Google

An Upstream Friendly Source Control Model And Tooling

adelg@google.com
dylanbhatch@google.com

Overview

- Kernel development at Google & its issues
- Our solution & its generally applicable themes
- Current progress & Looking forward

Kernel development at Google

Google

Kernel development at Google

- Prodkernel is a fork of the Linux kernel that Google deploys on its production systems
- It adds about 9000 patches on top of upstream Linux
  - Internal APIs (e.g. SwitchTo for Google Fibers)
  - Hardware support
  - Performance optimizations
- Every ~2 years we rebase all these patches over a ~2 year codebase delta

Kernel development at Google - Why do we have Prodkernel?

- We had internal needs and timelines that necessitated having our own fork of Linux
- Net: A method to set quality of service for outgoing network traffic
- MM: Specific rules for OOM kills for jobs running in our data center
- Sched: An new API to enable cooperative scheduling in userspace
- Perf: Disabling sampling of the user stack in perf tool due to user privacy concerns

Problems with the current process

Google

Upstreaming features

Google-made features are developed and tested in Prodkernel, which is up to 2 years behind upstream. This produces two major hurdles:

- Hurdle 1: Rebasing the feature across a multi year delta to upstream
- Hurdle 2: Re-testing on the newly rebased upstream feature
    - While a feature might have been validated against Google production workloads on the old upstream base, how does one replicate that on the new upstream base without the rest of Prodkernel?

Bug discovery

- We operate complex workloads at a very large scale
- This sets us up to discover lots of system bottlenecks, bugs, and deadlocks
- The nature of our rebase means that there is a large delay in discovery, diagnosis, and sharing of the issue upstream
- Being N majors behind upstream means that the bug we find is also X years old and we are X years late to presenting the issue to upstream
- By staying on a certain major for an extended period, a bug could be fixed upstream, but we won't benefit until we rebase again (or manually find & backport)

Platforms support & Backporting

- As upstream adds support for new platforms, we need to backport the patches
- This issue is actually generalizable to all backports
- We need to backport over a large delta, and even then, it ends up not being tested against the same kernel version -- hence bugs we encounter might not even be applicable to upstream

The Prodkernel Rebase

- Rebasing is **extremely** costly
- Individual patches must have their conflicts resolved against the new upstream base
- Entire kernel must be re-qualified against Google's workloads with the new base
- Dependencies between patches are inconsistently documented, making parallelization of the rebase effort extremely difficult
- The delay associated with a rebase worsens the aforementioned issues with upstream participation, which in turn increases the cost of the next rebase.

The Prodkernel Rebase

- Each subsequent Prodkernel rebase increases in number of patches rebased and time to complete the rebase
- Rather than see when the trendline becomes unsustainable, we want to proactively reduce our technical debt
- Effectively, reducing our rebase technical debt would have us realize more time available to spend participating upstream

The Prodkernel Rebase (cont)

- Not only is the upward trend in patches needing to be rebased and time delta increasing itself
- It also detracts from engineers ability to do anything else (e.g. contribute upstream)
- Structurally: They work on a years old kernel, this presents the hurdle of a possibly large rebase on a new feature for upstream
- Practically: Time is taken from an engineer’s finite resources to participate in rebasing, instead of participating upstream

Generalizing the Issues

- By not being close to upstream we run into two main issues
- **We cannot passively benefit from near upstream development**
- **We cannot contribute to upstream since we are far behind, leading to more internal patches, which just magnifies this point and the above one**

Icebreaker - An upstream friendly kernel

Google

General themes

- **Stay close to upstream** -- release a kernel on every major upstream version, creating a forcing function for developers to stay on board
- **Encourage upstream contribution** – make it easier for Googlers to participate in upstream, lessening our out of vanilla tree patches
- **Test, test , test** – test the kernel in production at a limited capacity, allowing us to observe bugs earlier and either fix them or ingest a fix from upstream

Why is focusing on these themes important?

- It mainly boils down to upstream engagement
- If our engineers work on a fork that is far from upstream, what benefit does it bring the engineer or **the upstream community** to engage
- Engineer develops a feature for our years old kernel, at this point they have a choice
  - Rebase and upstream it
  - Keep it internal and rebase it later
- Upstreaming it both presents an immediate cost and an uncertain outcome
- The second option, while still having a cost, is delayed and certain

