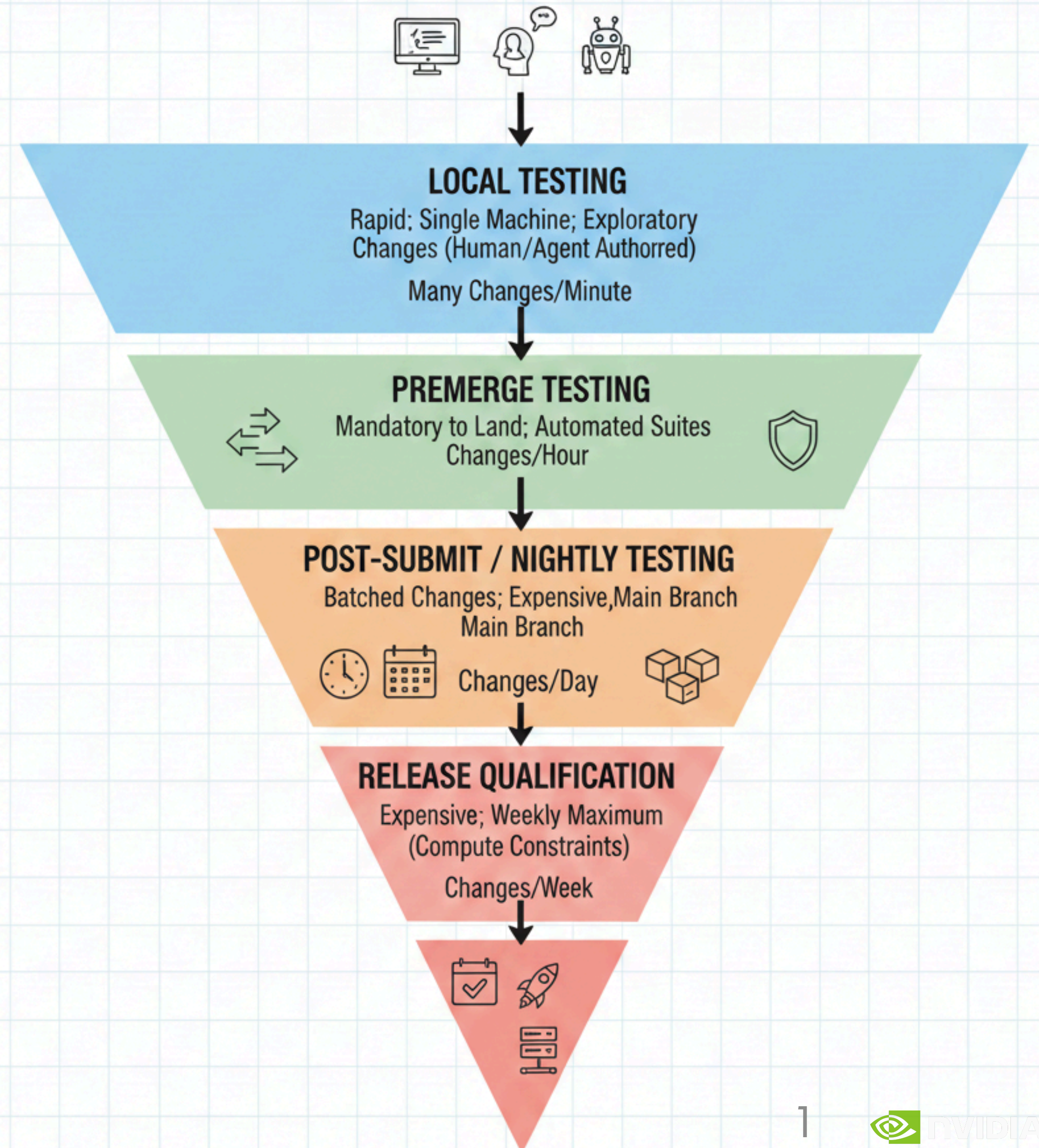
Testing Funnel: Validating LLVM at Scale

Author: Reid Kleckner / NVIDIA



Personal background

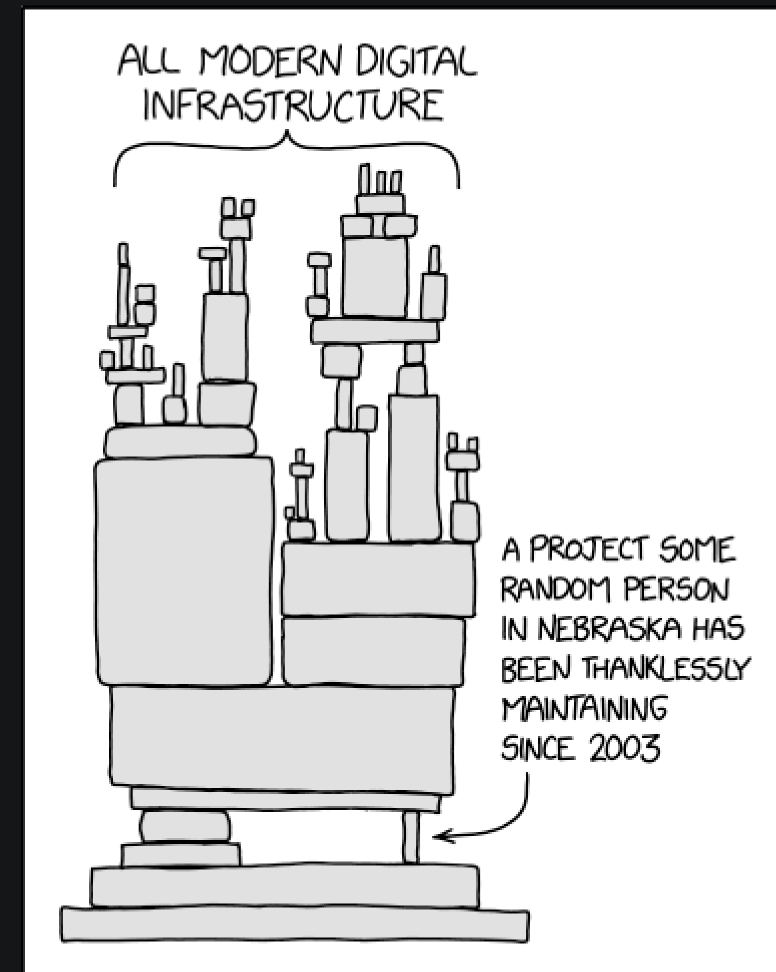
- Principle engineer @ NVIDIA on CUDA compiler team
  - Focused on upstreaming CUDA compiler changes to LLVM
- Involed in LLVM professionally since 2014
- Previous experience in release qualification for multiple downstream toolchains
- Years of conversations with folks in the community about release qualification challenges
- Want to share lessons to improve collaboration

Who is this talk for?

