



Sheltie

RESEARCH REPORT · OCTOBER 2026

Sheltie on real repositories

What Sheltie would have done on 23 open-source projects over 90 days, against what each project does today. Including where it does not help, what it would have let through, and what we could not prove.

PROJECTS

23 measured, 26 classified

METHOD

Exact footprints, replayed queues

WINDOW

2026-07-21 to 2026-10-19

DATA

github.com/softberries/sheltie, [research/real-world](#)

We measured what Sheltie would have done on 23 popular open-source repositories over 90 days (landings from 2026-07-21 to 2026-10-19). We compared it with what each project does today, not with a strawman. This page reports what we found, including where Sheltie does not help, where it would have let breakage through, the bugs in Sheltie the study exposed, and what we could not prove.

Every footprint here is the exact one Sheltie's adapters compute for that commit. The queue results come from replaying each project's real arrival times through Sheltie's scheduler. They are simulations, not production runs.

The scripts, the bench and every number are in `research/real-world`.

47–75%

less waiting in the merge queue, on five GitHub merge-queue projects (like for like)

72–85%

fewer CI runs where the queue runs the same CI as pull requests

Summary

Where projects use a merge queue, Sheltie would cut waiting substantially. For pull requests the real queue never ejected, total time from entering the queue to landing fell by 47–75% on five GitHub merge-queue projects (rust-analyzer, junit, trino, bevy, gradle). Where the queue runs the same CI as the pull request (rust-analyzer, junit, trino), Sheltie also needed 72–85% fewer CI runs than the queue's builds.

Most of that comes from landing a change without a new build when nothing that landed since its CI touches what it reads. That is only as safe as the pull request's CI. Three of the projects (bevy, gradle, rust-lang/rust) run more in their queue than on pull requests. For them, the fair comparison makes Sheltie run their full CI on every pull request, and the CI saving disappears: Sheltie would need about as many runs as their queue builds today.

Queue failures happen either way; a few would have reached `main`. On the five GitHub merge-queue projects, 223 pull requests that Sheltie would have landed at once failed a queue build at least once. 165 of them went back into the queue with no new commits and merged: the failure was most likely a flaky test or another pull request in the same build, and the same reruns would likely happen with Sheltie. 18 only rebased. In 40 the pull request's own diff changed after the failure:

- 14 failed checks that run only in the queue: 10 repository-wide checks at gradle and 4 jobs at bevy;
- 14 more at gradle followed failures in its fuller queue CI. We did not identify the failing tests, so "the test only ran in the queue" is likely but not confirmed;
- 2 changed CI workflows;
- 1 was a semantic conflict with `main` that footprints missed;
- for 9 we could not determine the cause.

That 1 and those 9, and possibly some of the 14 unconfirmed gradle cases, are where Sheltie's footprints may have missed a real interaction.

On projects without a merge queue, we could not show that Sheltie prevents red `main`. `main` went red and stayed red (two or more consecutive failed runs) 176 times across 13 projects. An overlapping landing was in range in 83 of those episodes; chance alone predicts 86. Stale merges are common (21–90% of pull requests merge after something else landed), but they do not explain the breakage we can see. In these projects Sheltie would add CI: about 89 days of CI wall time over 90 days across the 14 projects whose CI time we have, as an upper bound.

The study also found problems in Sheltie, fixed before these results were computed (see [What the study changed in Sheltie](#)).

The projects and how they merge

We classified 26 projects from their GitHub rulesets, Actions events and pull request timelines. Three have no GitHub pull request merge flow to compare against:

- go-lang/go reviews on Gerrit;
- guava is exported from Google's internal repository;
- spring-boot's maintainers merge locally.

