

accu
conference
2025

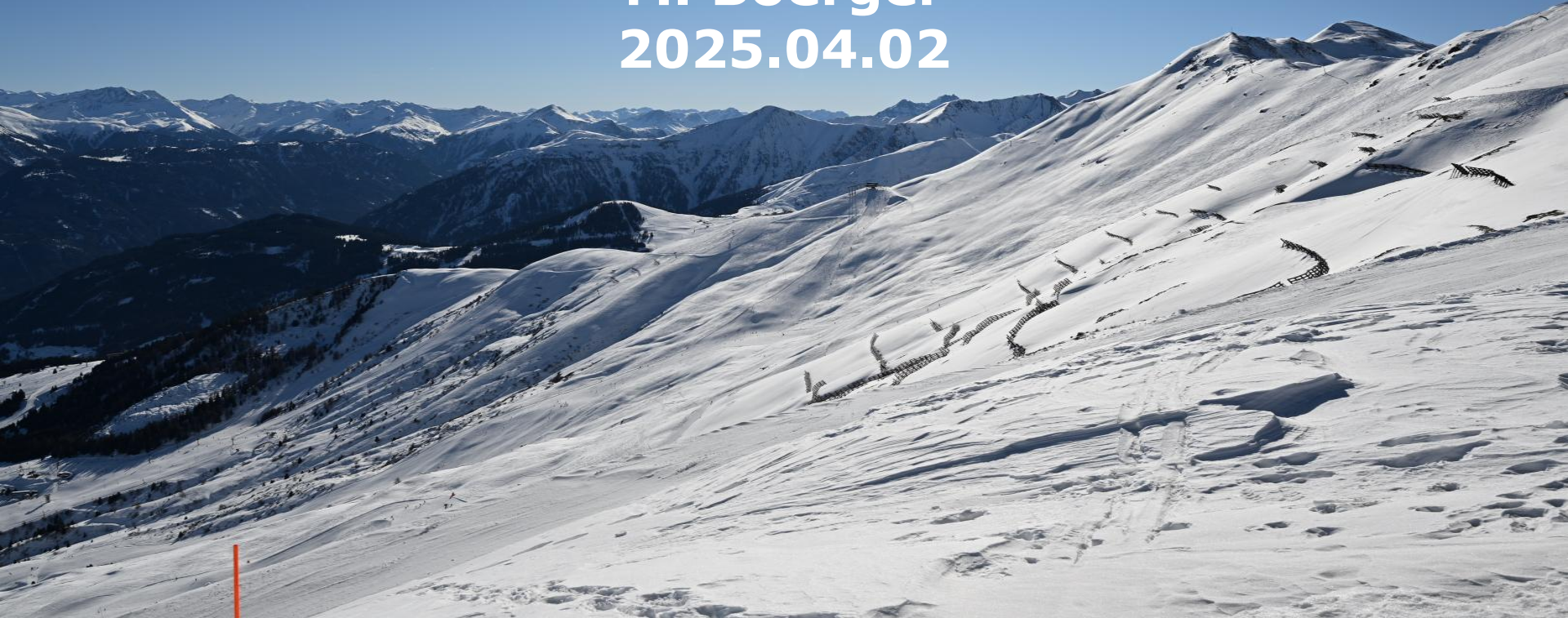
Bazel

Marcus Boerger



So, Bazel ?!

M. Boerger
2025.04.02



So, Bazel ?!

- What is Bazel
- Why Bazel?
- How to get started.
- Tips and Tricks.
- Tools

NOTE: I have not done autotools, cmake or anything beyond Bazel in > 10 years!



What is Bazel

Fast

Extensible

Reproducible

Maintainable

**Annoyingly
Different**

**Incredibly
Complex**

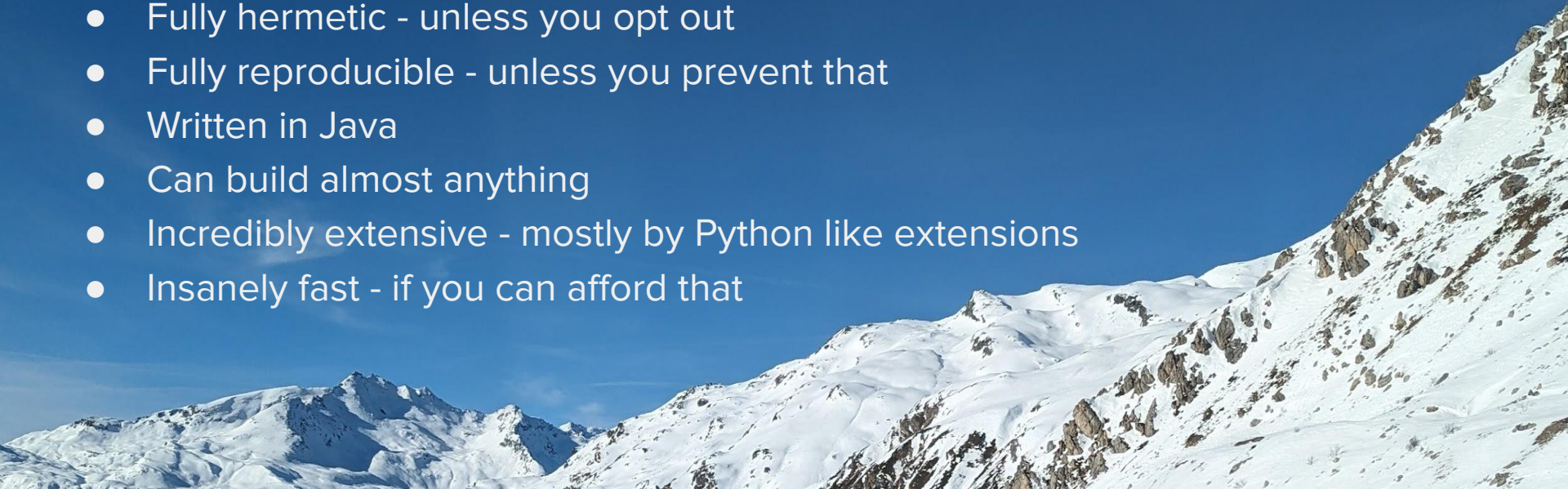
What is Bazel

Blaze is Google's latest gen build system in Emperor's ^G OpenSource clothing.

Google went through the dark ages: CC-Cache, Mach, Forge and then Blaze.

Blaze known as Bazel outside G is:

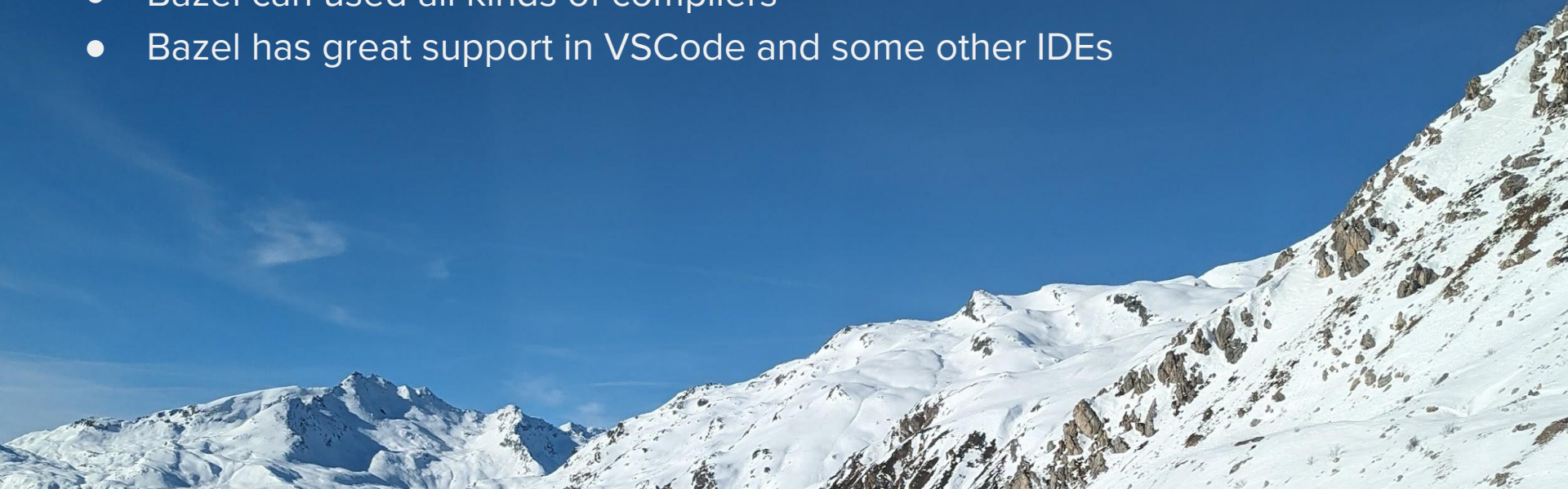
- Fully hermetic - unless you opt out
- Fully reproducible - unless you prevent that
- Written in Java
- Can build almost anything
- Incredibly extensive - mostly by Python like extensions
- Insanely fast - if you can afford that



