RELEASING C++ TOOLCHAINS WEEKLY IN A "LIVE AT HEAD" WORLD

Jorge Gorbe Moya

jgorbe@google.com

@slackito

Jordan Rupprecht

rupprecht@google.com

@jrupees

ABOUT US

C++ Production Toolchain Team @ Google

WHAT IS A C++ TOOLCHAIN?

The set of tools needed to build C++ programs:

- Compiler
- Linker
- Binary utilities
- C++ standard library
- ...

PRODUCTION @ GOOGLE

- Non-production: Android, iOS, Chrome, ...
- Production: Search, Ads, Gmail, ...

SOUNDS IMPORTANT

We must only use the stablest of all compilers, right?

HAHA, NOPE!

Google C++ prod toolchain is updated from the upstream HEAD once every <u>1-2 weeks</u>

WAIT A MINUTE...

- Doesn't everything break?
- Don't users get mad?
- Is a toolchain built from HEAD really stable?
- Most importantly: is it worth it?

IS IT WORTH IT?

Shorter feedback cycles

- Performance
- Diagnostics & sanitizers
- Security fixes
- Easier & safer upgrades

WHAT ABOUT TOOLCHAIN BUGS?

Yes, the toolchain has bugs

Just like any other piece of software

Living at head means those bugs are fixed faster

LET'S FIX TOOLCHAIN BUGS!

... or: are we just an elaborate build bot?

- Millions of tests
- Hundreds of MLoC (of C++)
- O(days) to revert bad patches

BACKGROUND FACTS

- Everything is in the same repository
- Everything is built from source
- We can fix our users' code!
  ⟶ transparent upgrades

BACKGROUND FACTS

- Build / test farm with CI and presubmit checks
- Strict Google-wide application of the Beyoncé rule

  > *"If you like it, then you shoulda put a ~~ring~~ test on it"*

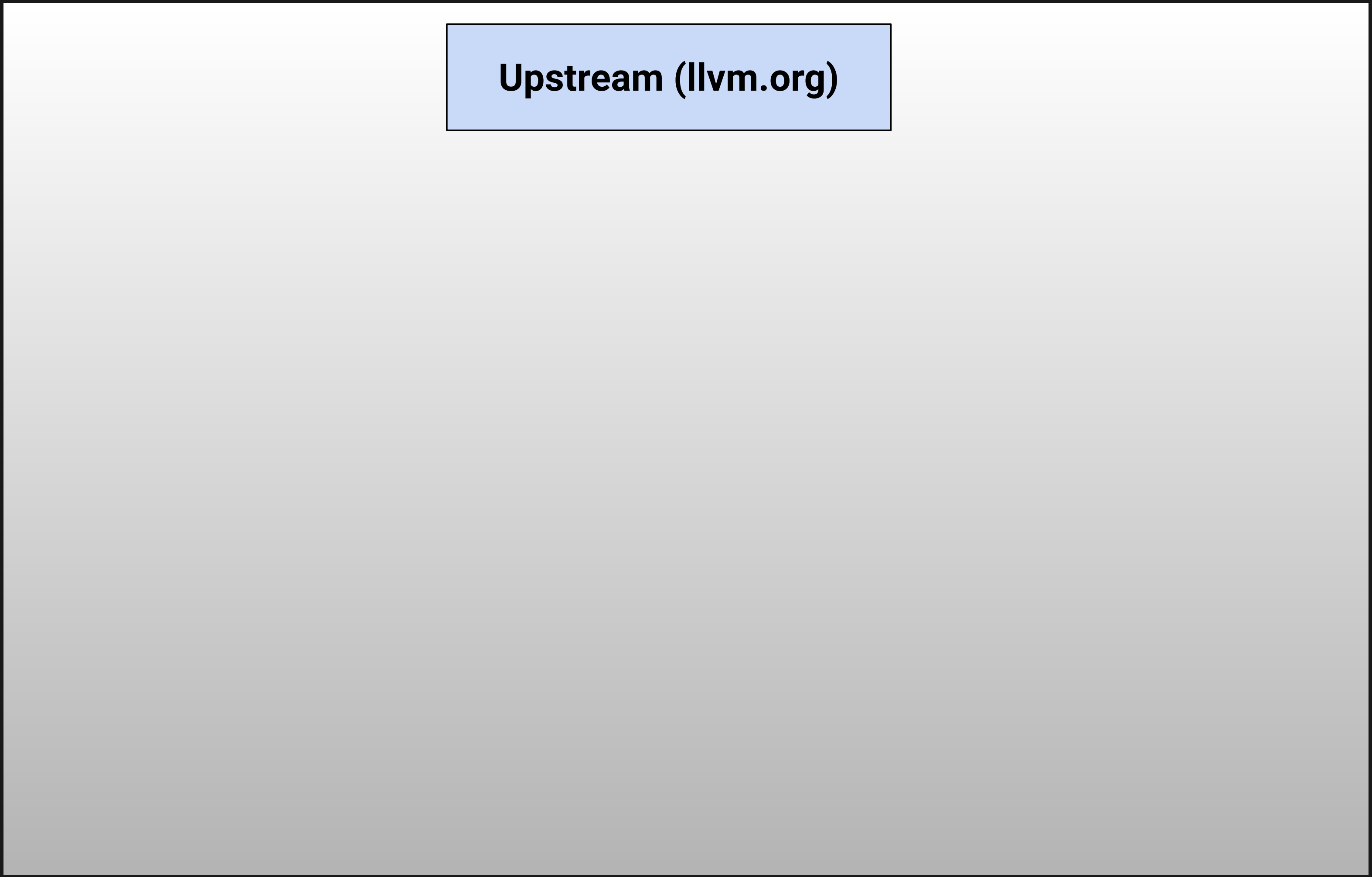
  ⟶ we can test everything our users care about!
- LLVM-based toolchain (but this is not an LLVM talk)

WHAT IS A RELEASE?

- Toolchain binaries in the repo
  - Repeatable builds at any given repository revision
- "release" = committing new toolchain binaries

