Working in the Lix codebase

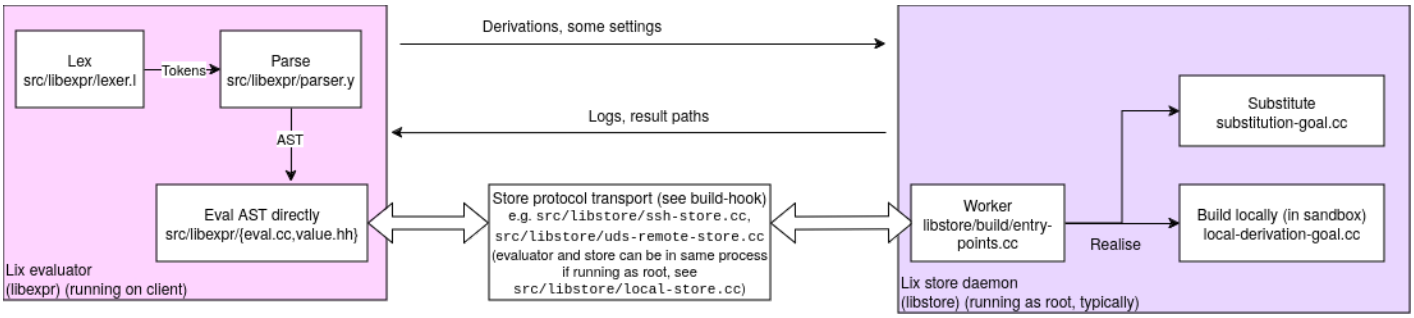
See also: See also: https://git.lix.systems/lix-project/lix/src/branch/main/doc/manual/src/contributing/hacking.md

Codebase overview

The Lix system is constituted of two broad parts, the evaluator and the store daemon. The two pieces may run on the same machine or on different machines.

For example, in a remote build setup like https://hydra.nixos.org, one node is running several evaluators in parallel, and builds are dispatched to several builder nodes.

(fyi to anyone editing this, double click the image in the preview to edit it)



Evaluator

The evaluator is an AST tree-walking evaluator with lazy semantics.

Notable files:

- libexpr/value.hh, which defines the interface for evaluated values' interactions.
- libexpr/eval.cc, where most of the evaluator is.
- libexpr/nixexpr.cc, where the most of the `nix::Expr` class hierarchy is implemented, which are the AST types for the evaluator.
- libexpr/primops.cc, defining builtins.
- libexpr/parser.y, the (current) yacc generated parser.
- libexpr/lexer.l, the bison-generated lexer.

Known design flaws

- GC issues (FIXME add details)
  - General tendencies to leak memory. Hydra restarts the evaluator every so often if it runs out of memory.
- AST based evaluator design limits perf
- Stack tracing has issues that make the traces confusing (FIXME add details)
- Funniness with attr ordering and equality that nixpkgs depends on, which is fragile

    - Currently no real tools to diagnose this and stop nixpkgs from depending on it.

      https://github.com/NixOS/nix/pull/8711 exists but regresses perf a lot and is not mergeable.
- Evaluation-time build dependencies (often called IFD) block the evaluator rather than allowing other evaluation to proceed
    - This has significant downstream effects such as typical derivations building hand-written large pieces rather than generated smaller pieces with IFD, since IFD is bad.
- The eval cache both has false hits and false misses, and needs redesign.

Lix team plans

- Rewrite parser (done, being ported for 2.91 by horrors)
- Rewrite evaluator to be amenable to moving to bytecode (horrors) (long term)
- Do something about GC (long term)

Store protocol

The store protocol is a hand rolled binary protocol with monotonically increasing versioning. It runs over a few different transports such as ssh (`src/libstore/ssh-store.cc`) or Unix sockets (`src/libstore/uds-remote-store.cc`).

Known design flaws

- We cannot extend the store protocol (not that it is Good) because of the monotonic version numbers: we must always be stuck at *some* released CppNix version. This significantly moves up the need to replace it.
    - The code is significantly tangled with the current protocol design.

Lix team plans

- Replace protocol with capnproto, transport with websockets?
    - Would likely be in addition to existing protocol; existing protocol likely would be run through a translator.

Store daemon

The store daemon takes derivations (≈ execve args and dependencies) and realises (builds or substitutes) them. It also implements store path garbage collection.

Lix's local store implementation currently uses a SQLite database as the source of truth for mapping derivation outputs to derivations as well as maintaining derivation metadata.

Notable files:

- libstore/build/local-derivation-goal.cc, which implements the local machine's builder including the sandbox
- libstore/build/entry-points.cc, the server side entry points of the store protocol
- libstore/daemon.cc
- libstore/uds-remote-store.cc, the client implementation of Unix socket stores

Known design flaws

- Sandbox is of dubious security especially on Linux (where it is actually expected to be somewhat secure)
  - Overall tangled code around the sandbox, particularly in platform specific parts
- Poor self-awareness: daemon doesn't know what it is building
  - Due to this plus the protocol being frozen, it would be very hard to implement e.g. dropping into a shell on failed builds
- Substitutions are inherently a kind of build so they can't happen out of dependency order or with better parallelism
- SQLite database has a habit of getting corrupted (probably due to Lix-side misuse)

Lix team plans

- Replace sandbox with other software, perhaps bwrap
- Fix daemon self-awareness, add protocol level features to make this better
- Rearchitect substitution to enqueue weakly ordered jobs that happen in parallel and can resume downloads
- Switch to xattrs as the source of truth of store path metadata such that the SQLite DB can be completely rebuilt

Freezes and recommended contributions

Suggested contributions

Consider taking an issue marked E-help wanted: assign it to yourself and have a go. Feel free to ask for help in the development channel. The Lix team wants these issues fixed, but they are not high on our agenda to fix ourselves.

When in doubt, please ask the Lix team before beginning work, to make sure it is in line with our current priorities.

Freezes

This document describes the state of freeze that Lix is in.

We *do* expect to have `main` always be in a state to run on machines you care about, unconditionally. Nightly builds should not be a problem to run in production in any freeze state.

The purpose of this policy is to set expectations of what we are looking for in contributions, rather than to set hard rules.

> Lix is currently in status "*milkshake*".

ice cube

No major features or code changes are accepted that touch the core (e.g. the hot paths of the evaluator, the store), absent a good justification, good test coverage and a strong belief that they will not cause regressions. In this state, we don't recommend external contributors do substantive work outside the roadmap without speaking with the Lix team first. However, this is a guideline, and such work can be done if discussed and planned carefully beforehand.

Simple surface features with low impact are likely to be accepted with tests, assuming that they do not impact reliability.

New tests are gleefully accepted.

Bug fixes (with tests) are gleefully accepted.

For example, the following would require discussion with the Lix team before work begins, as it is likely to not fit our goals:

- Adding new features to ca-derivations
- Doing substantial not-obviously-correct refactoring to the evaluator or daemon

For example, the following would likely be accepted assuming it has tests, without needing prior discussion:

- Backports of CLI UX features from CppNix
- New UX features in the REPL or in output of other commands considered to not have stable output
- Backtrace improvements that don't touch hot paths
- Bug fixes (to non load bearing bugs; be careful around evaluator semantics!)
- Improvements to development UX

hard ice cream

Changes that improve maintainability of the core are accepted, with careful review depending on their significance. Changes that add more complexity to the core need to pass scrutiny.

Features at the edge are accepted if they have low impact, assuming that they have tests.

soft serve

FIXME

milkshake

All kinds of changes are acceptable, but we still strive to keep `main` always as stable as possible and a safe decision to daily-drive for all your nixy needs. Please don't jam the ice cream machine!