- Presumably everyone here today is an **LLVM stakeholder**
- Many of you presumably wear one or both of these hats:
  - **LLVM contributor/maintainer:** Directly impacted by LLVM's test infra
  - **Downstream user/consumer:** Impacted by changes in the project
- Either way, the quality of LLVM's upstream CI should matter a lot to you
- Testing LLVM creates unique challenges — different from testing other software
  - Why is testing llvm-project difficult?

Why is compiler CI uniquely challenging?

- LLVM is foundational, think xkcd
  - … sometimes the skinny bar, sometimes the wide bar
- Toolchain support comes with unique challenges that affect our community
  - Foundation movement is like an earthquake
- Disruptive changes result in issue reports and **rework**
  - **Rework:** Doing the work anew through revert/reland or fixing forward
- Rework can be stressful, leads to burnout, and threatens project sustainability



NVIDIA

Hyrum's Law: Challenges of many users

> *With a sufficient number of users of an API,*
> *it does not matter what you promise in the contract:*
> *all observable behaviors of your system*
> *will be depended on by somebody.*
> *-- Hyrum Wright*

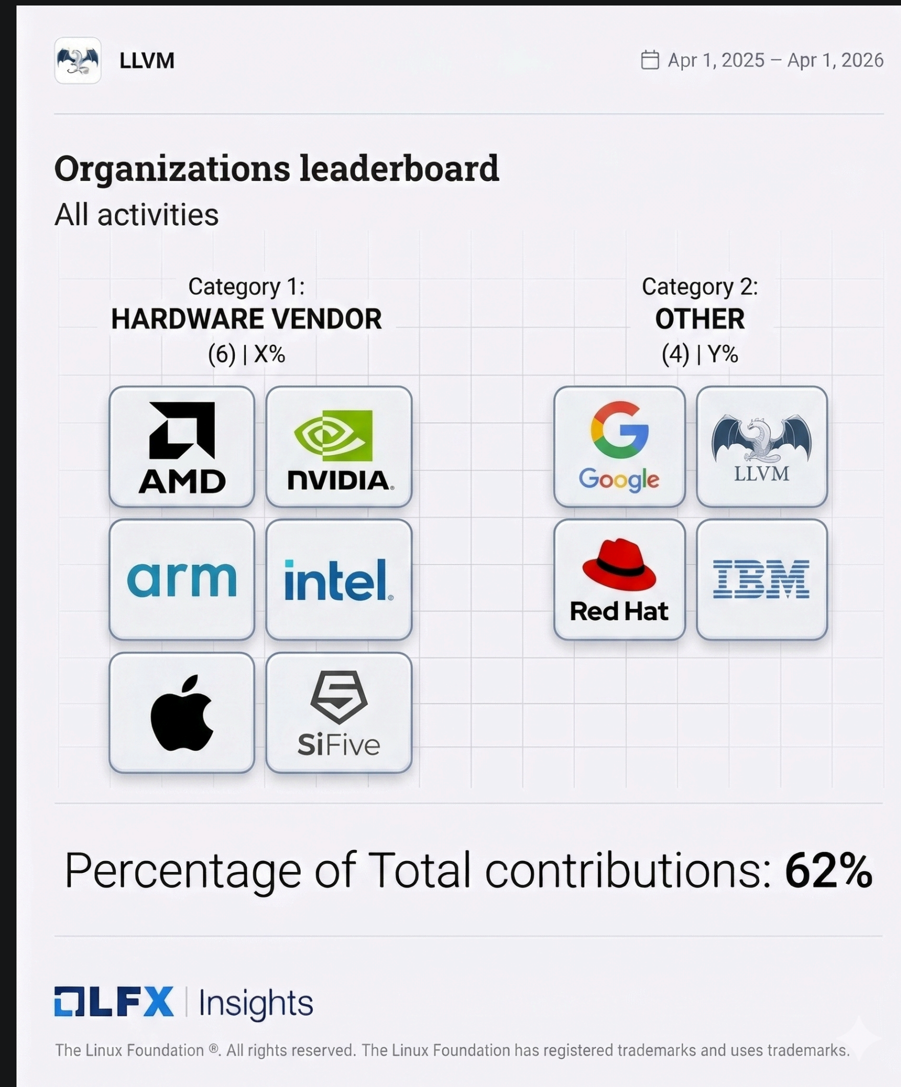
- Programming languages are a rich API surface
- Language semantics are the contract
  - Wiggle room in the contract is undefined behavior (or implementation defined)
- Regressions are a blame game: Is it a compiler bug or user error?

Contrast to conventional CI wisdom

- One team, one product, continuous delivery, shared definition of correctness, push-to-prod on green!
  - … this doesn't reflect us
- LLVM is vendored into many downstream toolchains with diverse requirements
- **Limited resources:** Upstream LLVM has limited compute and test engineering resources
- Downstream vendors can't hold upstream accountable to **hidden, ever-changing requirements**, no matter how business critical
  - Shifting left to upstream is impractical

Who contributes to LLVM?

