

Android Cast — Git flow (green_{master})

From GIT_FLOW.md

Field	Value
Author	Anton Afanasyeu
Revision	R1
Creation date	2026-05-23
Last modification date	2026-05-23
Co-authored	
Severity	medium
State	completed
Document type	technical

Table of contents

1. Goals
 2. Branches
 3. Branch diagram (steady state)
 4. Daily development
 - 4.1 Fast track on next (alpha sprint)
 5. Release: next → master
 6. Hotfixes — chosen strategy
 - Default: hotfix from master (option 1) ✓
 - Exception: fix only on next (option 2)
 7. Protection rules (CI + policy)
 - 7.1 How to protect branches (not local hooks alone)
 8. Sync cheat sheet
 9. Agent / Cursor checklist
 10. Comparison to classic Git Flow
 11. First-time setup (maintainer)
 12. Private primary + GitHub publish (dual remote)
 - 12.1 Add GitHub without replacing origin
 - 12.2 What to push where
 - 12.3 Daily habit
 - 12.4 Where branch protection lives
 - 12.5 One-way mirror (optional automation)
 - 12.6 Suggested order of operations (your situation)
- Related

Source: `GIT_FLOW.md` — section links work in PDF viewers that support internal anchors.

This document defines how we develop, integrate, release, and hotfix. **Cursor agents and humans follow the same rules.**

Field	Value
Version	1.4
Published	2026-06-08
Author(s)	Anton Afanaasyeu
Markdown source	docs/GIT_FLOW.md
PDF	docs/GIT_FLOW.pdf
Diagram sources	docs/_pdf_build/git_flow_diagram_*.mmd (render via build_git_flow_pdf.py)

1. Goals

Goal	How
Always shippable master	Only merges that passed CI/tests; tagged releases point here
Safe integration	Day-to-day work lands on next first
Simple mental model	Two long-lived branches + short-lived topic branches
Recoverable hotfixes	Production fixes branch from what users actually run

2. Branches

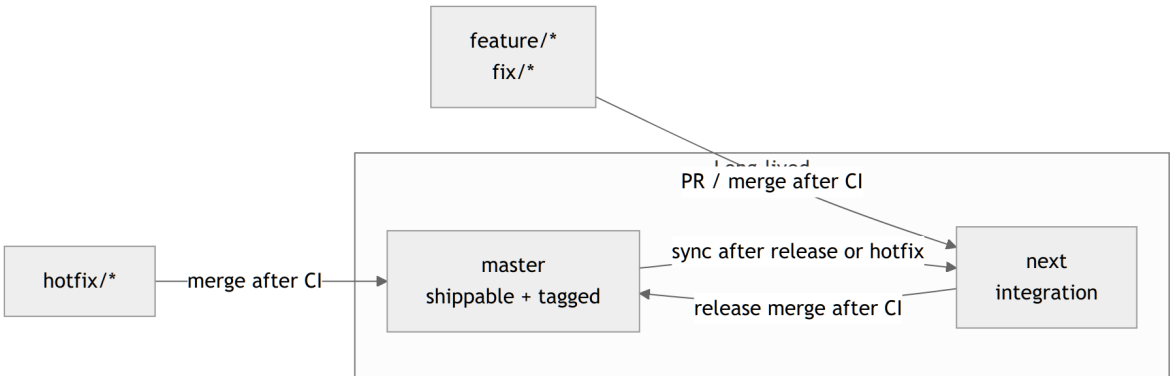
Branch	Role	Direct commits?
master	Production line; green (build + tests pass); release tags (v0.1.2)	No — merge only
next	Integration / release candidate; absorbs features until a release is approved	No — merge only (exception: alpha sprint direct commits when maintainer/agent explicitly commits here — §4.1)
feature/*, fix/*, topic/	Short-lived work; fork next	Yes (normal dev)
hotfix/	Urgent fix for released code	Yes; merge target master first

Naming: feature/cast-resolution-presets, fix/crash-upload-500, hotfix/v0.1.1-pin-crash.

