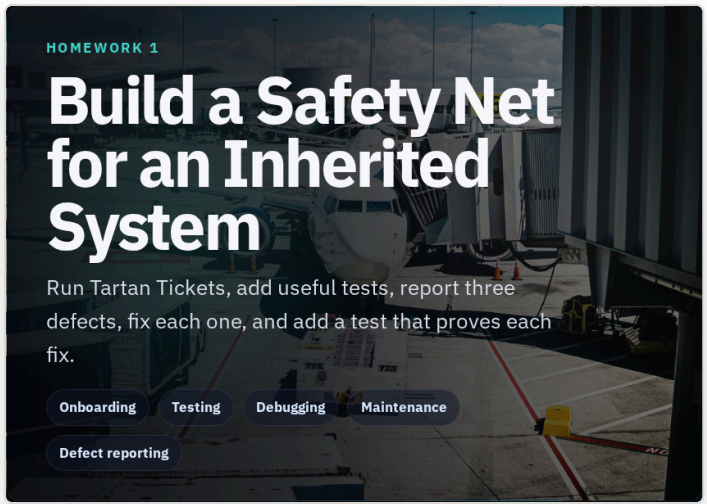


Source Code Management and Version Control

Chris Timperley · Monday, August 31, 2026

Administrivia: HW1: Build a Safety Net



Two graded checkpoints

A — Starting Point and Defect Reports
Sunday, August 30 · 11:59 p.m. ET

B — Fixes and Safety Net
Sunday, September 6 · 11:59 p.m. ET

https://cmu-devops.github.io/assignments/hw01

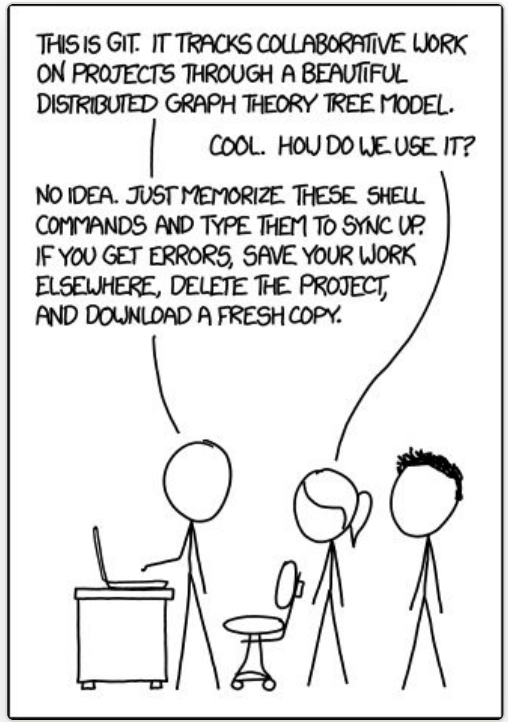
Administrivia: Lab 1

- Labs are designed to be completely independently and offline
  - No scheduled lab sessions
  - Labs are pass/fail
- Reach out to the course staff for support!
- Prepares you for the next assignment
- **Due by Friday**



🎯 Learning Goals

- Explain how branches, a healthy mainline, and checked integration let teams change software safely in parallel
- Predict how delayed integration increases divergence, slows feedback, and creates risks that Git may not detect
- Split a large change into coherent, reviewable batches that reach healthy intermediate states
- Compare branching, release, and repository choices by the coordination problem each solves and the costs each creates


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

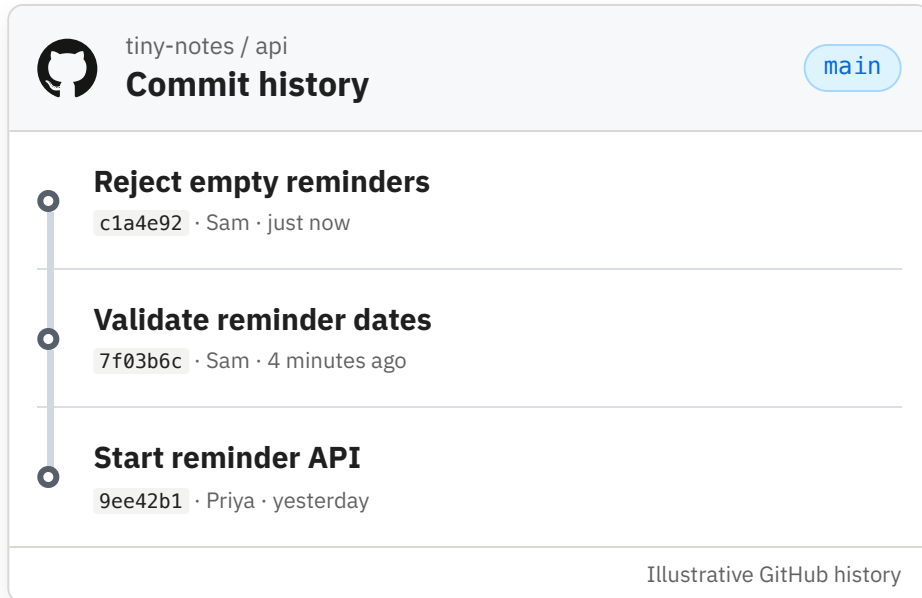
xkcd #1597

```
tiny-notes — zsh

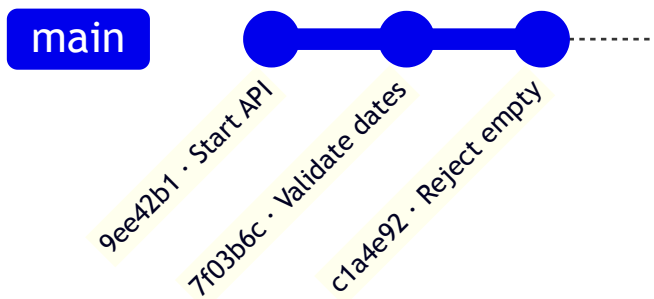
$ git add src/server.js
$ git commit -m "Start reminder API"
[main 9ee42b1] Start reminder API
2 files changed, 34 insertions(+)

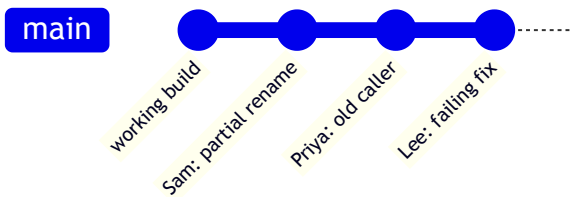
$ git add src/reminders.js
test/reminders.test.js
$ git commit -m "Validate reminder dates"
[main 7f03b6c] Validate reminder dates
2 files changed, 19 insertions(+), 3
deletions(-)

$ git add src/reminders.js
$ git commit -m "Reject empty reminders"
[main c1a4e92] Reject empty reminders
1 file changed, 8 insertions(+), 2
deletions(-)
```



Under the hood, our Git history is a graph





tiny-notes / api

Commit history

main

Working build
`3c9d201` · 08:55

Partially rename reminder API
`21bd4a8` · Sam · 09:02

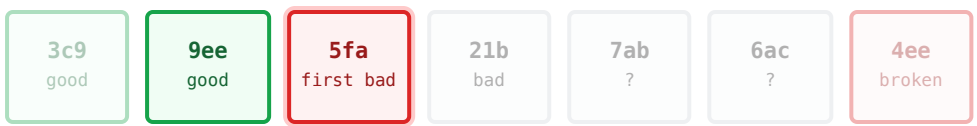
Add caller for old API
`7ab3f91` · Priya · 09:04

Try to repair build
`4ee81c0` · Lee · 09:06

No build or test status is recorded

The team can no longer trust its shared starting point.

Git can find the break, but diagnosis still costs time



NARROW THE HISTORY

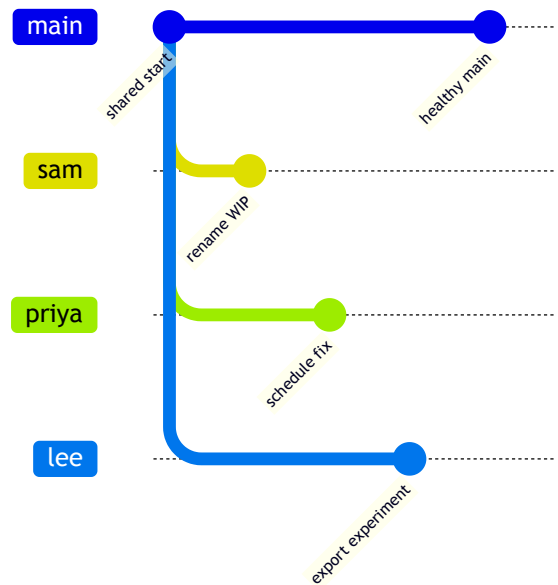
```
$ git bisect start
$ git bisect bad 4ee81c0
$ git bisect good 3c9d201
```

INSPECT THE SUSPECT LINE

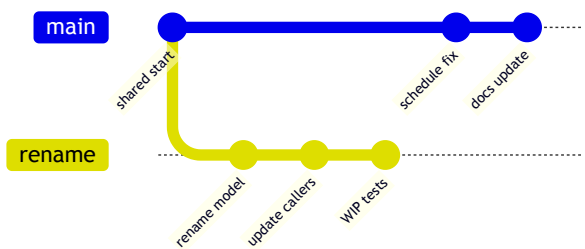
```
$ git blame -L 42,42
src/reminders.js
```

```
5fa83d2 (Sam 09:01) reminder.title.trim()
```

Teams need independent work and a healthy `main`



Branches let unfinished changes develop in isolation



`rename`

A developer records unfinished work

That line can move without replacing the team's shared state.

`main`

The team keeps working

The stable reference remains available while the rename develops.

A healthy mainline gives the team a stable starting point

VOCABULARY

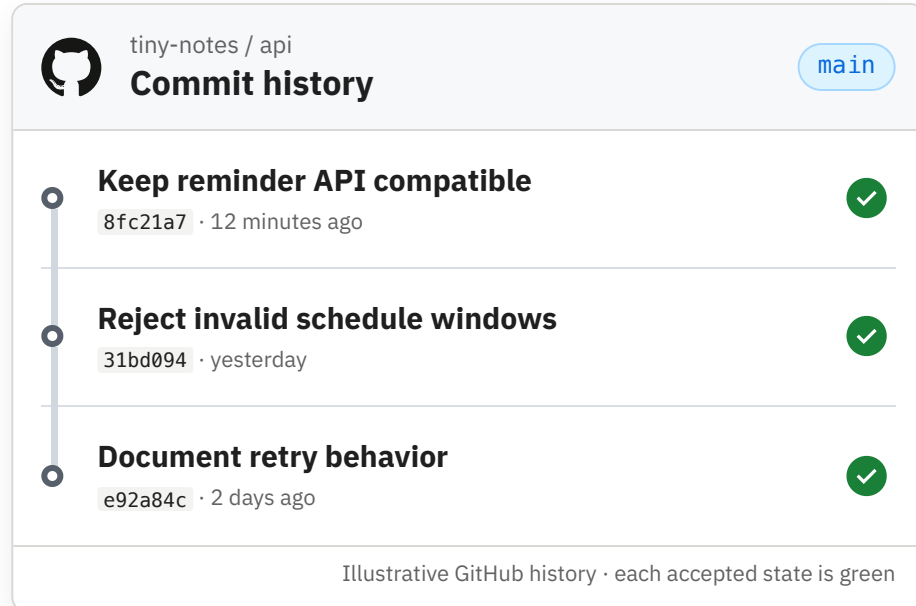
Mainline

The branch the team treats as its current shared source —often `main`, `master`, or `trunk`.