Bug tracker organisation

We have a repo of directly imported nix bugs at https://git.lix.systems/nixos/nix. Please don't file bugs in there, we want the IDs to match. When we import a bug, we might put notes on there as we triage it, and potentially close it.

Bug labels on NixOS/nix

- lix-import - Should be imported, we think it is still a bug
- lix-ignore - We don't care about this bug, it probably doesn't affect us
- lix-stability - Fixing this would improve the stability and reliability of Lix.

Dispositions:

- lix-norepro - Tried repro on upstream 2.18.1 and did not repro
- lix-retest-after-backports - Request that this be tested again once backports are done
- lix-reproduces-2.18 - Confirmed to repro in 2.18.
- lix-unclear-repro - Unsure how to repro but believe it affects lix
- lix-closed-libgit2 - Caused by libgit2
- lix-closed-lazy-trees - Caused by lazy trees

Closed, marginal

- post-build-hook doesn't print a warning if not trusted-user
  https://git.lix.systems/NixOS/nix/issues/9790#issuecomment-273
- complaints about builtins.derivation https://git.lix.systems/NixOS/nix/issues/9774
- rejecting flake config still asks for confirm again
  https://git.lix.systems/NixOS/nix/issues/9788
- complaints of "substituter disabled", but is their bin cache just broken?
  https://git.lix.systems/NixOS/nix/issues/9749
- warn on eol https://git.lix.systems/NixOS/nix/issues/9556

Gerrit

What is Gerrit and why do people like it?

Gerrit is a code review system from Google in a similar style to Google's internal Critique tool, but based on Git, and publicly available as open source. It hosts a Git repo with the ability to submit changes for review and offers mirroring to other repos (like https://git.lix.systems/lix-project/lix). It has an entirely different review model to GitHub (and Forgejo, GitLab, etc that copy GitHub's review model), where, instead of pull requests, you have changelists (CLs): reviews on individual commits, with each revision of a commit being a different "patchset", rather than reviewing an entire branch at a time. CLs may be merged one by one or in a batch.

Although this has some learning curve, we expect that you will find it pleasant to work with after figuring it out. It has some rough edges and strong opinions that take some getting used to, but it has served us well and saved us an inordinate amount of time both as reviewers and change authors. The rest of this document gives some pointers on the workflows we use with Gerrit.

People like Gerrit because it makes the following things trivial or easy, all of which are somewhere between annoying and impossible on GitHub modeled systems:

Gerrit produces better code:

- Gerrit enforces good commit messages, since there is no second "pr message" so peoples' commit messages get actually looked at with some care
- Gerrit enforces good commit *hygiene*, since adding another commit is really just splitting a commit with `git revise -c` or other tools; since there are no PR dependencies or branches to worry about, splitting commits is no longer a big ask.
  - Relatedly, this directly makes reviews smaller since the overhead of doing another change is low.

Gerrit makes reviewers' lives easier and reduces review round trips:

- As a reviewer, you can look at what changed since you last reviewed, even in the presence of rebases, by looking at the patchset history of a CL. This avoids pointless rereview; you can actually diff versions of changes properly.
- The *change author* generally merges the change after approval, without them needing commit access. This means that they can do a final once-over of the change and make sure that they are ok with its state before merging it. This reduces miscommunication causing merging of unfinished code.
- As a reviewer, you can edit someone's change and/or commit message to fix a typo (*in the web interface*) and then stamp it, while giving them the final say on merging the edited change.

- You can give feedback like the following: "I would merge this as-is but you can consider this feedback if you would like" and then let the change author decide to merge it.
  - Since the permission-requiring step in Gerrit is *approving* the change, not merging it, every change author can have final say in when the change gets merged.
- Review suggestions get applied as a batch without cluttering commit history in a confusing manner.
- You can download someone's change to look at it locally in one command that you can copy paste from the Gerrit interface.

Gerrit makes your life easier as a contributor:

- Submitting a new change is just a matter of committing it and pushing it. You don't need to think about branches or the web interface or extra commands. Want to do more changes building on it? Just commit them and push them.
- Branches are not required and you can easily build off of other peoples' changes by fetching them and rebasing against them; change dependencies are simply commit parents. They can then be merged in whichever manner they will be merged.
- If you are doing a larger change, it is natural to merge it piece by piece, adding little improvements as you go, and putting the highest risk parts of it at the tip, making the obviously good parts of the change land and keeping your diffs and rebases against `main` smaller.
- Gerrit makes it clear which comments still need action in a clean way, compared to GitHub where resolved comments get regularly broken or disappear altogether.
- Gerrit guesses (with reasonable accuracy) who a change is blocked on and shows it on the dashboard with a little arrow next to their name, allowing you to see at a glance which changes are your responsibility at a given time.
- There is a rebase button that just works. Trivial non-conflicting rebases do not require a rereview.

That being said, there are some downsides:

- Gerrit is very mean to you if you don't have your commit history in a clean presentable state, which takes some getting used to and Git does not make editing history easy, so it does involve a little more fighting of Git. However, this also means that the reviews can be of cleaner and smaller pieces of code with fewer unrelated changes.
  - This makes pushing work in progress code with questionable commit history harder; see below for solutions to this.
- Gerrit requires a little bit of local setup in the form of adding your SSH key or setting up the HTTP password. It also requires a Git commit-msg hook, but `nix develop` automatically does that for you.

Learning materials

- https://gerrit-review.googlesource.com/Documentation/intro-user.html

- https://docs.google.com/presentation/d/1C73UgQdzZDw0gzpaEqIC6SPujZJhqamyqO1XOHjH-uk/view

Our installation

Gerrit is at https://gerrit.lix.systems

The Gerrit SSH server is running on port 2022. The repo URLs are:

- `ssh://{username}@gerrit.lix.systems:2022/lix`
- `https://gerrit.lix.systems/lix` if using HTTP auth; see Gerrit settings for setting an HTTP password if desired

Hit the `d` key on any change to download it, which will give you the right URLs.

SSH config

You might like to add the following configuration to your `~/.ssh/config`:

```
Host gerrit.lix.systems
  User YOUR_GERRIT_USERNAME
  Port 2022
  # Keep sessions open for a bit in the background to make connections faster:
  ControlMaster auto
  ControlPath /tmp/ssh-%r@%h:%p
  ControlPersist 120
```

Basic workflow for a change

The unit of code review is a "change", which yields a single commit when "submitted" (merged). The commit message is taken from the change description in Gerrit; in our experience this tends to lead to more comprehensive commit messages.