Group	Projects
GitHub merge queue	rust-analyzer, junit-framework, trino, bevy, gradle (CI on TeamCity)
bors	rust-lang/rust
Prow/Tide	kubernetes
Serial label bot	traefik (the bot rebases a labelled pull request and merges after CI)
Plain merges, no queue	ruff, tokio, nushell, prometheus, hugo, vite, vitest, storybook, vscode, home-assistant, fastapi, pydantic, cpython, keycloak, okhttp

Whether the queue runs more than pull request CI decides how to read the results:

Project	Queue	Queue CI vs PR CI	Evidence
rust-analyzer	GitHub merge queue	same	same workflows on pull_request and merge_group, no merge_group-only jobs
junit-framework	GitHub merge queue	same	same workflows on pull_request and merge_group, no merge_group-only jobs
trino	GitHub merge queue	same	ci.yml: merge_group differs only by a checkout step for path filtering
bevy	GitHub merge queue	heavier	validation-jobs.yml: jobs with if: github.event_name == merge_group (iOS, Android, wasm, examples)
gradle	GitHub merge queue	heavier	PR commits get Quick Feedback (Linux) and unit tests; queue commits get the full TeamCity pipeline (integration, distribution, performance tests)
rust-lang/rust	bors	heavier	src/ci/github-actions/jobs.yml: 11 pr jobs, 82 auto (bors) jobs
traefik	label bot (serial)	same	the bot rebases and waits for the same PR CI
kubernetes	Prow/Tide	unknown	Prow; not visible on GitHub

Queue projects: the real queue against Sheltie

How it was measured

- **Ready time** is when a pull request entered the merge process:
 - for GitHub merge queues, its last `AddedToMergeQueue` event;
 - for bors, the last "approved by" comment;
 - for traefik, when the merge label was added;
 - for Tide, when `lgTM`, `approved`, passing presubmits and the removal of blocking labels were all in place.
- **Landed** is the pull request's `mergedAt`. (A merge-queue commit's own timestamp is when its build started, not when it landed.)
- **CI per build** is the median wall time of a queue build:
 - for GitHub merge queues, `merge_group` runs grouped by commit, from first start to last finish;
 - for bors, its "Duration" line;
 - for gradle, whose CI runs on TeamCity, from pending to final of the gating status `GitHub Merge Queue Check Pass` on each landed commit.
- **Sheltie** replays the same entries at the same times through the scheduler Sheltie runs (`plan_epoch`), with the same CI time per run:
 - Each change counts as tested on `main` as it was when its own pull request CI last started.
 - If anything that landed after that writes what the change reads, Sheltie rebuilds it on the new head. If it writes what the change writes, the change goes to the merge engine.
 - Otherwise it lands at once.
- **Like for like:** the comparison uses only pull requests the real queue never ejected, because the replay assumes every build passes.
- **Rebase merges:** trino merges pull requests by rebasing several commits onto `main`. Each such pull request counts as one change, with one footprint from its first parent to its last commit.

Waiting

Waits are median / 90th percentile / total.

Project	CI per build	PRs (never ejected)	Real wait: median / p90 / total	Sheltie wait: median / p90 / total	Total wait saved
rust-analyzer	8 min	279	9 min / 10 min / 43 h 35 min	0 s / 8 min / 13 h 57 min	68%
junit-framework	9 min	147	9 min / 18 min / 27 h 39 min	0 s / 9 min / 6 h 57 min	75%
trino	53 min	453	56 min / 1 h 55 min / 22 d 3 h	0 s / 1 h 21 min / 9 d 0 h	59%
bevy	35 min	511	38 min / 1 h 26 min / 17 d 18 h	34 min / 1 h 7 min / 9 d 10 h	47%
gradle	1 h 19 min	297	1 h 32 min / 2 h 51 min / 21 d 21 h	0 s / 2 h 24 min / 9 d 0 h	59%
rust-lang/rust	3 h 8 min	428	5 h 18 min / 17 h 2 min / 155 d 23 h	0 s / 3 h 8 min / 19 d 8 h	88%
traefik	13 min	37	2 min / 15 min / 9 h 37 min	0 s / 0 s / 0 s	100%

Read with care:

- **rust-lang/rust** (bors) runs 11 CI jobs on pull requests and 82 in bors. Landing at once would skip 71 job configurations, so its 88% is not a result Sheltie could deliver as is. Also, 222 of its 440 landings are rollups that bundle 2,151 pull requests; the replay sees a rollup as one change.

- **traefik**'s 37 changes in 90 days never overlapped in flight. The bot already lands them within minutes, so the saving is small in absolute terms (9 h 37 min in total).
- **kubernetes** is not in the table. Its waits are dominated by holds and policy (up to 35 days), and Prow's CI is not visible on GitHub, so we have no CI time to replay with.
- **bevy** saves least because its changes overlap often. Of 592 changes, Sheltie would send 115 to the merge engine, rebuild 92 and build 57 together with a conflicting change.