The source builds
Do not begin from a known broken build.

Required tests pass
Every accepted state meets the team's rule.

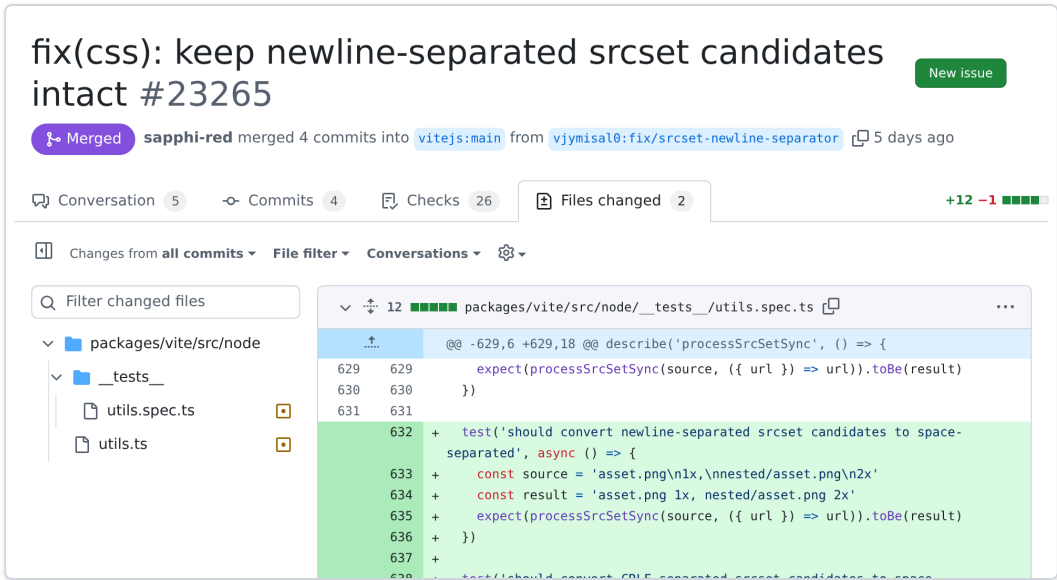
New work starts here
Developers branch from this shared state.



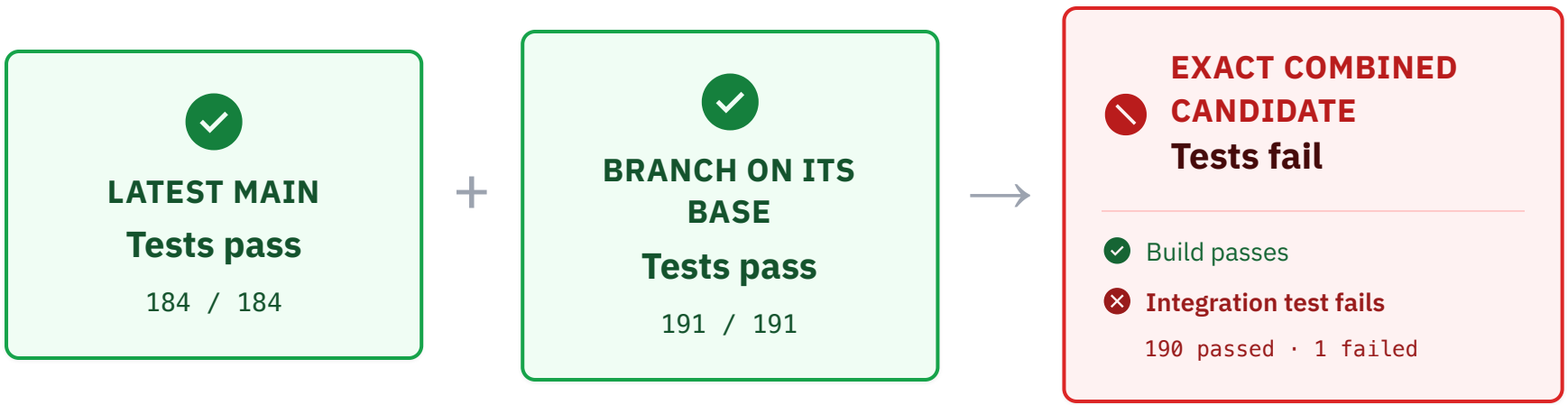
If the mainline breaks, fix it before integrating unrelated work.

We can then review and integrate changes individually

- Begin with the latest healthy `main`.
- Combine it with the **proposed change**.
- Build, test, and review that **exact candidate**.



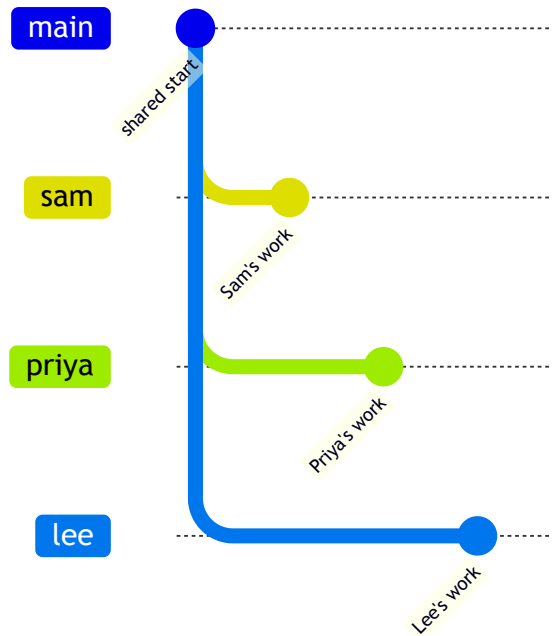
Two passing states can produce one failing combination



When do we create branches?
How do we use them?
When do we merge them?

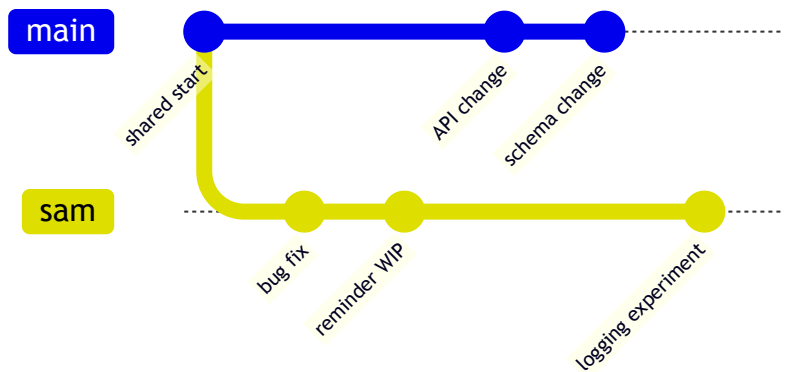
Branching Patterns

A contributor branch gives each developer a workspace



How could this go wrong?

A long-lived contributor branch diverges from mainline



READY NOW	NOT READY	MAY BE DISCARDED
Bug fix	Reminder feature	Logging experiment

shared start → more changes · more assumptions to reconcile

Divergent edits can produce a syntactic conflict

```
<<<<<<< HEAD
const timeout = 5_000
=======
const timeout = 10_000
>>>>>>> feature
```

Syntactic conflicts occur when Git cannot automatically combine overlapping edits.

A semantic conflict can survive a clean merge

A Branch A · `store.ts`
Rename the export

```
export function loadRecord(id) {
  return records.get(id)
}
```

✓ **Callers updated · tests pass**

Git merges cleanly
different files · no overlapping text

B Branch B · `duplicate.ts`
Call the old name

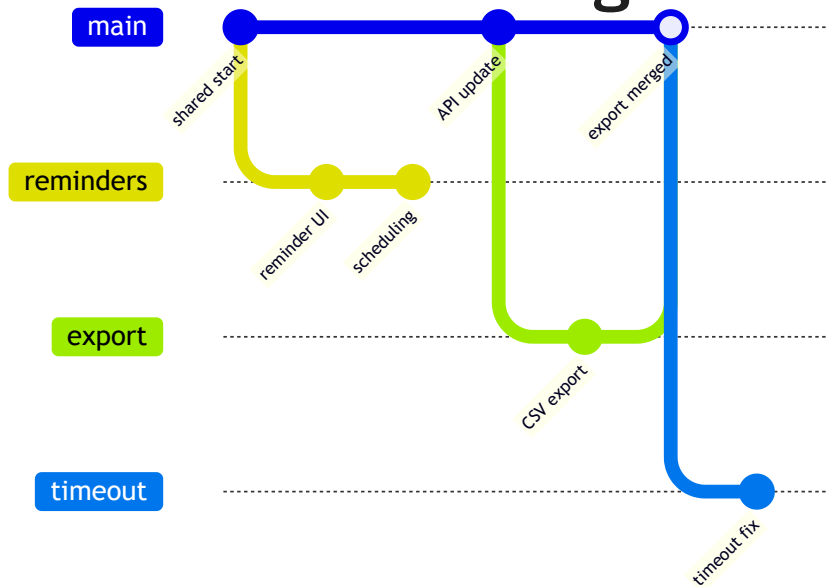
```
import { getRecord } from './store'

export const duplicate = id =>
  copyOf(getRecord(id))
```

✓ **Passes against current main**

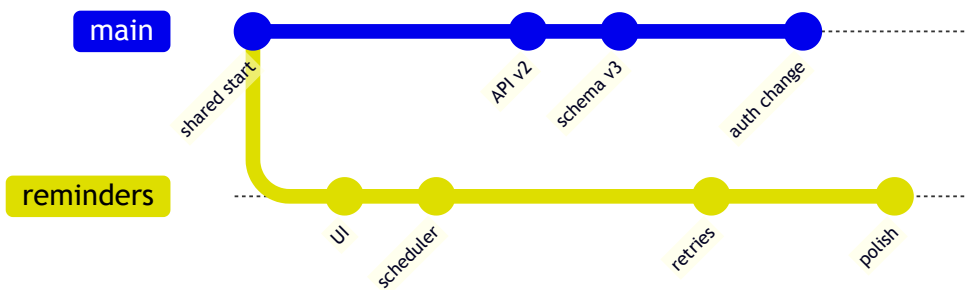
Combined build fails: `getRecord` is no longer exported
Semantic conflict: the text merges, but an assumption in the combined behavior is broken.