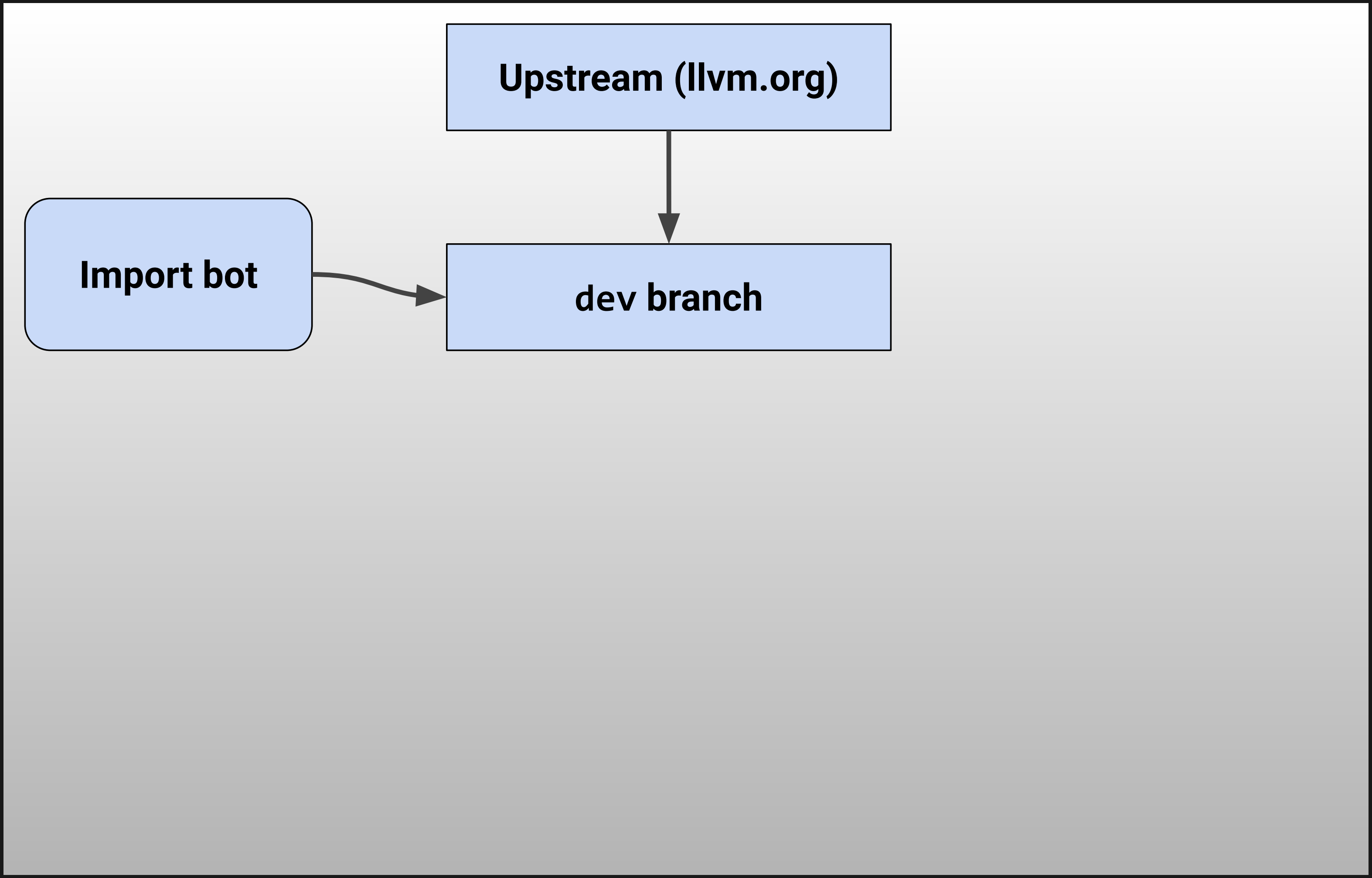
LIFE OF A TOOLCHAIN CHANGE

LIFE OF A TOOLCHAIN CHANGE



Upstream (llvm.org)

LIFE OF A TOOLCHAIN CHANGE

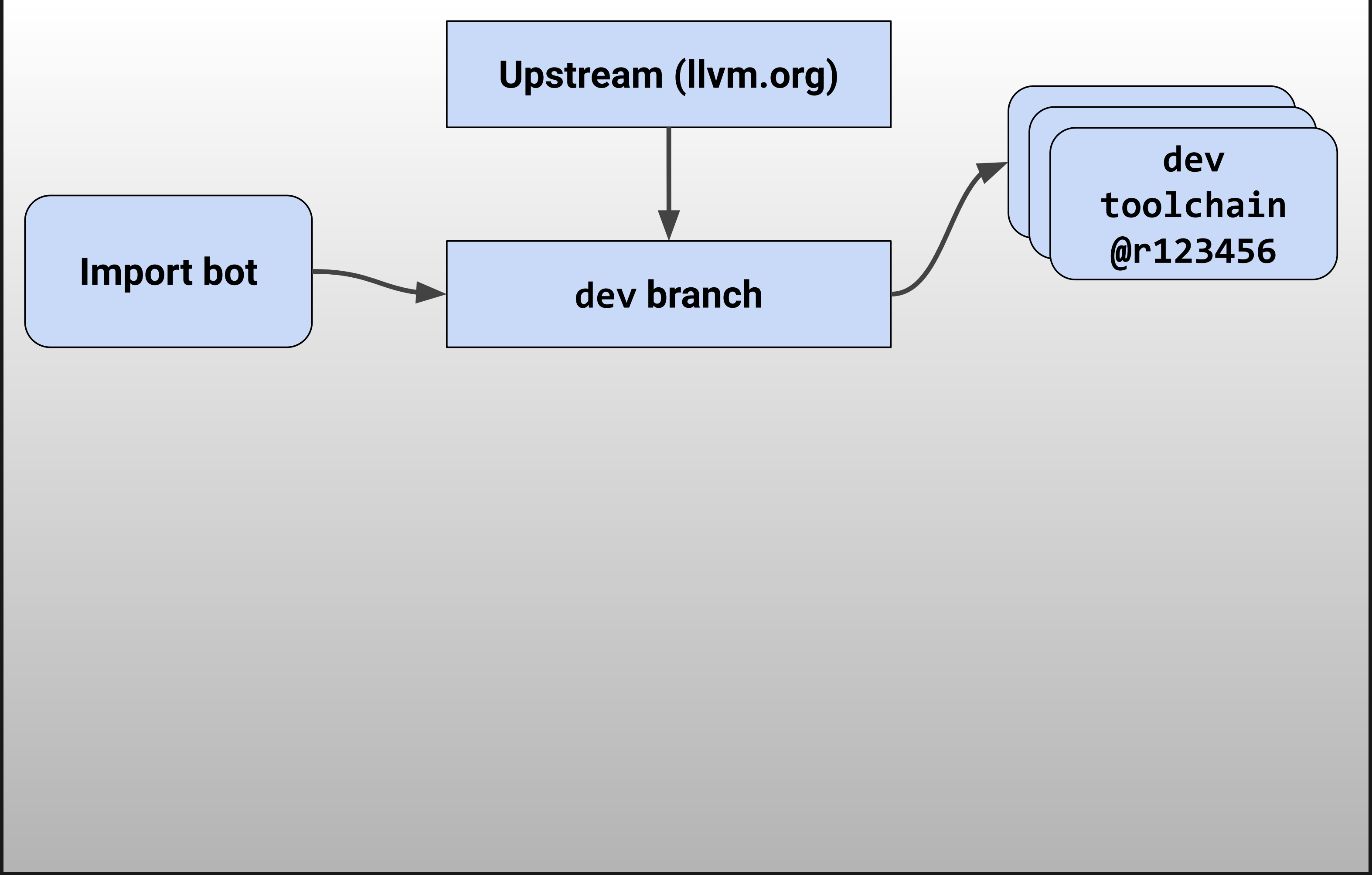


Upstream (llvm.org)