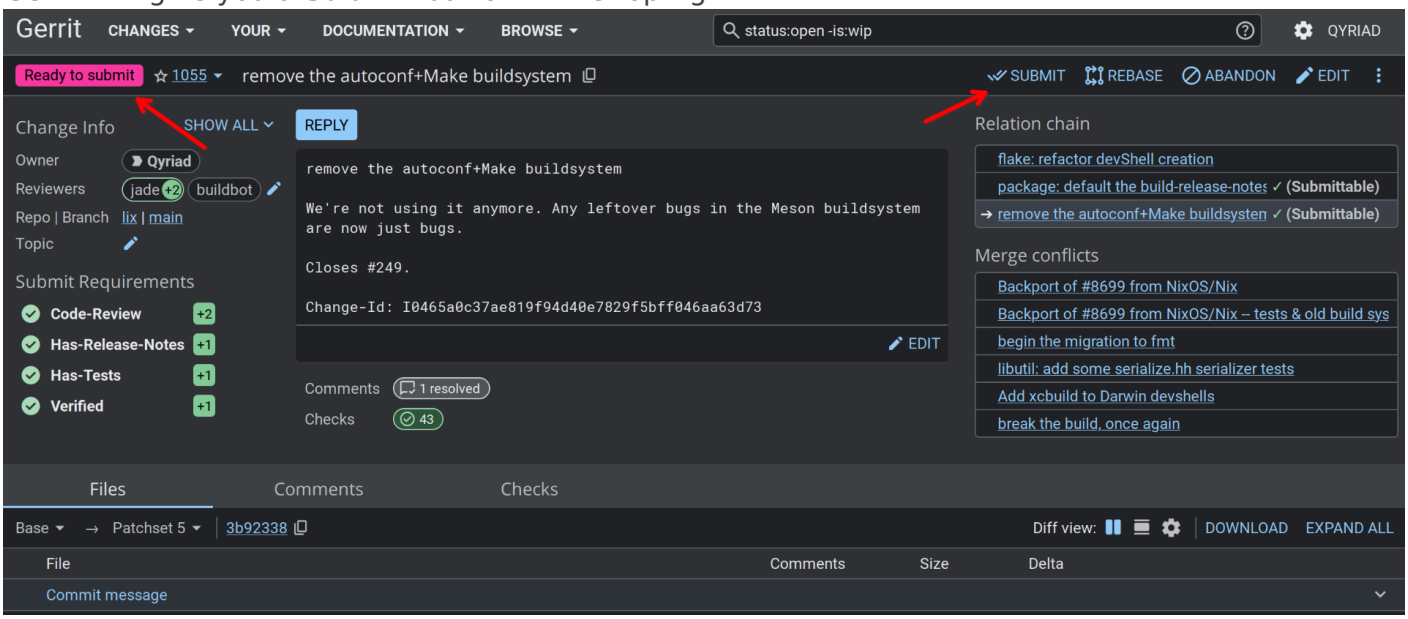
For a change to be merged, it must have the following four "votes", in Gerrit's terminology:

- Set by reviewers:
  - +2 Code-Review: the committer that reviewed this thinks it can be submitted as-is (all users can vote +1/-1, expressing a weaker view on code acceptability)
  - +1 Has-Release-Notes: means the reviewer thinks your commit added relevant release notes for that commit, or that it does not need any. This serves primarily as a reminder.
  - +1 Has-Tests: means the reviewer thinks your commit added all the tests that commit needs, or that it does not need additional tests. Like Has-Release-Notes, this

serves primarily as a reminder.
- Set automatically by CI:
  - +1 Verified: means CI successfully built for all our platforms and passed all tests

If you're newly part of the core team you will need to add yourself to the Gerrit `lix` group, otherwise you can't set the `Has-Release-Notes` or `Has-Tests` labels. If you're not, this doesn't affect you.

When all of those labels are set, a change becomes **Ready to submit**, in Gerrit's termology, and Gerrit will give you a **Submit** button in the top right:



By convention, ***the change author*** *has the final say on clicking the Submit button* (note: this is the opposite of the Github convention), and there is no special permission to merge a change once it has been fully reviewed (the permissions are in the reviewer +2'ing it). This gives you a last chance to have a look at your change before merging it.

Workflow tips

Local branches and commits

Gerrit is very mean to you if you don't have your local commit history in a linear presentable state, which takes getting used to but it is very low overhead once you get used to it. In short, amended commits become "patchsets", new commits become changes, and multiple commits help link your changes together as a "relation chain".

Note: if you're coming from Chromium, this is different to how they use Gerrit, where multiple commits become patchsets, and only the first commit on a local branch creates a new change.

Gerrit's commit-msg hook generates a new Change-Id for each commit you make, which in turn creates a new change that gets reviewed separately. To update an existing change after review feedback, amend or squash your changes into your old commit, keeping its Change-Id unchanged,

then push.

Consider not pushing for review before it is clean, or split commits up with `git-revise` (good) or `jj` (better) after the fact, amending as you work. If you want a backup of your changes, you can fork it on Forgejo and push to that fork.

Basic Pushing

If you cloned the repo from Forgejo, be sure to change your remote URL to point to Gerrit before continuing. Assuming your remote is called `origin` (which is the default):

`git remote set-url origin ssh://{username}@gerrit.lix.systems:2022/lix`

Then you can push to Gerrit with:

`git push origin HEAD:refs/for/main`

If you get tired of doing this every time, you can automate it by setting the `.git/config` as follows:

`git config remote.origin.push HEAD:refs/for/main`

You will have to do that in each fresh check-out. Once it's done, `git push` will work without additional options.

If you get a “remote unpack failed” error while pushing, run `git fetch` then try again.

If you wish to push a change and immediately mark it as WIP, you can push with `-o wip`, or make that the default behavior by checking `Set new changes to "work in progress" by default` in Gerrit's user settings, under "Preferences".

Topics & Push Arguments

A Gerrit topic may be set on push with:

```
git push origin HEAD:refs/for/main%topic=foo
```

Which will create all pushed changes with the topic "foo". Topics are helpful for grouping long series of related changes.

A change may also be marked as "work in progress" on push:

```
git push origin HEAD:refs/for/main%wip
```

Gerrit has documentation on other push arguments you can use here, but it also takes a `help` argument whose output is more canonical and might be easier to understand, which you can view with:

```
git push origin HEAD:refs/for/main%help
```

At the time of this writing (2024/06/26), that output looks like this:

```
$ git push origin @:refs/for/main%help
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Processing changes: refs: 1, done
remote:
remote: Help for refs/for/branch:
remote:
remote:  --base BASE                      : merge base of changes
remote:  --cc CC                          : add CC to changes
remote:  --create-cod-token                 : create a token for consistency-on-dema
remote:                                   nd (default: false)
remote:  --deadline NAME                    : deadline after which the push should
remote:                                   be aborted
remote:  --edit (-e)                       : upload as change edit (default: false)
remote:  --hashtag (-t) HASHTAG               : add hashtag to changes
remote:  --help (-h)                       : display this help text (default: true)
remote:  --ignore-automatic-attention-set-rules : do not change the attention set on
remote:  (-ias, -ignore-attention-set)          this push (default: false)
remote:  --label (-l) LABEL+VALUE             : label(s) to assign (defaults to +1 if
remote:                                   no value provided)
remote:  --merged                          : create single change for a merged
remote:                                   commit (default: false)
remote:  --message (-m) MESSAGE               : Comment message to apply to the review
remote:  --no-publish-comments (--np)         : do not publish draft comments
remote:                                   (default: false)
remote:  --notify [NONE | OWNER |             : Notify handling that defines to whom
remote:  OWNER_REVIEWERS | ALL]                 email notifications should be sent.
remote:                                   Allowed values are NONE, OWNER,
remote:                                   OWNER_REVIEWERS, ALL. If not set, the
remote:                                   default is ALL.
remote:  --notify-bcc USER                  : user that should be BCC'd one time by
remote:                                   email
remote:  --notify-cc USER                   : user that should be CC'd one time by
remote:                                   email
remote:  --notify-to USER                   : user that should be notified one time
remote:                                   by email
remote:  --private                         : mark new/updated change as private
```

```
remote:                                  (default: false)
remote:  --publish-comments                    : publish all draft comments on updated
remote:                                  changes (default: false)
remote:  --ready                         : mark change as ready (default: false)
remote:  --remove-private                      : remove privacy flag from updated
remote:                                  change (default: false)
remote:  --reviewer (-r) REVIEWER              : add reviewer to changes
remote:  --skip-validation               : skips commit validation (default:
remote:                                  false)
remote:  --submit                        : immediately submit the change
remote:                                  (default: false)
remote:  --topic NAME                    : attach topic to changes
remote:  --trace NAME                    : enable tracing
remote:  --wip (-work-in-progress)             : mark change as work in progress
remote:                                  (default: false)
remote:
To ssh://gerrit.lix.systems:2022/lix
 ! [remote rejected]     HEAD -> refs/for/main%help (see help)
error: failed to push some refs to 'ssh://gerrit.lix.systems:2022/lix'
```

Pulling

Pulling from Gerrit will work normally. It's worth keeping in mind that sometimes a CL you're working on has been edited in the web UI or by another contributor, so the commit in your repo isn't the latest. Rebasing will usually make the duplicate go away; this is part of the normal rebase semantics, not Gerrit magic. You might consider making rebase-on-pull your default.