Why Bazel

Why Bazel

- Bazel is widely used at some of the biggest industry players
- Bazel can build your stuff with hundreds of cores in parallel
- Bazel is primed for C++, Python, Java and Go
- Bazel has full support for cross compilation
- Bazel can used all kinds of compilers
- Bazel has great support in VSCode and some other IDEs



How did Google get there

No Google engineer wanted to use Autotools or CMake

CC-cache was, well slow

Mach was faster...

Most Google engineers knew Python

Next was Forge

Then came Blaze



Before we get started

Bazel has two modes:

- OLD: Workspaces

The old stuff is sometimes simpler for small projects or massive mono repos...

- NEW: Bazelmod

The new stuff is simply the future!



0. Theories

**Graphing &
Hashing**

What's at the core?

foo.cpp

foo.h

bar.cpp

bar.h

Let's compile this

What's at the core?

foo.cpp

foo.h

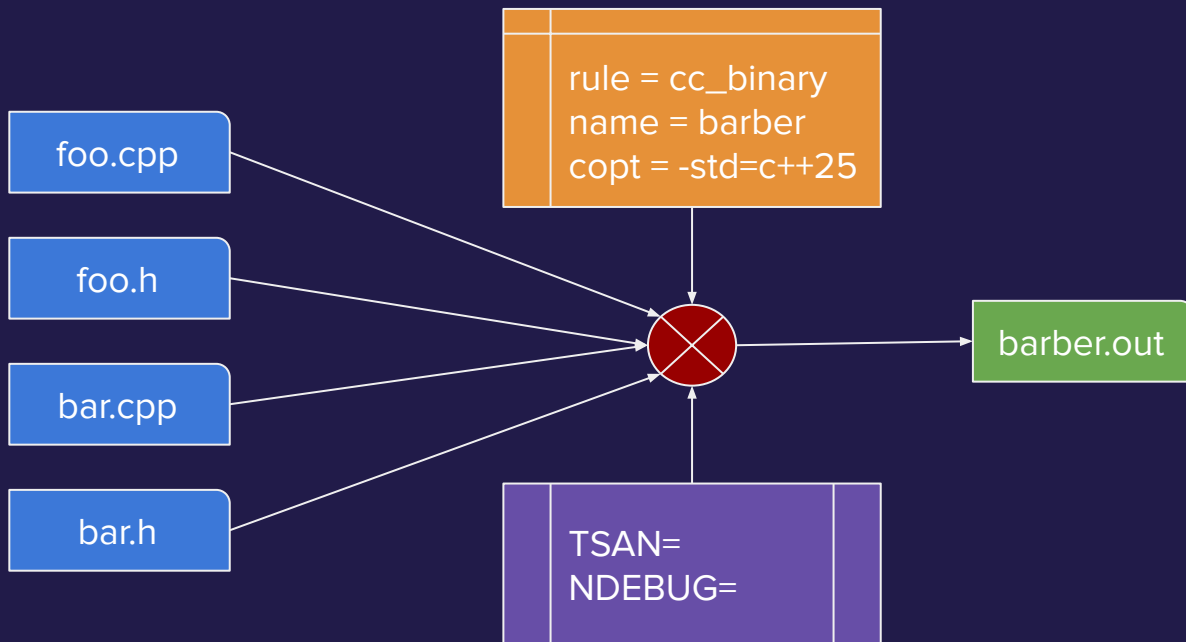
bar.cpp

bar.h

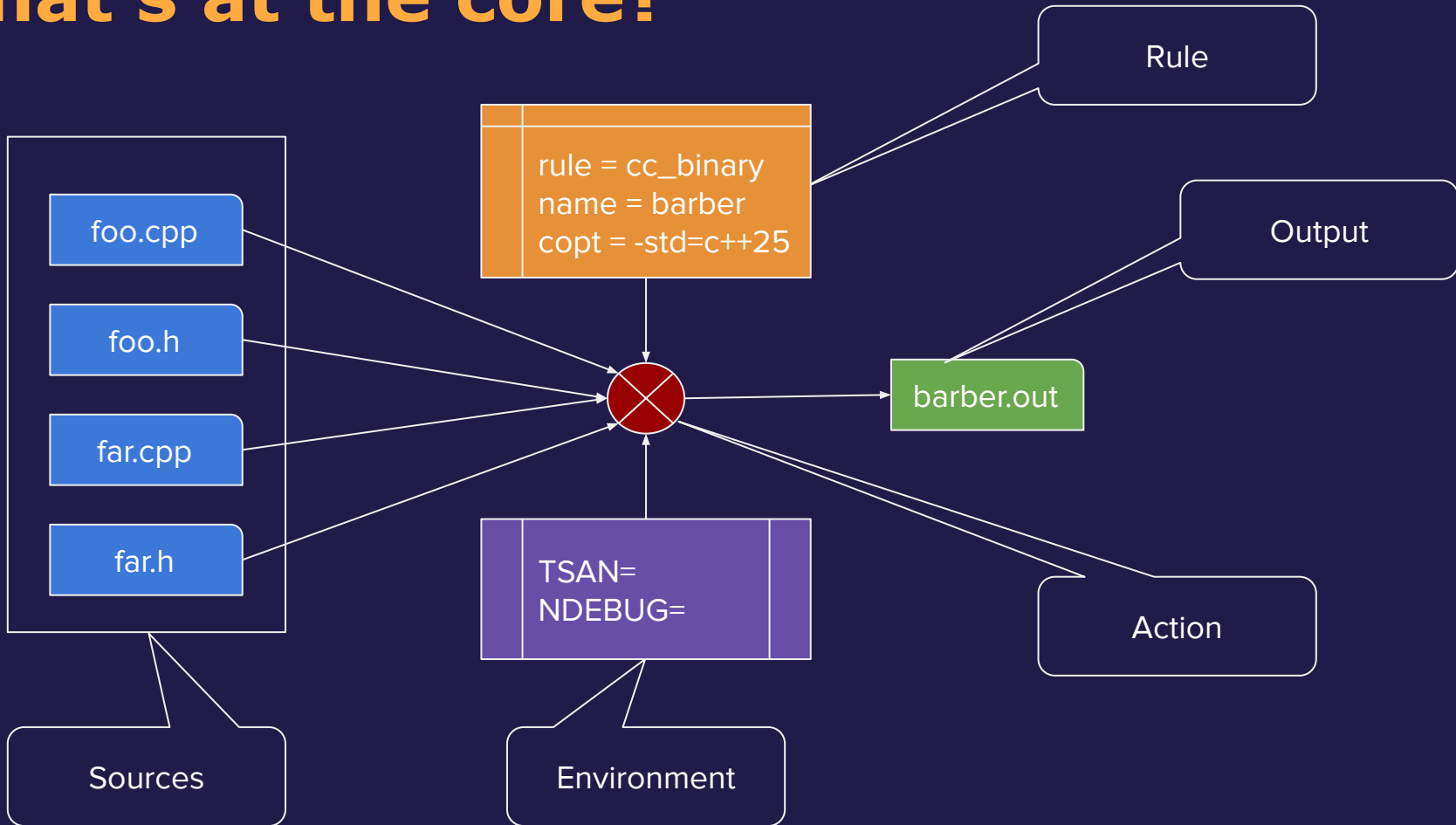
Let's compile this

In a controlled
environment

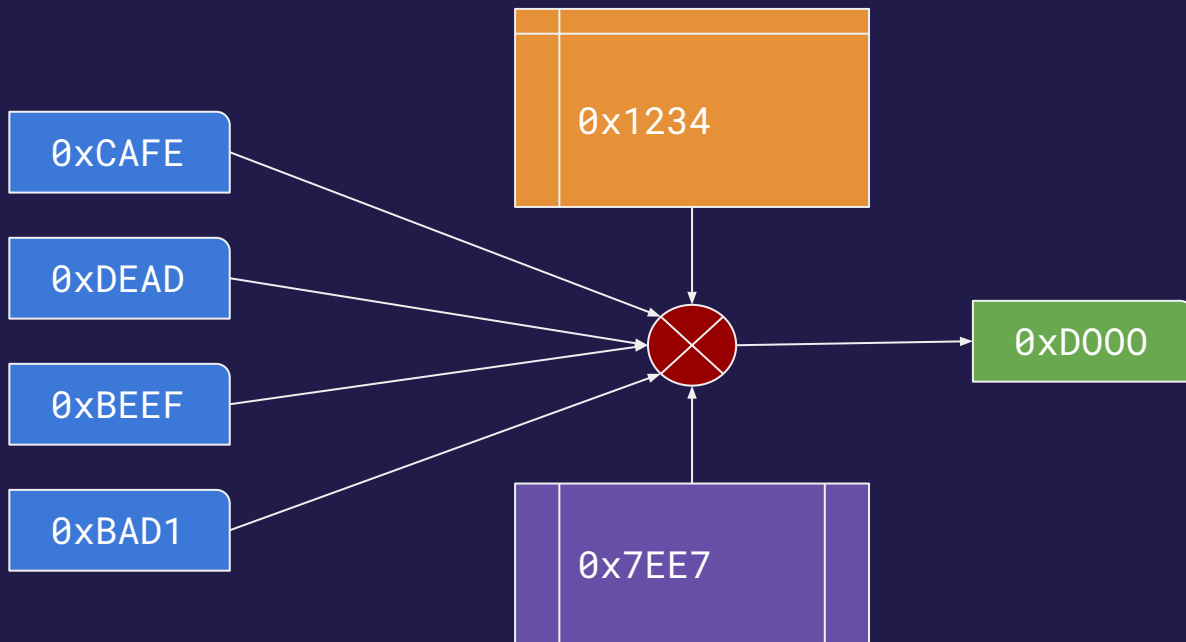
What's at the core?



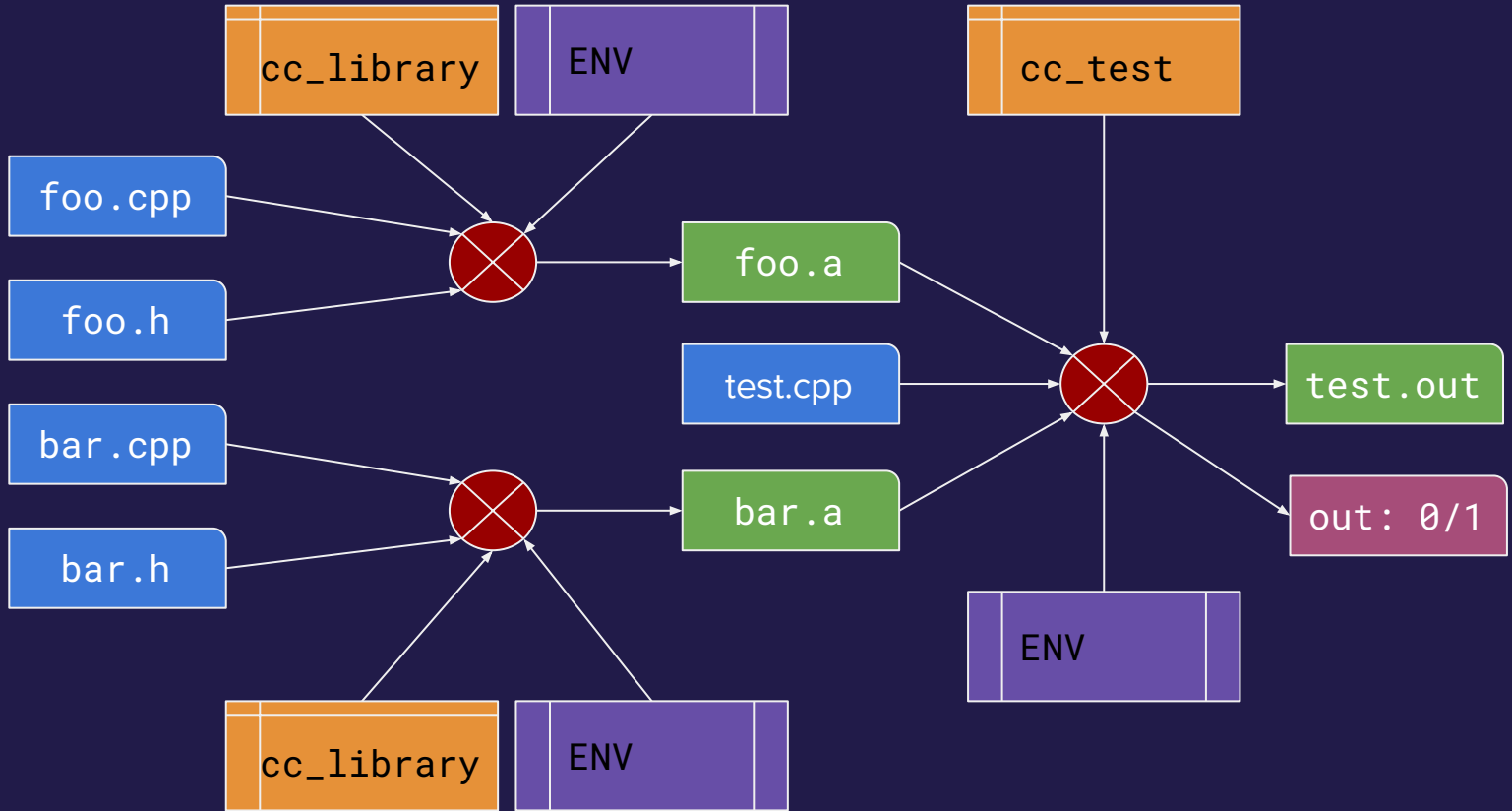
What's at the core?