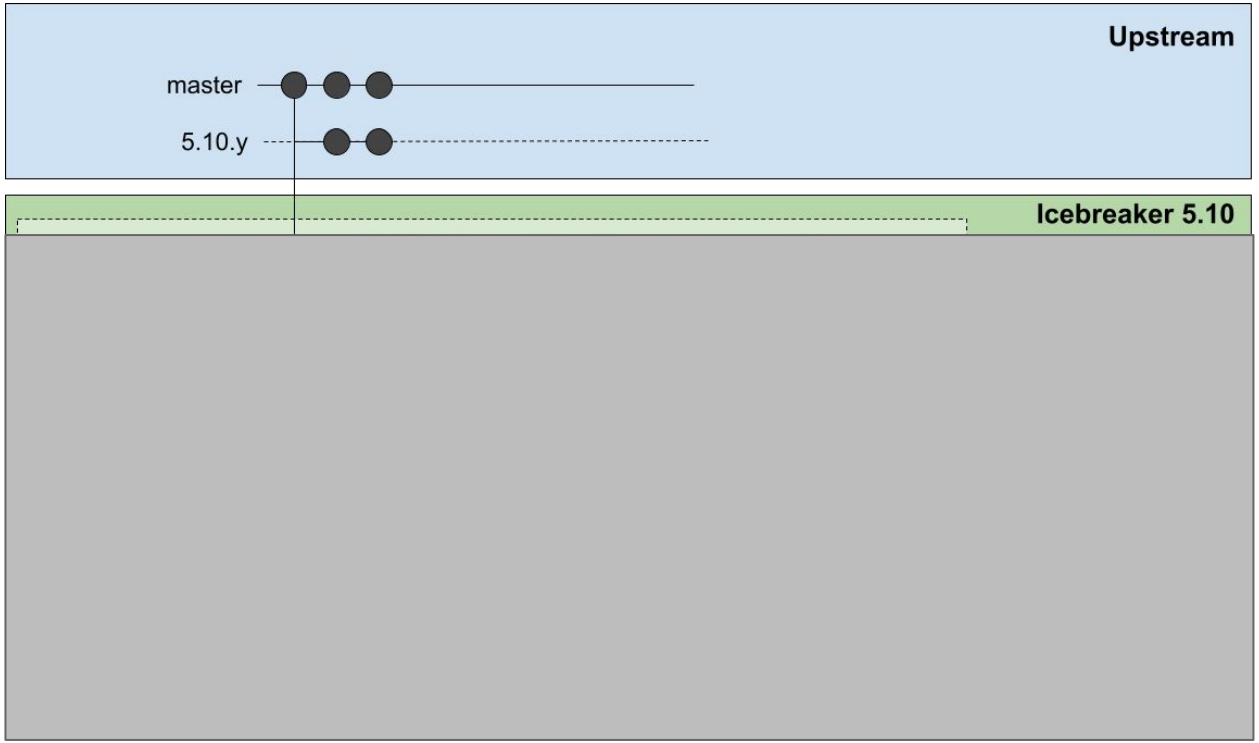
Why are these themes important? (cont)

- Qualification of upstream kernels
- We have lots of machines and a diverse variety of workloads
- Pragmatically, it is not efficient to try and fix every single kernel bug we come across, we need help from upstream
- Qualifying with upstream kernels helps this issue and also sets us up to be a better upstream citizen

So how can we take these themes and turn them into a reality?

