RABBIT HOLES

Accelerate · Manipal University Jaipur

15

PHASE

Being a maintainer

The job is mostly not code. Governance, review, saying no, and not burning out.

Contents

1 What the job actually is 3

2 The files that answer questions for you 3

3 Licensing 4

3.1 Contributions and provenance 5

4 Protecting the branch 5

4.1 CODEOWNERS 6

5 Reviewing well 6

5.1 What to actually look for, in order 7

5.2 How to say it 7

5.3 Receiving review 7

6 Triage 8

6.1 The triage question 8

7 Saying no 8

7.1 Wording that works 8

8 Releases and not breaking people 9

9 Security 9

9.1 Dependencies 10

10 Growing contributors 10

11 Governance 11

12 Burnout 11

13 Endings 12

1 What the job actually is

In Phase 3 the team leader merged pull requests. Every time they merged one, everyone else's branch went stale and somebody had to rebase. That small annoyance is the whole job in miniature: a maintainer's decisions are mostly decisions about other people's time.

The surprise, for anyone who takes on a project people actually use, is how little of the work is programming. The realistic split is closer to this:

Deciding What belongs in this project and what does not. The most valuable and least visible thing you do.

Reviewing Reading other people's code carefully enough to be useful, and quickly enough to be worth submitting to.

Explaining The same five questions, forever, to people who each ask once.

Triaging Sorting the incoming pile into real, duplicate, unclear, and not our problem.

Releasing Versioning, notes, and not breaking people.

Refusing Politely, clearly, and often.

Coding Whatever is left.

Nobody warns you about this, and it is the main reason maintainers quit.

> **WHY THIS EXISTS**
>
> This phase exists because Accelerate runs an organisation with repositories that outlive their authors' membership. The people who wrote `campus-mapper` will graduate. Somebody will inherit it. Everything here is the difference between inheriting a project and inheriting a mess.

2 The files that answer questions for you

Every question you answer twice should be answered in a file instead. GitHub has specific names it recognises and surfaces automatically.

File	Answers
`README.md`	What is this, why would I use it, how do I start
`CONTRIBUTING.md`	How do I submit a change and what will you reject
`CODE_OF_CONDUCT.md`	How people are expected to behave, and enforcement
`SECURITY.md`	How to report a vulnerability privately
`SUPPORT.md`	Where to ask questions that are not bugs
`LICENSE`	What others may legally do
`CODEOWNERS`	Who must review which files
`FUNDING.yml`	Adds a Sponsor button
`CITATION.cff`	How to cite this in a paper

Put them in the repository root or in `.github/`. An organisation-level `.github` repository provides defaults for every repository that lacks its own, which is exactly why Accelerate has one.

> **NOTE**
>
> The highest leverage sentence in `CONTRIBUTING.md` is the one describing what you will *not* accept. Scope creep arrives as a well-written, working pull request from someone who spent their weekend on it, and rejecting that is miserable for everyone. A sentence up front prevents the whole situation.

3 Licensing

> **TRAP**
>
> A public repository with no LICENSE file is **not** open source. Under copyright law, code with no licence is all rights reserved. Nobody may legally use, modify, or distribute it. GitHub's terms let others view and fork within GitHub, and that is all. Every serious company's legal team blocks dependencies with no licence. If you want people to use your work, add a licence on day one.

Licence	In short
MIT	Do what you like, keep the notice, no warranty. Short and universally understood. The default for student and club projects.
Apache 2.0	Like MIT, plus an explicit patent grant and a requirement to state changes. Preferred by companies for that patent clause.
BSD 3-Clause	Like MIT, plus you may not use the authors' names to endorse.
GPL v3	Anyone distributing a derivative must release their source under GPL too. Copyleft.
AGPL v3	GPL, and running it as a network service counts as distribution. Closes the cloud loophole.
MPL 2.0	Copyleft per file. Modified files stay open, the larger work need not be.
CC BY-SA	For documentation and creative work, not code.

```
gh repo create project --public --license mit
gh api repos/:owner/:repo --jq '.license.spdx_id'
```

3.1 Contributions and provenance

When someone contributes, who owns it? By default they retain copyright and licence it to you under the project's licence. Two mechanisms formalise it:

DCO, the Developer Certificate of Origin, is a statement that you wrote the code and may submit it, asserted by signing off commits:

```
git commit -s -m "Add parser"
```

That appends `Signed-off-by: Name <email>`. Lightweight, and enough for most projects, including the Linux kernel.