CI runs

Project	Queue CI	Real queue builds (failed)	Sheltie CI runs	Sheltie, if PR CI had to run the queue's full CI
rust-analyzer	same	297 (7)	83	not needed
junit-framework	same	165 (5)	36	not needed
trino	same	2357 (579)	348	not needed
bevy	heavier	768 (71)	224	816
gradle	heavier	≥599 (167)	170	580
rust-lang/rust	heavier	625 (190)	116	556
traefik	same	37	0	not needed

- Real queues build every entry, and rebuild entries behind every failure. Sheltie builds a change again only when something that landed after its CI overlaps it.
- For gradle the real count is a lower bound: one build per queue entry. Builds re-run after an ejection ahead of an entry are not visible.
- The last column is the fair cost for projects whose queue runs more than pull request CI: one full CI run per change plus Sheltie's rebuilds. It is about the same as the real queue.

What Sheltie would have done with each change:

Project	Changes	Land at once	Rebuild (reverify)	Merge engine	Built together	Leaky footprints
rust-analyzer	289	203 (70%)	35	46	5	160 (55%)
junit-framework	228	191 (84%)	3	34	0	24 (11%)
trino	746	391 (52%)	86	261	8	395 (53%)
bevy	592	328 (55%)	92	115	57	282 (48%)
gradle	410	239 (58%)	30	139	2	129 (31%)
rust-lang/rust	440	318 (72%)	74	38	10	417 (95%)
traefik	37	37 (100%)	0	0	0	25 (68%)

Would landing at once have let breakage through?

Queue failures, split by what Sheltie would have done with the same pull request:

Project	Failed in queue, Sheltie lands at once	Failed in queue, Sheltie rebuilds or merges
trino	112 / 388 (29%)	169 / 355 (48%)
gradle	58 / 239 (24%)	40 / 171 (23%)
bevy	44 / 328 (13%)	14 / 264 (5%)
junit-framework	5 / 117 (4%)	0 / 35 (0%)
rust-analyzer	4 / 203 (2%)	2 / 86 (2%)

Except at trino, queue failures hit pull requests that Sheltie would land at once as often as, or more often than, those it would rebuild. At trino and gradle, about a quarter of the pull requests Sheltie would land at once failed in the queue at least once. Footprint overlap is a weak predictor of which queue build fails.

Of the 223 pull requests that failed in the queue and that Sheltie would land at once, 165 went back into the queue with no new commits and merged. Those failures were most likely flaky or caused by another entry in the same build; we did not check each one. To find the rest, we took every pull request that failed in the queue, that Sheltie would land at once, and that got new commits afterwards. For each one we compared the pull request's own diff at the time of the failure with its final diff, and read the failed jobs and follow-up commits:

Outcome after the queue failure	trino	gradle	bevy	junit-framework	rust-analyzer	Total
change to CI workflow, only exercised in the queue	2	0	0	0	0	2
code change after a PR-CI job failed in the queue; cause not determined	0	0	1	0	0	1
code change; failing job and cause not determined	5	0	0	0	2	7
failing jobs run only in the queue (wasm, iOS/Android, examples, unused deps)	0	0	4	0	0	4
repository-wide check (distribution size, API list, version list, performance baseline)	0	10	0	0	0	10
semantic conflict with main ("Fix example merge conflict")	0	0	1	0	0	1
test fix after a PR-CI job failed in the queue; cause not determined	0	0	1	0	0	1
test or code fix; PR CI is a subset of queue CI, failing test not identified	0	14	0	0	0	14
diff unchanged: rebase or merge of main only	5	4	8	1	0	18
All candidates	12	28	15	1	2	58

- **18** only rebased or merged `main`; their diff did not change.
- **14** failed checks that run only in the queue, which Sheltie would need on the pull request:
 - 10 are repository-wide checks at gradle (distribution size, accepted API list, version lists, performance baselines). Any change can break these in combination with others, and no footprint captures them.
 - 4 at bevy, whose failing jobs run only in the queue.
- **14** are other gradle test or code fixes after a queue failure. Gradle's pull request CI is a subset of its queue CI, so the failing test most likely did not run on the pull request, but we did not identify it (gradle's queue CI is on TeamCity, outside the GitHub API).
- **2** changed trino's CI workflows, which are only exercised in the queue.
- **1** is a semantic conflict with `main` (bevy: "Fix example merge conflict"). Footprints should have caught it and did not.
- **9** have a code change whose cause we could not determine: 5 at trino, 2 at rust-analyzer, 2 at bevy.