If next does not exist yet on the remote:

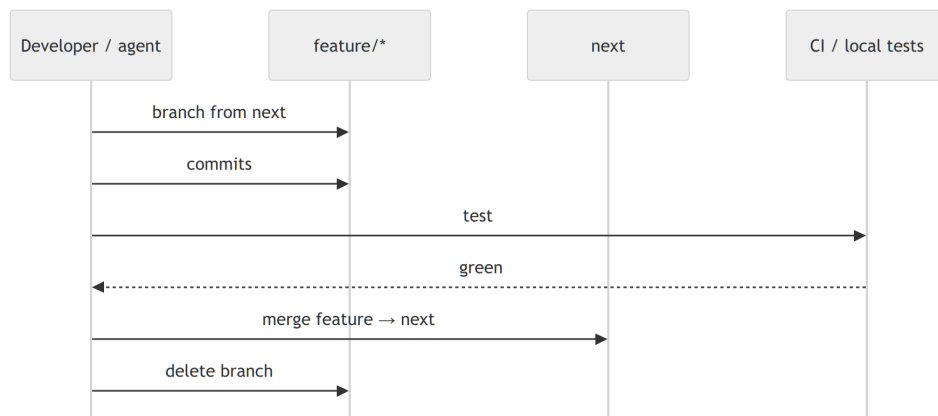
```
git checkout master
git pull
git checkout -b next
git push -u origin next
```

3. Branch diagram (steady state)



4. Daily development

- Start from next:** `git fetch && git checkout next && git pull`
- Create topic branch:** `git checkout -b feature/my-change`
- Commit** in small logical chunks on the topic branch.
- Before merge:** run project checks (Android: `./gradlew :app:testDebugUnitTest` or CI equivalent; backend: relevant tests/lint).
- Merge to next:** prefer merge commit or squash per team habit; **do not** merge unfinished work.
- Topic branch cleanup:** optional locally (`git branch -d feature/...`). On the server, prefer **delete branch on merge** (Gitea/GitLab/GitHub) so merged `feature/*` / `fix/*` are removed on origin automatically.



4.1 Fast track on next (alpha sprint)

During intensive alpha work, the team may commit **directly on next** instead of short-lived feature/* branches — to move faster and avoid branch overhead.

Trade-off: integration history on next is harder to read; **mitigation:** atomic, revert-friendly commits.

Rule	Why
One feature (or one logical slice) per commit	git revert <hash> removes only that slice
Do not mix unrelated areas in one commit (e.g. RSSH + graphs + stream dump)	Avoids reverting good work when one feature misbehaves
Commit message names the feature	e.g. dev: stream dump writer (FR 0.1) — easy to find in git log
Tests/docs for that slice in the same commit when small	Keeps revert self-contained
Prefer revert over edit when a feature is rejected	git revert on next; do not leave half-removed code across commits

Good split (three commits on next):

```

dev: network self-test – fix BACKEND head/uplink probes
dev: diagnostics – full advertisement info card
be: config.example mail stub for 2FA SMTP

```

Bad (one commit):

```

alpha wip: self-test, diagnostics, mail config, ingest script, docs

```

Revert one feature:

```

git log --oneline -20          # find the commit
git revert <hash>              # new commit undoing only that change

```

Agents: only commit to next when the user asks. When they do, follow the atomic rules above. Default remains **topic branch from next** for longer or risky work.

5. Release: next → master

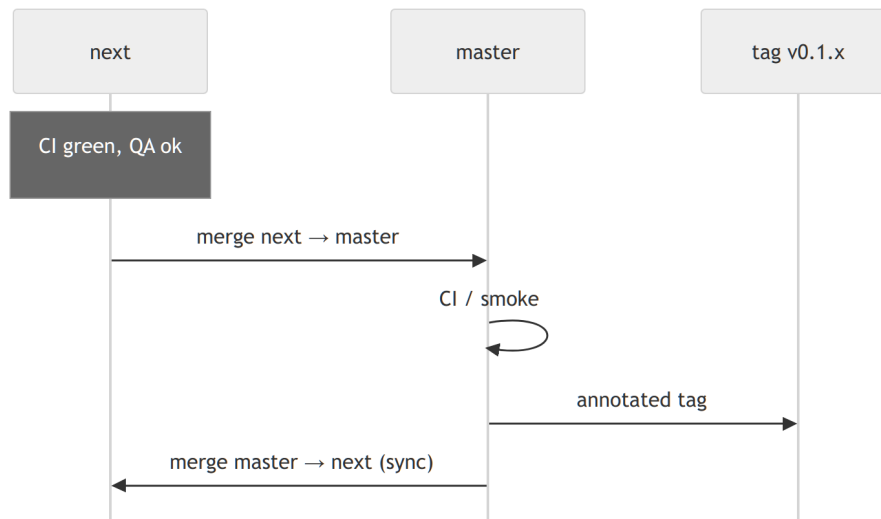
When: next is feature-complete for the version, soak/QA agreed, CI green.