A feature branch isolates one change



A feature branch separates unrelated work. What can still go wrong?

A long feature branch accumulates untested differences



Mainline moves

API, schema, and authentication change.

The feature grows

UI, scheduler, retries, and polish accumulate.

The combined state

Never built or tested together. Every change adds more assumptions that may be incompatible.

Delayed integration arrives as one massive pull request

Open

Pull request #482

Scheduled reminders overhaul

`sam/reminders` wants to merge into `main`

Changes requested

37 commits **42** files changed **+2,184** **−641** **18** days open

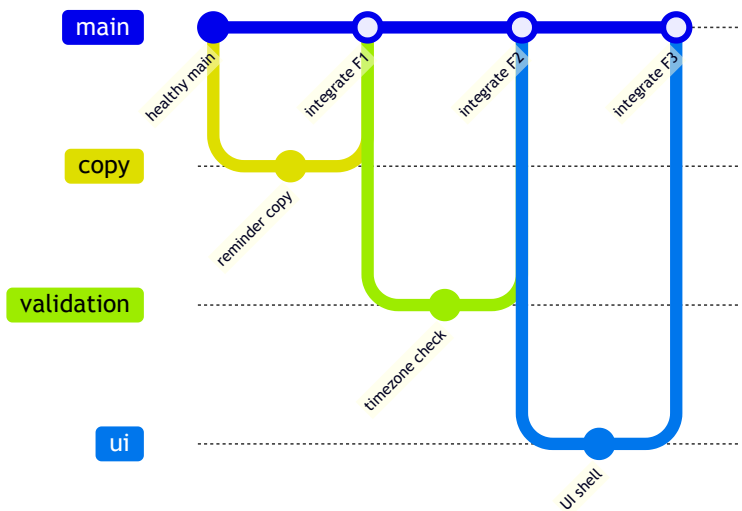
Some checks were not successful

- **Integration suite** — 18 scenarios failed — Failed
- **API contract** — Reminder payload no longer matches schema v3 — Failed
- **Unit tests** — 1,248 tests passed on the feature branch — Passed

Which change broke the combined behavior?

Review and diagnosis begin after weeks of work have accumulated.

Continuous integration shares small changes frequently



Integrate daily
Share one healthy increment at least daily.

Verify the combined state
Build and test the exact integrated result.

Restore mainline
Repair mainline before integrating more work.

How can a three-week feature integrate tomorrow?

Small Batch Development

What can we safely merge into `main` tomorrow?

Small batches shorten the path to feedback



KEY CONCEPT

Small batch

Related work the team can complete, test, and integrate independently—usually in hours or a few days.

“Small” does not mean few changed lines

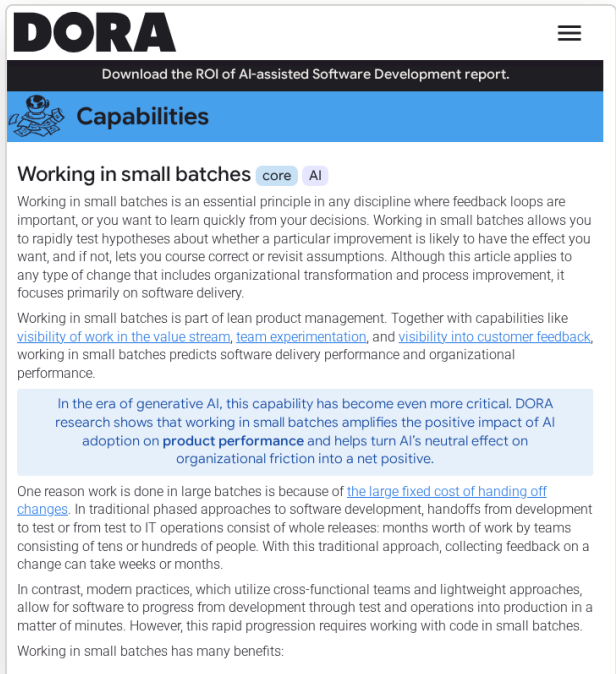
ILLUSTRATIVE PULL REQUEST

Open

Rename `parseDate` to `parseIsoDate`

`refactor/date-api` → `main`

236	+236	−236
files	lines	lines



DORA · Working in small batches

> In the era of generative AI, [small batch development] has become even more critical. DORA research shows that working in small batches amplifies the positive impact of AI adoption on product performance and helps turn AI's neutral effect on organizational friction into a net positive.
>
> — DORA

Small batches turn feedback into progress sooner



LEARN SOONER

Test assumptions early

Get feedback while the change is still easy to adjust.



FIX FASTER

Narrow the search

Inspect fewer changes when something fails.



CHANGE DIRECTION

Stop sooner

Avoid more work on an approach that is not working.



SEE PROGRESS

Finish work often

Frequent completed changes support efficiency and motivation.

DORA · Working in small batches

A small batch must still be safe to merge



FAILURE

What could go wrong?

The team can name likely failures and how its checks would detect them.



COMPATIBILITY

Can it merge without breaking today's behavior?

The change keeps behavior that people already use working.

Use INVEST to test a proposed batch

I Independent
It can be built and validated on its own.

N Negotiable
Feedback can still change it.

V Valuable
A stakeholder can use it.

E Estimable
There is enough information to size it.

S Small
It takes hours or a couple of days.

T Testable
The team can verify that it works.

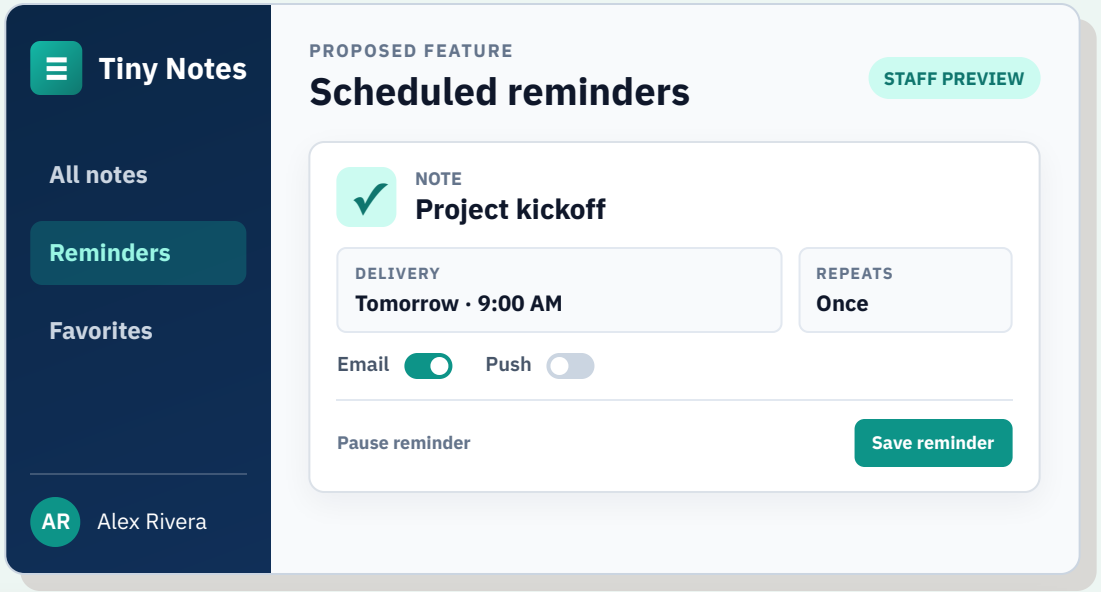
DORA · Working in small batches

Tiny Notes needs a feature that will take three weeks

Scheduled reminders

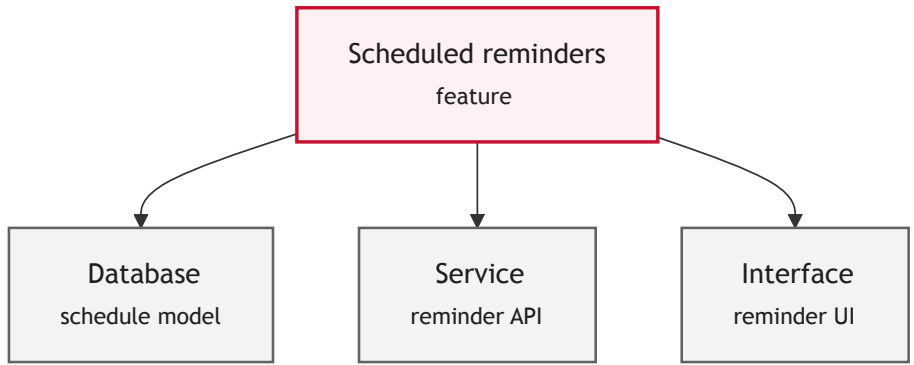
- Custom delivery times
- Recurring schedules
- Email and push delivery
- Pause and resume controls

ESTIMATED EFFORT **3 weeks**



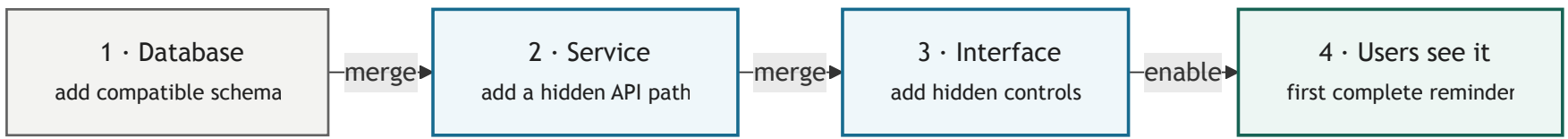
How would you split this request into small, safe batches?

We could split the feature by technical layer



- Clear ownership
- Familiar parts of the system
- Fewer components per task

Layer changes can merge before behavior is tested



Merging each part does not test the complete behavior.