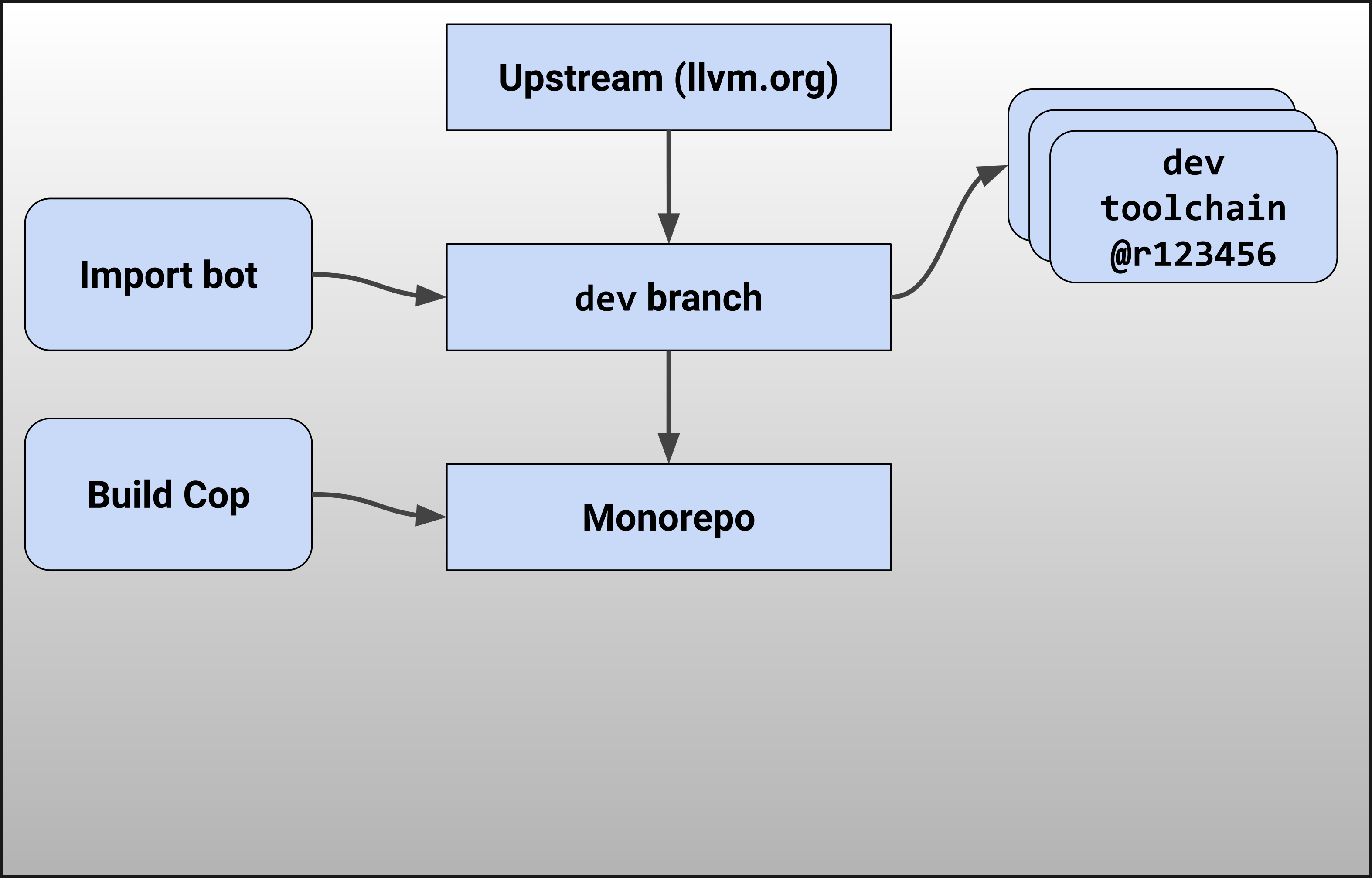
Import bot

dev branch

LIFE OF A TOOLCHAIN CHANGE



LIFE OF A TOOLCHAIN CHANGE


Upstream (llvm.org)