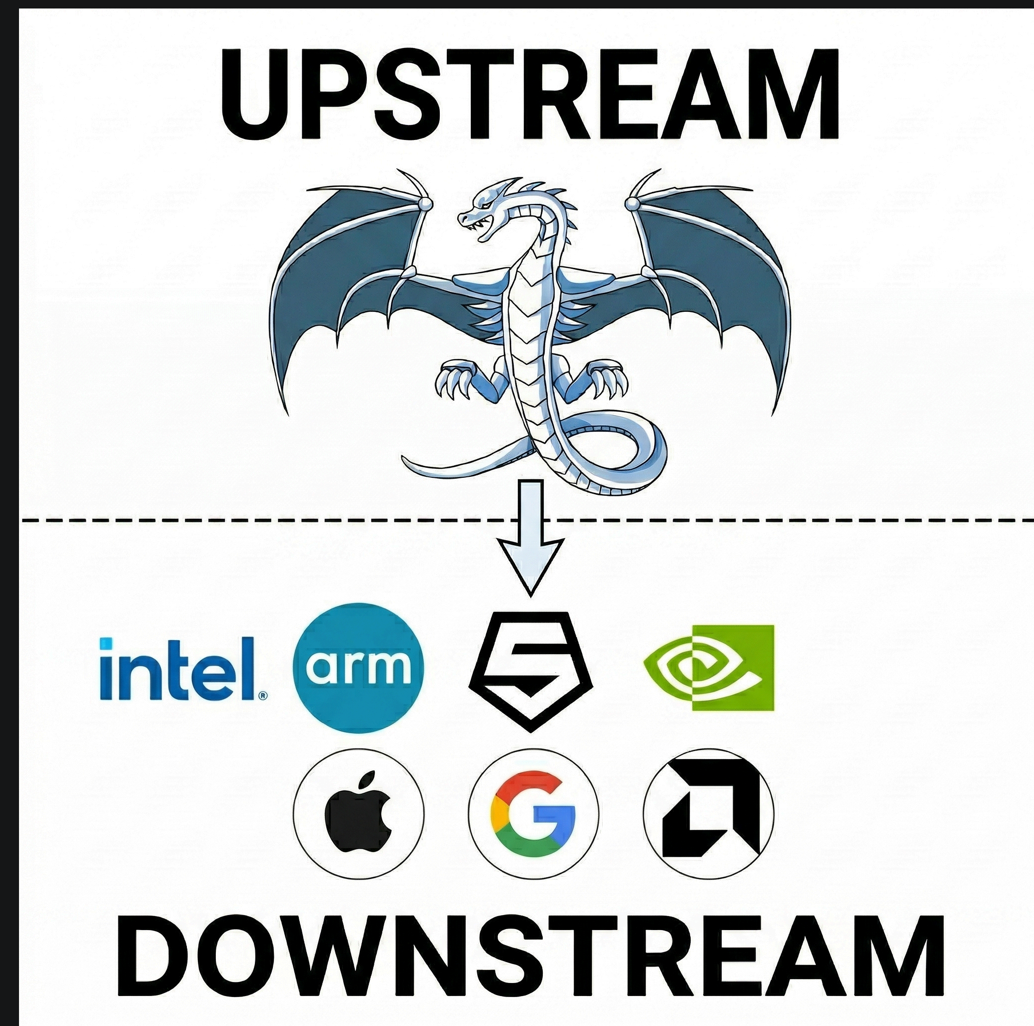
- Top 10 organizational contributors to LLVM, mostly based on github user org affiliation
  - LLVM has diverse industry support!
  - … But everyone has their own needs
  - Data from LFX Insights (link)
- **Hardware vendor** presence: 6/10 of these organizations sell compute hardware or designs
  - Hardware vendors heavily invested in performance



NVIDIA

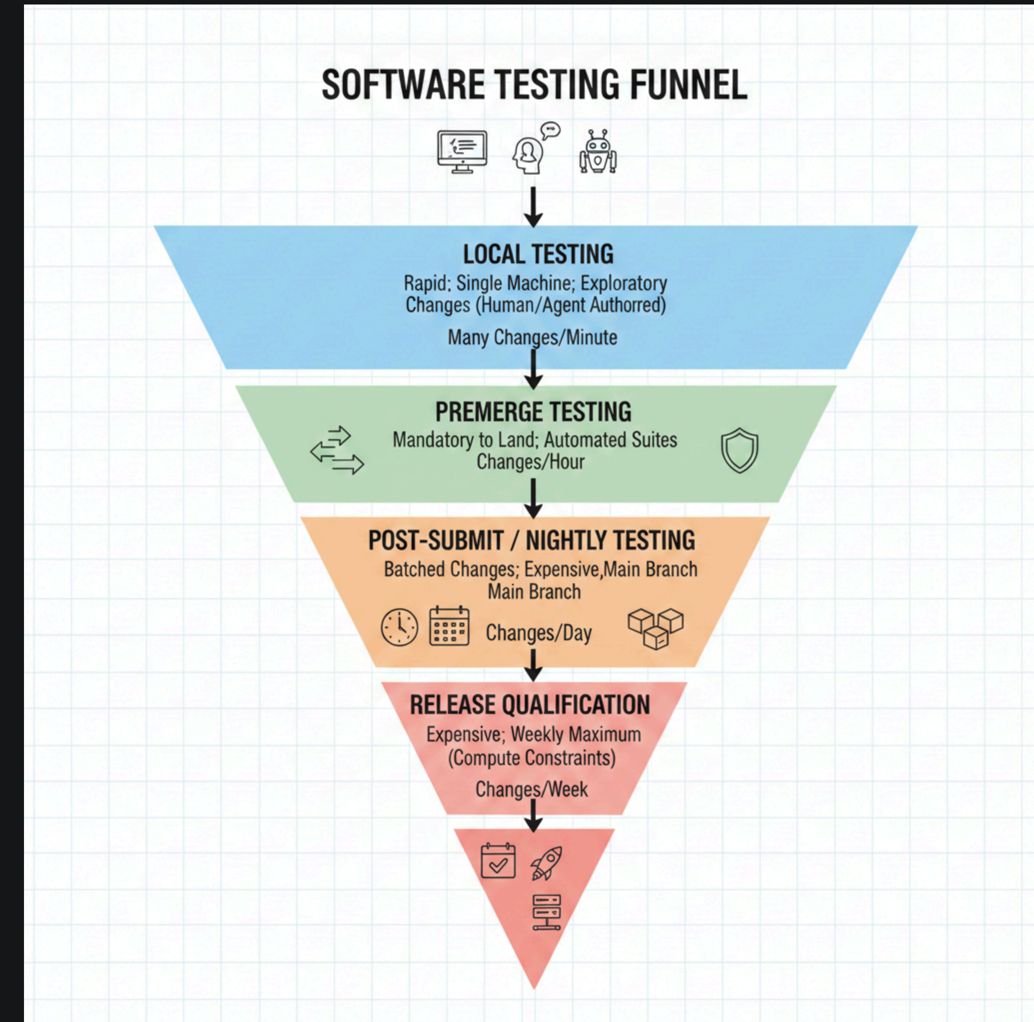
Downstream vendors

- Downstream requirements are important because they drive upstream contributions
- What happens when downstream CI fails?
  - The purpose of CI is to fail
- Typical outcomes:
  - Negotiate upstream fix
  - User error: fix downstream
  - Workaround downstream LLVM patch, long term tech debt
- Better testing can shift incentives to favor upstream contribution!