1. Ensure next is up to date and **all tests pass**.
2. **Merge** next → master (merge commit recommended for traceability).
3. On master: final smoke / CI.
4. **Tag** on master: git tag -a v0.1.2 -m "Release v0.1.2" then git push origin master --tags.
5. **Sync** next **with** master so both lines share the release commit and any last-minute release fixes:

```

git checkout next
git merge master # fast-forward or merge commit
git push origin next

```
6. Continue new work: branch **feature/* from next** again.



Versioning: tags on master only (vMAJOR.MINOR.PATCH). Bump per SemVer conventions you adopt for the app.

6. Hotfixes — chosen strategy

Default: hotfix from master (option 1) ✓

Use when the bug affects **what is already released** (code on master / latest tag).

1. `git checkout master && git pull`
2. `git checkout -b hotfix/v0.1.2-crash-on-stop`
3. Fix + tests
4. Merge `**hotfix/*` → `master**`, CI green
5. **Tag** patch on master: e.g. `v0.1.3`
6. `git checkout next && git merge master` — bring hotfix into integration (resolve conflicts if next diverged)
7. Push master, next, tags; delete `hotfix/*`

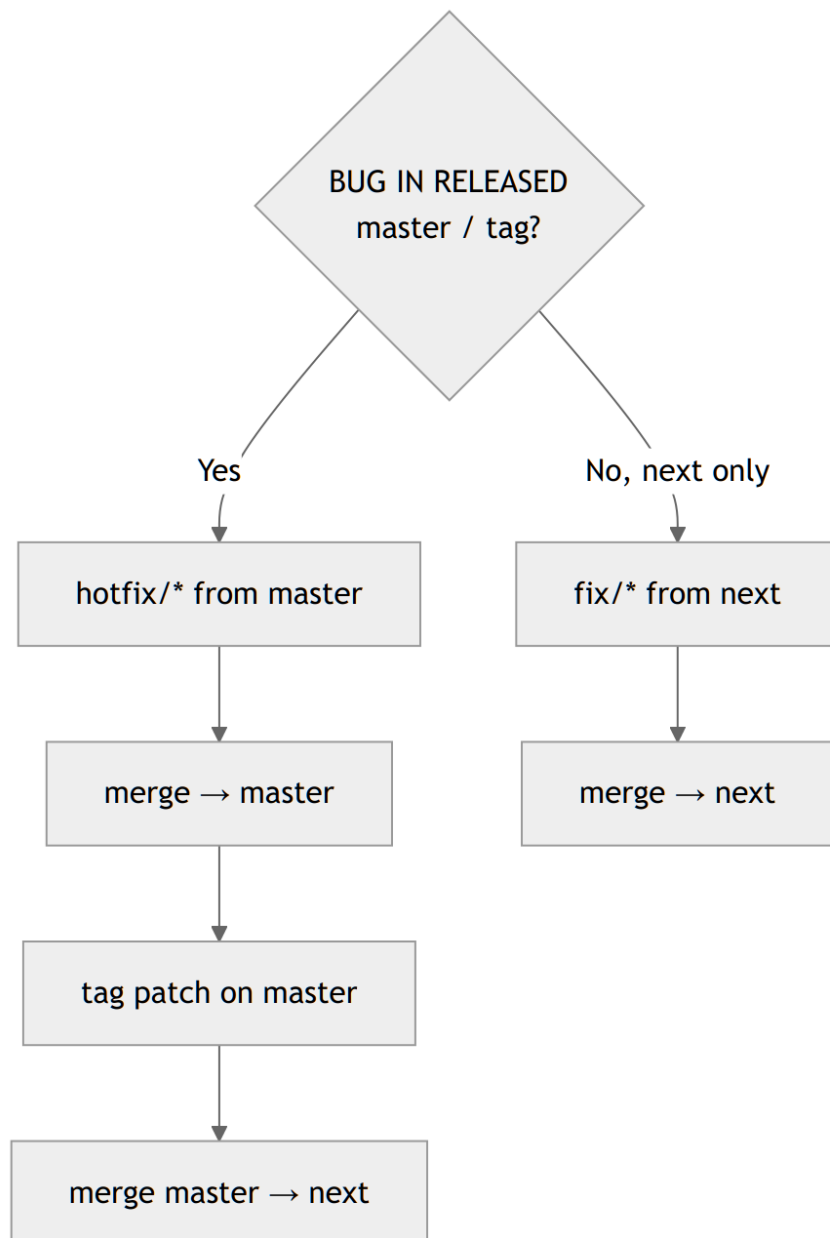
Why not hotfix only on next (option 2)?