Build a thin path, then add behavior gradually

START WITH A THIN PATH

1 · HIDDEN
Foundation
Service / API path

→

2 · STAFF ONLY
One-time email
First end-to-end check

→

3 · ALL USERS
One-time email
Same behavior, wider access

ADD BEHAVIOR GRADUALLY

4 · TIMING
Custom delivery
Choose when email arrives

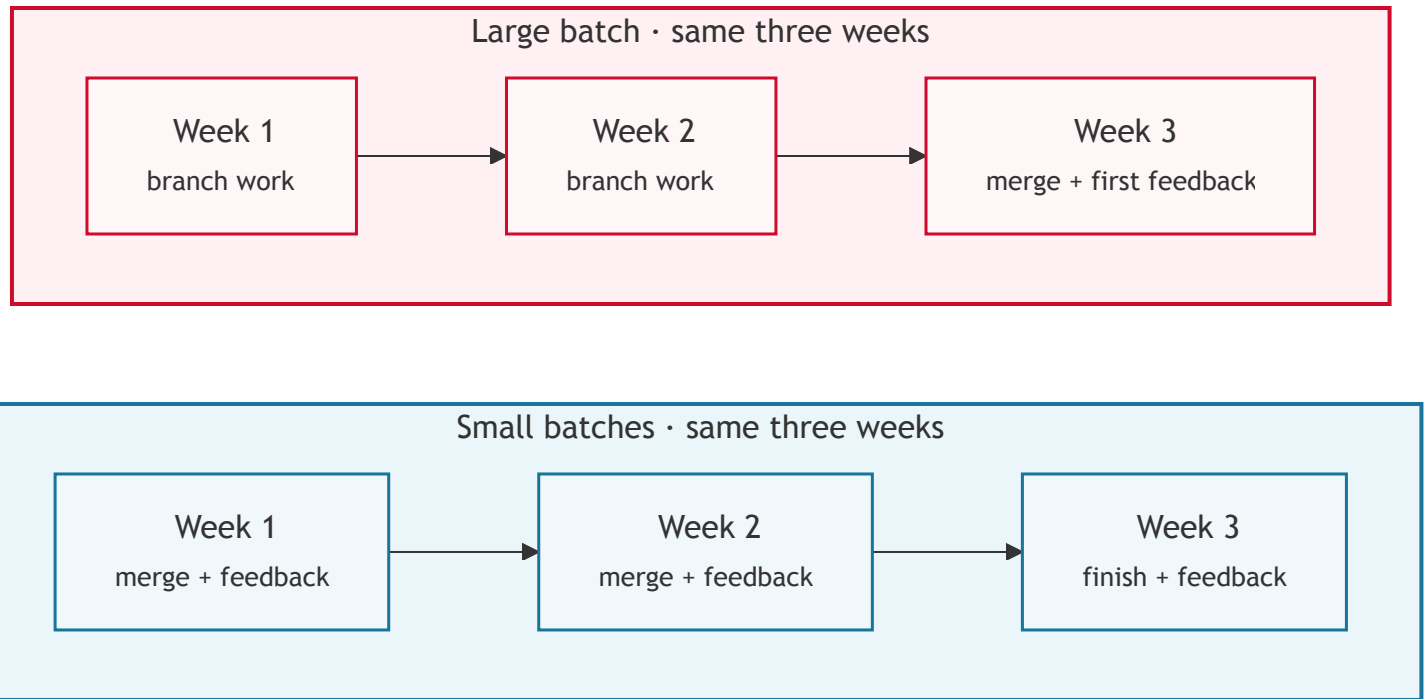
→

5 · SCHEDULE
Recurring email
Repeat a checked path

→

6 · MORE BEHAVIOR
Push and controls
Add push, pause, and resume

Smaller batches enable faster feedback



Must merging code also show it to users?

Tiny Notes has safe reminder code ready for `main`, but customers should not see it yet.

Feature flags let us merge code before users see it

- **Merge:** Put the code on the mainline while the behavior stays off.
- **Choose who sees it:** Turn it on for staff, a test environment, or part of the user base.
- **Clean up:** Test both states and remove the flag after everyone has the feature.

```
const remindersEnabled = featureFlags.isEnabled(
  'scheduled-reminders',
  currentUser,
)

if (remindersEnabled) {
  showReminderControls()
}
```



LaunchDarkly



Harness



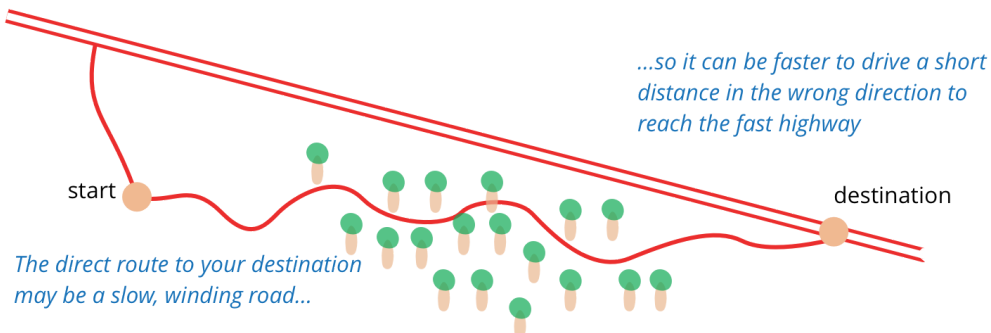
Optimizely

⚠ Feature flags do not make code modular

> Feature Branching is a poor man's modular architecture, instead of building systems with the ability to easy swap in and out features at runtime/deploytime they couple themselves to the source control providing this mechanism through manual merging.

— DAN BODART

Martin Fowler · Branching Patterns



Martin Fowler · An example of preparatory refactoring

Kent Beck 🌻
@KentBeck · **Follow**

for each desired change, make the change easy (warning: this may be hard), then make the easy change

7:07 PM · Sep 25, 2012

1.9K Reply C...

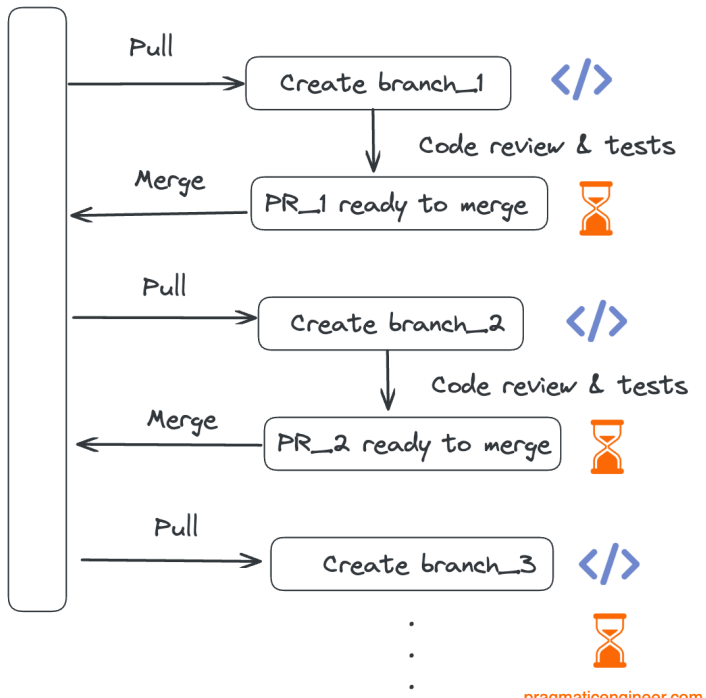
Read 21 replies

How can we review and merge PRs that depend on other PRs?

Stacked Pull Requests

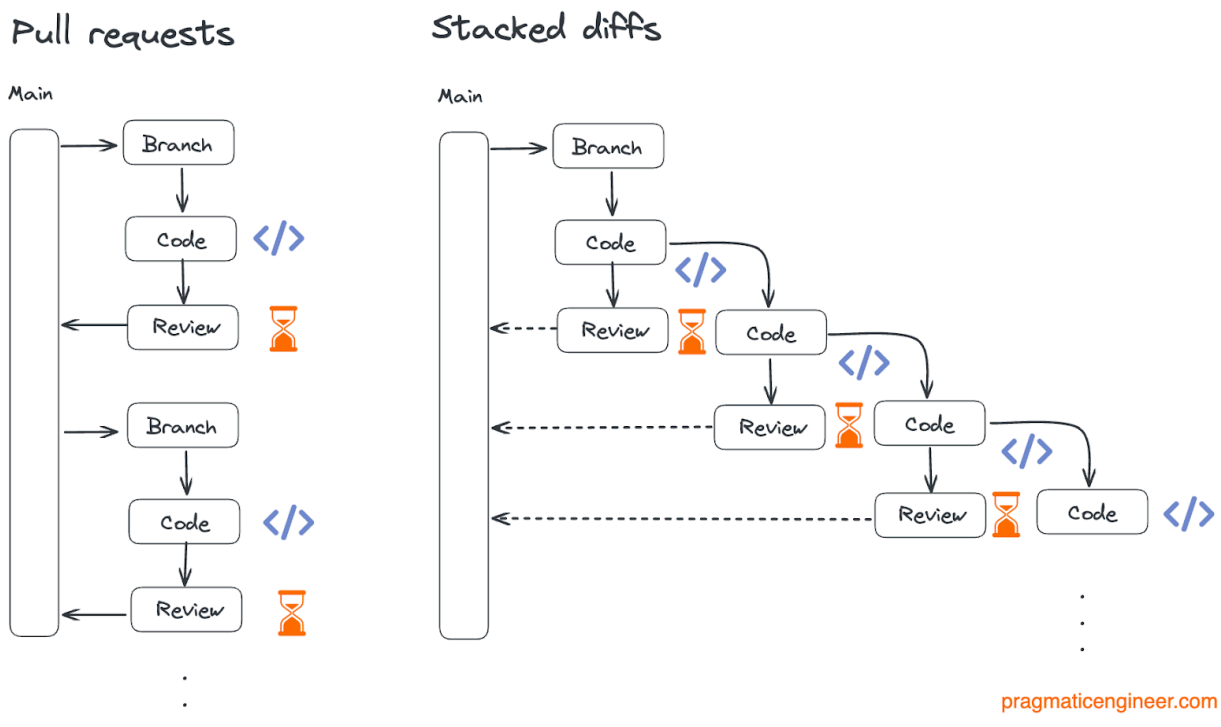
Keep dependent changes moving through review.