Sandbox branches

> This feature has some notable ways to shoot yourself in the foot. We still support it, since it allows for running CI builds on things before they become proper CLs. If you don't need that and don't want to worry about the footguns, consider using a branch on a Forgejo fork for sharing WIP code.
>
> In particular, if a commit is in *any* branch already including a `sb/` branch, it will be rejected with the error "no new changes" if it is later pushed to `refs/for/main`. This can be worked around by amending all the commits so they are distinct, or by `git push origin HEAD:refs/for/main%base=$(git rev-parse origin/main)`, which forces the merge-base

Use `refs/heads/sb/USERNAME/*`.

CI rerun

Push the CL again with a no-changes commit amendment if you want to force CI to rerun.

Finding CLs to review

Consider bookmarking: https://gerrit.lix.systems/q/status:open+-is:wip+-author:me+label:Code-Review%3C2

Improving build times

Setup

Use a clang stdenv:

`nix develop .#native-clangStdenvPackages`

Then delete `build/` if you were using gcc before. Enable build-time profiling with:

`just setup; meson configure build -Dprofile-build=enabled`

Then run the build: `just compile`.

Enabling build-time profiling itself costs about 10% of compile time but has no other disadvantage.

Build time reports

Use `maintainers/buildtime_report.sh build/` to generate a build time report. This will tell you where all our build time went by looking at the trace files and producing a badness summary.

Sample report

Build-time report sample

```
lix/lix2 » ClangBuildAnalyzer --analyze buildtimeold.bin
Analyzing build trace from 'buildtimeold.bin'...
**** Time summary:
Compilation (551 times):
  Parsing (frontend):         1465.3 s
  Codegen & opts (backend):   1110.9 s

**** Files that took longest to parse (compiler frontend):
 10478 ms: build/src/libstore/liblixstore.so.p/build_local-derivation-goal.cc.o
 10319 ms: build/src/libexpr/liblixexpr.so.p/primops.cc.o
  9947 ms: build/src/nix/nix.p/flake.cc.o
  9850 ms: build/src/libexpr/liblixexpr.so.p/eval.cc.o
```

9751 ms: build/src/nix/nix.p/profile.cc.o
9643 ms: build/src/nix/nix.p/develop.cc.o
9296 ms: build/src/libcmd/liblixcmd.so.p/installable-attr-path.cc.o
9286 ms: build/src/libstore/liblixstore.so.p/build_derivation-goal.cc.o
9208 ms: build/src/libcmd/liblixcmd.so.p/installables.cc.o
9007 ms: build/src/nix/nix.p/..._nix-env_nix-env.cc.o

**** Files that took longest to codegen (compiler backend):
24226 ms: build/src/libexpr/liblixexpr.so.p/primops_fromTOML.cc.o
24019 ms: build/src/libexpr/liblixexpr.so.p/primops.cc.o
21102 ms: build/src/libstore/liblixstore.so.p/build_local-derivation-goal.cc.o
16246 ms: build/src/libstore/liblixstore.so.p/store-api.cc.o
14586 ms: build/src/nix/nix.p/..._nix-build_nix-build.cc.o
13746 ms: build/src/libexpr/liblixexpr.so.p/eval.cc.o
13287 ms: build/src/libstore/liblixstore.so.p/binary-cache-store.cc.o
13263 ms: build/src/nix/nix.p/profile.cc.o
12970 ms: build/src/nix/nix.p/develop.cc.o
12621 ms: build/src/libfetchers/liblixfetchers.so.p/github.cc.o

**** Templates that took longest to instantiate:
42922 ms: nlohmann::basic_json<>::parse<const char *> (69 times, avg 622 ms)
32180 ms: nlohmann::detail::parser<nlohmann::basic_json<>, nlohmann::detail::i... (69 times, avg 466 ms)
27337 ms: nix::HintFmt::HintFmt<nix::Uncolored<std::basic_string<char>>> (246 times, avg 111 ms)
25338 ms: nlohmann::basic_json<>::basic_json (293 times, avg 86 ms)
23641 ms: nlohmann::detail::parser<nlohmann::basic_json<>, nlohmann::detail::i... (69 times, avg 342 ms)
20203 ms: boost::basic_format<char>::basic_format (492 times, avg 41 ms)
17174 ms: nlohmann::basic_json<>::json_value::json_value (368 times, avg 46 ms)
15603 ms: boost::basic_format<char>::parse (246 times, avg 63 ms)
13268 ms: std::basic_regex<char>::_M_compile (28 times, avg 473 ms)
12757 ms: std::__detail::_Compiler<std::regex_traits<char>>::_Compiler (28 times, avg 455 ms)
10813 ms: std::__detail::_Compiler<std::regex_traits<char>>::_M_disjunction (28 times, avg 386 ms)
10719 ms: std::__detail::_Compiler<std::regex_traits<char>>::_M_alternative (28 times, avg 382 ms)
10508 ms: std::__detail::_Compiler<std::regex_traits<char>>::_M_term (28 times, avg 375 ms)
9516 ms: nlohmann::detail::json_sax_dom_callback_parser<nlohmann::basic_json<... (69 times, avg 137 ms)
9112 ms: std::__detail::_Compiler<std::regex_traits<char>>::_M_atom (28 times, avg 325 ms)
8683 ms: std::basic_regex<char>::basic_regex (18 times, avg 482 ms)
8241 ms: std::operator<=> (438 times, avg 18 ms)
7561 ms: std::vector<boost::io::detail::format_item<char, std::char_traits<ch... (246 times, avg 30 ms)

7475 ms: std::vector<boost::io::detail::format_item<char, std::char_traits<ch... (246 times, avg 30 ms)
7309 ms: std::reverse_iterator<std::_Bit_iterator> (268 times, avg 27 ms)
7131 ms: boost::stacktrace::basic_stacktrace<>::basic_stacktrace (246 times, avg 28 ms)
6868 ms: boost::stacktrace::basic_stacktrace<>::init (246 times, avg 27 ms)
6518 ms: std::reverse_iterator<std::_Bit_const_iterator> (268 times, avg 24 ms)
5716 ms: std::__detail::_Synth3way::operator()<std::variant<nix::OutputsSpec:... (182 times, avg 31 ms)
5303 ms: nix::make_ref<nix::SingleDerivedPath, nix::DerivedPathOpaque> (178 times, avg 29 ms)
5244 ms: std::__uninitialized_move_a<boost::io::detail::format_item<char, std... (246 times, avg 21 ms)
4857 ms: std::make_shared<nix::SingleDerivedPath, nix::DerivedPathOpaque> (178 times, avg 27 ms)
4813 ms: std::__detail::_Synth3way::operator()<std::variant<nix::TextIngestio... (158 times, avg 30 ms)
4648 ms: nlohmann::detail::json_sax_dom_callback_parser<nlohmann::basic_json<... (69 times, avg 67 ms)
4597 ms: std::basic_regex<char>::basic_regex<std::char_traits<char>, std::all... (10 times, avg 459 ms)