`next` often contains unreleased features. Fixing there delays shipping the patch and risks dragging unrelated commits into production when you later merge `next` → `master`.

Exception: fix only on next (option 2)

Use when the bug **does not exist on** master (introduced on next only).

1. `git checkout -b fix/...` from next
2. Merge to next only
3. No tag until the normal **release** merge `next` → `master`



7. Protection rules (CI + policy)

Rule	master	next	topic branches
Required CI (build + tests) before merge	✓	✓	✓ (before PR/merge)
No direct pushes of WIP	✓	✓	—
Only maintainers merge to master	✓	optional	—
Tag only on master	✓	—	—
Force-push	never	maintainer only, rare	allowed on topic branches

See [§7.1 How to protect branches](#) below.

7.1 How to protect branches (not local hooks alone)

Short answer: use **server-side** rules on `f0xx.org` (or whatever hosts origin). **Local git hooks** on your laptop are optional helpers only — they do not stop you or someone else from pushing bad history from another machine.

Layer	Where it runs	Stops bad pushes?	Good for
Hosting UI / branch protection	Git server (GitHub, GitLab, Gitea, ...)	Yes	Most teams; use this if available
Server git hooks (pre-receive / update)	Bare repo on server	Yes	Self-hosted git : // SSH without a web UI

Local hooks (.git/hooks/pre-push)	Your PC only	No (easy to skip)	Personal reminders, format/tests before push
CI (build on push/PR)	CI runner	Indirect	Prove tests passed; pair with "required check" on server

What to enable on master and next

1. master: **never allow force-push** — production history must not be rewritten.
2. next: **force-push off by default** — enable only for maintainers when you intentionally clean integration history (squash/rebase of already-merged work). Prefer fixing history on feature/* before merging to next. After a rare next force-push, anyone with a clone must reset: `git fetch origin && git reset --hard origin/next`.
3. **Delete branch on merge** on the server — merged feature/* / fix/* disappear from origin without a manual delete step.
4. **No direct push** (optional but strong): changes only via merge from feature/* or from PR/MR.
5. **Require CI green** before merge (if you have Actions/GitLab CI/Gitea Actions).
6. **Restrict who can push** to you (and CI deploy key if needed).

feature/* branches stay **unprotected** — push freely.

If your host has a web UI (Gitea / GitLab / GitHub)

Look for **Branch protection** / **Protected branches**:

- Add rules for master and next.
- Enable: prevent force-push, require pull request (optional), require status checks.

No git internals required — the server rejects illegal pushes.

If you only have bare git on a server (git:// / SSH)

Ask whoever administers `git://f0xx.org/android_cast` to install **server-side hooks** in the **bare repository** (not in your clone):

- hooks/pre-receive or hooks/update — script receives branch name + old/new SHA; **exit 1** to reject (e.g. block direct push to refs/heads/master unless user is deploy; block all force-push to master/next).

Example policy in words: "reject push if ref is master or next and user is not in allowlist" or "reject non-fast-forward updates to master/next".

Local clone hooks live in .git/hooks/ and are **not copied** when others clone — so they are **not** team protection.

Local hooks (optional, for you only)

Useful on your machine, not security:

```
protected_regex='^(refs/heads/)?(master|next)$'
while read local_ref local_sha remote_ref remote_sha; do
  if echo "$remote_ref" | grep -qE "$protected_regex"; then
    echo "Blocked: push to master/next - use feature branch + merge (see docs/GIT_FLOW.md)"
    exit 1
  fi
done
```

Skip with `git push --no-verify` — another reason server rules matter.