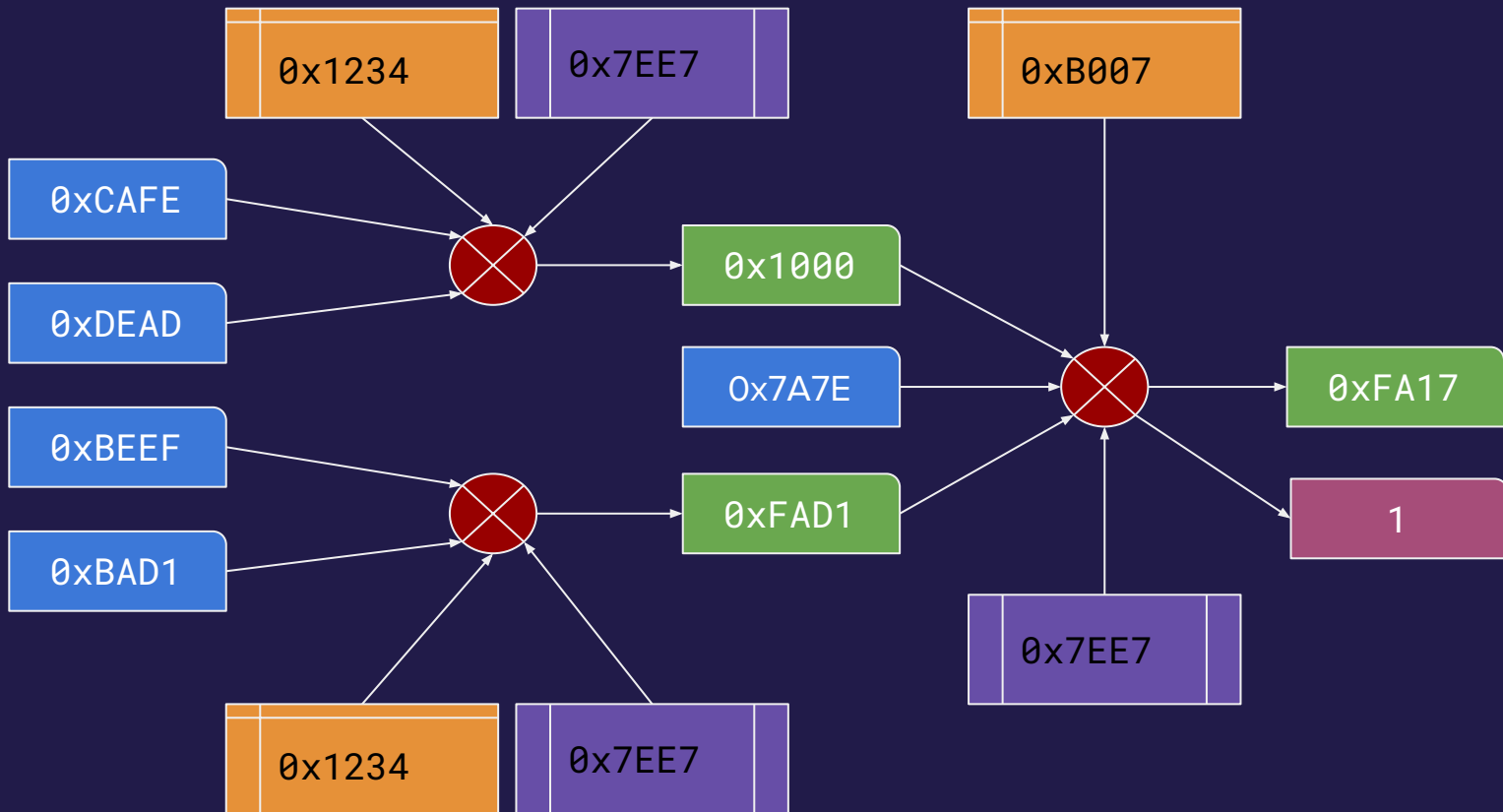
What's at the core?



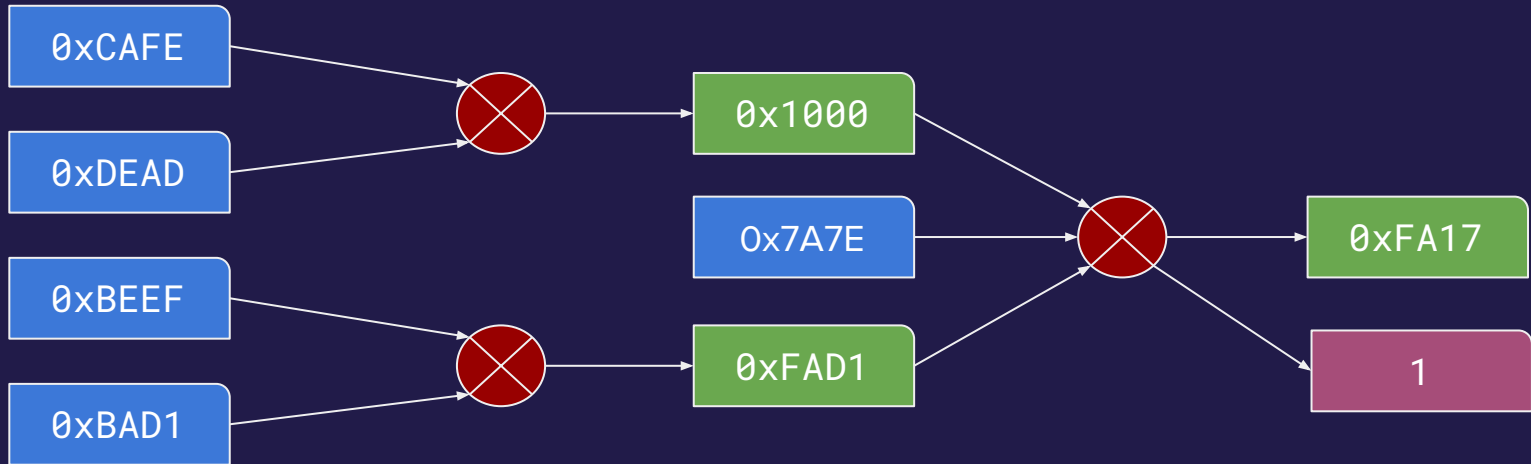
A Graph



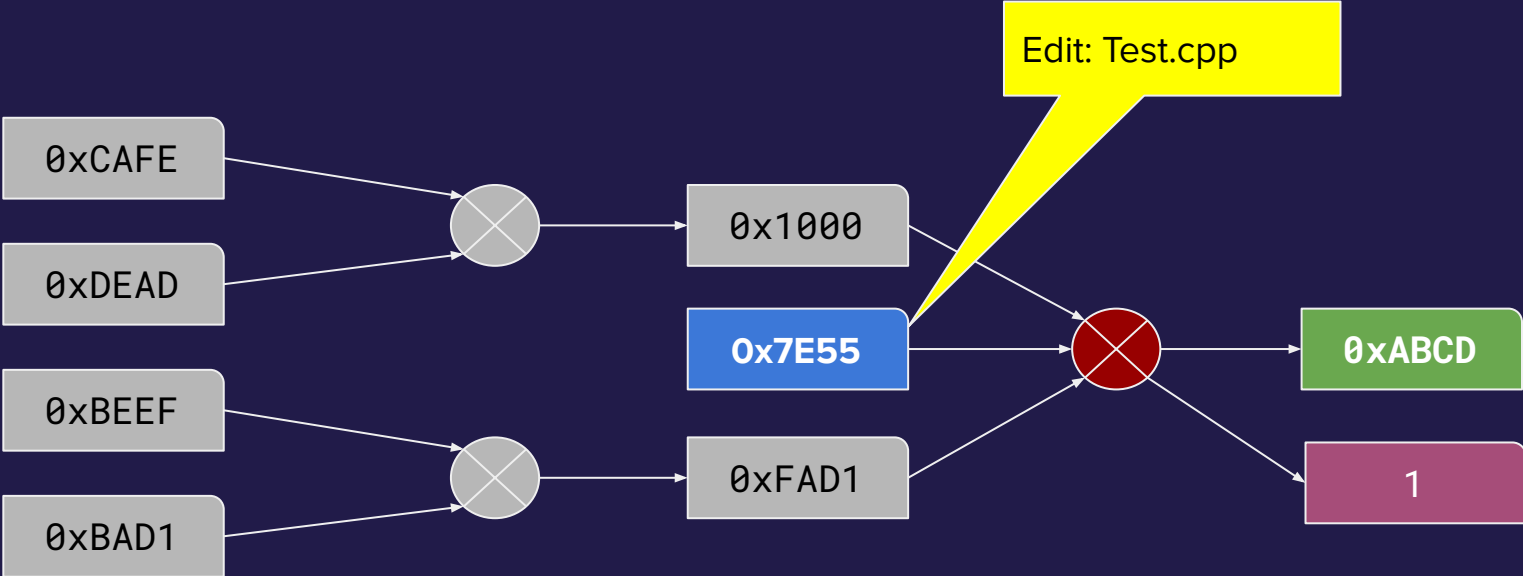
bazel test //...