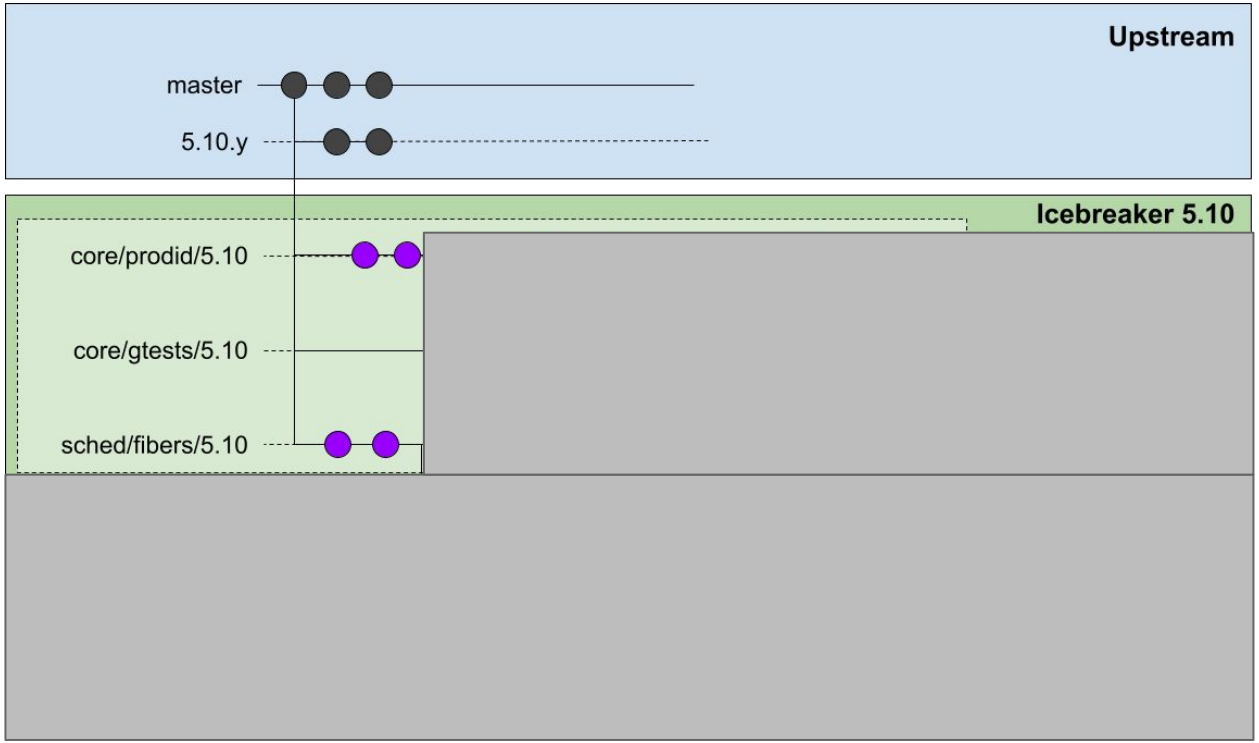
- There are two aspects to maintaining a kernel tree as it relates to source control
  - Development
  - Upgrading
- Icebreaker's SCM takes both of these workflows and sets them up for success with respect to interacting with upstream
- Focuses on making development for upstream easy and keeping code close to upstream easy

Icebreaker Structure



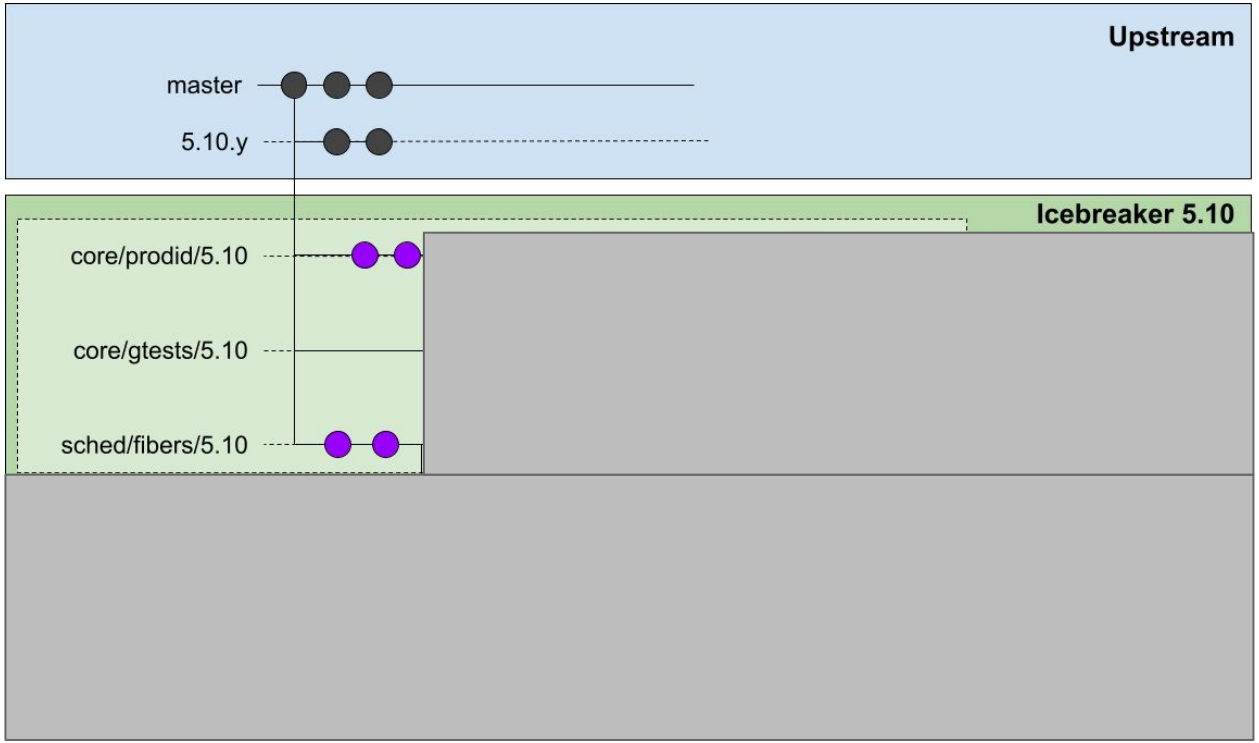
First, we fork off of an upstream Linux release.