Import bot

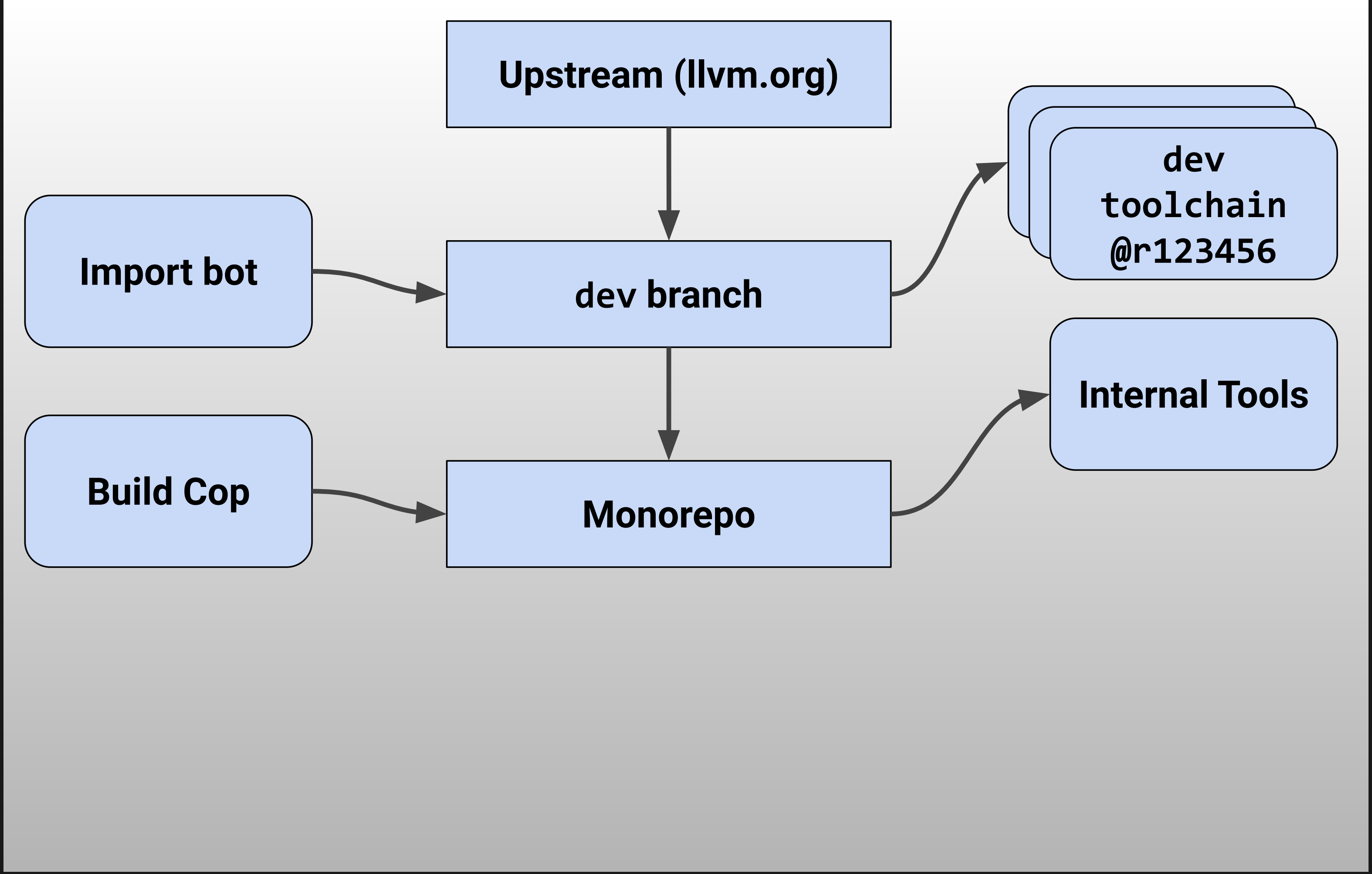
dev branch

dev
toolchain
@r123456

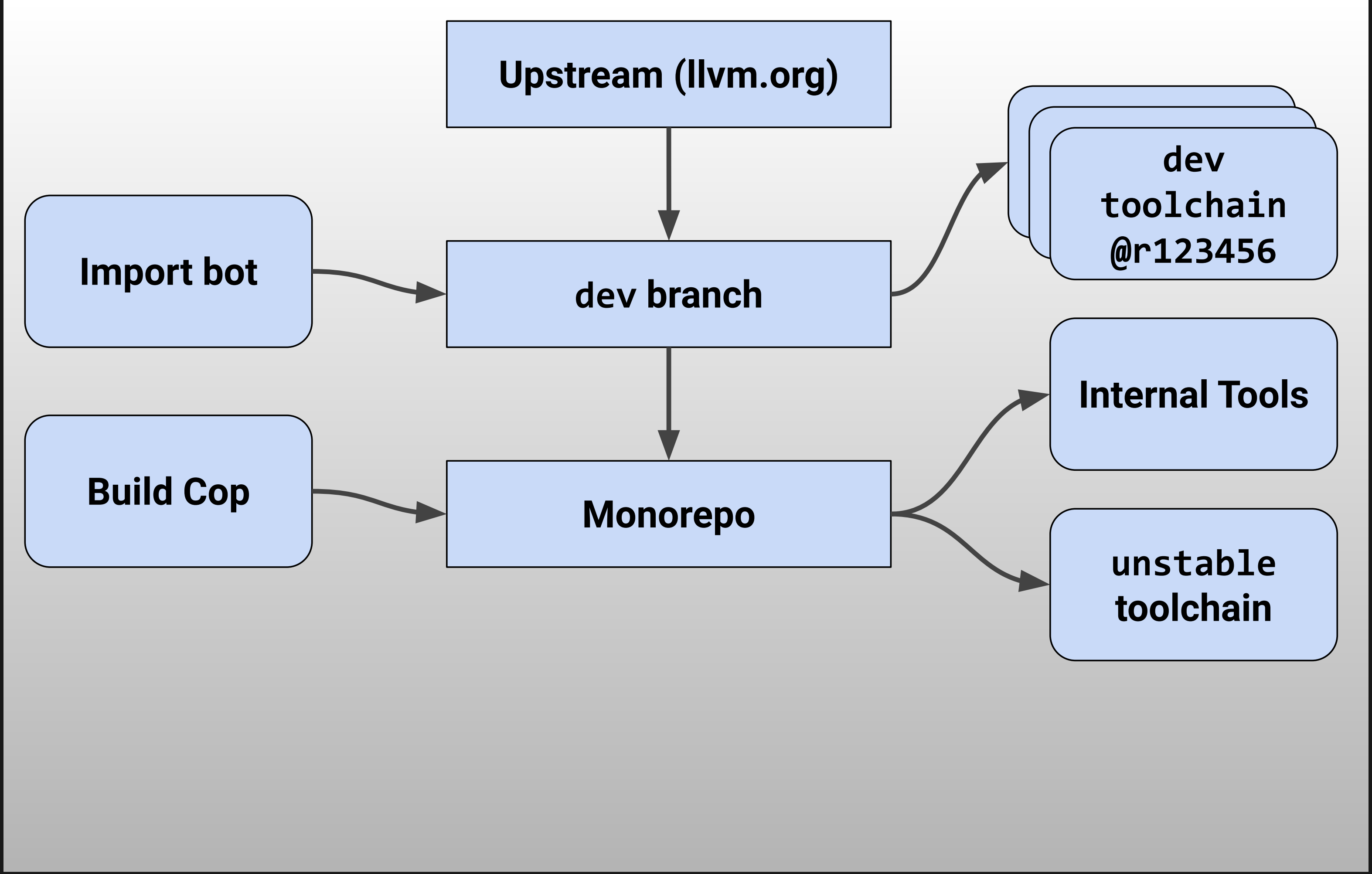
Build Cop

Monorepo

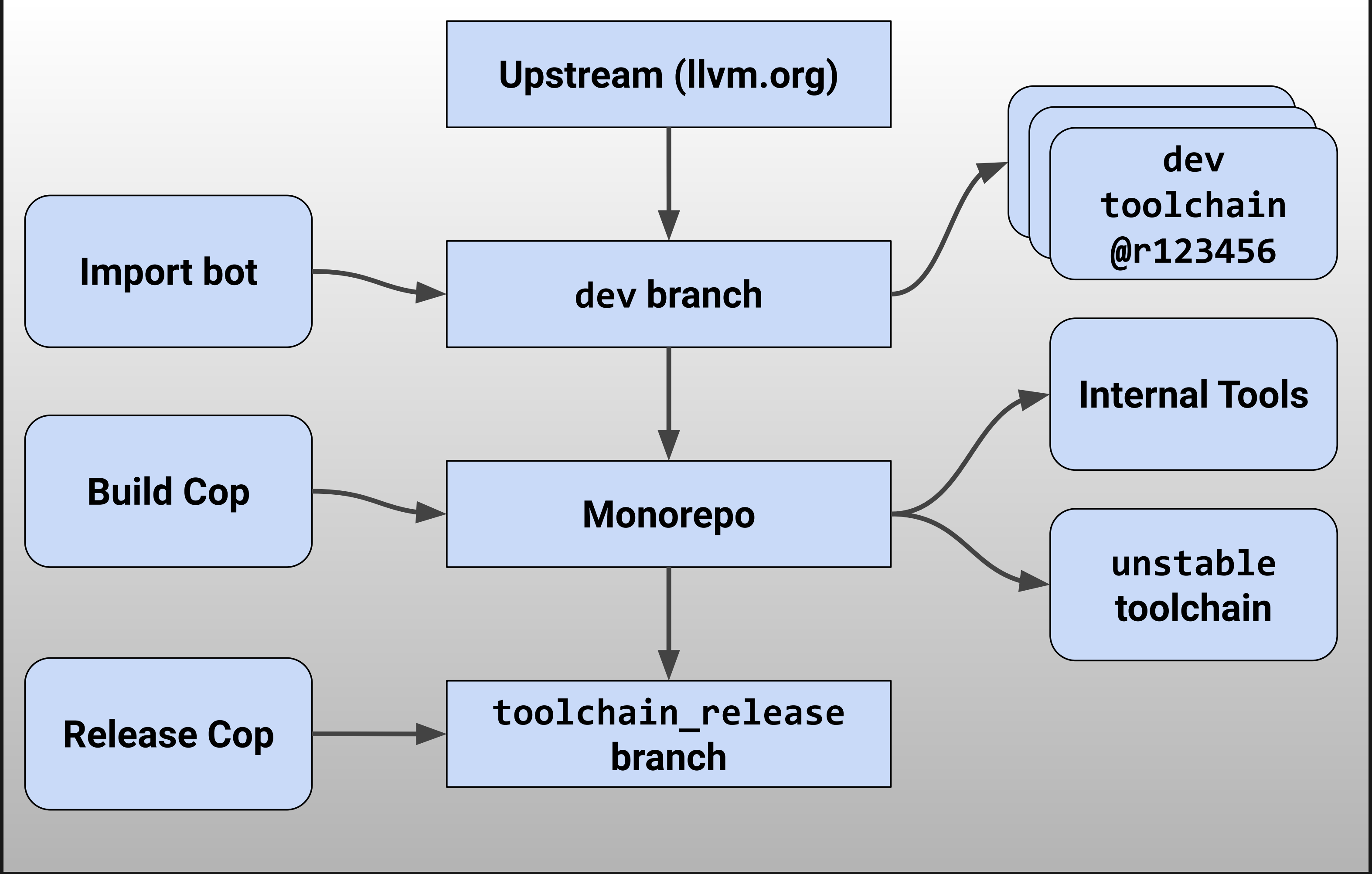
LIFE OF A TOOLCHAIN CHANGE



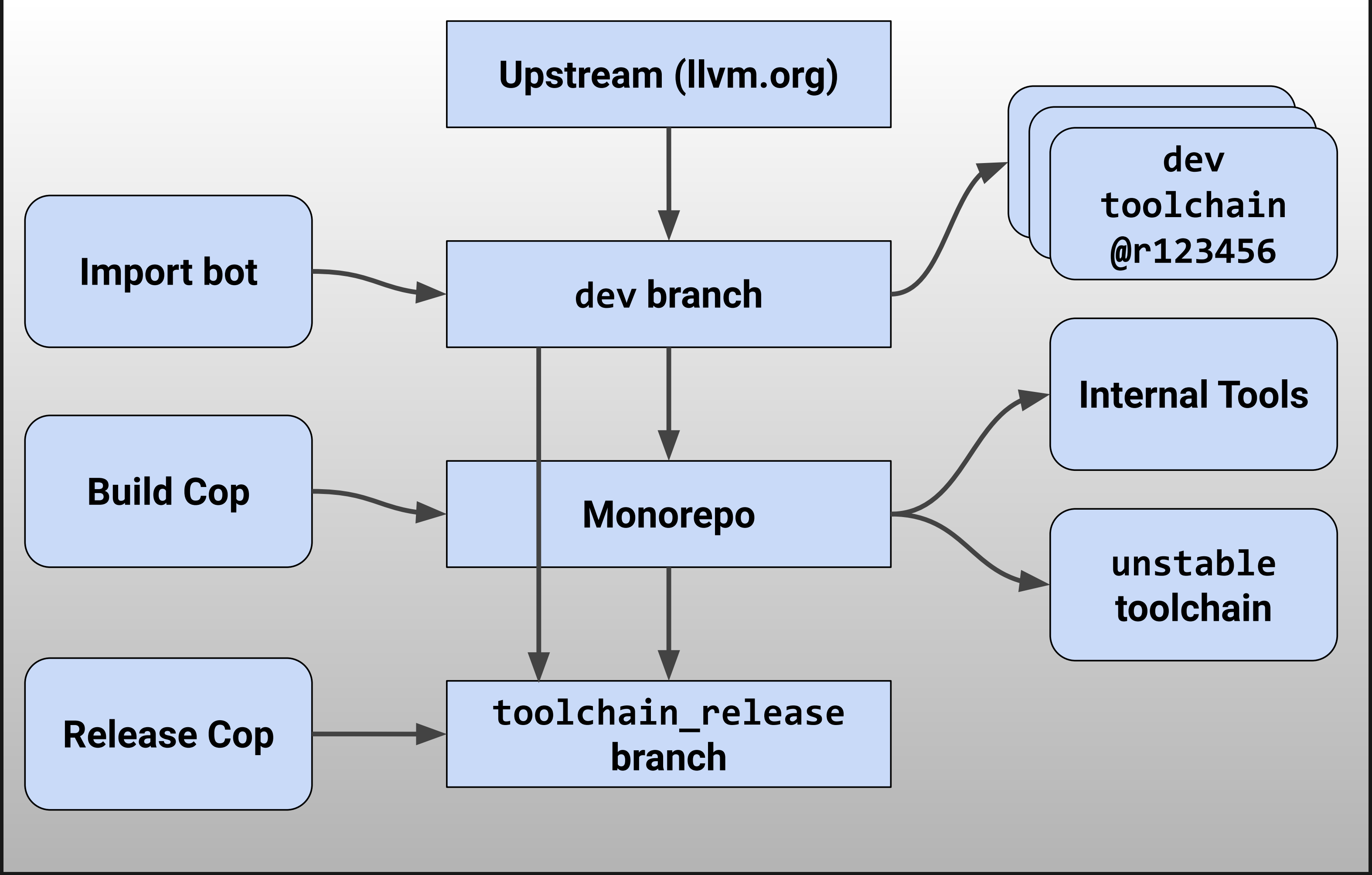