**** Template sets that took longest to instantiate:
55715 ms: std::__do_visit<$> (3603 times, avg 15 ms)
47739 ms: std::__detail::__variant::__gen_vtable_impl<$>::__visit_invoke (11132 times, avg 4 ms)
43338 ms: nlohmann::basic_json<$>::parse<$> (85 times, avg 509 ms)
43097 ms: std::__detail::__variant::__raw_idx_visit<$> (2435 times, avg 17 ms)
32390 ms: nlohmann::detail::parser<$>::parse (83 times, avg 390 ms)
30986 ms: nix::HintFmt::HintFmt<$> (1261 times, avg 24 ms)
30255 ms: std::__and_<$> (25661 times, avg 1 ms)
29762 ms: std::unique_ptr<$> (2116 times, avg 14 ms)
28609 ms: std::__tuple_compare<$>::__eq (2978 times, avg 9 ms)
27560 ms: nlohmann::detail::parser<$>::sax_parse_internal<$> (167 times, avg 165 ms)
27239 ms: std::variant<$> (1959 times, avg 13 ms)
26837 ms: std::__invoke_result<$> (10782 times, avg 2 ms)
25972 ms: std::tuple<$> (5714 times, avg 4 ms)
24247 ms: std::__uniq_ptr_data<$> (2116 times, avg 11 ms)
24061 ms: std::__result_of_impl<$> (9029 times, avg 2 ms)
23949 ms: std::__uniq_ptr_impl<$> (2116 times, avg 11 ms)
21185 ms: std::optional<$> (2502 times, avg 8 ms)
21044 ms: std::pair<$> (4989 times, avg 4 ms)
20852 ms: std::__or_<$> (24005 times, avg 0 ms)
20203 ms: boost::basic_format<$>::basic_format (492 times, avg 41 ms)
20184 ms: std::tie<$> (2895 times, avg 6 ms)
19938 ms: nlohmann::basic_json<$>::create<$> (668 times, avg 29 ms)
19798 ms: std::allocator_traits<$>::construct<$> (5720 times, avg 3 ms)
19182 ms: std::__detail::__variant::_Variant_base<$> (1959 times, avg 9 ms)

```
 19151 ms: std::_Rb_tree<$>::_M_erase (2320 times, avg 8 ms)
 19094 ms: std::_Rb_tree<$>::~_Rb_tree (2022 times, avg 9 ms)
 18735 ms: nlohmann::basic_json<$>::basic_json (243 times, avg 77 ms)
 18546 ms: std::__detail::_Synth3way::_S_noexcept<$> (2542 times, avg 7 ms)
 17174 ms: nlohmann::basic_json<$>::json_value::json_value (368 times, avg 46 ms)
 17111 ms: nlohmann::detail::conjunction<$> (907 times, avg 18 ms)

**** Functions that took longest to compile:
  2091 ms: _GLOBAL__sub_I_primops.cc (../src/libexpr/primops.cc)
  1799 ms: nix::fetchers::GitInputScheme::fetch(nix::ref<nix::Store>, nix::fetc... (../src/libfetchers/git.cc)
  1388 ms: nix::Settings::Settings() (../src/libstore/globals.cc)
  1244 ms: main_nix_build(int, char**) (../src/nix-build/nix-build.cc)
  1021 ms: nix::LocalDerivationGoal::startBuilder() (../src/libstore/build/local-derivation-goal.cc)
   918 ms: nix::LocalStore::LocalStore(std::map<std::__cxx11::basic_string<char... (../src/libstore/local-store.cc)
   835 ms: opQuery(Globals&, std::__cxx11::list<std::__cxx11::basic_string<char... (../src/nix-env/nix-env.cc)
   733 ms: nix::daemon::performOp(nix::daemon::TunnelLogger*, nix::ref<nix::Sto... (../src/libstore/daemon.cc)
   589 ms: _GLOBAL__sub_I_tests.cc (../tests/unit/libutil/tests.cc)
   578 ms: main_build_remote(int, char**) (../src/build-remote/build-remote.cc)
   522 ms: nix::fetchers::MercurialInputScheme::fetch(nix::ref<nix::Store>, nix... (../src/libfetchers/mercurial.cc)
   521 ms: nix::LocalDerivationGoal::registerOutputs[abi:cxx11]() (../src/libstore/build/local-derivation-goal.cc)
   461 ms: nix::getNameFromURL_getNameFromURL_Test::TestBody() (../tests/unit/libutil/url-name.cc)
   440 ms: nix::Installable::build2(nix::ref<nix::Store>, nix::ref<nix::Store>,... (../src/libcmd/installables.cc)
   392 ms: nix::prim_fetchClosure(nix::EvalState&, nix::PosIdx, nix::Value**, n... (../src/libexpr/primops/fetchClosure.cc)
   390 ms: nix::NixArgs::NixArgs() (../src/nix/main.cc)
   388 ms: update(std::set<std::__cxx11::basic_string<char, std::char_traits<ch... (../src/nix-channel/nix-channel.cc)
   340 ms: _GLOBAL__sub_I_primops.cc (../tests/unit/libexpr/primops.cc)
   332 ms: nix::flake::lockFlake(nix::EvalState&, nix::FlakeRef const&, nix::fl... (../src/libexpr/flake/flake.cc)
   305 ms: _GLOBAL__sub_I_lockfile.cc (../src/libexpr/flake/lockfile.cc)
   300 ms: nix_store::opQuery(std::__cxx11::list<std::__cxx11::basic_string<cha... (../src/nix-store/nix-store.cc)
   296 ms: nix::parseFlakeRefWithFragment(std::__cxx11::basic_string<char, std:... (../src/libexpr/flake/flakeref.cc)
   289 ms: _GLOBAL__sub_I_error_traces.cc (../tests/unit/libexpr/error_traces.cc)
```

278 ms: nix::ErrorTraceTest_genericClosure_Test::TestBody() (../tests/unit/libexpr/error_traces.cc)
274 ms: CmdDevelop::run(nix::ref<nix::Store>, nix::ref<nix::Installable>) (../src/nix/develop.cc)
269 ms: nix::flake::lockFlake(nix::EvalState&, nix::FlakeRef const&, nix::fl... (../src/libexpr/flake/flake.cc)
257 ms: nix::NixRepl::processLine(std::__cxx11::basic_string<char, std::char... (../src/libcmd/repl.cc)
251 ms: nix::derivationStrictInternal(nix::EvalState&, std::__cxx11::basic_s... (../src/libexpr/primops.cc)
249 ms: toml::result<toml::basic_value<toml::discard_comments, std::unordere... (../src/libexpr/primops/fromTOML.cc)
238 ms: nix::LocalDerivationGoal::runChild() (../src/libstore/build/local-derivation-goal.cc)