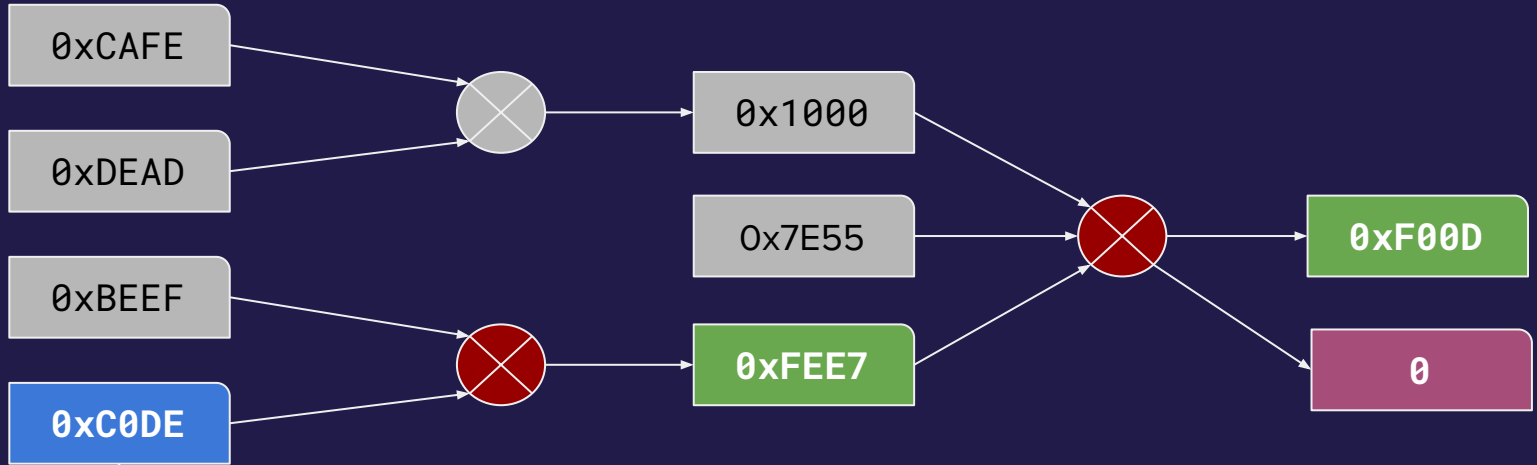
Illustrate with logical const removed



Changes

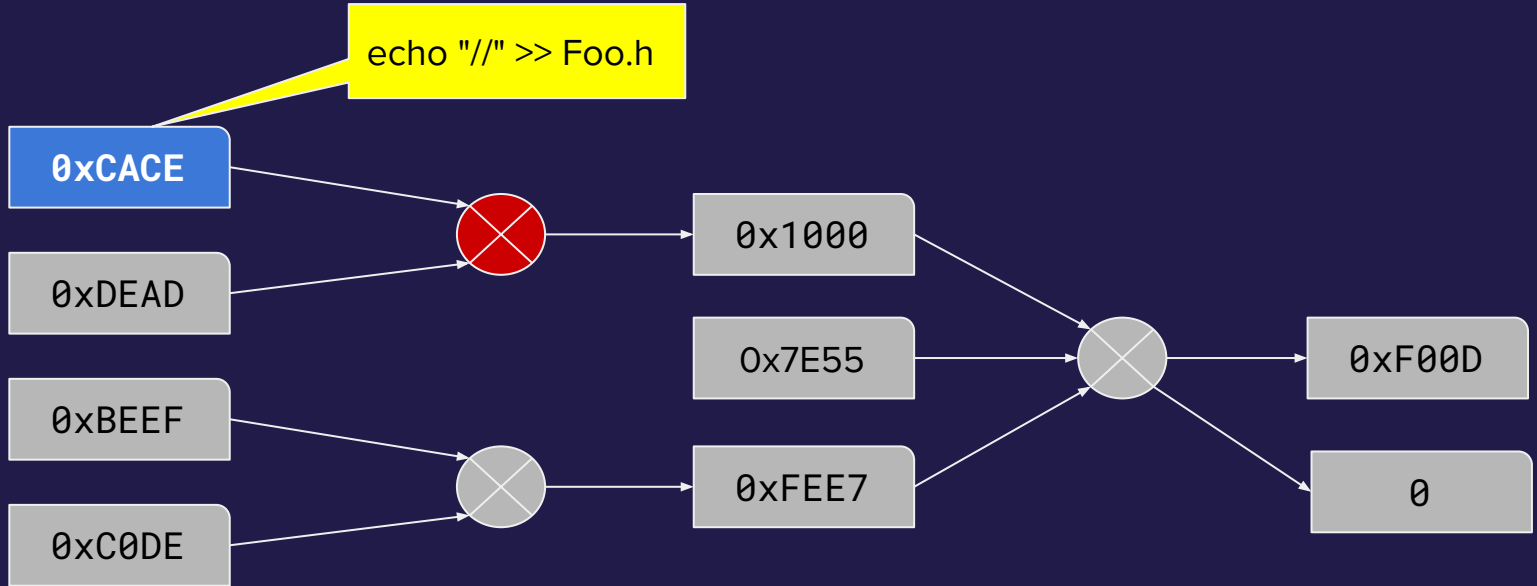


Edit a source

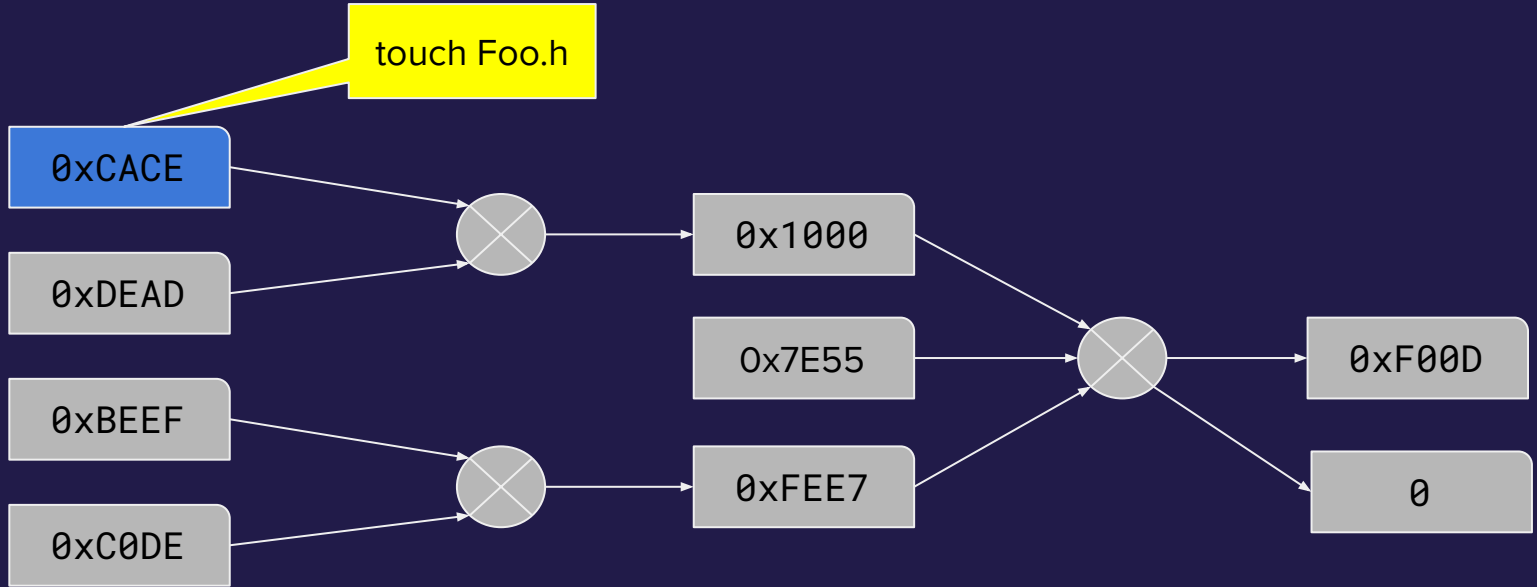


Edit: Bar.h

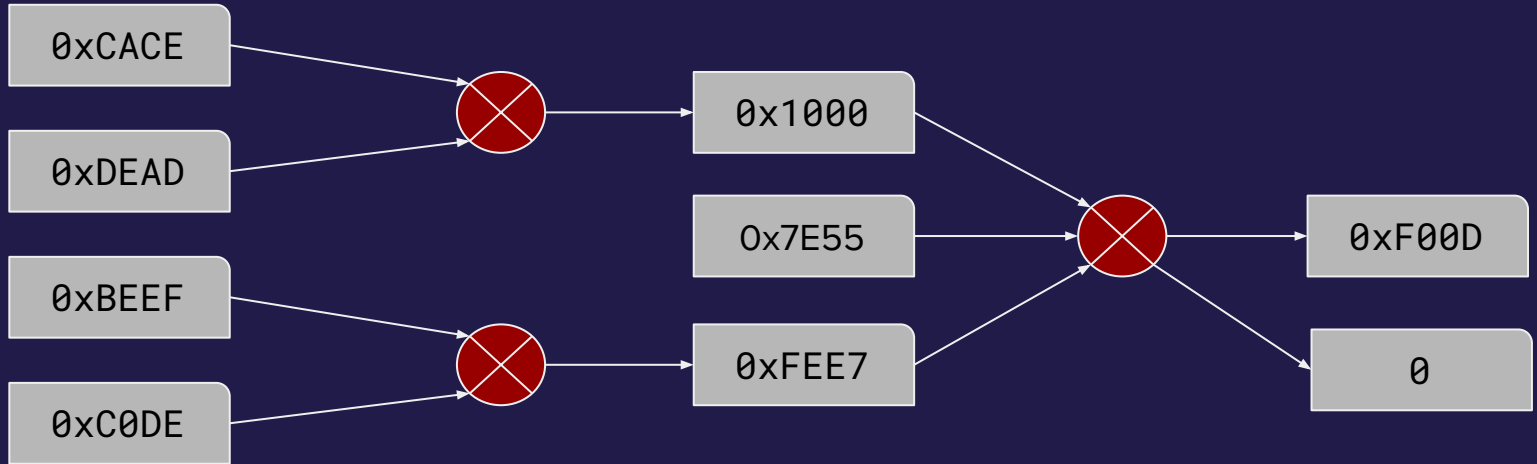
Edit a comment



touch



bazel clean --async ; bazel build //...



1. Practical Bazel

**Bazel & C++
Rule Attributes**

1.1. Bazel & C++

How to not be confused all the time

Simple guidelines to make everyone's life easier

Who's there?

- To write a rule you need to declare your inputs, output and what you want to do.
- Bazel works in rules that transform sources.
 - `cc_library` C++ library
 - `cc_test` C++ test
 - `cc_binary` C++ binary

 - `py_library` Python library
 - `py_test` Python test
 - `py_binary` Python binary

 - `sh_library` Shell library
 - `sh_test` Shell test
 - `sh_binary` Shell binary

 - `genrule` Generates files which may become executable

A typical BUILD file

```
cc_library(  
  name = "barber",  
  srcs = [  
    "bar.cpp",  
    "foo.cpp"  
  ],  
  hdrs = [  
    "bar.h",  
    "foo.h"  
  ],  
)  
  
cc_test(  
  name = "func_test",  
  srcs = ["func-tester.cpp"],  
  deps = [  
    ":barber",  
    "@com_google_googletest//:gtest",  
  ],  
)
```

A typical BUILD file!

A typical BUILD file

```
cc_library(  
  name = "barber",  
  srcs = [  
    "bar.cpp",  
    "foo.cpp"  
  ],  
  hdrs = [  
    "bar.h",  
    "foo.h"  
  ],  
)  
  
cc_test(  
  name = "func_test",  
  srcs = ["func-tester.cpp"],  
  deps = [  
    ":barber",  
    "@com_google_googletest//:gtest",  
  ],  
)
```

A typical BUILD file!

A C++ library called "barber".

A C++ test called "func_test".

A typical BUILD file

```
cc_library(  
  name = "barber",  
  srcs = [  
    "bar.cpp",  
    "foo.cpp"  
  ],  
  hdrs = [  
    "bar.h",  
    "foo.h"  
  ],  
)  
  
cc_test(  
  name = "func_test",  
  srcs = ["func-tester.cpp"],  
  deps = [  
    ":barber",  
    "@com_google_googletest//:gtest",  
  ],  
)
```

A typical BUILD file!

The library has sources and headers:

- bar.cpp
- bar.h
- foo.cpp
- foo.h

The test has a source file:

- func-tester.cpp

A typical BUILD file

```
cc_library(  
  name = "barber",  
  srcs = [  
    "bar.cpp",  
    "foo.cpp"  
  ],  
  hdrs = [  
    "bar.h",  
    "foo.h"  
  ],  
)  
  