LIFE OF A TOOLCHAIN CHANGE



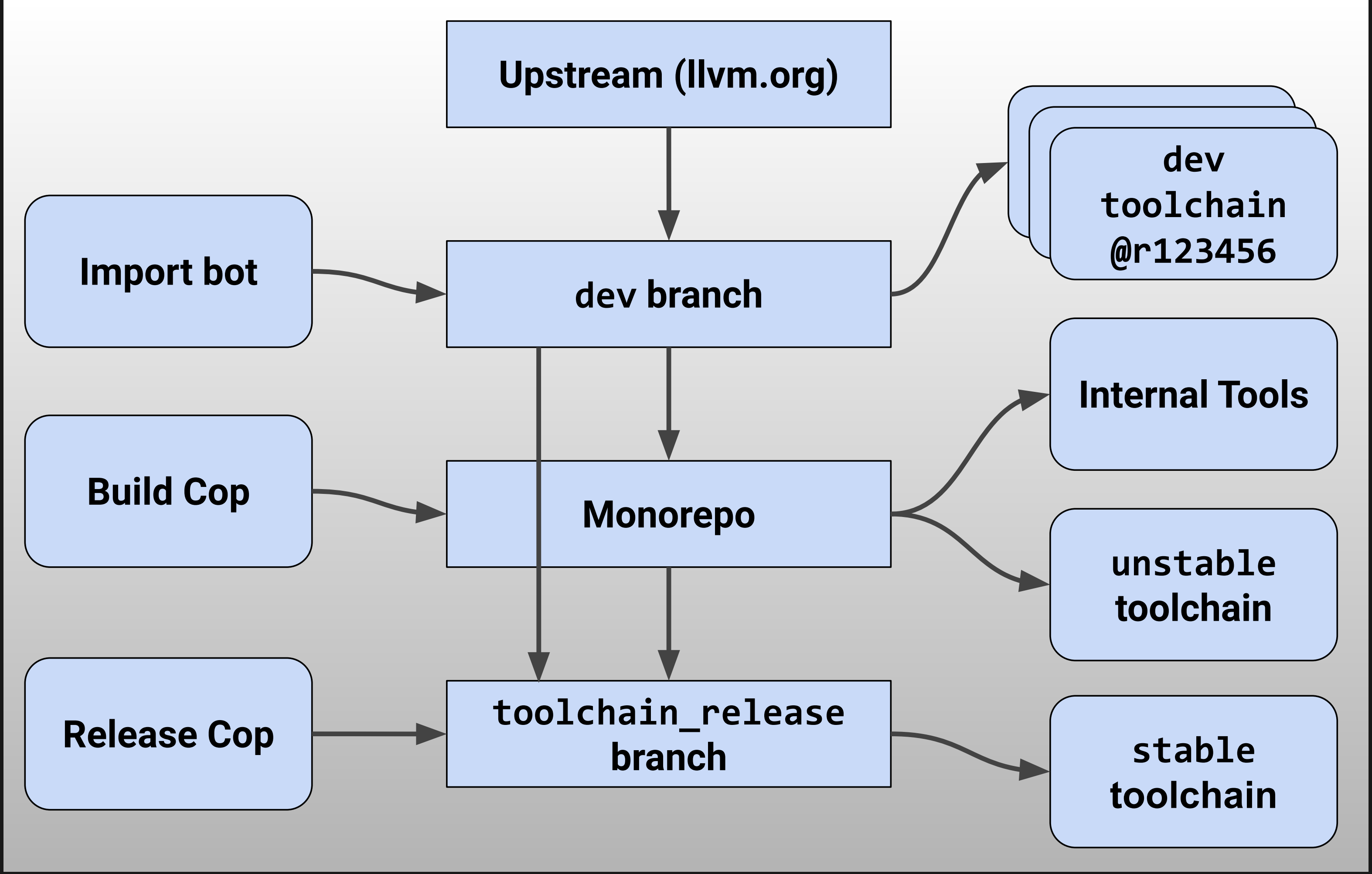
LIFE OF A TOOLCHAIN CHANGE



LIFE OF A TOOLCHAIN CHANGE



LIFE OF A TOOLCHAIN CHANGE

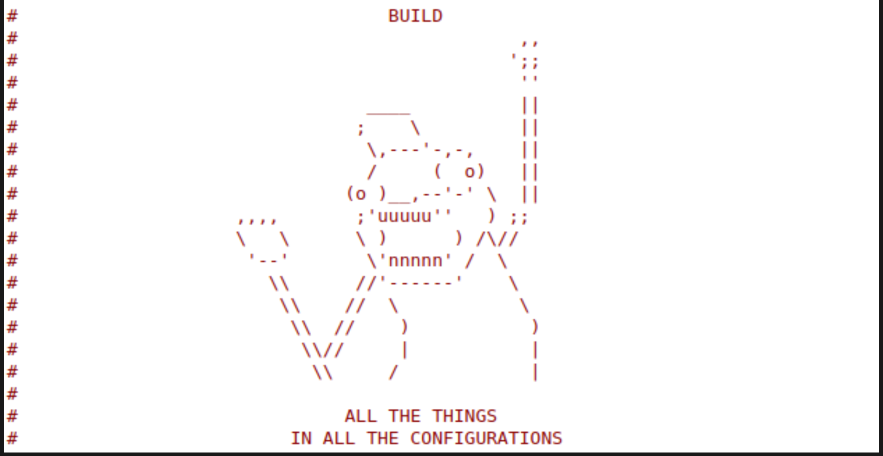


HOW DO WE ACTUALLY DO THAT?