Icebreaker Structure



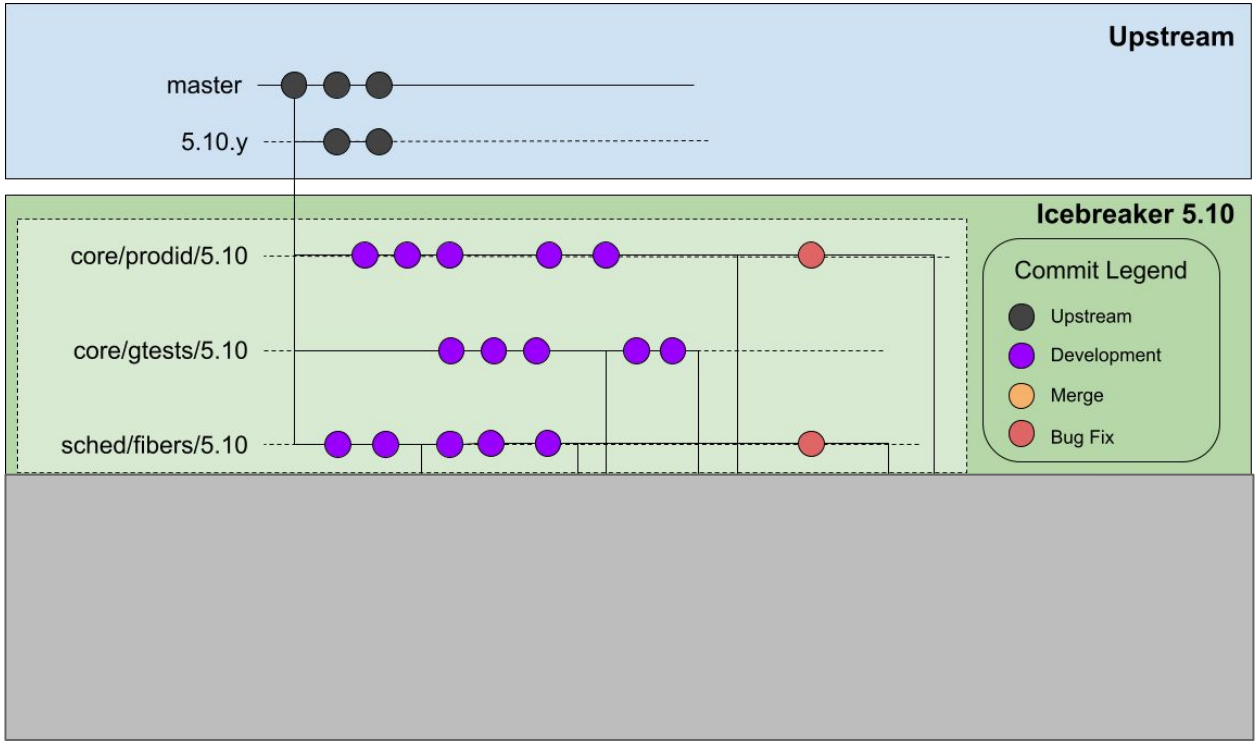
We then create *feature branches*. *Feature branches* are a set of patches on top of vanilla Linux source that augment the kernel.