Dependent work can stall on review



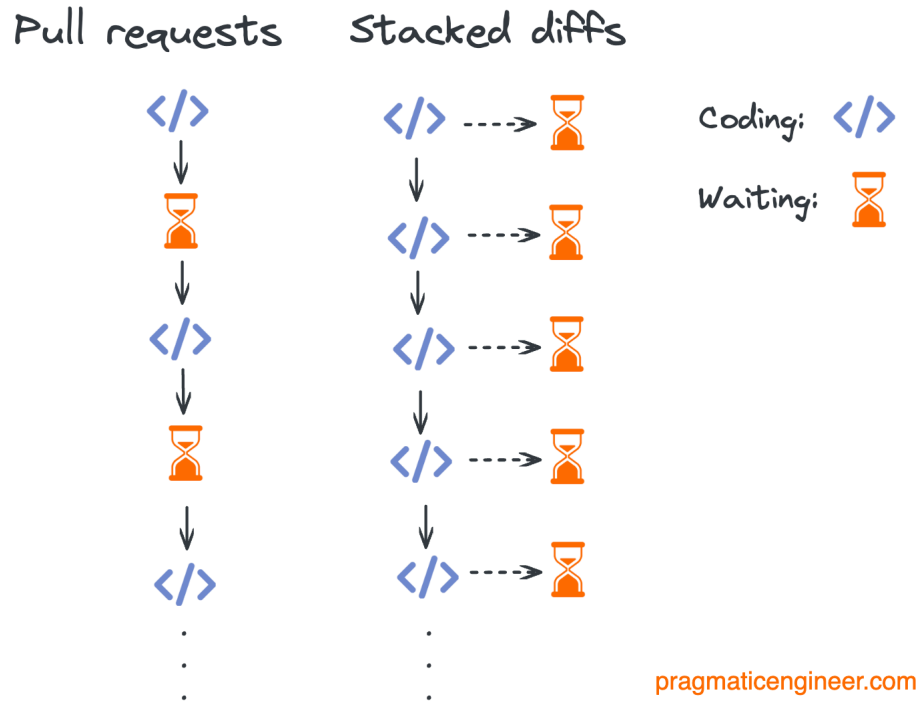
Source: The Pragmatic Engineer

Stacked PRs build each change on the one before it



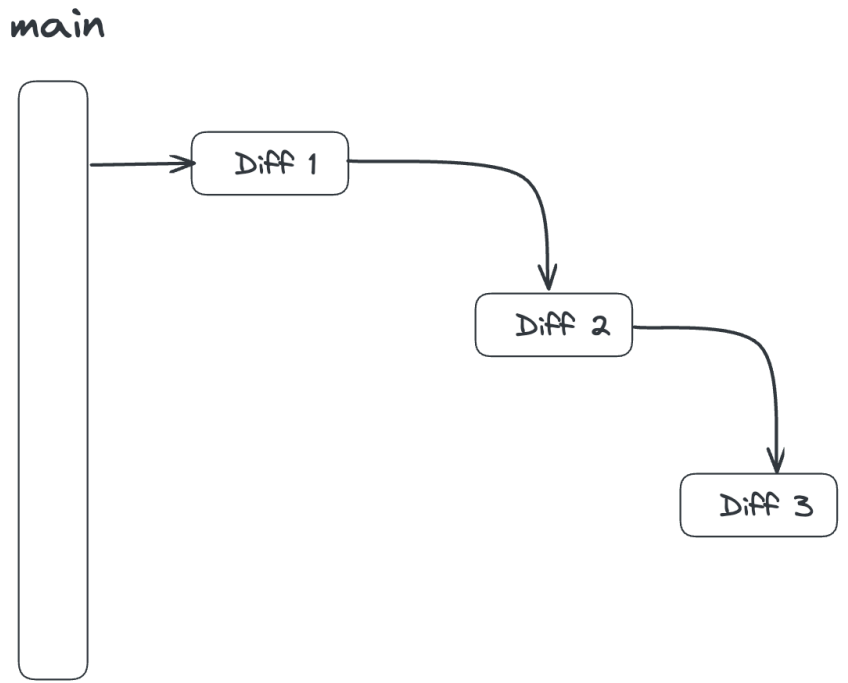
Source: The Pragmatic Engineer

Stacked PRs overlap coding with review



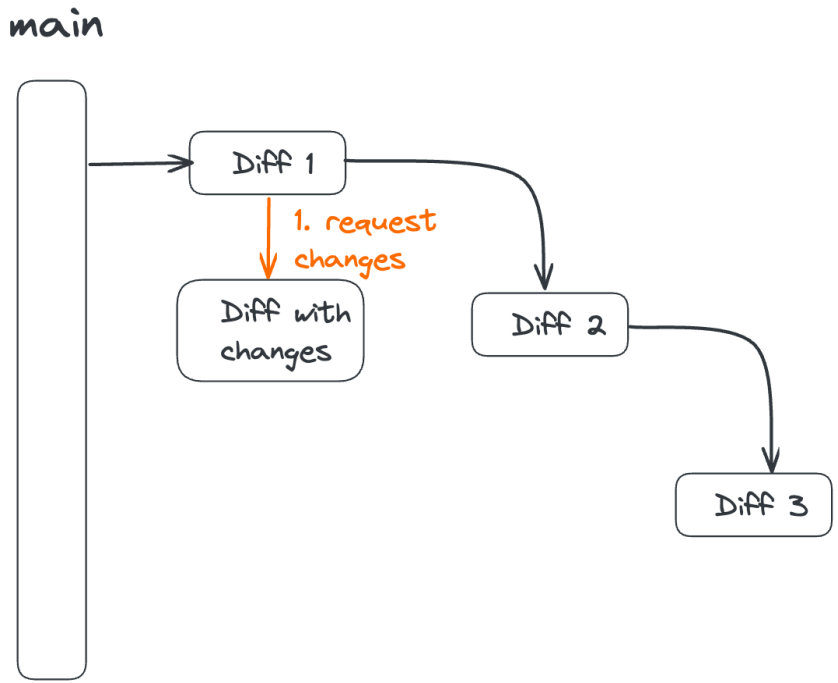
Source: The Pragmatic Engineer

The catch: earlier feedback affects later PRs



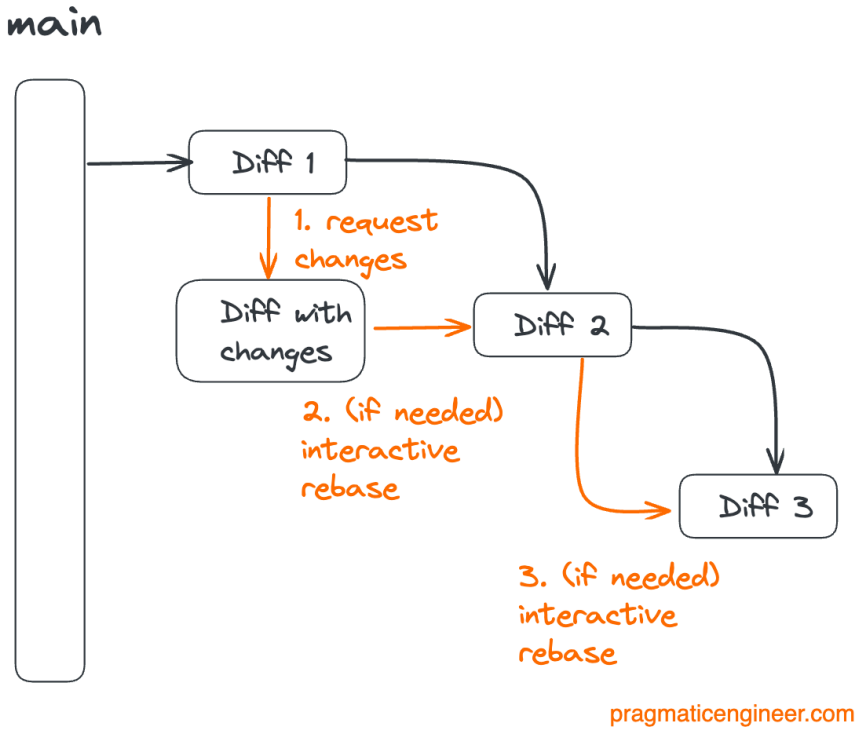
Source: The Pragmatic Engineer

The catch: earlier feedback affects later PRs



Source: The Pragmatic Engineer

Keep stacks short! Changes ripple across later PRs

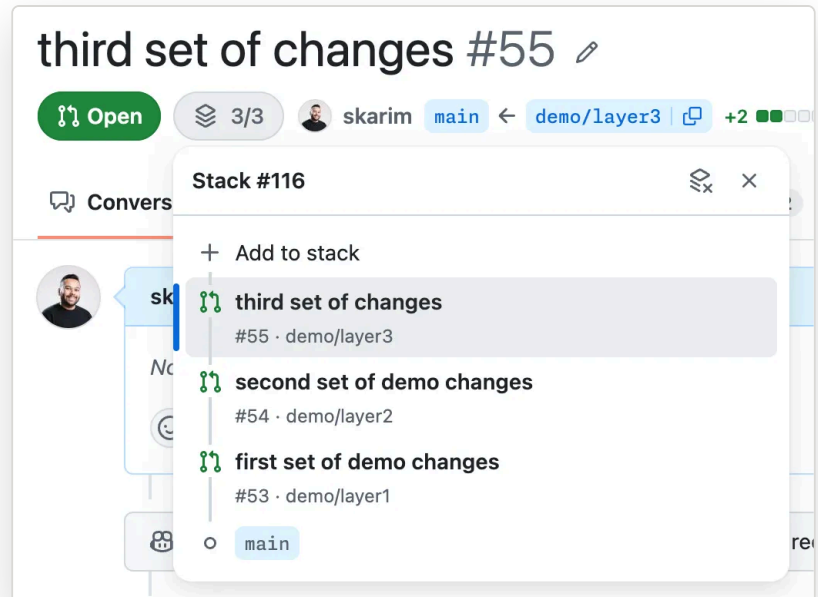


Source: The Pragmatic Engineer

Stack-aware tools reduce branch bookkeeping

- **GitHub (public preview):** View, rebase, and merge a stack from the website.
- `gh stack`: Create branches, open linked PRs, and propagate updates.
- **Other tools:** Graphite, Sapling, Jujutsu, and git-town also manage stacks.