**** Function sets that took longest to compile / optimize:
10243 ms: std::vector<$>::_M_fill_insert(__gnu_cxx::__normal_iterator<$>, unsi... (190 times, avg 53 ms)
9752 ms: bool boost::io::detail::parse_printf_directive<$>(__gnu_cxx::__norma... (190 times, avg 51 ms)
8377 ms: void boost::io::detail::put<$>(boost::io::detail::put_holder<$> cons... (191 times, avg 43 ms)
5863 ms: boost::basic_format<$>::parse(std::__cxx11::basic_string<$> const&) (190 times, avg 30 ms)
5660 ms: std::vector<$>::_M_fill_insert(std::_Bit_iterator, unsigned long, bo... (190 times, avg 29 ms)
4264 ms: non-virtual thunk to boost::wrapexcept<$>::~wrapexcept() (549 times, avg 7 ms)
4023 ms: std::_Rb_tree<$>::_M_erase(std::_Rb_tree_node<$>*) (1238 times, avg 3 ms)
3715 ms: boost::stacktrace::detail::to_string_impl_base<boost::stacktrace::de... (166 times, avg 22 ms)
3705 ms: std::vector<$>::_M_fill_assign(unsigned long, boost::io::detail::for... (190 times, avg 19 ms)
3326 ms: boost::basic_format<$>::str[abi:cxx11]() const (144 times, avg 23 ms)
3070 ms: void boost::io::detail::mk_str<$>(std::__cxx11::basic_string<$>&, ch... (191 times, avg 16 ms)
2839 ms: boost::basic_format<$>::make_or_reuse_data(unsigned long) (190 times, avg 14 ms)
2321 ms: std::__cxx11::basic_string<$>::_M_replace(unsigned long, unsigned lo... (239 times, avg 9 ms)
2213 ms: std::_Rb_tree<$>::_M_get_insert_hint_unique_pos(std::_Rb_tree_const_... (203 times, avg 10 ms)
2200 ms: boost::wrapexcept<$>::~wrapexcept() (549 times, avg 4 ms)
2093 ms: std::vector<$>::~vector() (574 times, avg 3 ms)
1894 ms: bool std::__detail::_Compiler<$>::_M_expression_term<$>(std::__detai... (112 times, avg 16 ms)
1871 ms: int boost::io::detail::upper_bound_from_fstring<$>(std::__cxx11::bas... (190 times, avg 9 ms)
1867 ms: boost::wrapexcept<$>::clone() const (549 times, avg 3 ms)
1824 ms: std::_Rb_tree_iterator<$> std::_Rb_tree<$>::_M_emplace_hint_unique<$... (244 times, avg 7 ms)
1821 ms: toml::result<$> toml::detail::sequence<$>::invoke<$>(toml::detail::l... (93 times, avg 19 ms)
1814 ms: nlohmann::json_abi_v3_11_2::detail::serializer<$>::dump(nlohmann::js... (39 times, avg 46 ms)
1799 ms: nix::fetchers::GitInputScheme::fetch(nix::ref<$>, nix::fetchers::Inp... (1 times, avg 1799 ms)
1771 ms: boost::io::detail::format_item<char, std::char_traits<char>, std::al... (190 times, avg 9 ms)
1762 ms: std::__detail::_BracketMatcher<$>::_BracketMatcher(std::__detail::_B... (112 times, avg 15 ms)
1760 ms: std::_Function_handler<$>::_M_manager(std::_Any_data&, std::_Any_dat... (981 times, avg 1 ms)
1733 ms: std::__detail::_Compiler<$>::_M_quantifier() (28 times, avg 61 ms)
1694 ms: std::__cxx11::basic_string<$>::_M_mutate(unsigned long, unsigned lon... (251 times, avg 6 ms)

```
  1650 ms: std::vector<$>::vector(std::vector<$> const&) (210 times, avg 7 ms)
  1650 ms: boost::io::basic_altstringbuf<$>::overflow(int) (190 times, avg 8 ms)

**** Expensive headers:
178153 ms: ../src/libcmd/installable-value.hh (included 52 times, avg 3426 ms), included via:
  40x: command.hh
  5x: command-installable-value.hh
  3x: installable-flake.hh
  2x: <direct include>
  2x: installable-attr-path.hh

176217 ms: ../src/libutil/error.hh (included 246 times, avg 716 ms), included via:
  36x: command.hh installable-value.hh installables.hh derived-path.hh config.hh experimental-features.hh
  12x: globals.hh config.hh experimental-features.hh
  11x: file-system.hh file-descriptor.hh
  6x: serialise.hh strings.hh
  6x: <direct include>
  6x: archive.hh serialise.hh strings.hh
  ...

173243 ms: ../src/libstore/store-api.hh (included 152 times, avg 1139 ms), included via:
  55x: <direct include>
  39x: command.hh installable-value.hh installables.hh
  7x: libexpr.hh
  4x: local-store.hh
  4x: command-installable-value.hh installable-value.hh installables.hh
  3x: binary-cache-store.hh
  ...

170482 ms: ../src/libutil/serialise.hh (included 201 times, avg 848 ms), included via:
  37x: command.hh installable-value.hh installables.hh built-path.hh realisation.hh hash.hh
  14x: store-api.hh nar-info.hh hash.hh
  11x: <direct include>
  7x: primops.hh eval.hh attr-set.hh nixexpr.hh value.hh source-path.hh archive.hh
  7x: libexpr.hh value.hh source-path.hh archive.hh
  6x: fetchers.hh hash.hh
  ...

169397 ms: ../src/libcmd/installables.hh (included 53 times, avg 3196 ms), included via:
```

  40x: command.hh installable-value.hh
  5x: command-installable-value.hh installable-value.hh
  3x: installable-flake.hh installable-value.hh
  2x: <direct include>
  1x: installable-derived-path.hh
  1x: installable-value.hh
  ...

159740 ms: ../src/libutil/strings.hh (included 221 times, avg 722 ms), included via:
  37x: command.hh installable-value.hh installables.hh built-path.hh realisation.hh hash.hh serialise.hh
  19x: <direct include>
  14x: store-api.hh nar-info.hh hash.hh serialise.hh
  11x: serialise.hh
  7x: primops.hh eval.hh attr-set.hh nixexpr.hh value.hh source-path.hh archive.hh serialise.hh
  7x: libexpr.hh value.hh source-path.hh archive.hh serialise.hh
  ...

156796 ms: ../src/libcmd/command.hh (included 51 times, avg 3074 ms), included via:
  42x: <direct include>
  7x: command-installable-value.hh
  2x: installable-attr-path.hh

150392 ms: ../src/libutil/types.hh (included 251 times, avg 599 ms), included via:
  36x: command.hh installable-value.hh installables.hh path.hh
  11x: file-system.hh
  10x: globals.hh
  6x: fetchers.hh
  6x: serialise.hh strings.hh error.hh
  5x: archive.hh
  ...

133101 ms: /nix/store/644b90j1vms44nr18yw3520pzkrg4dd1-boost-1.81.0-dev/include/boost/lexical_cast.hpp (included 226 times, avg 588 ms), included via
:
  37x: command.hh installable-value.hh installables.hh built-path.hh realisation.hh hash.hh serialise.hh strings.hh
  19x: file-system.hh
  11x: store-api.hh nar-info.hh hash.hh serialise.hh strings.hh
  7x: primops.hh eval.hh attr-set.hh nixexpr.hh value.hh source-path.hh archive.hh serialise.hh strings.hh