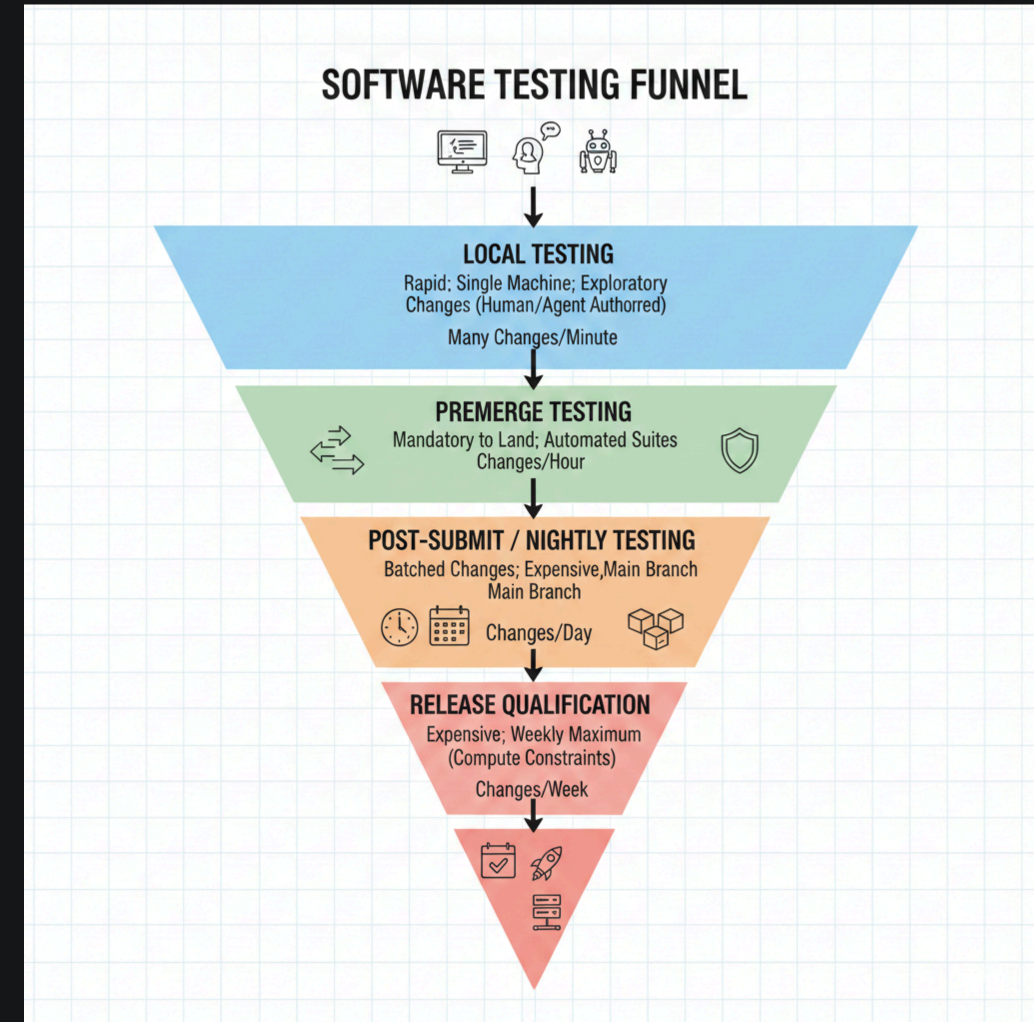
Enter: Testing Funnel

- **Testing Funnel:** Metaphor for release pipeline
  - Each stage has time and compute budget
- Stage width represents **change bandwidth**:
  - Quantity of changes that can be validated in the expected time budget
- Meeting the time budget requires batching changes together
- Final validation is expensive
  - Filter bad changes at high levels to mitigate



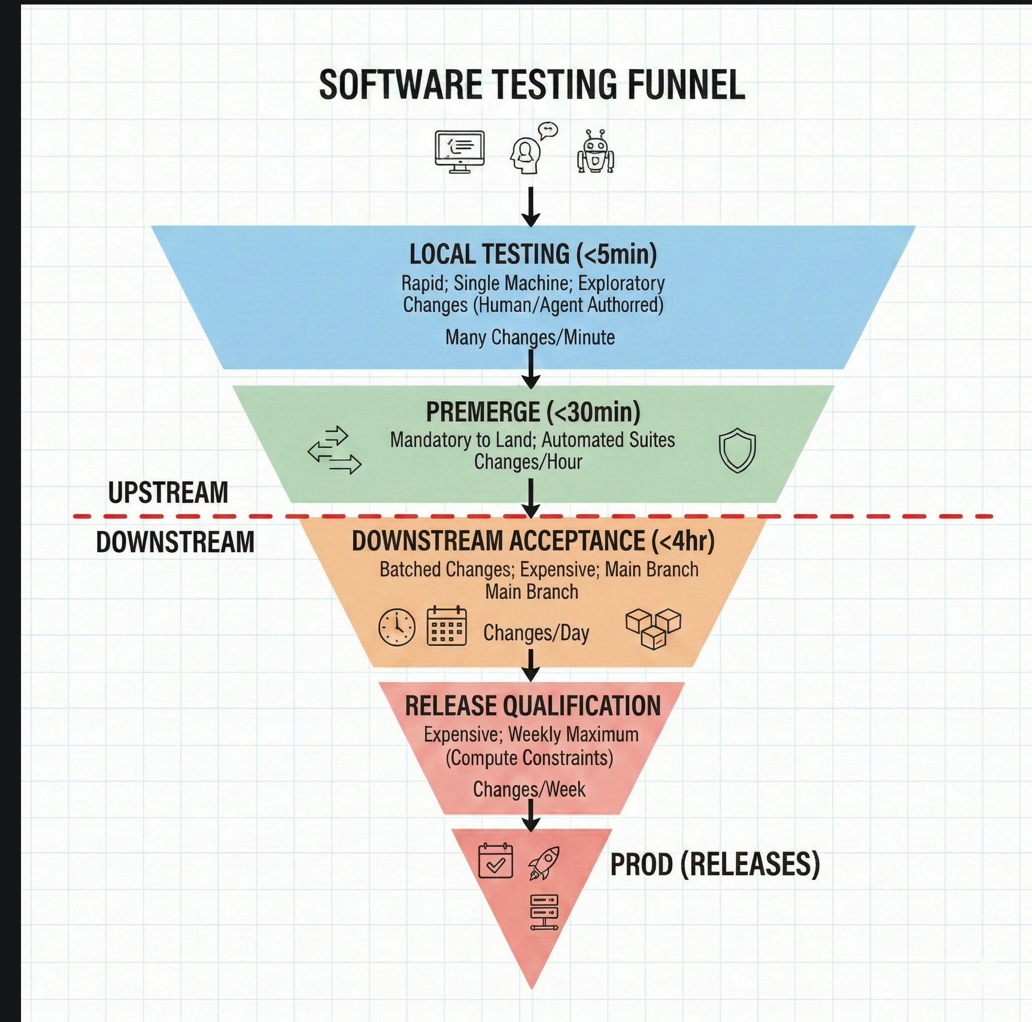
Budgets are important

- Testing resources are *always* constrained and demand is elastic
  - Test suites grow until it hurts 😬
- Development is about feedback loops, think DORA metrics
  - Fast feedback is critical to productivity
- **Rust BORS contrast:** queue flush time can be 5-10hrs (link), vs 30min LLVM premerge



Downstream testing

- **Problem:** Communicating across this boundary is expensive
  - Friction for authors & maintainers, not just a downstream problem
- Downstream hardware vendors often have performance tests
  - Upstream cannot afford to maintain low-noise perf tests
  - Dowstreams must denoise, root-cause, and lift into higher-level tests