GitHub Docs: Stacked pull requests

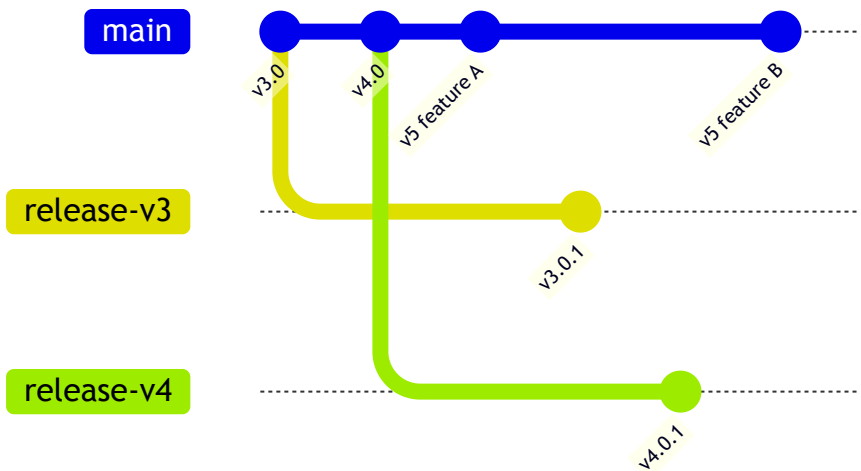


GitHub Stacked PRs · public preview

Supporting Released Versions

Customers need a v4 fix, but `main` already contains v5 work

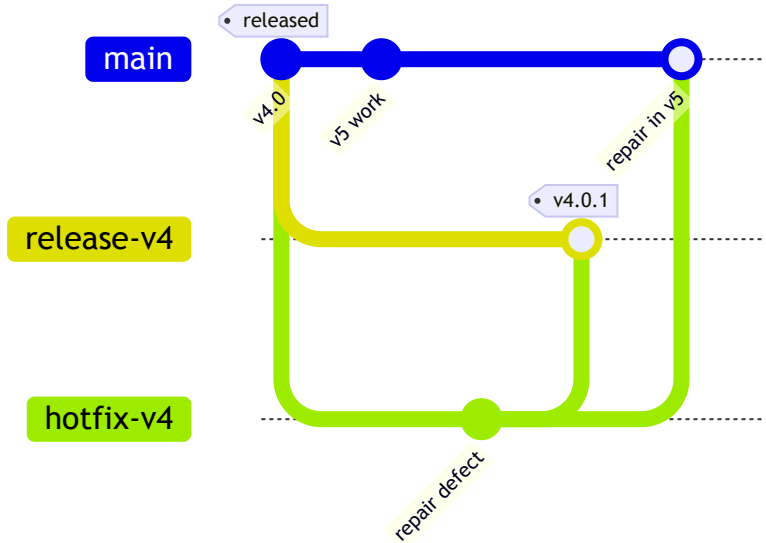
We can use release branches to separate maintenance from development



What it enables

- start from the released commit
- accept focused fixes for that version
- leave newer mainline work out

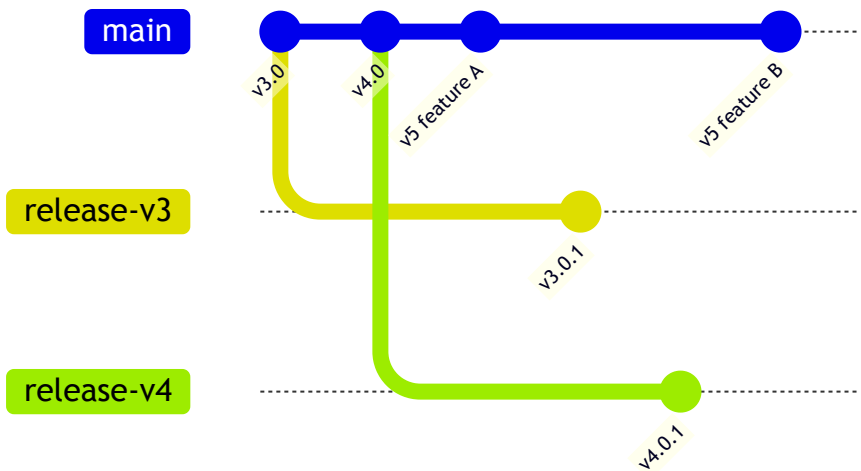
A hotfix must reach every affected line



1. Branch from the release line that contains the defect.
2. Make and check only the focused repair.
3. Merge the repair into the supported release.
4. Propagate it to every newer affected line.

What does supporting release branches cost?

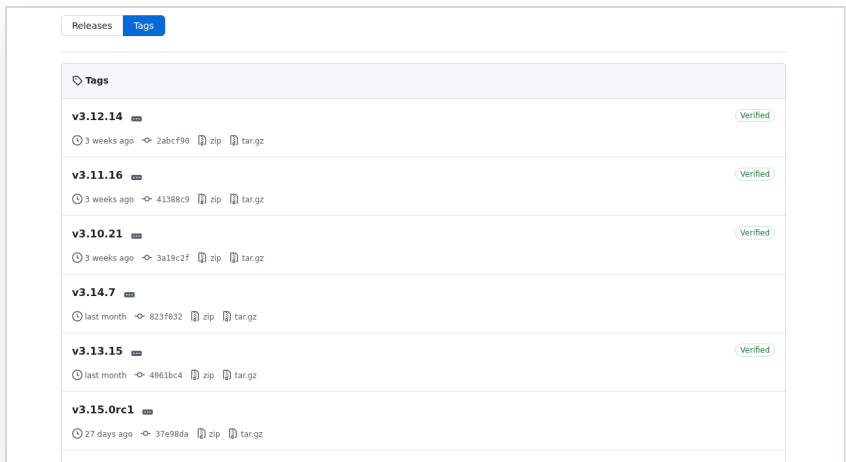
Supporting more releases multiplies maintenance work



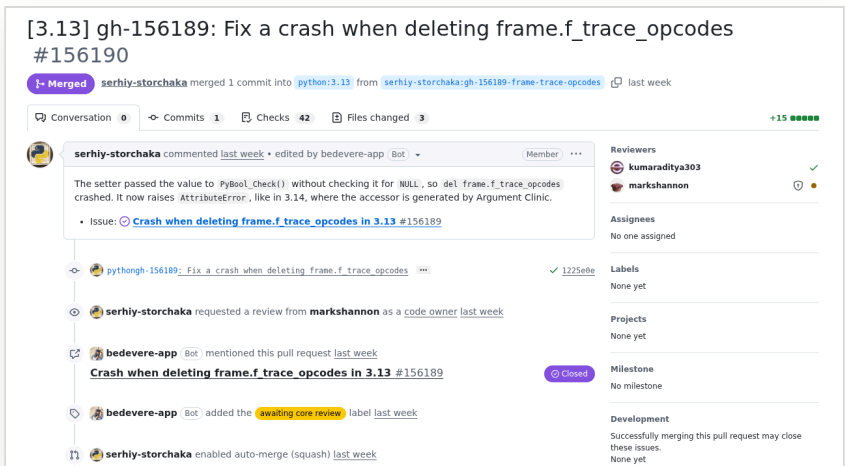
What it costs

- build and test another branch
- manage differences as release lines diverge
- move and verify fixes on every affected line

Example: CPython patches supported releases

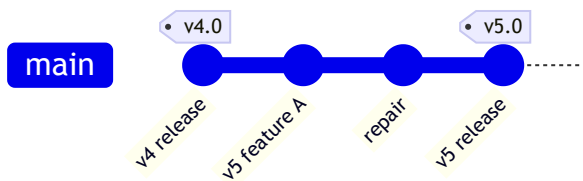


Published versions: CPython's tags show releases from several maintained version lines.



CPython: PR #156190 targets `3.13`; its description notes that `3.14` already behaves correctly.

Release-ready `main` needs fewer branches



- Every accepted `main` commit is a valid release candidate.
- A current-version fix follows the normal review and integration path.
- The fix appears in the next release from `main`.
- No long-lived release branch needs to be reconciled.

This policy cannot patch v4 without v5 work once `main` has advanced.

Which branching approach fits our support promise?

Homebrew supports only its current release

Release-ready `main`

- keep `main` release-ready
- fix through the normal path
- maintain fewer long-lived branches

Policy: Homebrew is a rolling release that supports only its latest release and latest `main` commit.

Supported Versions

Homebrew is a rolling release package manager. This means:

- only the latest release and latest commit on the `main` (or `master` if not yet migrated to `main`) branch of Homebrew/brew are supported.
- only the latest commit on the `main` (or `master` if not yet migrated to `main`) branch of all Homebrew repositories is supported.

Threat Model

Homebrew protects the integrity of Homebrew-maintained code, official formula and cask metadata, official bottle metadata, checksum verification, update and install logic, CI and release infrastructure, build sandboxing and credential filtering.

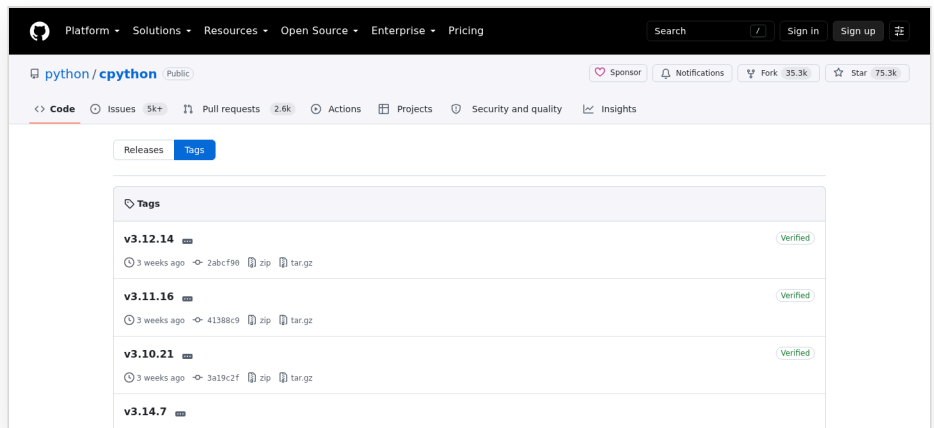
Homebrew documents a current-version-only support policy.

CPython maintains older supported versions

Release branches

- preserve a release branch per supported line
- start urgent fixes from the affected line
- propagate and test fixes on newer lines

Policy: CPython maintenance branches keep supported feature versions available for bugfix releases.

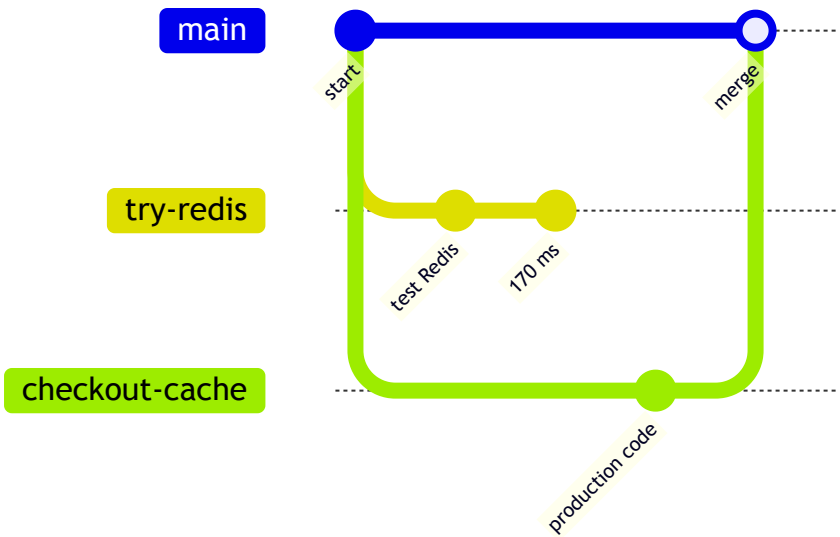


CPython tags show releases from several maintained version lines.