cc_test(  
  name = "func_test",  
  srcs = ["func-tester.cpp"],  
  deps = [  
    ":barber",  
    "@com_google_googletest//:gtest",  
  ],  
)
```

A typical BUILD file!

The library has sources and headers:

- bar.cpp
- bar.h
- foo.cpp
- foo.h

The test has a source file:

- func-tester.cpp

The test depends on the internal Library:

- func_test -> barber

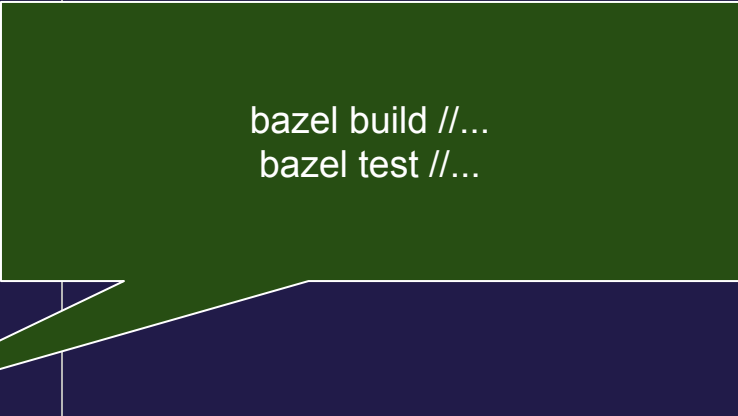
The test also has an external dependency:

- GoogleTest

Now what?

```
cc_library(  
  name = "barber",  
  srcs = [  
    "bar.cpp",  
    "foo.cpp"  
  ],  
  hdrs = [  
    "bar.h",  
    "foo.h"  
  ],  
)  
  
cc_test(  
  name = "func_test",  
  srcs = ["func-tester.cpp"],  
  deps = [  
    ":barber",  
    "@com_google_googletest//:gtest",  
  ],  
)
```

A typical BUILD file!



bazel build //...
bazel test //...

How many cc_library rules?

```
cc_library(  
  name = "barber",  
  srcs = [  
    "bar.cpp",  
    "foo.cpp"  
  ],  
  hdrs = [  
    "bar.h",  
    "foo.h"  
  ],  
)
```

A typical BUILD file!

This is what (lazy) people always end up.
And then they want magic tools.

```
cc_test(  
  name = "func_test",  
  srcs = ["func-tester.cpp"],  
  deps = [  
    ":barber",  
    "@com_google_googletest//:gtest",  
  ],  
)
```

How many cc_library rules?

```
cc_library(  
  name = "barber",  
  srcs = glob(  
    include = ["*.cpp"],  
    exclude = ["func-tester.cpp"],  
  ),  
  hdrs = glob(["*.h"]),  
)
```

A typical BUILD file!

This is what we all (fatally) dream of.
Well, before tooling.
And before AI.

```
cc_test(  
  name = "func_test",  
  srcs = ["func-tester.cpp"],  
  deps = [  
    ":barber",  
    "@com_google_googletest//:gtest",  
  ],  
)
```

How many cc_library rules?

```
cc_library(  
  name = "barber",  
  srcs = [  
    "bar.cpp",  
    "foo.cpp"  
  ],  
  hdrs = [  
    "bar.h",  
    "foo.h"  
  ],  
)  
  
cc_test(  
  name = "func_test",  
  srcs = ["func-tester.cpp"],  
  deps = [  
    ":barber",  
    "@com_google_googletest//:gtest",  
  ],  
)
```

A typical **convoluted** rule!

No discernible structure:

- Makes it needlessly hard to find rules or sources, because there is no relationship between them.
- Makes Bazel's life needlessly hard.
- Humans cannot look up anything.
- Instead we must always use search tools.

