"))' > prs.json || true

[ -s prs.json ] || { echo "No bot PRs"; exit 0; }

for PR in $(jq -r '.[].number' prs.json); do

BASE=$(gh pr view -R ${GITHUB\_OWNER}/${GITHUB\_REPO} "$PR" --json baseRefName -q .baseRefName)

HEAD=$(gh pr view -R ${GITHUB\_OWNER}/${GITHUB\_REPO} "$PR" --json headRefName -q .headRefName)

rm -rf repo && git clone --quiet https://github.com/${GITHUB\_OWNER}/${GITHUB\_REPO}.git repo

cd repo

git fetch origin "$BASE" "$HEAD"

git checkout "origin/$BASE"

# Close if already merged (empty diff)

if git merge-base --is-ancestor "origin/$HEAD" "origin/$BASE"; then

gh pr close -R ${GITHUB\_OWNER}/${GITHUB\_REPO} "$PR" --comment "Closing: changes already present in ${BASE}."

cd ..; continue

fi

# Rebase HEAD on BASE (best effort)

git checkout -b tmp "$HEAD" || git checkout -B tmp "origin/$HEAD"

set +e

git rebase "origin/$BASE"

REB=$?

set -e

if [ $REB -eq 0 ]; then

git push --force-with-lease origin tmp:"$HEAD"

else

git rebase --abort || true

fi

cd ..

# Re-enable auto-merge unless blocked

LABELS=$(gh pr view -R ${GITHUB\_OWNER}/${GITHUB\_REPO} "$PR" --json labels -q '.labels[].name' || true)

echo "$LABELS" | grep -Fx "do-not-automerge" >/dev/null || gh pr merge -R ${GITHUB\_OWNER}/${GITHUB\_REPO} "$PR" --auto --squash || true

done

'''

}

}

}

}

**4) (Optional) Shared Library helpers**

Create a shared lib (e.g., vars/ghAuto.groovy) so Jenkinsfiles become one-liners:

def prEnsure(String base, String head, String title, String body) {

sh """

set -euo pipefail

export GH\_TOKEN="${GH\_TOKEN}"

OPEN=\$(gh pr list -R ${GITHUB\_OWNER}/${GITHUB\_REPO} --base ${base} --head ${head} --state open --json number -q 'length(.)')

if [ "\$OPEN" -eq 0 ]; then

gh pr create -R ${GITHUB\_OWNER}/${GITHUB\_REPO} --base ${base} --head ${head} --title "${title}" --body "${body}"

fi

PR=\$(gh pr list -R ${GITHUB\_OWNER}/${GITHUB\_REPO} --base ${base} --head ${head} --state open --json number -q '.[0].number' || true)

[ -n "\$PR" ] && gh pr merge -R ${GITHUB\_OWNER}/${GITHUB\_REPO} "\$PR" --auto --squash || true

"""

}

return this

Then in a Jenkinsfile:

@Library('your-shared-lib') \_

pipeline {

agent any

environment { GH\_TOKEN = credentials('GH\_TOKEN\_BOT'); GITHUB\_OWNER='your-org'; GITHUB\_REPO='your-repo' }

stages {

stage('Auto PRs'){ steps { script {

ghAuto.prEnsure('main', "release/${LINE}", "Release → main", "Publish ${TAG}")

ghAuto.prEnsure('develop', "release/${LINE}", "Release → develop", "Sync ${TAG}")

}}}

}

}

**Tips / gotchas**

* **Permissions:** GitHub App tokens are safer than PATs; configure Jenkins “GitHub App” credentials if you prefer and swap GH\_TOKEN usage accordingly.
* **Idempotency:** All scripts **check for an existing PR** before creating.
* **Merge policy:** Keep required checks minimal-but-meaningful on main/release/\*, and use **merge queue** if you mirror it via GitHub settings.
* **Security:** Lock who can trigger these jobs (webhook token, folder auth).
* **Monorepo:** Add path filters in your webhook or early in the pipeline to only act when specific service folders changed.

If you tell me your Jenkins version, plugin set, and whether you prefer **GitHub App** over PAT, I’ll tweak the snippets exactly to your environment.

develop ← feature/\* or fix/\* branches

│

│ (features, fixes merged continuously)

▼

release/x.y (optional)

│ → CI runs extended tests, UAT, etc.

▼

tag vX.Y.0

│

├─→ merge to main (publishes)

└─→ merge to develop (syncs hotfixes & docs)

🔹 Developer experience (simple mental model)

Daily work:

“Always branch off develop, always PR into develop.”

Before release:

“We’ll cut release/x.y on Friday; freeze new features.”

“Fix bugs in that branch only until it stabilizes.”

After release:

“Tag vX.Y.0. The bot merges release → main → develop. Done.”

Hotfix:

“Branch from main, PR to main, bot back-merges to develop. Done.”

That’s it.

🔹 Example release schedule (bi-weekly)

Day Activity

Mon–Thu (Week 1) Devs merge PRs → develop. CI auto-deploys to staging env.

Fri (Week 1) Cut release/x.y. Freeze feature merges. Only bugfix PRs allowed.

Mon–Wed (Week 2) Test & stabilize.

Thu (Week 2) Tag vX.Y.0 on release/x.y. CI deploys to prod.

Fri (Week 2) Autobot merges release → main → develop. Delete release branch.

Mon (Week 3) Continue new work on develop.

🤖 Automation (minimal but powerful)

You already have most of this from your Autobot flows earlier.

Here’s how it applies in GitFlow-Lite:

Event Automation Action

Tag on release/x.y Open PRs: release/x.y → main (publish), release/x.y → develop (sync). Auto-merge.

Hotfix tag on main Open PRs: hotfix/x.y.z → develop. Auto-merge.

Nightly sweeper Rebase & close stale bot PRs.

Backport label (auto-backport:release/x.y) Cherry-pick a merged fix from develop into the active release. Auto-merge if green.

Release creation Auto-generate changelog & attach artifact (image digest, SBOM).

✅ Benefits (keeps GitFlow’s strengths)

GitFlow Benefit Still retained in Lite version

Controlled releases with change freeze ✅ via short-lived release/x.y

Hotfix isolation and quick rollback ✅ still separate hotfix/x.y.z

Stable production branch ✅ main only updated via tags

Predictable integration branch ✅ develop as next release

Auditable release tags ✅ vX.Y.Z created and linked

Reduced merge chaos ✅ one PR target for developers

Easy automation hooks ✅ all flows can be event-driven (tag, label, schedule)

🧭 Governance checklist (simple guardrails)

🔒 Protect main and develop (require reviews & checks).

🧱 Allow only bugfix PRs to release/\*. Enforce via branch rule or PR template.

🚦 Require successful build/test on all PRs.

🧾 Auto-generate release notes from conventional commits.

🕒 Delete release/\* branches automatically post-release.

🧩 Optional enhancements

Feature flags → allow merging incomplete features without blocking release.

Version bump automation → derive version from tags or CHANGELOG.md.

Multi-service release orchestration → group tags with a release.json manifest.

Dependency gating → detect when Service A’s release depends on Service B’s version.

🧭 TL;DR Summary

Concept GitFlow Classic GitFlow-Lite (Recommended)

Branches main, develop, many releases main, develop, 1 short-lived release

Merge flow develop → release → main → develop develop → release (auto to main & back)

Hotfix main → develop/release main → develop (auto)

Tags on release branch same

Dev PR target depends (develop or release) always develop

Automation optional core to reduce cognitive load

Ideal for Monoliths, big-bang releases Microservices, scheduled releases