Every supported release adds a line that can diverge and a fix-propagation obligation.

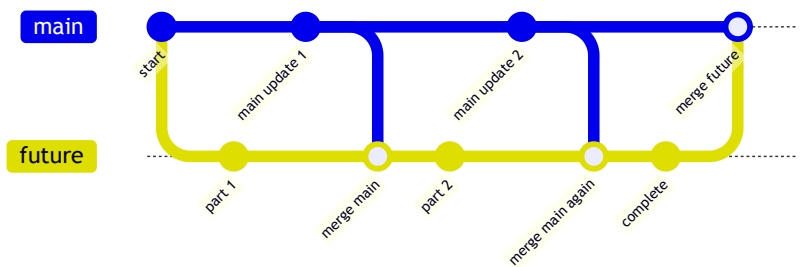
Branching Patterns (Continued)

An experimental branch answers one question



- **Ask one question:** Can Redis reduce latency? Can this library handle the load?
- **Run a focused test:** do only enough work to answer the question.
- **If the answer is no:** delete the branch.
- **If the answer is yes:** start from `main` and write the production version.

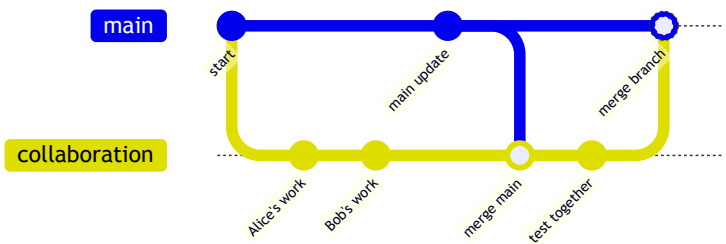
A future branch holds a large change until it is ready



- Use only when a large change cannot be split into smaller, safe merges.
- Merge mainline changes into it often. Keep its build and tests passing.
- When the change is ready, merge it into mainline and delete the branch.

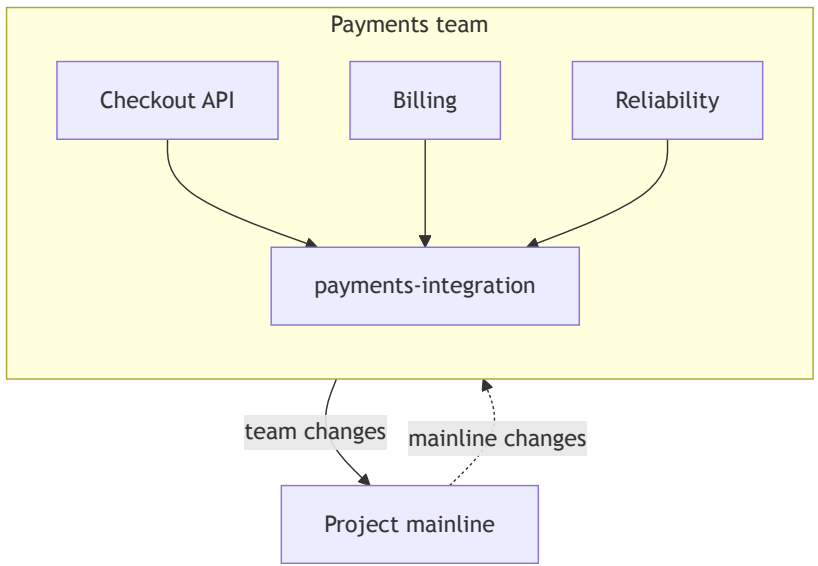
Use at most one future branch.
Delete it when the change is merged or canceled.

A collaboration branch lets a small group share work



- A few people share it for one short task.
- They share unfinished code without putting it on mainline.
- Merge or delete it when the task is complete.

A team integration branch is a second mainline



- A permanent team (e.g., payments, infrastructure) shares the branch.
- Members merge work there before it reaches the project mainline.
- The team keeps both branches working and merges changes in both directions.

Branching Policies

A policy lets developers make the same decision



REUSABLE OPTION

Pattern

A feature branch isolates unfinished work.



STARTER BUNDLE

Named workflow

GitHub Flow combines branches, reviewed PRs, and a healthy main.



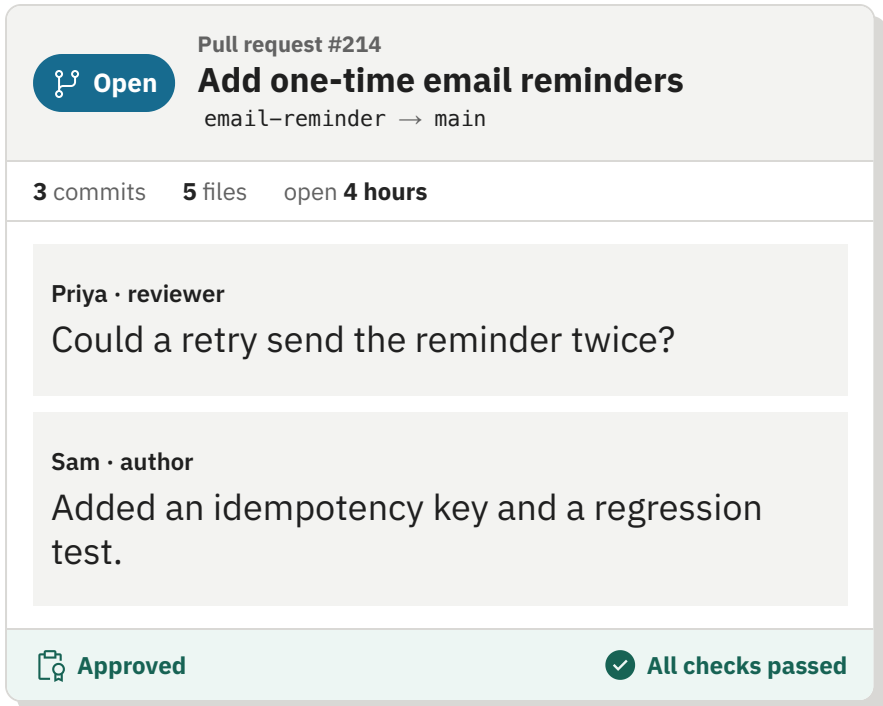
TEAM AGREEMENT

Policy

Merge reviewed PRs within one day; keep main releasable.

Without shared limits, one developer may merge today while another waits two weeks.

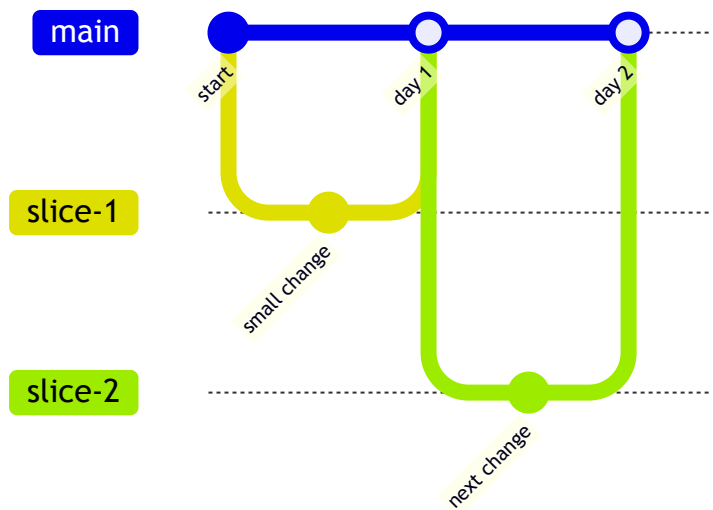
GitHub Flow requires review but leaves timing open



- Branch from a healthy `main`.
- Review and check the pull request.
- Merge back; release from `main`.

- **Addresses:** review and automated checks before integration.
- **Policy still decides:** PR batch size and waiting time. A same-day slice and a two-week feature branch can both follow GitHub Flow.

Trunk-based development requires daily integration



- **Merge daily:** each developer integrates at least once per day.
- **Keep branches short:** branches or forks live for less than one day.
- **Keep few branches:** the team maintains fewer than three active branches.

- **Addresses:** branch divergence by limiting how long work stays apart.
- **Policy still decides:** review evidence and release support. A same-day reviewed PR can satisfy both GitHub Flow and trunk-based development.

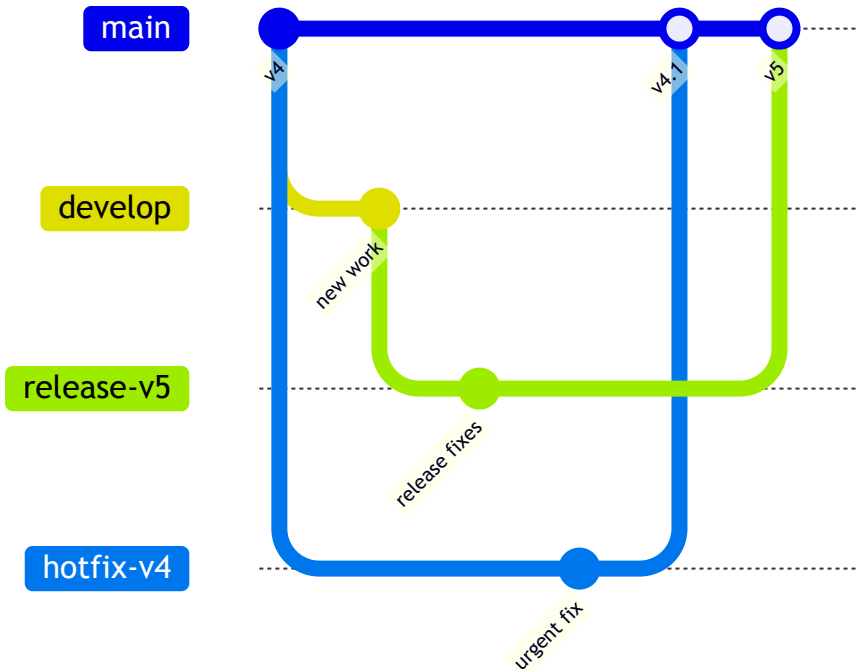


DORA · Accelerate State of DevOps · 2021

> Elite performers who meet their reliability targets are **2.3 times more likely** to use trunk-based development.

— DORA · 2021

Git-flow adds branches for release support

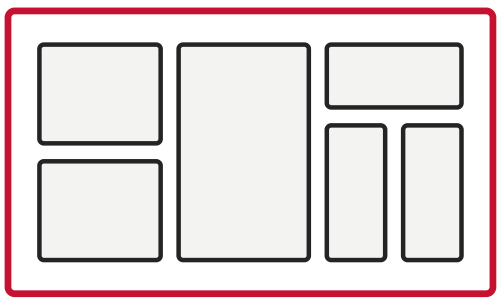


- `develop`: feature work integrates toward the next version.
- `release-*`: feature scope freezes; only defect fixes and release preparation continue before merging to `main`.
- `hotfix-*`: urgent repairs begin from the released line and propagate forward.