NVIDIA

Case Studies of Status Quo

- Walk through some examples of reported regressions
  - Data gathered from Github PR API: comments created after `merged_at`
- **Blameless:** Contributors not accountable for untested requirements
- **Valuable:** Negotiating requirements is important work
  - Want to help, not do less
- **Laborious:** Consumes author and maintainer time, not just reporter time

Case: PR 138227, Clang modules

- Reported issue 12 days after merge
- Comments: 34
- Words: 6,464 — ~2 hours just to type @ 50 WPM
- Distinct authors: 9

Case: PR 183347, destructor `dead_on_return`

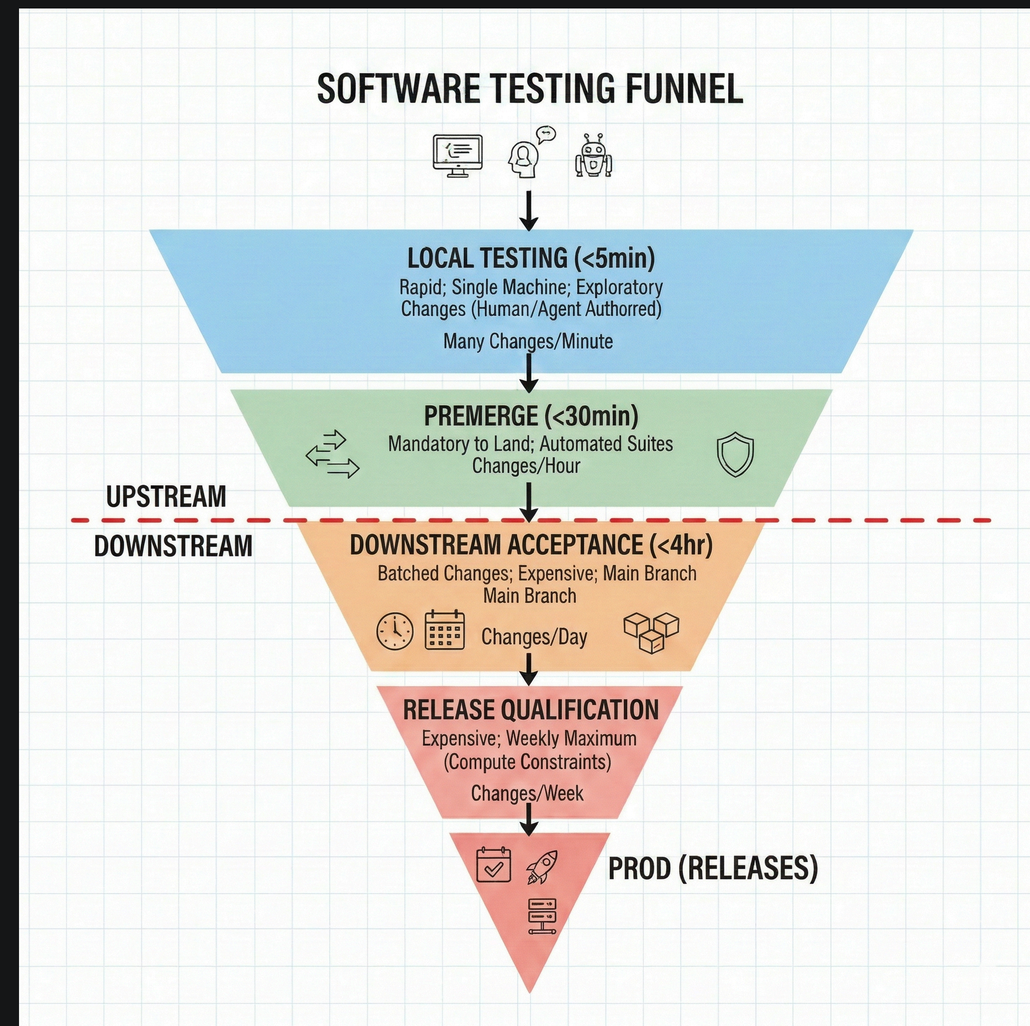
- Comments: 14
- Words: 1,504
- Distinct authors: 5
- Reporting delay: 1 days, fast, but manual!

Case: PR 133091, Revectorization

- Reporting delay: 5 days
- Comments: 9
- Words: 5,998 — 2hr typing @ 50wpm
- Distinct authors: 3

What to do?

- Invest in shared LLVM infrastructure:
  - The ROI in accelerating upstream contribution is huge
- Build tools that automatically lift interesting findings to higher layers
  - Design them to be adapted to downstream testing contexts
- Invest in lower layers of the testing funnel upstream

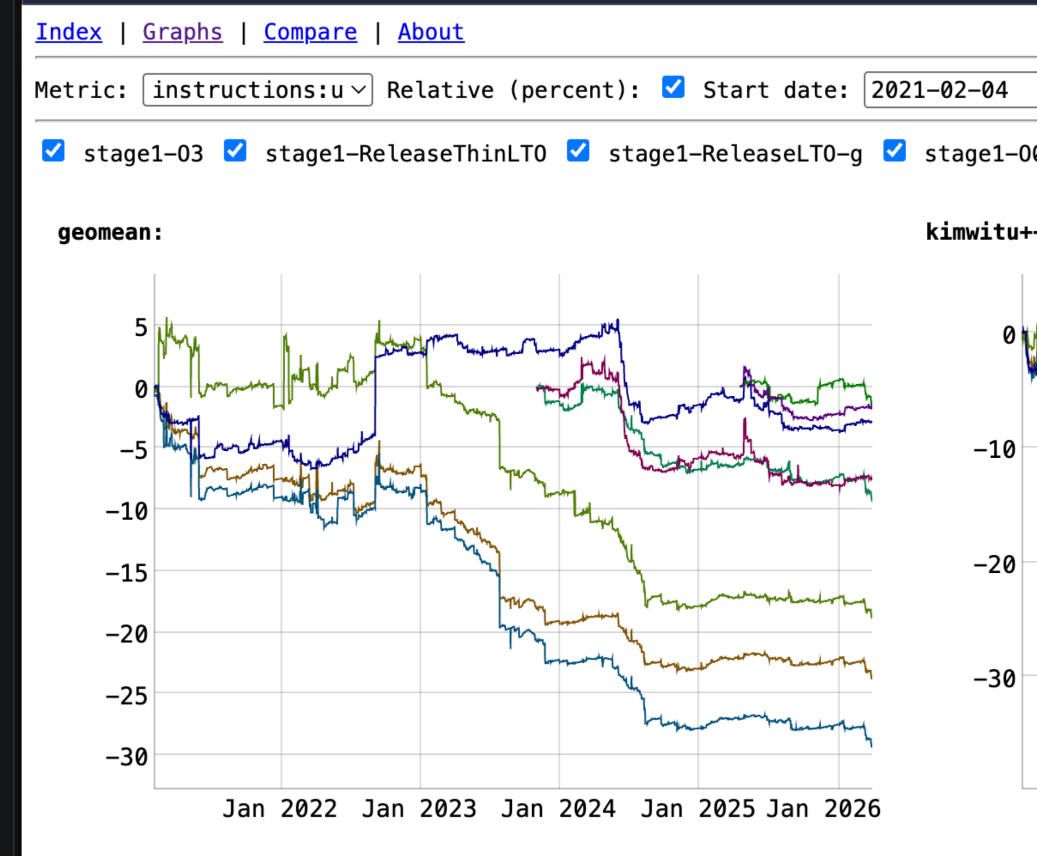


Principles for Infrastructure

- **Shared Truth:** Avoid "my machine" setup problems
- **Automate everything:** Need low-maintenance, self-healing systems
- **Static metrics:** Fast, deterministic, observable metrics are better than true metrics
  - Performance is hard to measure, but predictive proxies exist (spills, tokens)
- … we already do this! Let's see how we already apply these principles