- How do we verify correctness and performance?
- How can things go wrong?
- How do we fix it?

TOOLCHAIN TESTING

- Build & test ALL the things
- In ALL the configurations
- Automatically



```
#                                  BUILD
#                                              ,,
#                                             ';;
#                                              ''
#                                ____          ||
#                               ;    \         ||
#                                \,---'-,-,    ||
#                                /     (  o)   ||
#                              (o )__,--'-' \  ||
#                    ,,,,       ;'uuuuu''   ) ;;
#                    \   \      \ )      ) /\//
#                     '--'       \'nnnnn' /  \
#                       \\      //'------'    \
#                        \\    //  \           \
#                         \\  //    )           )
#                          \\//     |           |
#                           \\     /            |
#
#                              ALL THE THINGS
#                         IN ALL THE CONFIGURATIONS
```

(ascii art from our testing script, based on a drawing by Allie Brosh)

BENCHMARKING

Closer to regular testing than it seems at first glance

TESTING

Tests allow us to make changes while preserving correctness

BENCHMARKING

Benchmarks allow us to make changes while preserving performance

TESTING

We need a mix of unit tests and integration tests

BENCHMARKING

We need a mix of micro- and macrobenchmarks

TESTING

Test things that actually matter

BENCHMARKING

Measure things that actually matter

TESTING

Flaky / brittle tests are bad

BENCHMARKING

Noisy benchmarks are bad

NOT EVERYTHING IS THE SAME...

Mostly, benchmarks are fuzzier

- Tests are PASS/FAIL, benchmarks give you a number
- Performance decisions need non-local information

IF YOU LIKE IT...

Put a benchmark on it!

WHAT WE BENCHMARK

- Microbenchmarks for hot code (CPU or latency):
    - in individual projects — e.g. matrix math
    - in common infrastructure — e.g. protobufs
- A few macrobenchmarks
- Public benchmarks, for upstream-friendly test cases

HOW WE BENCHMARK

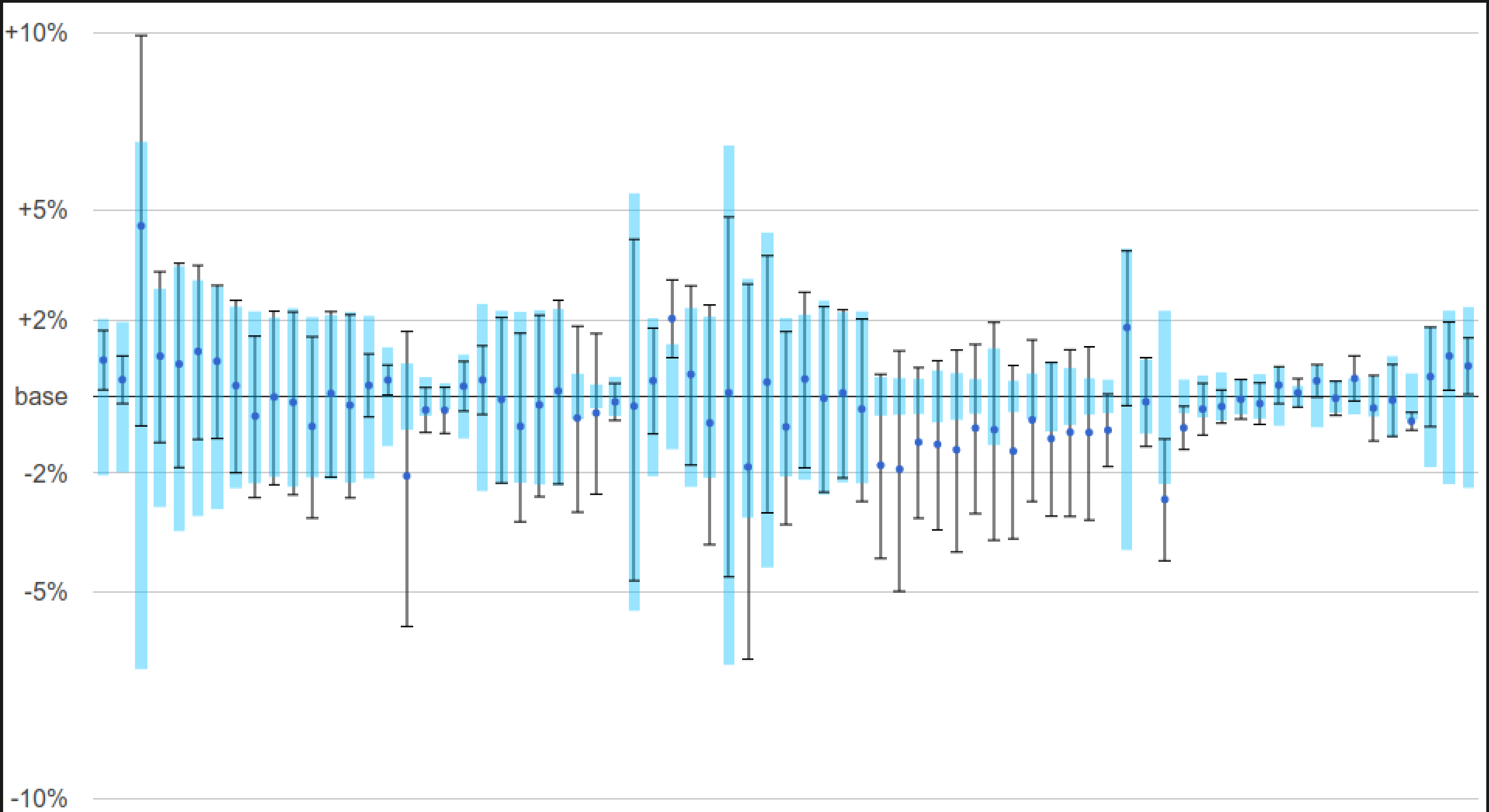
Focus on **reproducibility**

If we can't reproduce it, we can't debug it effectively

OUR FIRST TRY AT STABILITY

- Put some prod-like machines in an dedicated cluster
- Multiple runs, compute confidence intervals
- Run an A/A test to see how noisy it is