This is the honest worst case: on the three projects whose queue runs the same CI as pull requests (1,263 changes), 9 pull requests changed after a queue failure, and Sheltie would have landed them without a rebuild. For 7 of them we cannot say whether the failure was an interaction footprints missed. In those projects Sheltie's sweeper (post-landing CI with bisect and revert) would be the safety net. We did not simulate it.

Projects without a merge queue

How it was measured

- **CI** is the build and test workflows that run both on pull requests and on pushes to `main`. Security audits, release, CodeQL, Copilot setup, zizmor, typos, CLA and labeller workflows are left out.
- **Stale at merge**: something else landed between the start of the pull request's last CI and its merge.
- **Sheltie would rebuild** a pull request when one of those landings writes something it reads.
- **Extra CI** costs each such rebuild one full pull request CI run of wall time. That is an upper bound: Sheltie reruns only affected tests.
- **Persistent red episode**: two or more consecutive failed runs of a CI workflow on `main`. The candidates are the landings since the last green run. An episode counts as "with an overlapping landing" when a candidate was stale and something that landed in its gap overlaps it.
- **Chance baseline**: how many episodes would show an overlapping landing if overlap had nothing to do with breaking. Each candidate is taken to overlap at that project's overall rate.

Results

Project	Adapter	PRs	PR CI	Stale at merge	Sheltie would rebuild	Extra CI (wall time, upper bound)	Leaky footprints	Persistent red episodes	...with an overlapping landing: seen vs by chance
ruff	cargo	1388	12 min	726/1295 (56%)	440 (34%)	3 d 23 h	790 (61%)	6	2 vs 2.4
nushell	cargo	364	23 min	186/346 (54%)	87 (25%)	33 h 21 min	179 (52%)	0	0 vs 0.0
tokio	cargo	163	1 h 16 min	75/162 (46%)	36 (22%)	45 h 38 min	8 (5%)	1	0 vs 0.2
hugo	go	144	1 h 5 min	81/143 (57%)	64 (45%)	2 d 21 h	21 (15%)	5	1 vs 2.2
prometheus	go	414	28 min	325/414 (79%)	220 (53%)	4 d 6 h	95 (23%)	20	12 vs 10.6
okhttp	gradle	147	29 min	21/38 (55%)	14 (37%)	6 h 59 min	6 (16%)	2	2 vs 0.7
keycloak	maven	844	59 min	758/840 (90%)	559 (67%)	23 d 4 h	515 (61%)	n/m	n/m
vscode	node	5221	8 min	4675/5216 (90%)	4550 (87%)	28 d 4 h	5110 (98%)	11	10 vs 9.6
storybook	node	142	n/m	89/125 (71%)	74 (59%)	n/m	8 (6%)	n/m	n/m
vite	node	321	6 min	162/317 (51%)	98 (31%)	11 h 9 min	36 (11%)	21	7 vs 6.5
vitest	node	370	15 min	76/367 (21%)	57 (16%)	14 h 19 min	19 (5%)	30	2 vs 6.0
fastapi	python	178	5 min	57/178 (32%)	11 (6%)	1 h 3 min	10 (6%)	1	0 vs 0.1
home-assistant	python	6293	4 min	5660/6291 (90%)	2068 (33%)	6 d 3 h	3029 (48%)	52	34 vs 33.6
pydantic	python	220	12 min	77/220 (35%)	55 (25%)	11 h 16 min	54 (25%)	5	2 vs 1.2
cpython	python	1211	32 min	919/1211 (76%)	691 (57%)	15 d 10 h	914 (75%)	22	11 vs 12.6
Total								176	83 vs 85.8

n/m means not measurable:

- keycloak runs no CI on pushes to `main`, so red `main` is invisible;
- storybook's main CI is not on GitHub Actions;
- only 38 of okhttp's 147 pull requests had matching CI suites.