Compile time tracker

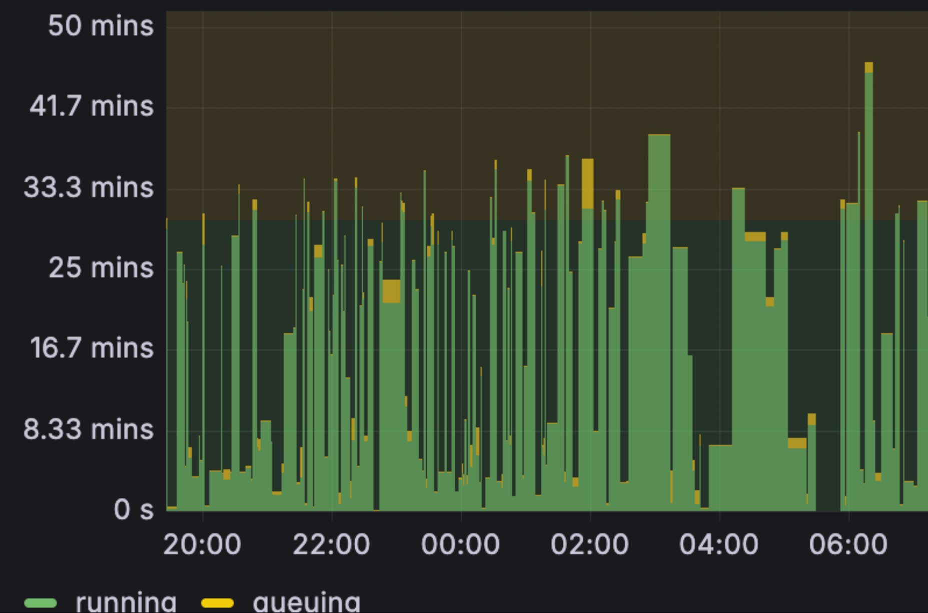
- Amazing example of leadership through metrics, thank you Nikita Popov!
- **Shared truth principle:** We can all see it
- **Static metric principle:** Instructions are an imperfect proxy for walltime
- **Automation principle:** These jobs can be triggered by pushing to a branch
- We should do this again for another quality metric!



NVIDIA

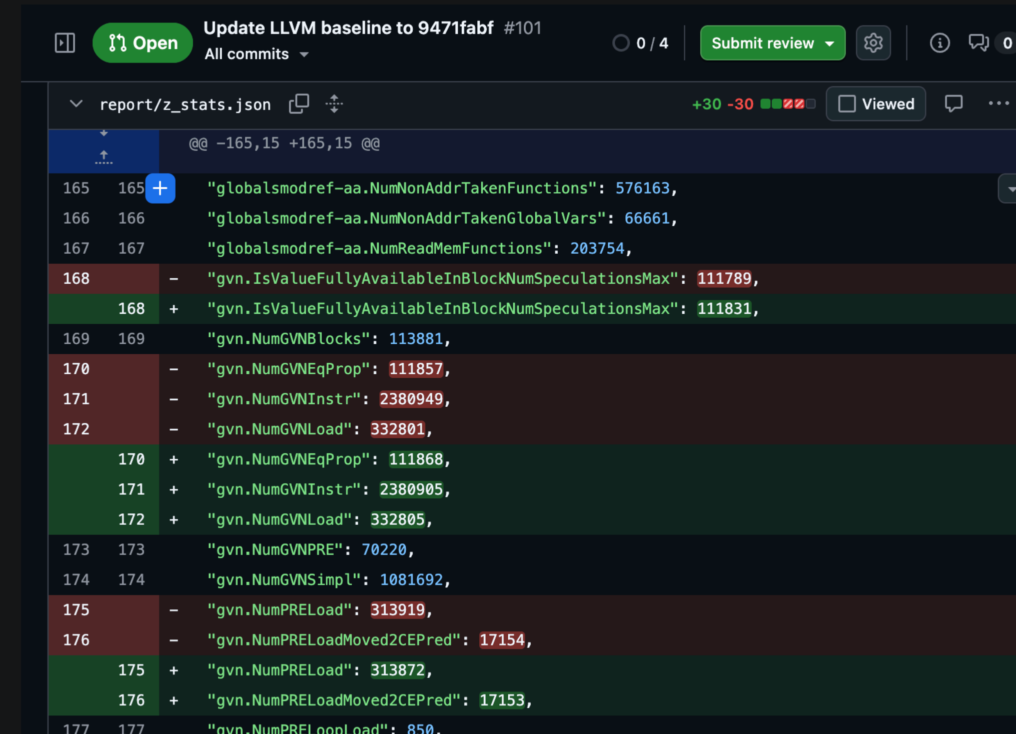
Premerge tests

- Premerge has come a long way since 2024
- Thank you to Aiden Grossman and Lucile Nihlen for maintaining reliable, 30min Linux & Windows github action checks!
- Obvious application of principles: **Publicly visible, automated**