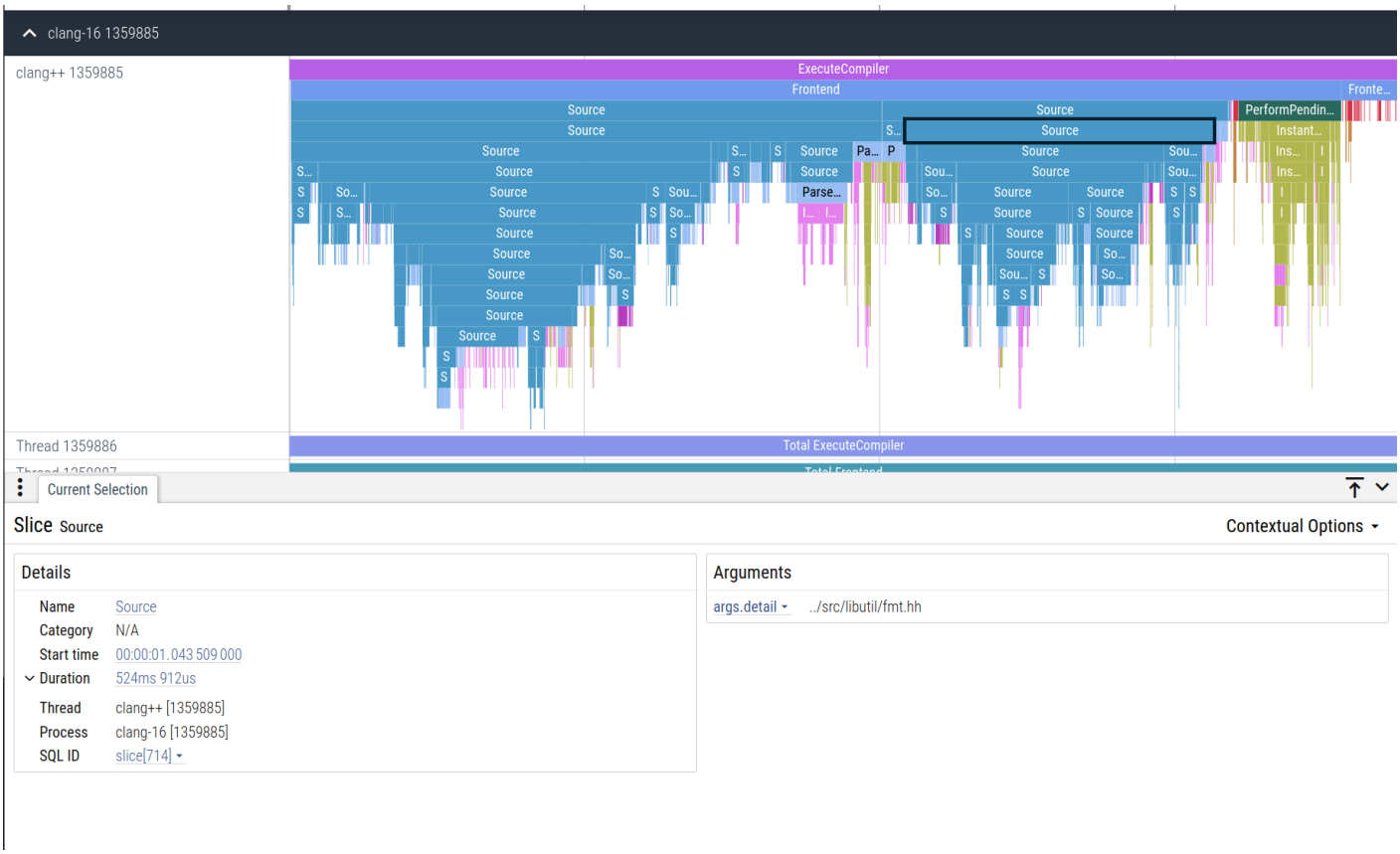
```
  7x: libexpr.hh value.hh source-path.hh archive.hh serialise.hh strings.hh
  6x: eval.hh attr-set.hh nixexpr.hh value.hh source-path.hh archive.hh serialise.hh strings.hh
  ...

132887 ms: /nix/store/h2abv2l8irqj942i5rq9wbrj42kbsh5y-gcc-12.3.0/include/c++/12.3.0/memory (included
262 times, avg 507 ms), included via:
  36x: command.hh installable-value.hh installables.hh path.hh types.hh ref.hh
  16x: gtest.h
  11x: file-system.hh types.hh ref.hh
  10x: globals.hh types.hh ref.hh
  10x: json.hpp
  6x: serialise.hh
  ...

  done in 0.6s.
```

Manually looking at traces

Note that the summary in the report can miss details like *why* one particular header is bad; to find that out, use a trace viewer to inspect the JSON trace file; we suggest `rg -t json -uu error\.hh build/ | less` to find some `.cc` trace that the bad header (in this example, `error.hh`) appears in.

You can look at individual file traces by opening some file like `build/src/libcmd/liblixcmd.so.p/command.cc.json` in https://ui.perfetto.dev or another Chrome-trace-json compatible trace viewer like Speedscope.

This will produce a flamegraph of the trace (screenshot shows Perfetto):



The most general spans of compile time are at the top, and the constituent spans are shown as you go down.

Successful build time reduction CLs

- cl/588: 15%
- cl/1351: 10%

See the build-time-optimisation Gerrit topic for more related things.

Backport guide

Don't forget, using Gerrit is a bit different than other systems.

single commits

try `git cherry-pick -x` first. if this works, excellent. if not, apply the usual cherry-picking procedures:

- track down apply failures to intermediate changes. maybe cherry-pick those first if they're not too awful, but experience shows that they usually are too awful
- anything that touches the store or contains the word `Accessor` in the vicinity probably needs to be picked about and rewritten from scratch
- *always* test ofc, even if something applies cleanly it will sometimes just fail to build (or worse, run)
- nix prs tend to have broken inner commits. it is often necessary to pick parts from later commits in a pr to fix ci, in that case note this down as `fixes taken from <commits...>`
- sometimes a commit may be so broken that it can't reasonably be fixed except by squasing it with some other commit, in that case just squash them and not it down somehow (either with multiple `cherry picked from` commit hashes or multiple commit message+cherry-pick-hash blocks, depending on whether the fix messages were any useful)

full prs

single-commit prs were mostly picked using `cherry-pick -x -m1` to keep the association with the upstream pr number for clarity. this implicitly squashes the pr into a single commit so it's only useful for single-commit prs. (some prs that have broken intermediate commits also benefit from this, but see above for that)

when pushing these to gerrit please set a topic like `backport-<pr-number>` using push options (`-o topic=backport-<pr-number>` in `git push`) to delineate one picked pr from a pr that depends on it

Misc tips

buildbot user style to make the pulsing pills bearable

```
@keyframes pulse_animation {
  0% { transform:scale(.9) }
  50% { transform:scale(1) }
  to { transform:scale(.9) }
}

.pulse {
    animation-duration: 10s !important;
}
```

FIXME: someone should PR this, now that we have the ability to patch buildbot

run all lix vm tests locally

```
tests=$(
    nix eval --json --impure \
        --apply '
            let f = n: t:
                if __isAttrs t
                then (if t.type or "" == "derivation"
                    then (if t.system == __currentSystem
                            then [ n ]
                            else [])
                    else __concatMap (m: f "${n}.${m}" t.${m}) (__attrNames t))
                else [];
            in f ".#hydraJobs.tests"
        ' \
        .#hydraJobs.tests \
        | jq -r '.[]'
)
```

```
nix build --no-link -L ${tests[@]}
```

check out current patchset of a cl by git alias

put this in a gitconfig that can configure aliases:

```
[alias]
	cocl = !\
	 ps=$(\
	   ssh $(git config remote.origin.gerriturl) \
	     gerrit query --format=json --current-patch-set $1 \
	       | jq -sr .[0].currentPatchSet.ref \
	 ) && git fetch origin $ps && git checkout FETCH_HEAD && true
```

then run as `git cocl <cl-number>`. needs a `git config remote.origin.gerriturl gerrit.lix.systems`, or some other url that ssh can connect to. (could've extracted it from the remote url but we didn't want to do that much shell)

git stuff

git-revise

git-revise is a cool tool for splitting and shuffling commits in-memory without breaking your working tree. it's great.

It also has some broken stuff with respect to gerrit commit-msg hooks. However, this can be fixed (this is opt-in because some commit-msg hooks make unsound assumptions but the gerrit one should be fine):

```
# allow gerrit git hooks to run on git-revise
[revise "run-hooks"]
	commit-msg = true
```

Making `git clean` clean the stuff that isn't removed by `make clean`

If you don't want to use `git clean -x` to remove all git-ignored stuff, but want to remove things that are generated in Lix's build process but aren't removed by `make clean`, apply this patch: no-ignore-not-cleaned.patch

Building Locally

See hacking.md in the Lix repo for the main documentation. Extra tips can go here.