All persistent red episodes, by what their candidates looked like:

- stale base, overlapping landing: Sheltie rebuilds before landing: 83
- tested on the current main: not an interaction: 64
- stale base, no overlap: Sheltie lands it too: 27

- direct push or untested change in range: 2
- **Stale merges are the norm, but they do not explain red `main`**. Across 176 persistent episodes, an overlapping landing was in range 83 times, against 85.8 expected by chance. We found no evidence that Sheltie's rebuilds would have prevented these breakages.
- **64 episodes started at changes tested on the current `main`**. Flaky tests, infrastructure and environment changes break `main` without any interaction; vitest alone had 26.
- **27 episodes had stale changes with no overlap**. If any of them was a real interaction, Sheltie's footprints would have missed it too.
- **Sheltie would cost CI here**. It would rebuild 6–87% of pull requests. The largest shares come from coarse footprints: vscode's root `install` script puts the whole repository in every read set, and keycloak's Maven build plugins do the same for its test suite.

Adapter coverage

A footprint is leaky when the change reads something the adapter cannot model exactly:

- a build script;
- a proc macro;
- an install script;
- a build plugin;
- a file outside the module graph.

Its read set then widens to everything that leak could reach. Leaky does not mean serial, but it makes rebuilds more likely. This is the share of landings with a leaky footprint after the fixes below:

Project	Adapter	Leaky footprints	Most common reason
ruff	cargo	858/1416 (61%)	build script crates/ty/build.rs
bevy	cargo	282/592 (48%)	included file ci_testing_config.ron not in tree
fastapi	python	10/326 (3%)	scripts/translate.py is outside the module graph
hugo	go	35/182 (19%)	testscripts/commands/no_symlinks.txt is outside the module graph
gradle	gradle	129/410 (31%)	released-versions.json is outside the module graph
home-assistant	python	3029/6298 (48%)	setup.py builds extensions
junit-framework	gradle	24/228 (11%)	documentation/.tool-versions is outside the module graph
keycloak	maven	517/846 (61%)	build plugin runs code (maven-antrun-plugin)
kubernetes	go	271/925 (29%)	test/compatibility_lifecycle/reference/versioned_feature_list.yaml is outside...
vscode	node	5172/5282 (98%)	install script
nushell	cargo	195/368 (53%)	build script scripts/build.rs
prometheus	go	95/415 (23%)	web/ui/pnpm-lock.yaml is outside the module graph
pydantic	python	54/221 (24%)	pydantic-core/src/serializers/shared.rs is outside the module graph
cpython	python	917/1214 (76%)	setup.py builds extensions
rust-lang/rust	cargo	417/440 (95%)	cannot parse compiler/rustc_trait_selection/src/error_reporting/infer/need_ty...
rust-analyzer	cargo	160/289 (55%)	build script crates/rust-analyzer/build.rs
okhttp	gradle	9/153 (6%)	okhttp/api/android/okhttp.api is outside the module graph
storybook	node	18/162 (11%)	scripts/build/entry-configs.ts is outside the module graph
tokio	cargo	8/166 (5%)	proc-macro crate

Project	Adapter	Leaky footprints	Most common reason
traefik	go	25/37 (68%)	integration/gateway-api-conformance-reports/v1.6.1/experimental-v3.8-default-...
trino	maven	395/746 (53%)	build plugin runs code (exec-maven-plugin)
vite	node	37/328 (11%)	vitest.config.e2e.ts is outside the module graph
vitest	node	19/370 (5%)	knip.jsonc is outside the module graph

The causes that remain are mostly deliberate caution:

- **Code that runs at build time:** Cargo build scripts and proc macros, npm install scripts, Maven plugins that run code, Python `setup.py` extensions.
- **Languages an adapter does not read:**
 - ruff's Python packaging;
 - pydantic's Rust core;
 - prometheus' web UI;
 - files under `test/` read at run time.
- **rust-lang/rust (95%):** `syn` cannot parse the unstable syntax the compiler uses, so Sheltie claims nothing about those crates.
- **bevy:** a `wasm32`-only `include_str!` of a file that CI generates.