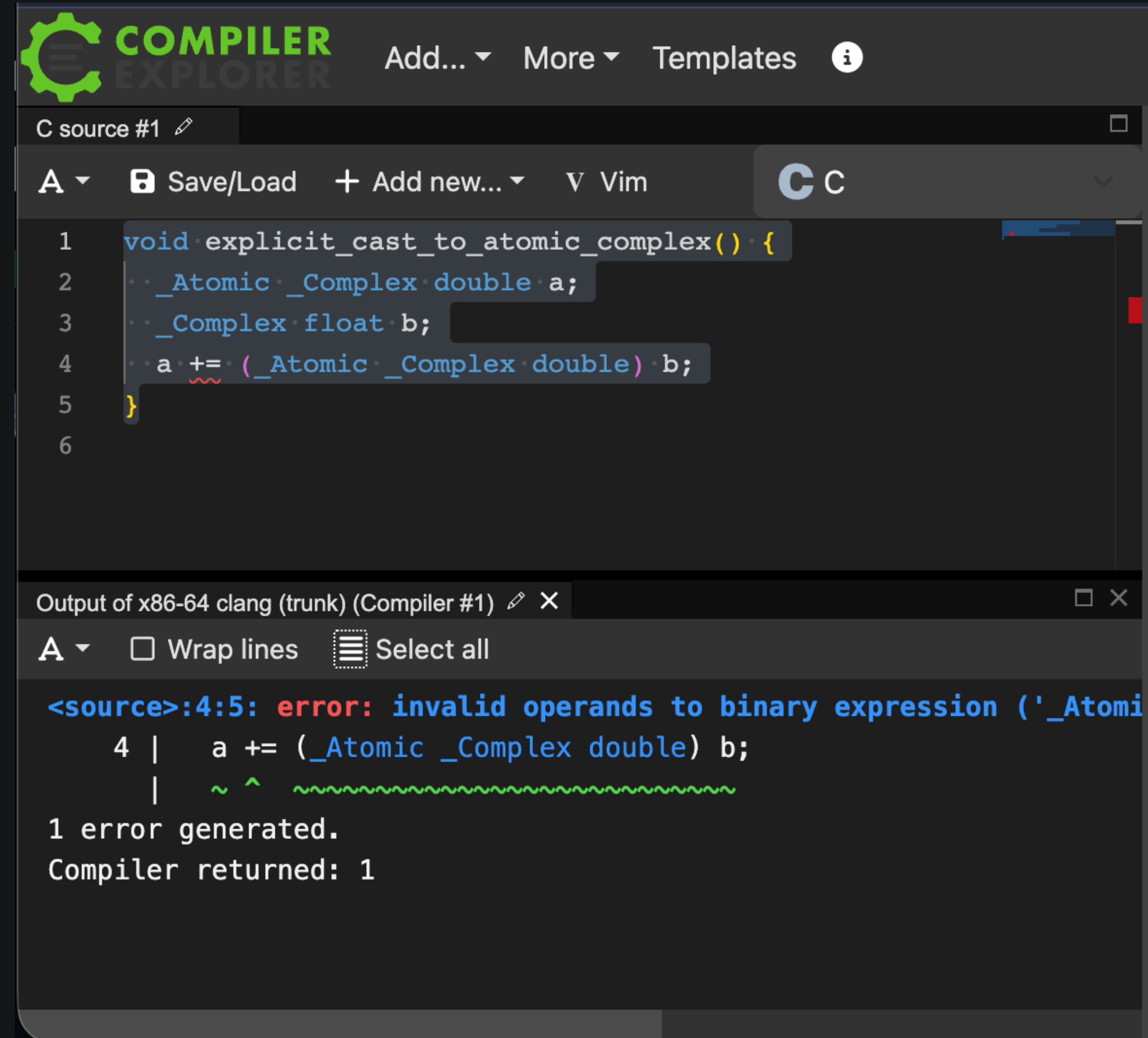
llvm-opt-benchmark

- Thank you, Yingwei Zheng! 🙏
- Builds corpus of Rust, C, C++ code in deterministic containers
  - Dumps IR & statistics during build
  - Uses Github PR view to diff (example)
- Limited to release builds for CPU targets
  - Should we contribute resources to expand scope (GPU, LTO, PGO, etc)?
- Principles: **Publicly visible, automated, static metrics**



Compiler Explorer / Godbolt

- Thank you, Matt Godbolt! 🙏
- Compiler explorer is **the exemplar for the power of shared truth**
- Live web demos are more real than shell transcripts



LNT

- Hosted **shared, public metrics** on debug info size and quality
- lnt.llvm.org is down
- Thank you to Luke Lau from Igalia for setting up a new instance at cc-perf.igalia.com 🙏
- There is also interest in a rewrite

LNT Database Suite nts Baselines System

nts / sifive-p550-rv64gc_zba_zbb-ReleaseLTO__clang_DEV__riscv64:3 / Run Results

- Machine Info
- Run Info
- View Options
- Report
- execution_time
- code_size

Runs:

- 04/08/2026 00:11:39
- 04/07/2026 00:07:46
- **04/06/2026 00:08:33**
- 04/05/2026 00:03:53
- 04/04/2026 00:10:05
- 04/03/2026 00:11:51

Compare To:

- 04/08/2026 00:11:39
- 04/07/2026 00:07:46
- 04/06/2026 00:08:33
- **04/05/2026 00:03:53**
- 04/04/2026 00:10:05

Run-Over-Run Changes Detail

Performance Improvements - execution_time	Δ	Previous	Cu
MicroBenchmarks/LCALS/SubsetCRawLoops/lcalsCRaw.test:BM_ICCG_RAW/44217	-6.46%	294.177	2
MicroBenchmarks/harris/harris.test:BENCHMARK_HARRIS/2048/2048	-1.65%	341726.000	336(
MicroBenchmarks/LCALS/SubsetALambdaLoops/lcalsALambda.test:BM_DEL_DOT_VEC_2D_LAMBDA/2	-1.37%	2.445	
MicroBenchmarks/LCALS/SubsetCRawLoops/lcalsCRaw.test:BM_HYDRO_1D_RAW/44217	-1.21%	139.240	1

Run-Over-Baseline Changes Detail

Performance Regressions - code_size		Δ (B)	Baseline	Current	σ (B)	Δ	
MultiSource/Benchmarks/MiBench/security-sha/security-sha	Profile	53.18%	2734.000	4188.000	0.000	0.00%	0.(
SingleSource/Benchmarks/Misc/himenobmtxpa	Profile	51.56%	2176.000	3298.000	0.000	0.00%	0.(

NVIDIA

What's Working, What's Missing?

- These projects demonstrate the principles **actually work**
  - Community adopts tools that are public, automated, and trustworthy
- Coverage so far: compile time, middle-end static metrics, some RISC-V perf
- **Gaps:**
  - Root cause analysis is manual or automated on private infra
  - Lack robust integration testing — the middle of the funnel
  - AI/ML/HPC workloads not covered above