Icebreaker Structure



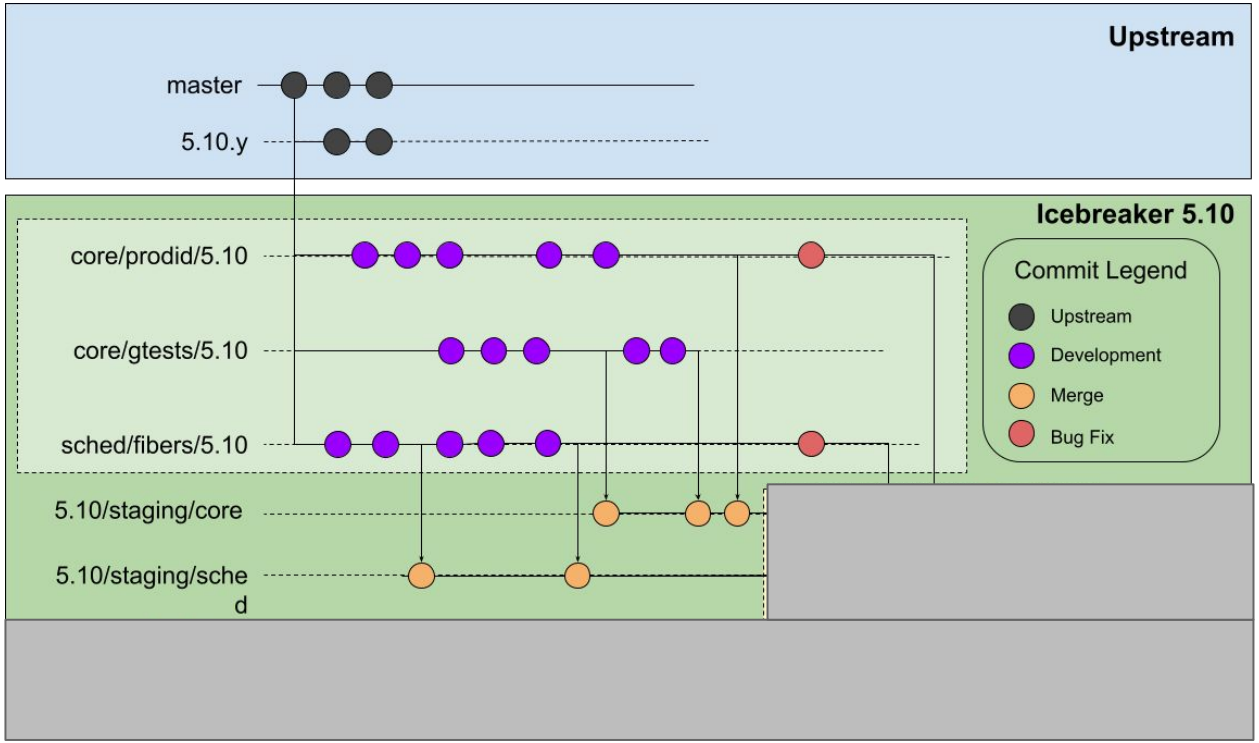
The nice thing about feature branches is that since they are just patches on-top of a vanilla kernel, meaning they can very easily be upstreamed.

Icebreaker Structure



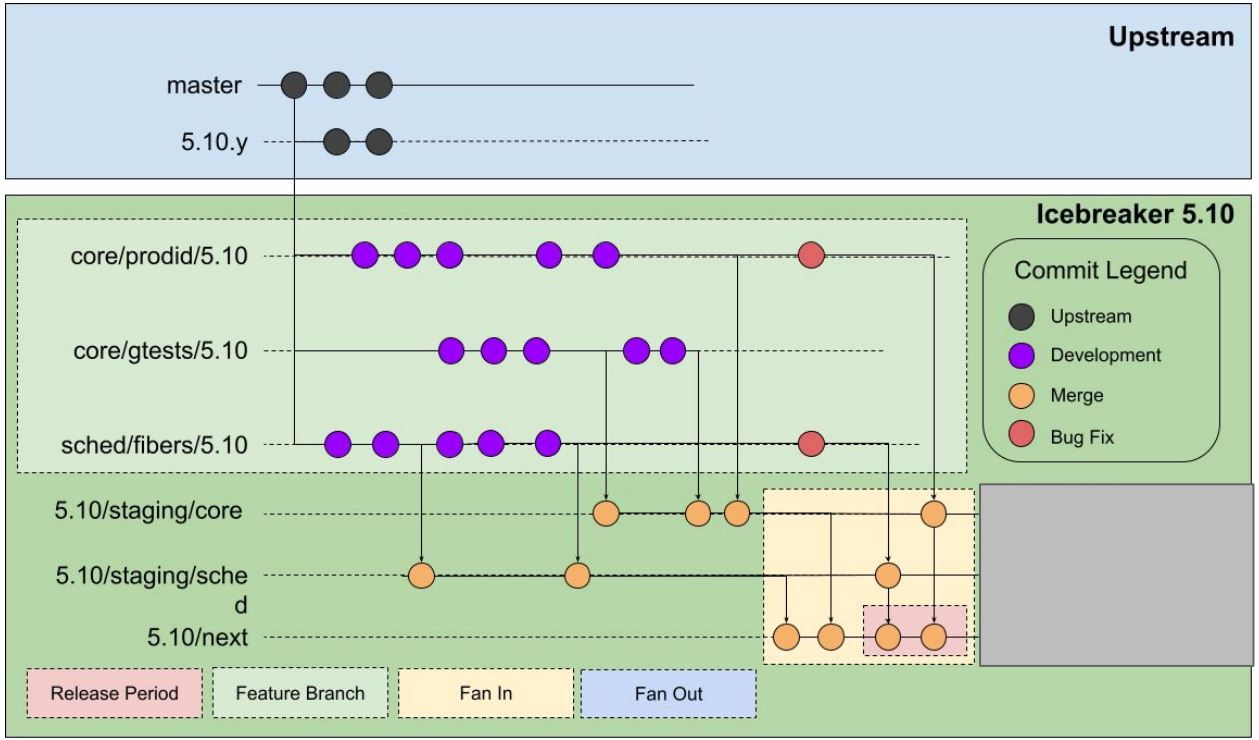
Development then happens on these feature branches.