How many `cc_library` rules?

```
cc_library(  
  name = "bar",  
  srcs = ["bar.cpp"],  
  hdrs = ["bar.h"],  
)
```

```
cc_library(  
  name = "foo",  
  srcs = ["foo.cpp"],  
  hdrs = ["foo.h"],  
  deps = [":bar"],  
)
```

Use exactly **1 `cc_library`** rule for exactly **1 source file**.

Attributes `srcs` and `hdrs` use a **single line**.

Sources (.cpp):

- may only be used in a single rule.

Headers (.h):

- should only be used in a single rule.
- may not be used in **`textual_hdrs`**.
- may export other headers from other rules.

How many cc_library rules?

```
cc_library(  
  name = "bar",  
  srcs = ["bar.cpp"],  
  hdrs = ["bar.h"],  
)  
  
cc_library(  
  name = "foo",  
  srcs = ["foo.cpp"],  
  hdrs = ["foo.h"],  
  deps = [":bar"],  
)  
  
cc_test(  
  name = "foo_test",  
  srcs = ["foo_test.cpp"],  
  deps = [  
    ":foo",  
    "@com_google_googletest//:gtest",  
  ],  
)
```

Use exactly **1 cc_library** rule for exactly **1 source file**.

Attributes **srcs** and **hdrs** use a **single line**.

Sources (.cpp):

- may only be used in a single rule.

Headers (.h):

- should only be used in a single rule.
- may not be used in **textual_hdrs**.
- may export other headers from other rules.

Make test names follow library names.

How many cc_library rules?

```
cc_library(  
  name = "bar",  
  srcs = ["bar.cpp"],  
  hdrs = ["bar.h"],  
)  
  
cc_library(  
  name = "foo",  
  srcs = ["foo.cpp"],  
  hdrs = ["foo.h"],  
  deps = [":bar"],  
)  
  
cc_test(  
  name = "bar_test",  
  srcs = ["bar_test.cpp"],  
  deps = [":bar", ...],  
)  
  
cc_test(  
  name = "foo_test",  
  srcs = ["foo_test.cpp"],  
  deps = [":foo", ...],  
)
```

A cleaner structure.

How many cc_library rules?

```
cc_library(  
  name = "bar",  
  srcs = ["bar.cpp"],  
  hdrs = ["bar.h"],  
)  
  
cc_library(  
  name = "foo",  
  srcs = ["foo.cpp"],  
  hdrs = ["foo.h"],  
  deps = [":bar"],  
)  
  
cc_test(  
  name = "bar_test",  
  srcs = ["bar_test.cpp"],  
  deps = [":bar", ...],  
)  
  
cc_test(  
  name = "foo_test",  
  srcs = ["foo_test.cpp"],  
  deps = [":foo", ...],  
)
```

The rule that shares the path name is the default rule.

```
bazel query //dir:*
```

- Shows the available rules in dir/BUILD.

```
bazel build //dir:bar
```

- Only bar.

```
bazel build //dir:foo
```

- Only foo.

```
bazel build //dir
```

- Only "dir" if it exists.

```
bazel build //dir:*
```

- All (non manual) in dir.

```
bazel build //dir/...
```

- All (non manual) in dir and sub-directories.

More on Bazel Query in part 2.

How many cc_library rules?

```
cc_library(  
  name = "bar",  
  srcs = ["bar.cpp"],  
  hdrs = ["bar.h"],  
  visibility = ["//visibility:private"],  
)
```

```
cc_library(  
  name = "foo",  
  srcs = ["foo.cpp"],  
  hdrs = ["foo.h"],  
  deps = [":bar"],  
  visibility = ["//visibility:public"],  
)
```

Make rules private, if possible.

Do not repeat headers if rules are private.

Instead export the headers using IWYU annotations.

How many cc_library rules?

```
package(default_visibility = [  
    "//visibility:private",  
])  
  
cc_library(  
    name = "bar",  
    srcs = ["bar.cpp"],  
    hdrs = ["bar.h"],  
)  
  
cc_library(  
    name = "foo",  
    srcs = ["foo.cpp"],  
    hdrs = ["foo.h"],  
    deps = [":bar"],  
    visibility = ["//visibility:public"],  
)
```

Make rules private, if possible.

Do not repeat headers if rules are private.

Instead export the headers using IWYU annotations.

Use package / default_visibility.

How many cc_library rules?

dir/BUILD

```
package(default_visibility = [
    "//visibility:private",
])

cc_library(
    name = "bar",
    srcs = ["bar.cpp"],
    hdrs = ["bar.h"],
)

cc_library(
    name = "foo",
    srcs = ["foo.cpp"],
    hdrs = ["foo.h"],
    deps = [":bar"],
    visibility = ["//visibility:public"],
)
```

dir/bar.h

```
#pragma once

#IWYU pragma: private, include "dir/foo.h"
#IWYU pragma: friend dir/*
```

dir/foo.h

```
#pragma once

#include "dir/bar.h" // IWYU pragma: export
```

How many cc_library rules?

dir/BUILD

```
package(default_visibility = [
    "//visibility:private",
])

cc_library(
    name = "bar",
    srcs = ["bar.cpp"],
    hdrs = ["bar.h"],
)

cc_library(
    name = "foo",
    srcs = ["foo.cpp"],
    hdrs = ["foo.h"],
    deps = [":bar"],
)

cc_library(
    name = "dir",
    hdrs = ["dir.h"],
    deps = [":bar", ":foo"],
    visibility = ["//visibility:public"],
)
```

dir/bar.h

```
#pragma once

#IWYU pragma: private, include "dir/dir.h"
#IWYU pragma: friend dir/*
```

dir/foo.h

```
#pragma once

#IWYU pragma: private, include "dir/dir.h"
#IWYU pragma: friend dir/*

#include "dir/bar.h" // IWYU pragma: export
```

dir/dir.h

```
#pragma once

#include "dir/bar.h" // IWYU pragma: keep
#include "dir/foo.h" // IWYU pragma: export
```

General Structure

```
"""Title/Docstring"""  
  
# Comments...  
  
load(...)  
  
package(  
    default_visibility = [  
        "//visibility:private",  
    ],  
    features = [  
        "layering_check",  
        "parse_headers",  
    ],  
)  
  
# rules
```

If you can: Add a title and file comment.

All `load` statements must be at the top of the file just after the title/docstring.

Next provide package defaults:

- Rules should be private by default
- Enable `layering_check`:
 - Ensures dependency correctness
 - Goes hand in hand with IWYU
- Enable `parse_headers`:
 - Make bazel parse headers even if it otherwise would not to force correctness.

CC Rules

```
cc_library(  
  name = "foo",  
  srcs = ["foo.cpp"],  
  hdrs = ["foo.h"],  
)
```

Per C++ source file there should be:

- ONE header.
- ONE cc_library rule.
- All names should match up.
- Not matching makes maintenance hard.

Preferably:

- A header should only have one class.
- All header code in a namespace.

CC Rules

```
cc_library(  
  name = "foo",  
  srcs = ["foo.cpp"],  
  hdrs = ["foo.h"],  
  deps = [  
    ":internal",  
    "//workspace",  
    "@external",  
  ],  
)
```

Per C++ source file there should be:

- ONE header.
- ONE cc_library rule.
- All names should match up.
- Not matching makes maintenance hard.

Preferably:

- A header should only have one class.
- All header code in a namespace.

Dependencies:

- Internal dependencies start with `:`
- Absolute repo deps start with `//`
- External Repos start with `@`

CC Rules

```
cc_library(  
  name = "foo",  
  srcs = ["foo.cpp"],  
  hdrs = ["foo.h"],  
)  
  
cc_test(  
  name = "foo_test",  
  srcs = ["foo_test.cpp"],  
  deps = [  
    ":foo",  
    "@com_google_googletest//:gtest",  
  ],  
)
```

Per C++ source file there should be:

- ONE header.
- ONE cc_library rule.
- All names should match up.
- Not matching makes maintenance hard.

Preferably:

- A header should only have one class.
- All header code in a namespace.

Each cc_library rule needs a test:

- All test names should end in "_test".