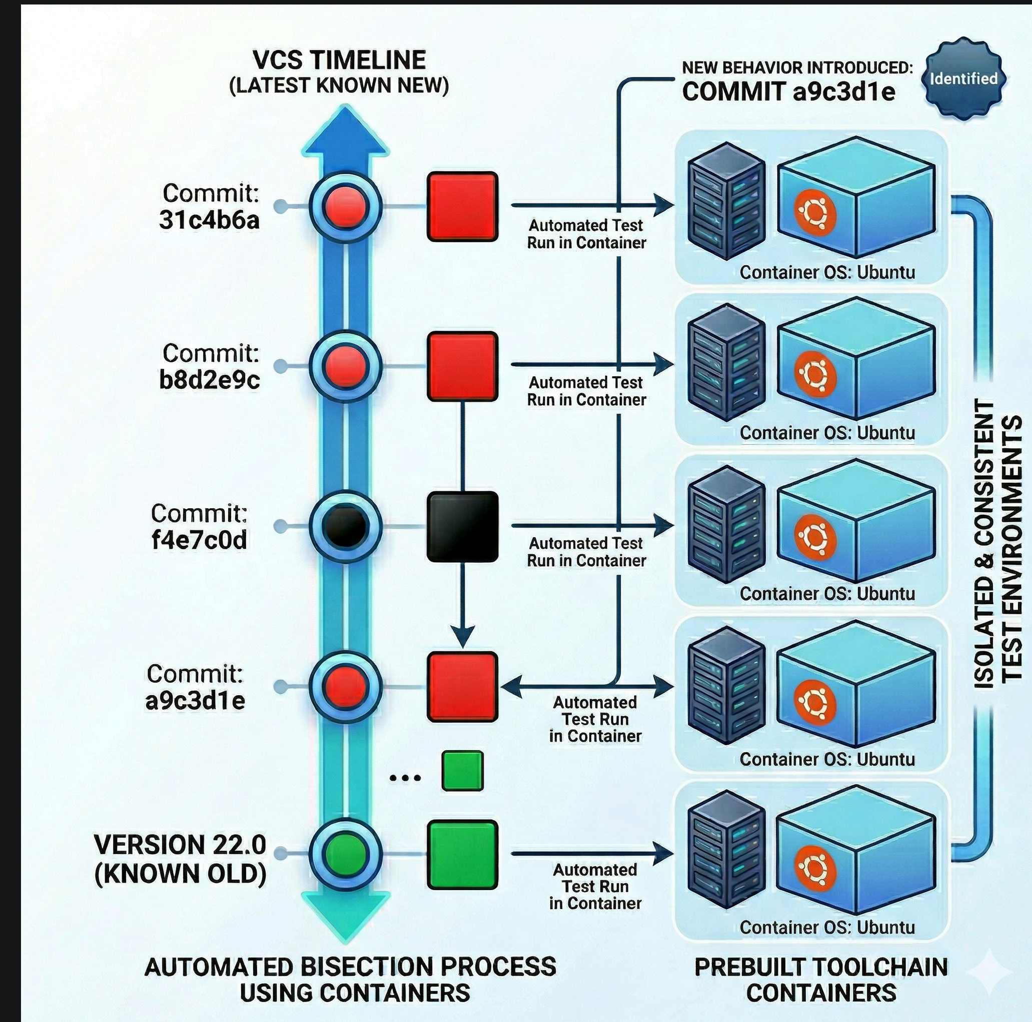
Idea: Standard finding format

- Standardize on interestingness scripts using LLVM tools as the format of findings (think cvise, llvm-reduce)
  - Execute tests in pre-defined container with varying versions of LLVM tools
  - **Shared truth:** Public record of analysis is valuable
- Downstream release testing should translate internal regressions to this format

```bash
#!/bin/bash
# Does input have more than 3 spills?
((llc $1 | grep SPILL | wc -l > 3))
```

Idea: Automate root cause analysis

- Manual, client-side `git bisect` is not enough
- Three-dimensional search space:
  - LLVM revisions: git bisect
  - User source code: cvise, object file bisection
  - Compilation phase: opt bisect
- **Vision:** Human conversation begins with a link to an in-progress analysis



Idea: Integration testing

- Integration tests are a tool to expedite code review
  - Reviewing untested to high-risk areas (MSSA or your favorite) is expensive
  - Re-reviewing relands is expensive
- **Beyond premerge:** Integration tests don't fit in premerge resource budget
  - 🤔 On-demand Clang self-host trybot? Reconsider David Spickett's RFC
- **Problem:** Who monitors? How to improve signal?
  - Automation principle: Feed findings straight into root cause analyzer, no manual action required
  - See Clang `-fcrash-diagnostics-dir=`

Distro Testing

- Linux distros support GCC through distro testing (musttail ref)
- Clang diagnostic changes often stuck due to lack of real world data
  - Recall `-Wstrict-prototypes` conversation
- **Non-goal:** Maintain a whole distro!
  - **Automation principle:** Must self-heal, fallback to old Clang or GCC to keep going
  - Feed root cause tool automatically, ignore non-actionable failures

Static Metrics for GPUs

- GPU compiler quality is critical for HPC, AI frameworks, and DL workloads
- Static proxies exist: spill counts, vector instructions, unroll counts, etc
- **Gap:** No public, shared, automated tracking for GPU targets
    - Equivalent of llvm-compile-time-tracker for NVPTX, AMDGPU, SPIR-V
- Downstream vendors currently run this privately — results are invisible upstream

What can downstreams do differently?

- Build open, dual-purpose infra that can be deployed downstream and upstream
  - How many auto-reduction root cause pipelines do we need to build?
  - Expanded scope is expensive, but the payoff of shared visibility in smooth upstream collaboration is **huge**
- Design release pipelines to **automatically lift** findings to tests for top of the funnel
  - Much easier to talk about spill counts and code size than benchmark swings
- **LLVM can help:** Bake in tools to produce reproducers, similar to Clang & LLD

Conclusion: What should we do?

- Create **publicly visible, automated** project infrastructure that tracks correctness and **static quality metrics**
  - Everyone, both upstream and downstream, should be interested
  - Negotiating requirements between upstream/downstream is a forever problem
- **Beyond premerge:** Build out lower layers of testing funnel
  - post-submit, nightly, and release
  - Automate analysis of any findings
- Work together to meet everyone's needs: Upstream & downstream, working together 😄
- **Call to action:** Reach out to infra area team if you can help make this a reality!

Backup slides

What about LLMs?

- LLMs can help! However, there's a lot of value capturing repeated LLM actions in deterministic, repeatable actions in a **deterministic control loop**
- See **Bazel bot** which uses a deterministic program to fix deps 40+% of the time
  - The LLM acts as a backstop: recovers from obscure edge case errors
- LLMs can be used to debug, not just generate code:
  - LLMs can pre-analyze issue to make regressions more actionable
  - LLMs can be a reduction aide in classic reduction pipelines
- Core value is **automation principle**, LLMs mop up unstructured, edge-case failures

Dev containers

Consider building a live demo container for the keynote: time an untuned LLVM build with default settings on a MacBook, then compare it against an opinionated dev container with PCH and other fast-build defaults. Show the audience what first-time contributors could have if we invested in the tooling.

Concrete proposals

- Containerized auto-bisection:
  - Accelerate bug finding with buckets of pre-built toolchain artifacts (consider ELF shaker)
- Builtin IR and pipeline capture:
  - Why do we rebuild this in every LLVM frontend?
  - Build once, reuse for Clang, Swift, Rust, CUDA, ROCM, etc