Icebreaker Structure



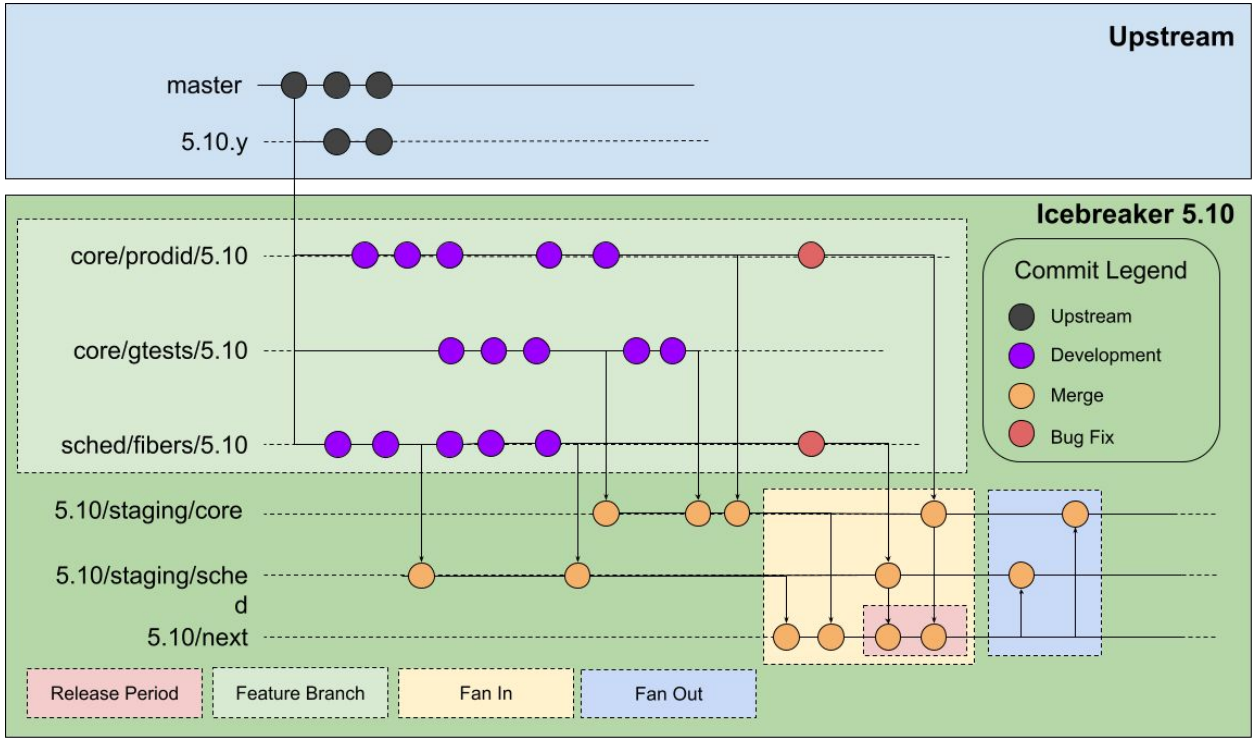
Feature branches, get merged into staging branches, based on their subsystem, for testing.

Icebreaker Structure



Staging branches, get merged into the next branch for release.