RISC-V support

Goal: install lix on a riscv64-linux system

The target is a DevTerm R-01, so it's an AllWinner D1 RISC-V processor @ 1GHz, with 1GB of memory and 32GB of microSD.

We can't run the Lix installer without building it, because there's no canned build for it. So let's try building it natively:

```
$ rustup
$ git clone https://git.lix.systems/lix-project/lix-installer
$ cd lix-installer
$ RUSTFLAGS="--cfg tokio_unstable" cargo install --path .
```

This doesn't work because there's some conditional complication^Wcompilation that doesn't cover riscv64. So we need to open `self_test.rs` and add an entry:

```
    #[cfg(all(target_os = "linux", target_arch = "riscv64"))]
    const SYSTEM: &str = "riscv64-linux";
```

At this point, it will, in principle, build. In practice, however, 1GB is just not enough RAM. If you add some swap it'll make it to the last step, but then it wants 1.5GB+ for that. I wouldn't try it on a system with less than 2GB, and ideally more.

Ok, native build is a bust unless I want to let it thrash all night. So let's cross-compile it on `ancilla`, which is, conveniently, already running nixos.

The nix-installer flake doesn't come with riscv64 cross support, and rather than try to figure it out I just winged it with nix-shell. I am skipping over a lot of false starts and blind alleys here as I ran into things like dependency crates needing a cross-compiling gcc, or rust not having a stdlib on riscv64-musl.

```
$ git clone https://git.lix.systems/lix-project/lix-installer
$ cd lix-installer
$ $EDITOR shell.nix
with import <nixpkgs> {
  crossSystem.config = "riscv64-unknown-linux-gnu";
};
mkShell {
  nativeBuildInputs = with import <unstable> {}; [ cargo rustup ];
```

```
}

$ nix-shell
[long wait for gcc to compile]

$ export RUSTUP_HOME=$PWD/.rustup-home
$ export CARGO_HOME=$PWD/.cargo-home
$ rustup default stable
$ rustup target add riscv64gc-unknown-linux-gnu
$ edit src/self_test.rs
[apply that same patch to SYSTEM]
```

The build invocation is a bit more complicated here, because we need to tell it where to find the linker:

```
$ RUSTFLAGS="--cfg tokio_unstable" cargo build \
    --target riscv64gc-unknown-linux-gnu \
    --config target.riscv64gc-unknown-linux-gnu.linker='"riscv64-unknown-linux-gnu-gcc"'
[another long wait]

$ file target/riscv64gc-unknown-linux-gnu/debug/lix-installer
target/riscv64gc-unknown-linux-gnu/debug/lix-installer:
  ELF 64-bit LSB pie executable, UCB RISC-V, RVC, double-float ABI,
  version 1 (SYSV), dynamically linked,
  interpreter /nix/store/g4xam7gr35sziib1zc033xvn1vy9gg8m-glibc-riscv64-unknown-linux-gnu-2.38-44/lib/ld-
linux-riscv64-lp64d.so.1,
  for GNU/Linux 4.15.0, with debug_info, not stripped
```

Since we couldn't do a static musl build it needs the nix ld.so, but we can get around that!

```
$ scp target/riscv64gc-unknown-linux-gnu/debug/lix-installer root@riscv:.
$ ssh root@riscv
# ./lix-installer
-bash: ./lix-installer: no such file or directory

# ldd ./lix-installer
/nix/store/.../ld-linux-riscv64-lp64d.so.1 => /lib/ld-linux-riscv64-lp64d.so.1
[other output elided]
```

```
# /lib/ld-linux-riscv64-lp64d.so.1 ./lix-installer
The Determinate Nix installer (lix variant)
[...]
```

Sadly we can't actually use it to install, because `nix_package_url` needs a default value, and on RISC-V, it doesn't have one! It's `self_test.rs` all over again except it doesn't manifest until runtime.

So, off to `src/settings.rs` we go. It doesn't need to be a valid URL, just something URL-shaped.

```
/// Default [`nix_package_url`](CommonSettings::nix_package_url) for unknown platforms
pub const NIX_UNKNOWN_PLATFORM_URL: &str =
    "https://releases.lix.systems/unknown-platform";
```

```
    #[cfg_attr(
        all(target_os = "linux", target_arch = "riscv64", feature = "cli"),
        clap(
            default_value = NIX_UNKNOWN_PLATFORM_URL,
        )
    )]
```

Rebuild, re-push, re-run:

```
# /lib/ld-linux-riscv64-lp64d.so.1 /opt/lix-installer install linux
Error:
   0: Planner error
   1: `nix-installer` does not support the `riscv64gc-unknown-linux-gnu` architecture right now
```

Ok, missed a few places in settings.rs, let's put a quick and dirty hack in there:

```
            #[cfg(target_os = "linux")]
            (_, OperatingSystem::Linux) => {
                url = NIX_UNKNOWN_PLATFORM_URL;
                nix_build_user_prefix = "nixbld";
                nix_build_user_id_base = 30000;
                nix_build_user_count = 32;
            },
```

```
            #[cfg(target_os = "linux")]
            (_, OperatingSystem::Linux) => {
                (InitSystem::Systemd, linux_detect_systemd_started().await)
            },
```

It also needs a tarball to install; jade_ kindly updated the flake for it to support riscv64, so we just check it out (or, well, check out review branch 1444) and then `nix build -L .#nix-riscv64-linux.binaryTarball` and away we go.

This, it turns out, also doesn't work, because the installer is hardcoded to expect the directory the tarball contains to start with `nix-*`. You can either unpack and repack the tarball to meet that requirement, or find all the places in lix-installer that assume that and edit them -- they're in `src/action/base/move_unpacked_nix.rs` and `src/action/base/setup_default_profile.rs`.

Finally, this particular kernel lacks `seccomp` support -- in order to get it working, I had to edit the lix (not lix-installer) `package.nix` and add `(lib.mesonEnable "seccomp-sandboxing" false)` to the meson flags.

And with that done, it works!

```
root@devterm-R01:~# uname -a && nix --version
Linux devterm-R01 5.4.61 #12 PREEMPT Wed Mar 30 14:44:22 CST 2022 riscv64 riscv64 riscv64 GNU/Linux
nix (Lix, like Nix) 2.90.0pre20240613_dirty
```

Branches

The Lix repository contains multiple releases in parallel. The branches work as follows:

- `main`. This contains *major* tags (except for 2.90 because of an early branch-off. We might fix that manually?), and is for the *next* major version of the software. This is where new development typically happens.
- `release-*`. These contain tags for `*.0` and further minor releases on a major release. We generally try to not backport things, since we would much rather get another major release out. (subject to revision; we would really like to not have LTS releases, but distro may make us do it?). These branches are *development branches* for a given release after it is released.
- (suggestion?) `stable-*` - Branch which is always pointed at the latest tag in that given major version.

Version types

- Full release, e.g. 2.90.0. This is a snapshot of HEAD that we believe is stable for release and that we have fully performed out-of-tree validation on.
- Beta release, e.g. 2.90.0-beta.1. This is something that we would consider running in more or less any environment, given that we all run HEAD ourselves. This is an arbitrarily selected snapshot of HEAD that we are deciding to produce installers for, and is not special.
- Release candidate, e.g. 2.90.0-rc1. This is something that we would just release but it needs a bit more out-of-tree validation.

Git tags

Git tags are created with the format `2.90.0`.

Docker tags

- `latest` - The latest minor version of the latest major version
- `2.90` and similar - The latest minor version of the `2.90` major release.
- `2.90.0` - Exactly `2.90.0`.