CLA, a Contributor Licence Agreement, is a real contract, usually assigning broader rights to a company. Heavier, and a genuine barrier: a first-time contributor fixing a typo will not sign a legal document, and you lose that contribution.

For a student club, DCO or nothing. A CLA is a corporate tool for corporate reasons.

4 Protecting the branch

```
gh api -X PUT repos/:owner/:repo/branches/main/protection \
  -F required_pull_request_reviews[required_approving_review_count]=1 \
  -F required_pull_request_reviews[dismiss_stale_reviews]=true \
  -F required_status_checks[strict]=true \
  -F required_status_checks[contexts][]=test \
```

```
  -F enforce_admins=false \
  -F restrictions=null
```

The options worth understanding:

Required reviews How many approvals before merging. One is usually right. Two on a project where a mistake is expensive.

Dismiss stale reviews New commits invalidate old approvals. Turn this on. Without it, someone can get approval and then push something else.

Require branches up to date The `strict` flag. Prevents a green pull request merging into a `main` that has moved. Correct, and it causes the constant rebasing that merge queues solve.

Required status checks Which CI jobs must pass.

Enforce admins Whether the rules apply to you. Leaving this off is a pragmatic escape hatch and a slow corrosion of the rules. Turn it on once the project is stable.

Require signed commits Phase 10.

Require linear history Blocks merge commits, forcing squash or rebase.

Require conversation resolution All review comments resolved before merge.

> NOTE
>
> **Rulesets** are the newer system and are better. They can apply to several repositories at once, target branches by pattern, be evaluated in a dry-run mode before enforcement, and layer rather than override. If you are setting this up fresh in an organisation, use rulesets rather than per-repository branch protection.

4.1 CODEOWNERS

```
*                       @accelerate-muj/core-committee
/handbook/              @TheQMLGuy
/.github/workflows/     @TheQMLGuy @shashwat-deep
*.tex                   @TheQMLGuy
/phase-2/poems/
```

Last matching rule wins, unlike `.gitignore`. The empty last line means poem submissions need no specific owner, which is what you want during a workshop with thirty simultaneous pull requests.

Combined with “require review from code owners” in branch protection, the right person is requested automatically and nothing merges without them.

5 Reviewing well

Code review is a skill and most people are never taught it. Reviewing badly is worse than not reviewing.

5.1 What to actually look for, in order

1. **Is this the right change at all?** Wrong-problem is the most expensive thing to catch late and the cheapest to catch here. Ask before reading closely.
2. **Does it work?** Check it out and run it. `gh pr checkout 42`.
3. **What happens at the edges?** Empty input, no network, concurrent runs, huge input.
4. **Can the next person understand it?** Including the author in a year.
5. **Is it tested?** Enough that a future change breaking it is caught.
6. **Style.** Last, and ideally not by you. A formatter should own this.

5.2 How to say it

> **TRAP**
>
> The two failure modes are the rubber stamp and the pile-on. Both are common and both drive contributors away.
> Things that help, concretely:
>
> - **Say what is good.** Not flattery. Reviews that are exclusively negative read as hostile even when every point is correct.
> - **Separate must from nice.** Prefix optional remarks with `nit:` or `optional:`. Without that signal every comment reads as blocking and a two-line change takes a week.
> - **Ask rather than assert.** "What happens if this is empty?" beats "this crashes on empty input", because sometimes it does not and you were wrong.
> - **Explain the why.** "Use a set here" teaches nothing. "A set makes this lookup constant time, and this runs per request" teaches once and forever.
> - **Use suggested changes** for anything trivial. A one-click fix is kinder than a paragraph.
> - **Review promptly or say you cannot.** A pull request sitting untouched for two weeks is the single most common reason people stop contributing. "I cannot look at this until Friday" costs you ten seconds and keeps them.

```
gh pr diff 42
gh pr checkout 42
gh pr review 42 --comment --body "Two questions inline, nothing blocking"
gh pr review 42 --approve --body "Nice catch on the empty case"
gh pr review 42 --request-changes --body "See the note about the token"
```

5.3 Receiving review

The other half, and it is a skill too. The reviewer is not attacking you. If a reviewer misread your code, that is evidence the code is easy to misread, and the useful response is usually a clearer name or a comment rather than an explanation in the thread that nobody will ever read again.