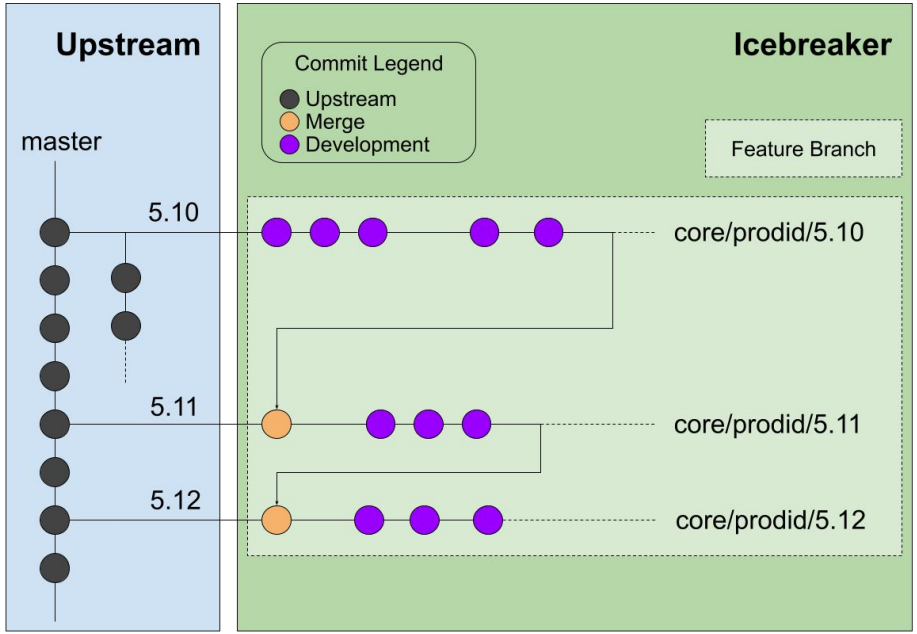
Icebreaker Structure



The next branch “fans-out” to staging branches post release.

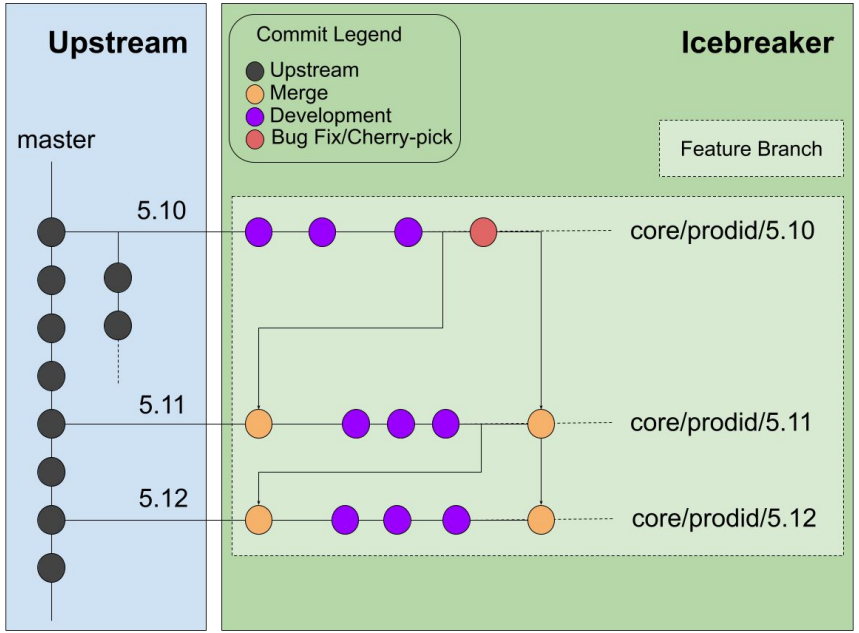
One Feature Branch, Many Icebreakers

- Instead of a rebase, features merge “onto” upstream major releases, but do not merge LTS releases.
- Conflict resolutions live in the merge commits, and development commits keep the same stable SHA1 after upgrading.
- Feature upgrades do not depend on other feature upgrades to take place.



One Feature Branch -- Bug fixes

- Bugs can be fixed at the oldest supported version of a feature, and carried forward to the new ones via merge.
- This way, exactly one buggy SHA1 can be fixed by exactly one fixup SHA1. A “`Fixes: <SHA1>`” commit footer is very meaningful in this context.
- No need to include extra metadata to track patches, even when supporting multiple versions.