What the study changed in Sheltie

Problem found	Effect in the data	Fix
Cargo: <code>mod r#type;</code> and <code>#[path]</code> in non- <code>mod.rs</code> files not resolved	leaky footprints: rust-analyzer 64% → 55%, ruff 76% → 61%	#104
Cargo: modules declared inside macros (<code>cfg_rt! { mod pool; }</code>) not followed	tokio 71% → 5%	<code>sheltie-cargo</code> follows <code>mod</code> items in macro bodies
Go: every <code>//go:embed</code> made a package non-hermetic	hugo 76% → 21%, prometheus 62% → 23%, kubernetes 50% → 29%	embed patterns resolved by the go command's rules; embedded files belong to the package
Node: a <code>package.json</code> with a byte order mark was rejected	vite 25% → 11%	the mark is stripped, as Node does
<code>.circleci/**</code> treated as build input	hugo 21% → 19%	plain for every adapter, like <code>.github/**</code>
Two checkouts of one commit in one process collided	14 of about 24,000 footprints failed in the bench	worktree names are unique per call
ROI report assumed every change was tested on <code>main</code> as of landing	With that assumption, Sheltie needed 0 CI runs on five of six queue projects (3 at junit) instead of 36–348. The shipped report overstated savings.	the report finds each change's base from merge commits or fetched pull request heads, and says how many changes lack one (ROI report)

What we cannot prove

These are the limits of this study. Read every result above with them in mind.

- **These are simulations.** No project ran Sheltie.
 - The replay assumes every CI run passes and every merge-engine resolution succeeds.
 - It uses one CI time per project (the median) for every build.
 - Real projects would also change how they work if they adopted it.

- **Whether Sheltie is safe to land at once on these projects.** 9 pull requests on projects with equal queue and pull request CI changed after a queue failure that Sheltie would have skipped. For 7 of them we could not identify the failing job or the cause. The causes of the other 31 changed pull requests come from reading their commits and changed files, and for 14 at gradle the failing test was not identified. We did not rerun anything.
- **Why `main` went red.** The red-episode analysis is statistical. We did not read the 176 failures, and the chance baseline treats overlap as independent of everything else.
- **Required checks.** GitHub does not expose which checks a project requires, so failures are counted from the queue's own ejection events, not from workflow conclusions.
- **Ready time is inferred** from timeline events. Review and policy delays that happen outside those events count as queue time; this is why kubernetes is left out.
- **The sample is not random.** We picked popular projects with clear setups across six build tools.
 - They are mostly human-written pull requests; Sheltie is built for agent-generated ones, where volume and overlap are higher.
 - Nothing here measures that case.
- **One window.** 90 days, ending 2026-10-19. Release cycles and freezes inside it are not separated out.

What was verified, and how

Claim	How it was checked	Verified
Each project's merge setup	<code>merge_group</code> runs, bors comments, Tide statuses, label events in the collected data	Yes
Merge-queue landing time is <code>mergedAt</code> , not the commit time	the queue commit's committer time equals its build start (checked on junit)	Yes
Queue CI vs pull request CI	workflow files (bevy <code>validation-jobs.yml</code> , trino <code>ci.yml</code> , rust <code>jobs.yml</code> : 11 PR / 82 auto jobs); gradle statuses on PR and queue commits	Yes
gradle queue CI time (1 h 19 min)	GitHub Merge Queue Check Pass status on all 410 landed queue commits	Yes
Ejections and their reasons	<code>RemovedFromMergeQueueEvent.reason</code>	Yes
Footprints are what Sheltie computes	the released adapters at each landing; 0 errors in the final set	Yes
The replay is Sheltie's	the bench checks every replay against <code>sheltie_scheduler::roi::replay</code> and stops on any difference	Yes
18 escape candidates only rebased	pull request diff at the failure equals the final diff	Yes
Causes of the 40 escape candidates whose diff changed	read from failed jobs, follow-up commits and changed files	Partly (9 undetermined; for 14 at gradle the failing test is not identified)
Red <code>main</code> is not explained by stale overlap	statistical baseline only	No (no per-episode cause)
Required status checks	not public	No
Sheltie's behaviour in production on these projects	simulation only	No

Reproduce it

`research/real-world/README.md` lists the steps: survey, collection, footprints, analyses and table generation.
`results/tables.md` holds the tables above, generated from `results/*.json`.