snappy benchmark, 10 runs



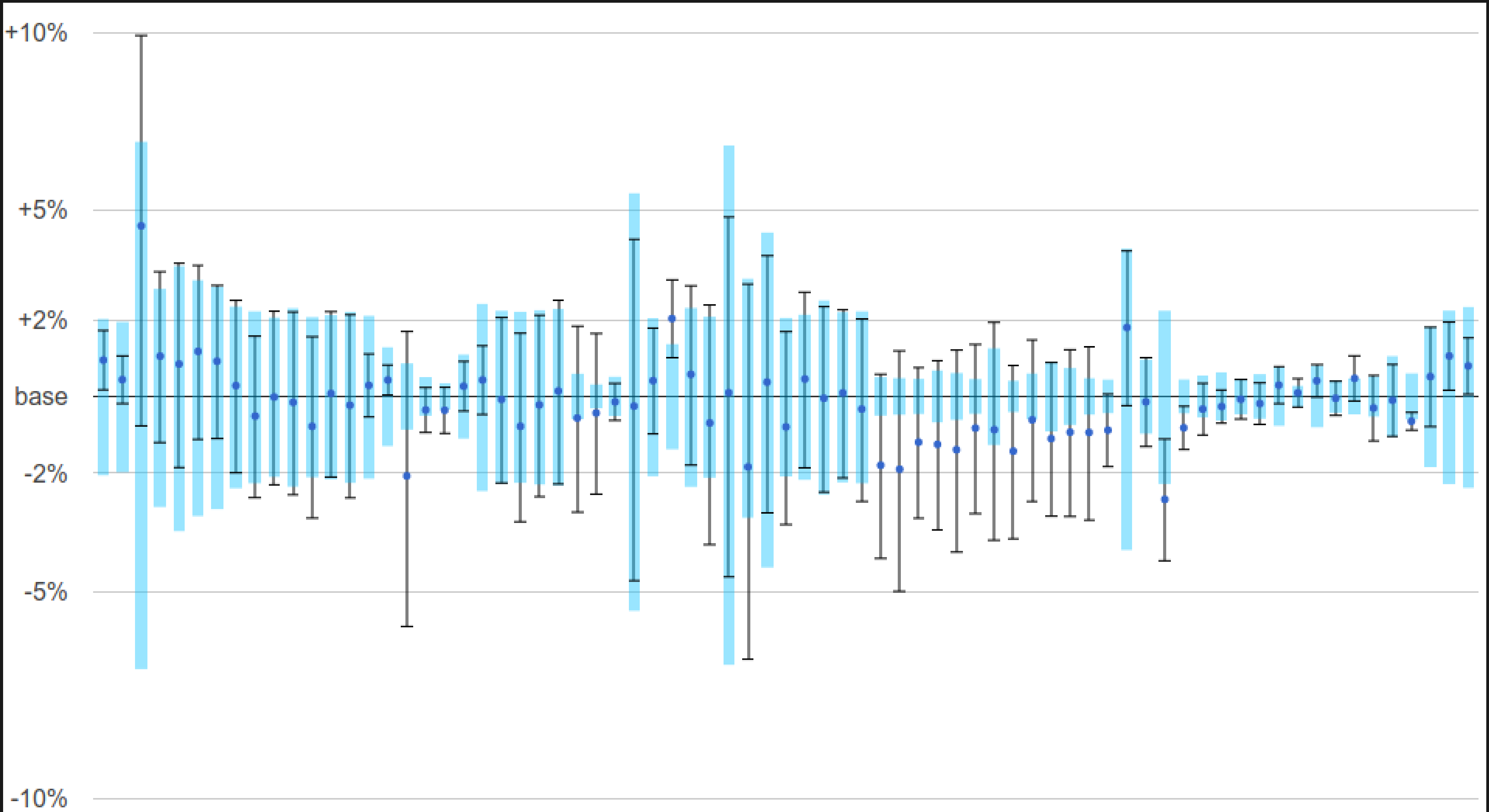
SOME THINGS WE TWEAKED

- Interference from other processes
- OS settings (e.g. ASLR)
- CPU frequency scaling
- Number of runs per benchmark
- Dry runs

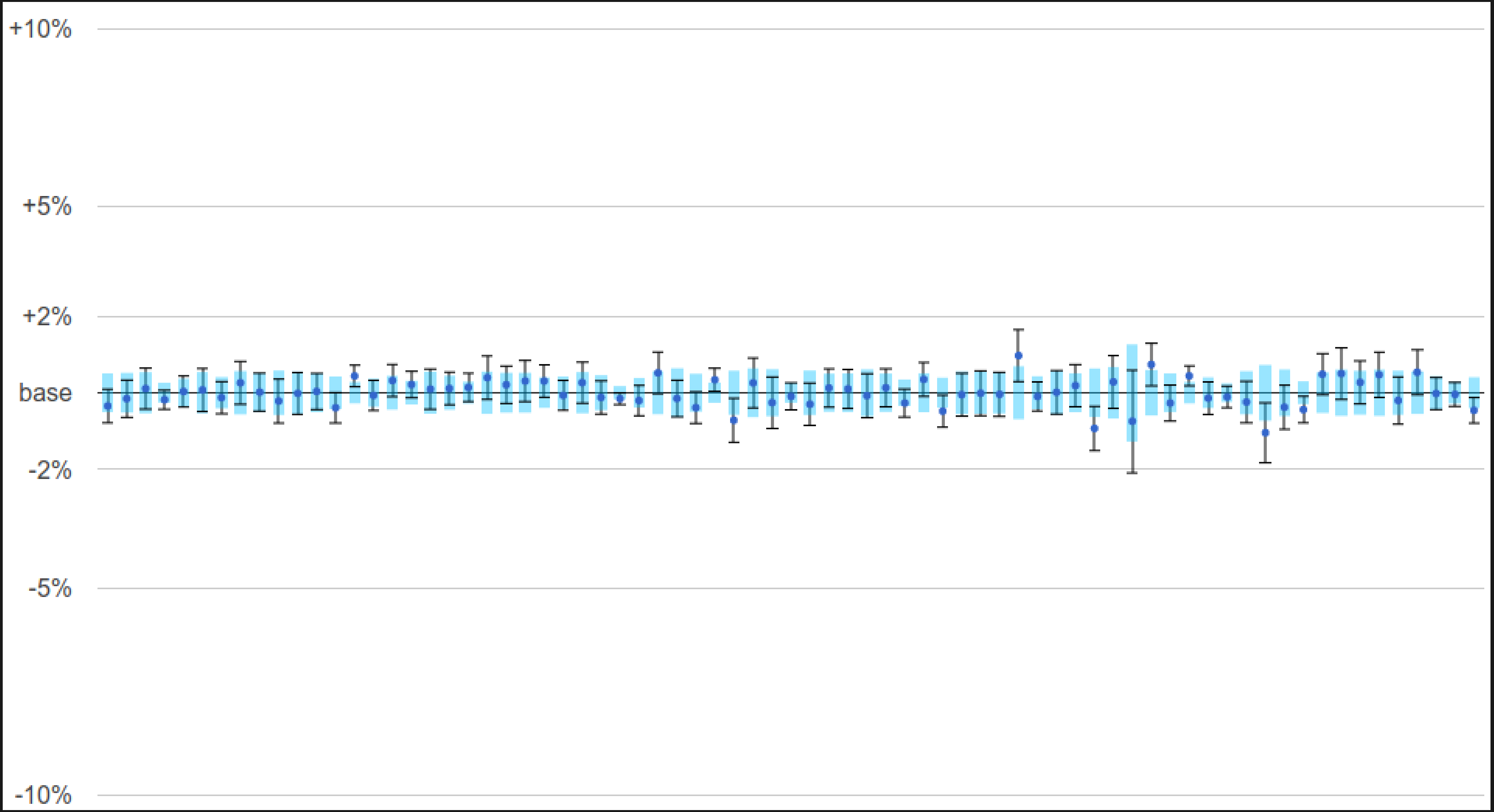
IF YOU CAN'T STABILIZE SOMETHING, RANDOMIZE IT

Example: relative run ordering of A/B variants

snappy benchmark, 10 runs



snappy benchmark, 10 runs



RESULT FILTERING AND VISUALIZATION

tens of benchmark binaries ×
multiple metrics per binary ×
several platforms ×
several build configs = a lot of results

SHIP IT!