Push back when you disagree. A review is a conversation, not a verdict. Just do it about the technical point.

6 Triage

Incoming issues are not a to-do list. They are unsorted input, and about half are not work.

A workable label scheme, kept small:

```
type:    bug  feature  docs  question
status:  needs-info  confirmed  blocked  wontfix  duplicate
effort:  good-first-issue  help-wanted
```

```
gh issue list --label "status:needs-info" --json number,updatedAt
gh issue edit 42 --add-label "status:confirmed" --milestone "v1.1"
gh issue close 17 --reason "not planned" --comment "Duplicate of #12"
```

6.1 The triage question

For every new issue, in order: *Can I reproduce it? Is it already reported? Is it in scope? Is it actionable as written?*

Anything failing the last test gets one request for specifics and a `status:needs-info` label. If it goes quiet for a few weeks, close it kindly. Closing is not rejection, and reopening is free.

> **TRAP**
>
> Stale bots that close inactive issues automatically are widely used and widely resented. From the maintainer's side it keeps the list honest. From the reporter's side, being told by a robot that a real, unfixed bug is now closed for inactivity is insulting, and it is a recurring source of public complaints about major projects.
> If you use one, exclude confirmed bugs, give a long window, and write the message like a person. Better: do not close on inactivity at all, and use a label plus a saved search instead. An issue nobody has fixed is still true.

7 Saying no

The hardest part of the job, and the one that decides whether your project stays maintainable.

Every accepted feature is permanent. You will maintain it, document it, keep it working through refactors, and break someone if you remove it. A feature used by four percent of users costs you forever.

7.1 Wording that works

For an out-of-scope request:

> *Thanks for writing this up. I do not think this belongs in the core project, because it only applies when you are using X and it would add a dependency for everyone else. It would make a good extension, though, and I would happily link it from the README.*

For a pull request nobody asked for:

> *I appreciate the work here and I am not going to merge it, so let me explain why rather than leaving it open. This changes the default behaviour, and there are people relying on the current default. If you want to pursue it, open an issue and we can talk about a migration path.*

The pattern in both: decide, say so plainly, give the actual reason, offer a path forward if there is one. Do not soften it into ambiguity.

> **TRAP**
>
> The worst thing you can do is leave it open. An unanswered pull request is a person waiting, checking, and slowly concluding they were ignored. A clear no on the day it arrives is far kinder than silence for six months, and people generally respect it.
> If you are not sure, say that: "I need to think about this, give me a week." Then actually do.

8 Releases and not breaking people

Phase 10 covered the mechanics. The judgement is here.

Compatibility is a promise. Once someone depends on your project, changing behaviour breaks their build at a time you chose and they did not. Semantic versioning is how you communicate that in advance.

Deprecate before removing. The sequence that works:

1. Release a version where the old way still works but warns, and the new way exists.
2. Document the migration, with a before and after example.
3. Leave it for at least one minor release, longer if the project is widely used.
4. Remove it in the next MAJOR, and say so prominently in the notes.

Write release notes for humans. A list of commit messages is not release notes. Lead with what changed for the user, then what to do about it, and put the full log at the bottom for anyone who wants it.

9 Security

`SECURITY.md` states which versions you support and how to report privately.

> **TRAP**
>
> Never ask people to report vulnerabilities as public issues. The whole point is a window to fix it before it is exploited.
> GitHub has **private vulnerability reporting**, free, which gives you a private thread and a place to draft an advisory and request a CVE:
>
> ```
> gh api -X PUT repos/:owner/:repo/private-vulnerability-reporting
> ```
>
> Turn it on for anything anyone else runs.