- **Addresses:** planned releases and urgent fixes across supported lines.
- **Policy still decides:** batch limits, review time, required checks, and how fixes propagate between branches.

Monorepos vs. Polyrepos

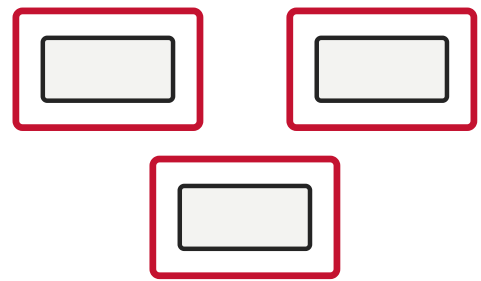
The same system can be split across repositories in different ways



Monorepo



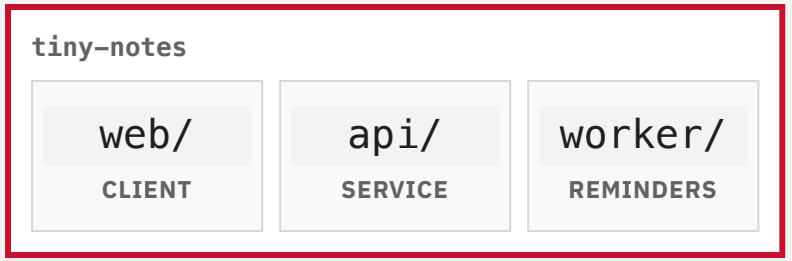
Single-repo monolith



Polyrepo

How should Tiny Notes store its source?

One repository



Three repositories



A monorepo can make a source change atomic

Scheduled reminders touch `web`, `API`, and `worker`.

Monorepo

ATOMIC

ONE COMMIT

`web + API + worker`

All source changes land together.

Polyrepo

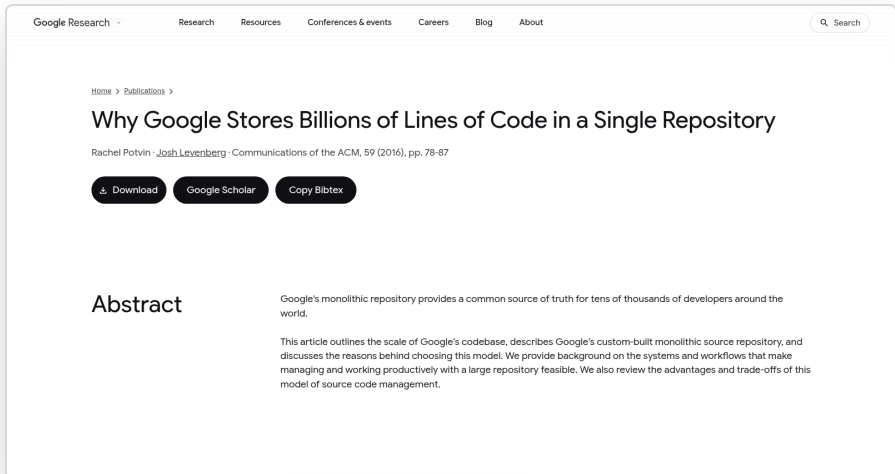
STAGED

SEVERAL COMMITS

1. **Add compatible API**
2. **Update web + worker**
3. **Remove old API**

Atomic means one source commit—not one deployment.

Monorepos keep related source changes together



Google Research · Potvin & Levenberg (2016)

Google

Meta

MAKES EASIER

Change together

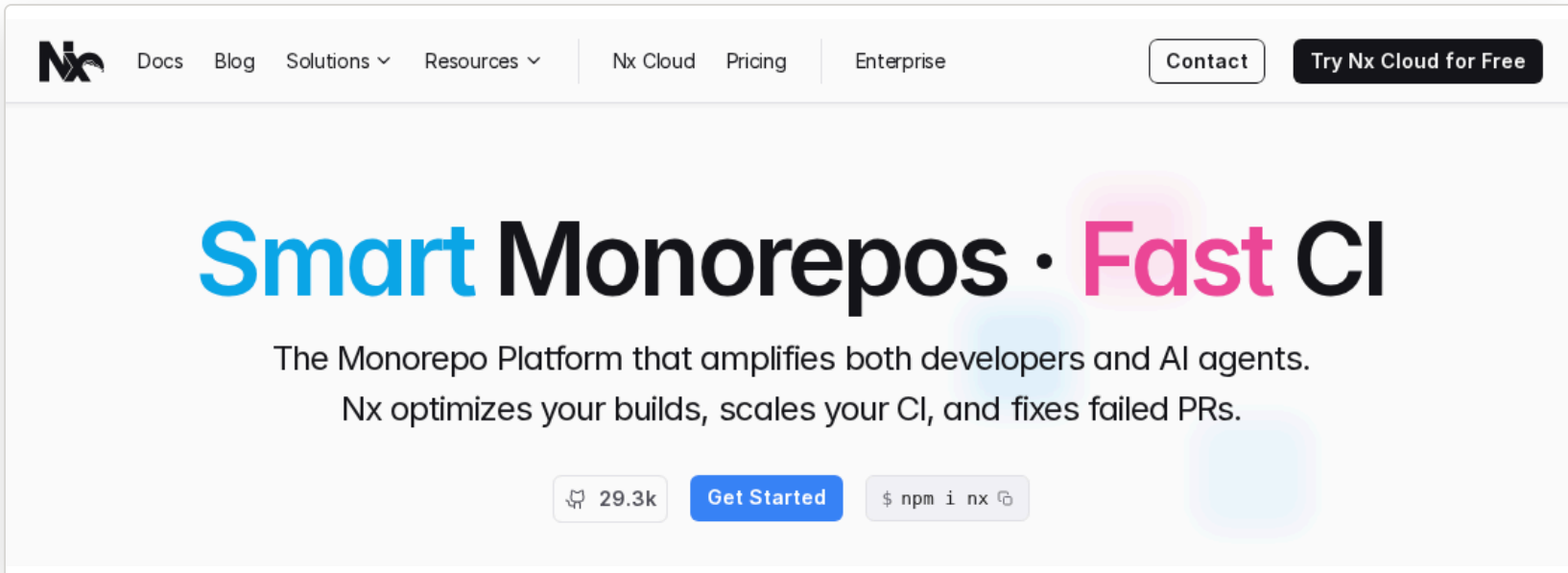
- One commit can update several components
- Search, review, and test one source state
- Share standards and reusable code

REQUIRES

Scale the shared boundary

- Select affected builds and tests
- Define path ownership and access
- Keep shared feedback fast

AI agents are changing the monorepo pitch



Nx homepage

Polyrepos keep ownership and releases separate



Java Platform @ Netflix · JVM Ecosystem Team

NETFLIX

Spotify

MAKES EASIER

Own and release independently

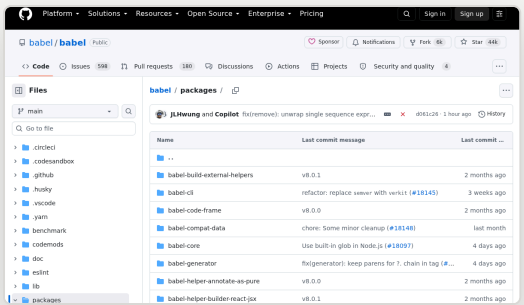
- Focus access, history, and checks
- Choose a component's release cadence
- Limit local coordination

REQUIRES

Coordinate across histories

- Version compatible interfaces
- Automate multi-repository updates
- Test the assembled system

Repository shapes can nest

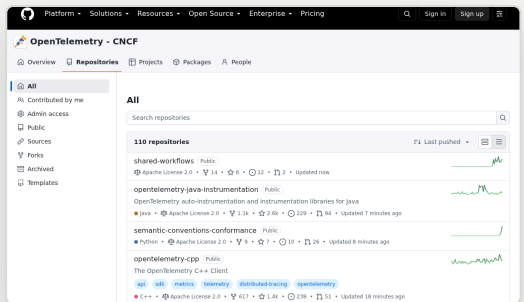


babel/babel · packages/

ONE REPOSITORY

Babel

Many published packages share one Git history.

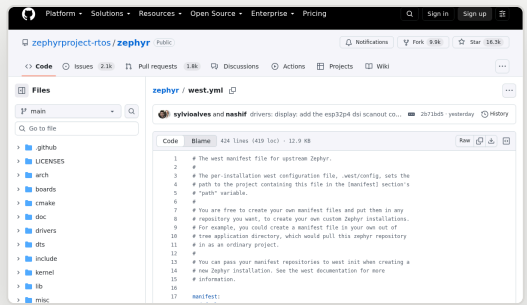


open-telemetry · repositories

MANY REPOSITORIES

OpenTelemetry

Languages and concerns have separate histories.



zephyrproject-rtos/zephyr · west.yml

COMPOSED WORKSPACE

Zephyr

One manifest assembles independently versioned repositories.

A project can be polyrepo at one boundary while each repository contains many packages.

No single VCS strategy fits every team and situation

Three teams have different coordination needs



SCENARIO A

PocketTasks

A hosted to-do app delivered as one service. Two engineers deploy several times per day and peer-review every change.



SCENARIO B

CampusHub

An enrollment and advising platform with a web app, API, and shared database. Ten engineers release monthly; student-data changes need security review.



SCENARIO C

MarketGrid

A global marketplace built from independently deployed services. Forty teams share customer and checkout APIs; payment changes need formal approval.

Recommend a VCS strategy for each team

For **PocketTasks**, **CampusHub**, and **MarketGrid**, decide:

1. Should the code live in one repository or several?
2. How should branches, reviews, and merges work?
3. What checks must pass before a change reaches `main`?
4. How should the team manage releases or supported versions?

Use at least one fact from each scenario to explain your choices.

The team's needs should explain its strategy

- **PocketTasks:** one repository, short-lived branches, quick peer review, frequent merges, and a `main` branch that is ready to release
- **CampusHub:** related code together, frequent integration, security review for student-data changes, and monthly releases
- **MarketGrid:** repositories owned by service teams, versioned shared APIs, automated compatibility checks, and formal approval for payment changes

For each example: **Which fact about the team makes this strategy a good fit?**

🚀 Today, we learned how to ...

- Explain how branches, a healthy mainline, and checked integration let teams change software safely in parallel
- Predict how delayed integration increases divergence, slows feedback, and creates risks that Git may not detect
- Split a large change into coherent, reviewable batches that reach healthy intermediate states
- Compare branching, release, and repository choices by the coordination problem each solves and the costs each creates