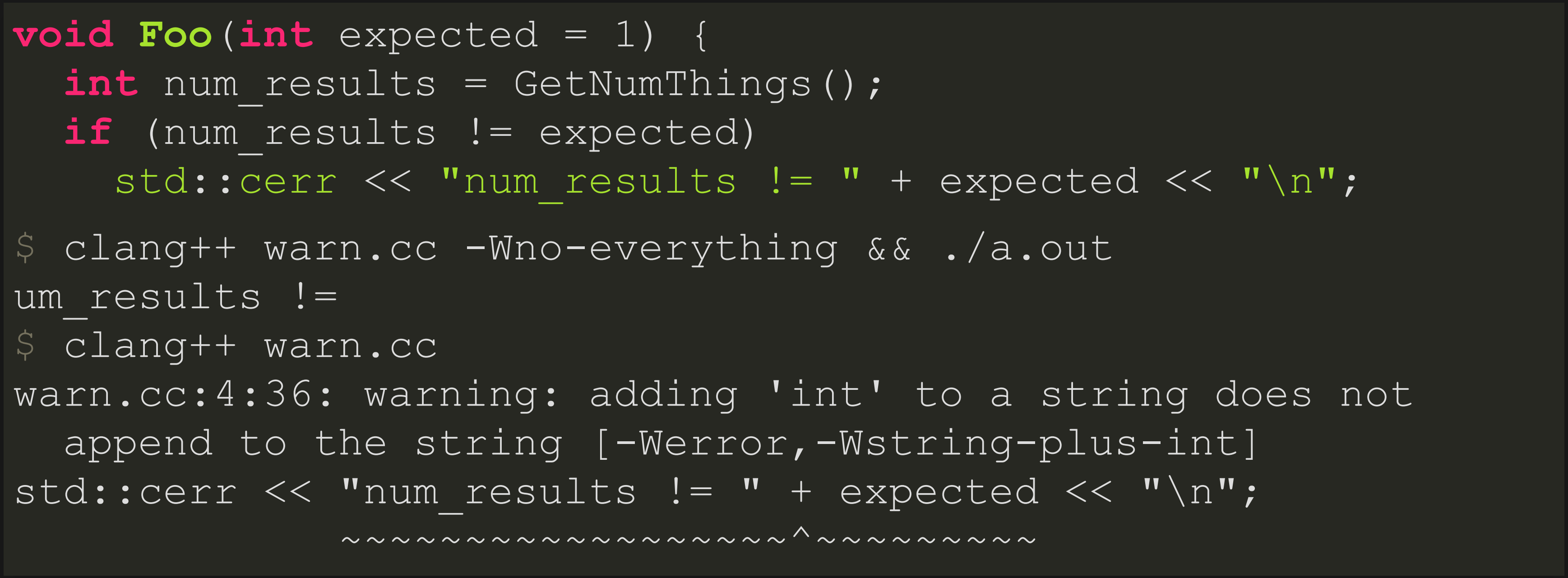
If tests pass and benchmarks don't regress, we can ship the release now

But that never happens

FAILURES AND REGRESSIONS

1. Deflake
2. Identify revision
3. Bad revision or just exposing a bug?

COMPILER DIAGNOSTICS



```
void Foo(int expected = 1) {
  int num_results = GetNumThings();
  if (num_results != expected)
    std::cerr << "num_results != " + expected << "\n";
```

```
$ clang++ warn.cc -Wno-everything && ./a.out
um_results !=
$ clang++ warn.cc
warn.cc:4:36: warning: adding 'int' to a string does not
  append to the string [-Werror,-Wstring-plus-int]
std::cerr << "num_results != " + expected << "\n";
             ~~~~~~~~~~~~~~~~~~^~~~~~~~~
```

COMPILER CRASHES

```
$ clang -O1 -c ...
clang: Assertion `Dead.count(Pred) && "All predecessors must
be dead!"' failed.
```

AUTOMATED REDUCTION

Open source tools like C-Reduce, llvm-reduce, or bugpoint produce minimal test cases

```c
int a, b, *c, d, e;
void f() {
  int g;
  for (;;) {
    for (; e; e = b) {
      c = g;
      for (; c; c = d)
        if (a) break;
      if (c) break;
    }
  }
}
```

QUALITY REGRESSIONS

- Object file/debug section bloat
- Compile time/memory usage regressions
- Debug info quality regressions

UNDEFINED BEHAVIOR

```
static void Test(const char *base, int offset) {
  printf("%p + %d => %s\n", base, offset,
         base + offset ? "true" : "false");
}
```

```
$ clang++ -O3 ub.cc && ./a.out
(nil) + 0 => false
0x100 + 0 => true
0x100 + 8 => true
(nil) + 8 => false 🤔


ubsan: ub.cc:3:15: runtime error: applying non-zero offset 8
    to null pointer
... but only with a patch: https://reviews.llvm.org/D67122
```

BRITTLE TESTS

Hyrum's Law:

> *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.*

... and toolchains have lots of implementation details!

BRITTLE TESTS

Implementation detail: ordering

```
std::unordered_set<int> ComputeSet() { ... }
```

```
TEST(Widget, ComputeSet) {
  std::cout << "Values are: " << ComputeSet() << "\n";
```

```
$ ./WidgetTest
Values are: {2, 3, 1}
```

```
TEST(Widget, ComputeSet) {                                          ✘
  EXPECT_THAT(ComputeSet(), ElementsAre(2, 3, 1));
```

```
TEST(Widget, ComputeSet) {                                          ✔
  EXPECT_THAT(ComputeSet(), UnorderedElementsAre(1, 2, 3));
```

BRITTLE TESTS

Implementation detail: "precision" of `-ffast-math`

```
double MagicNumber() { /* ... lots of math ... */ }
```

```
TEST(Foo, MagicNumber) {
  std::cout << "Magic number: " << MagicNumber() << "\n";
```

```
$ ./FooTest
Magic number: 5.270125784910
```

```
TEST(Foo, MagicNumber) {                                          ✗
  EXPECT_EQ(MagicNumber(), 5.270125784910);
```

```
TEST(Foo, MagicNumber) {                                          ✓
  EXPECT_THAT(MagicNumber(), FloatNear(5.27, 0.1))
```

MISCOMPILES

¯_(ツ)_/¯

MISCOMPILES

- Discuss on revision review thread
- Identify the affected object file
- Compare generated assembly/IR to find the code being miscompiled

PERFORMANCE REGRESSIONS

- Rerun to make sure it's reproducible
- Bisect using a suitable threshold
- Talk to patch authors /
  hand off to our performance team to investigate
- Revert or fix

REPRODUCIBLE NOISE

Sometimes we see things that look like regressions, and survive increased iteration counts

PROFILE COLLECTION NOISE

PGO (profile-guided optimization) builds:

- Build an initial version (possibly instrumented)
- Run initial version, collect a profile
- Build again using the profile for optimization hints
- Run again, measure performance

PROFILE COLLECTION NOISE

Problem:

- Profiles are not compatible across revisions
- Each side of the A/B test has its own profile
  ⟶ different optimization decisions for each binary

Solution:

- Longer profiling runs so random sampling converges
- or do everything again, get all new A/B profiles
  ⟶ average out the bias

ALIGNMENT-SENSITIVE CODE

Problem:

- Instruction alignment can have performance implications
- The compiler doesn't have a good model for this
- Same good/bad alignment even with rebuilds

Solution:

- ¯_(ツ)_/¯
- In extreme cases, manually align hot code

SHIP IT!

Once all the blockers are solved, we can finally ship a release

NOT AS HARD AS IT SEEMS!

- Release cop is a full time job
- Normal hours; no oncall
- In a normal week:
  - 1 to 2 root causes accounting for most test failures
  - some UB in existing code caught by sanitizers

HOW CAN I DO THIS?

Key components:

- Test the universe
- … and then fix things
- Identify culprits & reduce failures
- Feedback loop with community/toolchain vendor
- Optional: deal with scale

RECAP

- Toolchains are just software
- Good tests and benchmarks are the foundation
- Frequent cadence & shorter feedback cycle saves in the long run

QUESTIONS?

Jorge Gorbe Moya

jgorbe@google.com

@slackito

Jordan Rupprecht

rupprecht@google.com

@jrupees