The process when a report arrives: acknowledge within a day or two even if only to say you have seen it; agree a disclosure date with the reporter, ninety days being the industry norm; fix privately using a draft security advisory, which gives you a private fork; release the fix and publish the advisory at the same moment; credit the reporter, because it costs nothing and it is the entire compensation most researchers get.

9.1 Dependencies

```
version: 2
updates:
  - package-ecosystem: "pip"
    directory: "/"
    schedule: {interval: "weekly"}
    open-pull-requests-limit: 5
    groups:
      dev-dependencies:
        patterns: ["pytest*", "ruff"]
```

Group updates, or Dependabot will file eleven pull requests a week and you will start ignoring them, which is worse than not having it.

10 Growing contributors

A project with one maintainer has a bus factor of one. Growing that is a deliberate activity.

Label real starter issues `good first issue` on something genuinely small, with the file named and the approach sketched. A vague hard task with a friendly label is worse than nothing.

Answer the first pull request properly Someone's first contribution is disproportionately important. Review it that week. Be generous about small things.

Promote people Someone who has landed several good changes and reviews others should be given commit access before they ask. Most people will never ask.

Write down the tacit knowledge The reason for the odd workaround, the thing that always breaks in CI. `CONTRIBUTING.md`, or an `ARCHITECTURE.md`.

Let go of things The instinct to fix it yourself because it is faster is correct in the short run and fatal in the long run.

11 Governance

Small projects run on "whoever started it decides". That works until it does not.

BDFL One person has final say. Efficient, clear, and a single point of failure in every sense.

Meritocracy or committer model Contributors earn commit access over time, decisions by rough consensus. Most mid-sized projects.

Elected steering committee Formal, elections, defined terms. Overhead worth paying once enough people depend on it. Accelerate's constitution is this.

Foundation Neutral legal owner, for projects where several companies need assurance nobody can capture it.

Whatever you use, write it down. Unwritten governance is the founder's preferences, and it is invisible until the first real disagreement, at which point everyone discovers they believed something different.

12 Burnout

The predictable ending. Open source maintainer burnout is common enough to be studied, and the pattern is consistent: growing obligation, static reward, no boundary between the project and the rest of life.

What actually helps:

Set office hours Fixed times you look at the project. Outside them, do not. The alternative is a background hum of obligation that never stops.

Say the project is finished Software can be done. "Complete, accepting bug fixes only" in the README is a legitimate and underused state.

Reduce the surface Fewer features, fewer platforms, fewer promises. Every reduction is permanent relief.

Recruit before you need to Phase 15's whole previous section.

Turn off notifications Genuinely. Phase 12.

Remember what you owe You published something for free. You do not owe anyone support, a fix, a response, or your weekend. Entitled demands are the other person being wrong.

TRAP

The specific thing to watch for is a project that has become an obligation you resent and cannot put down. The exits are all fine: hand it over, add maintainers, mark it unmaintained, or archive it.
Archiving is honest and much better than abandoning it silently while people keep filing issues.

```
gh repo archive old-project
```

Say why in the README, and point at a fork if a good one exists.

13 Endings

Handing over. Add them as an admin, let them merge things for a while, then transfer. `Settings, Transfer ownership` moves it with issues, pull requests and stars intact, and leaves a redirect. Never delete and re-upload, which destroys all of that.

Being forked. Anyone may fork your project. If someone forks because they disagree with your direction, that is the licence working as intended, and it is not a betrayal. If the fork is better maintained than yours, the honest move is to link it.

Succession for a club. Accelerate's specific case. Every repository should name a current owner, and that name should be updated when officers change. A repo whose owner graduated two years ago is unmaintained regardless of what the settings say.

CHECK YOURSELF

1. What can people legally do with a public repository that has no LICENSE?
2. Why turn on "dismiss stale reviews"?
3. Name three things that make a code review useful rather than discouraging.
4. Why is leaving a pull request open worse than rejecting it?
5. What are the four steps to remove a feature without breaking people?
6. Someone reports a vulnerability in a public issue. What now?
7. Your project has one maintainer and it is you. What is the actual risk, and what reduces it?

Next: Phase 16, the last one, on the parts of git that only show up at scale, and on where git came from, which explains several things that look strange until you know.