CI without branch UI

Run tests on every push; you still **choose** not to merge until green. Branch protection + "required check" automates that on modern hosts.

8. Sync cheat sheet

Event	Action
After release (next → master + tag)	<code>git checkout next && git merge master && git push</code>
After hotfix to master + tag	Same: merge master → next
Feature finished	merge topic → next only
Started wrong base	rebase topic onto latest next (coordinate if already shared)

9. Agent / Cursor checklist

Agents working in this repo **must**:

1. **Never** commit or push directly to master or next unless the user explicitly asks for that merge/release step.
2. **Branch from** next for features and non-production fixes (feature/*, fix/*).
3. **Run tests** (or state clearly if not run) before recommending merge to next.
4. **Releases**: only describe merging next → master and tagging after user confirms QA.
5. **Hotfixes on released code**: branch from master, merge back to master, tag, then **merge** master **into** next.
6. **Set active branch** metadata when using a named topic branch (project convention).
7. **Do not** force-push master; avoid force-push on next unless the user explicitly requests integration history rewrite.
8. **Fast track on** next: when the user commits directly on next (alpha sprint), use **atomic commits** — one feature per commit so `git revert <hash>` can drop a bad feature without touching others (§4.1).

10. Comparison to classic Git Flow

Git Flow	This repo
main	master (green, tagged)
develop	next
feature/*	same, base = next
release/*	optional short branch; default = test on next then merge to master
hotfix/*	from master, merge to master, sync to next

We intentionally avoid a permanently divergent develop without a green production branch: master **is always releasable**.

11. First-time setup (maintainer)

```
git checkout master && git pull
git checkout -b next 2>/dev/null || git checkout next
git push -u origin next
```

12. Private primary + GitHub publish (dual remote)

Canonical repo: your server — today origin → git://f0xx.org/android_cast.

Daily work: always git push / git pull against origin first.

GitHub: secondary remote for visibility, issues, and (optional) GitHub Actions — not the source of truth unless you decide otherwise.

12.1 Add GitHub without replacing origin

```
git remote add github git@github.com:YOUR_USER/android-cast.git
git remote -v
```

Keep origin = private server. Name the public remote github (or publish) so commands stay obvious.

12.2 What to push where

Ref	Private origin	GitHub github
feature/*, WIP	✓ always	optional (omit for cleaner public)
next	✓ always	your choice: public integration or keep private
master + tags	✓ always	✓ publish releases you want visible
Secrets, local.properties, logs	never in either	never

Typical OSS-style: push everything to origin; push only master and **tags** to github until you are ready to open next.

```
git push origin master --tags
git push github master --tags
```

Push next to GitHub only when you want public integration:

```
git push origin next
git push github next
```

12.3 Daily habit

```
git fetch origin
git checkout next && git pull origin next
git push -u origin feature/foo
git push origin next
```

```
git push github master --tags
```

Set upstream on topic branches to **origin**, not github:

```
git push -u origin feature/my-change
```

12.4 Where branch protection lives

Server	Protect
f0xx.org (origin)	master, next — server hooks or admin UI (primary enforcement)
GitHub (github)	master (and next if public) — Settings → Branch protection

GitHub rules only affect pushes **to GitHub**. They do not protect your private server unless you also configure origin.

12.5 One-way mirror (optional automation)

On the private server (cron or post-receive hook), after a successful push to origin:

```
git push --prune github +refs/heads/master:refs/heads/master +refs/tags/*:refs/tags/*
```

Or use GitHub's "mirror repository" importers only for initial import; ongoing sync is usually git push github ... from CI or hook.

Avoid git push github --mirror unless GitHub is a full duplicate of **private** history — it copies all branches and deletes remote branches you deleted locally.

12.6 Suggested order of operations (your situation)

1. On **f0xx.org**: create next from master, push origin next.
2. On **f0xx.org**: enable server protection for master / next (admin).
3. Create **GitHub** repo; git remote add github
4. First publish: git push github master and tags you want public.
5. Enable **GitHub branch protection** on master (block force-push).
6. Continue feature flow on origin only; publish to GitHub when you merge/tag on master.

Related

- README.md — documentation index (all guides)

- ROADMAP.md — product tracks
- .cursor/rules/git-flow.mdc — agent enforcement
- docs/_pdf_build/build_git_flow_pdf.py — PDF generator