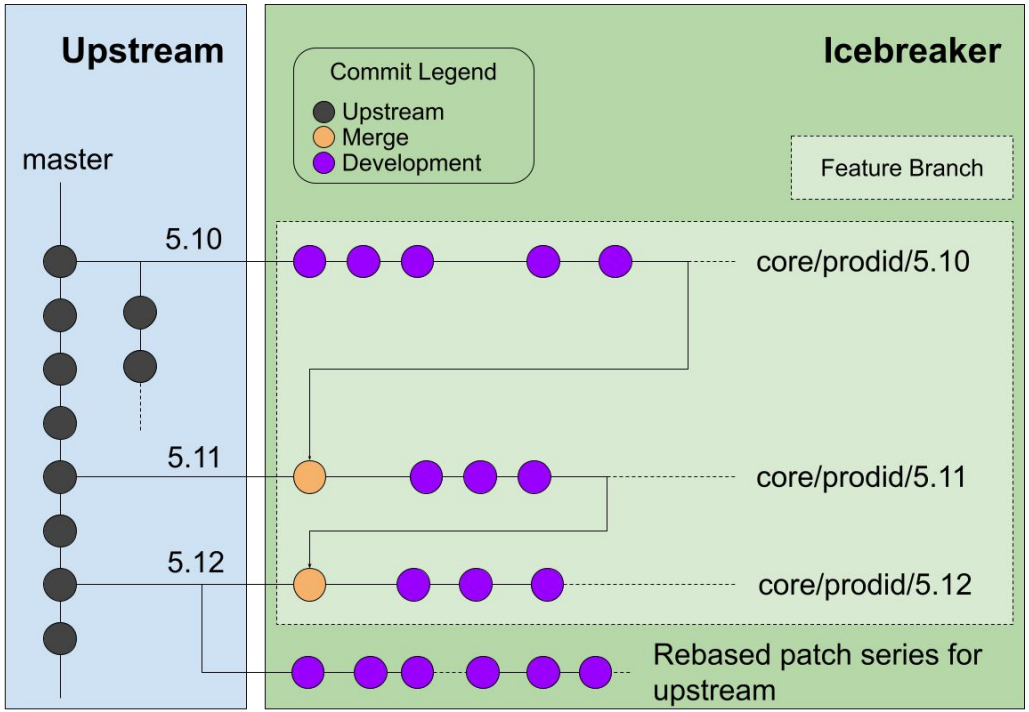


Back to our themes

- **Stay close to upstream** -- feature branches can **easily** be upgraded every kernel release
- **Encourage upstream contribution** – feature branches are basically patchsets
- **Test, test , test** – multiple levels of granularity make it easy to test at different steps of the proccess

Back to our themes

- **Encourage upstream contribution:**
- The separation of features and relative stability of their contents make it easier for developers to upstream
- With git-rerere enabled, setting up a clean patch series for upstream can be as simple as git-rebase



Icebreaker - Tooling

Google

Tooling

- In theory, the aforementioned ideas should work
- But there are many steps that need to be taken and it is likely the case the a developer does not want to be responsible for certain mechanical tasks
- For these process to work we need well oiled automation

Feature branch testing

- When a developer uploads a patch, we automatically run build tests across a variety of CONFIGs and ARCHs
- We validate the commit message and its metadata
- All of this ensures that if a developer ends up wanting to send a branch upstream, it is already in a good state

Kernel testing

- Staging branches have a select subset of all our tests run against them automatically as a smoke test
- This lets the subsystem maintainer have some semblance of confidence in the health of their subsystem
- Our release branches runs all our tests, and if we find failure, we can bisect back to the faulty subsystem
- This makes the job of the subsystem maintainer and release manager easy in that they just have to look at results instead of manually figuring out what to run

Kernel composition

- We have automated rules that merge feature branches into its staging branch depending on the maintainer preference
- We have a tool to generate and upload proposed “fan-in”s out staging branches
- This automation makes it so that the overhead after committing a patch to feature branch is minimal

Upgrading the kernel

- We built automation to:
  - Resolve dependencies among feature branches
  - Automatically attempt to upgrade them to the next version
- When combined with our testing automation, we basically get tests on the proposed upgraded feature branch for free
- When combined with our composition automation, we get combining the feature branches for free
- Big idea here is to build reusable automation and then chain it

Recap on tooling and themes

- **Stay close to upstream** -- Upgrade automation makes this easy
- **Encourage upstream contribution** – Testing at feature branch level with commit validation also makes this easy
- **Test, test, test** – Testing at the release branch levels provides another level of testing before prod

Current status & Looking forward

Google

How is Icebreaker performing?

- We are on Icebreaker 5.15 (at time of writing 5.16 just came out)
- Our time to get from 5.X to 5.X+1 is less than an upstream release cycle, implying this is sustainable and we actually have wiggle room
- Sustained practice and automation development has yielded quicker and quicker upgrades despite moving more and more code

Looking forward

- Fully automate everything that is still human based
  - We have some CLI tools, but we want to encapsulate this logic in a persistent job
- Move to testing feature branches atop release candidates, so we can participate in testing Linus’ release candidates
- Open up the scope of Icebreaker to allow for more upstream development

Takeaways

Google

Takeaways

- **Stay close to upstream** -- Being close to the tip of where everyone else is makes life easier and is a worthwhile goal despite effort to get there. Automating these process means you can eventually get close to set and forget.
- **Encourage upstream contribution** – Make it as easy as possible for developers to develop for upstream
- **Test, test test** – Automate testing at all levels. No one wants to manually kick off tests. Make it so that git push is the last thing you have to type.

Questions

Google