CC Rules

```
cc_library(  
  name = "foo_cc",  
  srcs = ["foo.cpp"],  
  hdrs = ["foo.h"],  
)  
  
cc_test(  
  name = "foo_test",  
  srcs = ["foo_test.cpp"],  
  deps = [":foo_cc", ...],  
)  
  
cc_binary(  
  name = "foo",  
  srcs = ["foo_main.cpp"],  
  deps = [":foo_cc", ...],  
)
```

Per C++ source file there should be:

- ONE header.
- ONE cc_library rule.
- All names should match up.
- Not matching makes maintenance hard.

Preferably:

- A header should only have one class.
- All header code in a namespace.

Each cc_library rule needs a test:

- All test names should end in "_test".

Binary rules:

- Source file names end in "_main.cpp".
- If a library shares the binary name:
 - Add "_cc" to the library rule, or
 - Move one rule to another dir.

CC Rules

```
cc_library(  
  name = "foo_cc",  
  srcs = ["foo.cpp"],  
  hdrs = ["foo.h"],  
)
```

```
cc_test(  
  name = "foo_test",  
  srcs = ["foo_test.cpp"],  
  deps = [":foo_cc", ...],  
)
```

```
cc_binary(  
  name = "foo",  
  srcs = ["foo_main.cpp"],  
  deps = [":foo_cc", ...],  
)
```

```
sh_test(  
  name = "foo_sh_test",  
  srcs = ["foo_sh_test.sh"],  
  data = [":foo"],  
)
```

Per C++ source file there should be:

- ONE header.
- ONE cc_library rule.
- All names should match up.
- Not matching makes maintenance hard.

Preferably:

- A header should only have one class.
- All header code in a namespace.

Each cc_library rule needs a test:

- All test names should end in "_test".

Binary rules:

- Source file names end in "_main.cpp".
- If a library shares the binary name:
 - Add "_cc" to the library rule, or
 - Move one rule to another dir.
- Should be tested with sh_test.

CC Rules

```
cc_library(  
    name = "foo_cc",  
    srcs = ["foo.cpp"],  
    hdrs = ["foo.h"],  
)  
  
cc_test(  
    name = "foo_test",  
    srcs = ["foo_test.cpp"],  
    deps = [":foo_cc", ...],  
)  
  
cc_binary(  
    name = "foo",  
    srcs = ["foo_main.cpp"],  
    deps = [":foo_cc", ...],  
)  
  
py_test(  
    name = "foo_py_test",  
    srcs = ["foo_py_test.py"],  
    data = [":foo"],  
)
```

Per C++ source file there should be:

- ONE header.
- ONE cc_library rule.
- All names should match up.
- Not matching makes maintenance hard.

Preferably:

- A header should only have one class.
- All header code in a namespace.

Each cc_library rule needs a test:

- All test names should end in "_test".

Binary rules:

- Source file names end in "_main.cpp".
- If a library shares the binary name:
 - Add "_cc" to the library rule, or
 - Move one rule to another dir.
- Should be tested with `sh_test`, or `py_test`.

CC Rules & CPP Files

```
cc_library(  
    name = "foo_cc",  
    srcs = ["foo.cpp"],  
    hdrs = ["foo.h"],  
)  
  
cc_test(  
    name = "foo_test",  
    srcs = ["foo_test.cpp"],  
    deps = [":foo_cc", ...],  
)  
  
cc_binary(  
    name = "foo",  
    srcs = ["foo_main.cpp"],  
    deps = [":foo_cc", ...],  
)  
  
sh_test(  
    name = "foo_main_test",  
    srcs = ["foo_main_test.sh"],  
    data = [":foo"],  
)
```

- Avoid forward declarations:
 - ODR violations.
 - Issues when changing struct/class.
 - Problems with IWYU.
 - Problems with dependencies.
 - Does not truly help with compile times.
- Move code into a namespace (same as dir).

CC Rules & CPP Files

```
cc_library(  
  name = "foo_cc",  
  srcs = ["foo.cpp"],  
  hdrs = ["foo.h"],  
)  
  
cc_test(  
  name = "foo_test",  
  srcs = ["foo_test.cpp"],  
  deps = [":foo_cc", ...],  
)  
  
cc_binary(  
  name = "foo",  
  srcs = ["foo_main.cpp"],  
  deps = [":foo_cc", ...],  
)  
  
sh_test(  
  name = "foo_main_test",  
  srcs = ["foo_main_test.sh"],  
  data = [":foo"],  
)
```

- Avoid forward declarations:
 - ODR violations.
 - Issues when changing struct/class.
 - Problems with IWYU.
 - Problems with dependencies.
 - Does not truly help with compile times.
- Move code into a namespace (same as dir).
- If you test with shell - make it reusable/reliable.
 - File diff testing is usually available by: github.com/bazelbuild/bazel-skylib
 - There are more involved alternatives: github.com/helly25/bashtest

CC Rules & CC test source

dir/BUILD

```
cc_library(  
    name = "foo_cc",  
    srcs = ["foo.cpp"],  
    hdrs = ["foo.h"],  
    deps = [...],  
)  
  
cc_test(  
    name = "foo_test",  
    srcs = ["foo_test.cpp"],  
    deps = [  
        "foo_cc",  
        "@com_google_googletest//:gtest",  
    ],  
)
```

dir/foo_test.cpp

```
#include "dir/foo.h"  
  
#include <gmock/gmock.h>  
#include <gtest/gtest.h>  
  
namespace dir {  
    namespace {  
  
        using ::testing::...  
  
        struct FooTest : ::testing::Test {};  
  
        TEST_F(FooTest, Basics) { /* TEST */ }  
  
    } // namespace  
} // namespace dir  
  
int main(int argc, char** argv) {  
    testing::InitGoogleTest(&argc, argv);  
    InitLoggingAndFlags(argc, argv);  
    return RUN_ALL_TESTS();  
}
```

CC Rules & CC test source

dir/BUILD

```
cc_library(  
    name = "foo_cc",  
    srcs = ["foo.cpp"],  
    hdrs = ["foo.h"],  
    deps = [...],  
)  
  
cc_test(  
    name = "foo_test",  
    srcs = ["foo_test.cpp"],  
    deps = [  
        "foo_cc",  
        "@com_google_googletest//:gtest",  
        "@com_google_googletest//:gtest_main",  
    ],  
)
```

dir/foo_test.cpp

```
#include "dir/foo.h"  
  
#include <gmock/gmock.h>  
#include <gtest/gtest.h>  
  
namespace dir {  
    namespace {  
  
        using ::testing::...  
  
        struct FooTest : ::testing::Test {};  
  
        TEST_F(FooTest, Basics) { /* TEST */ }  
  
    } // namespace  
} // namespace dir  
  
int main(int argc, char** argv) {  
    testing::InitGoogleTest(&argc, argv);  
    InitLoggingAndFlags(argc, argv);  
    return RUN_ALL_TESTS();  
}
```

CC Rules & CC test source & args/env

dir/BUILD

```
cc_library(  
    name = "foo_cc",  
    srcs = ["foo.cpp"],  
    hdrs = ["foo.h"],  
    deps = [...],  
)  
  
cc_test(  
    name = "foo_test",  
    srcs = ["foo_test.cpp"],  
    deps = [  
        "foo_cc",  
        "@com_google_googletest//:gtest",  
        "@com_google_googletest//:gtest_main",  
    ],  
    args = ["--arg", ...],  
    env = {"key": "value", ...},  
)
```

dir/foo_test.cpp

```
#include "dir/foo.h"  
  
#include <gmock/gmock.h>  
#include <gtest/gtest.h>  
  
namespace dir {  
    namespace {  
  
        using ::testing::...  
  
        struct FooTest : ::testing::Test {  
            static void SetupTestCase() {  
                // flags, env, whatever  
            }  
        };  
  
        TEST_F(FooTest, Basics) { /* TEST */ }  
  
    } // namespace  
} // namespace dir
```

CC Rules & CC test source & private code

`dir/foo.h`

```
#include <set>

namespace dir {

class Foo {
public:
    Foo() = default;

private:
    friend struct FooTest;

    std::set<std::string> GetNames() const;
};

} // namespace dir
```

`dir/foo_test.cpp`

```
#include "dir/foo.h"

#include <gmock/gmock.h>
#include <gtest/gtest.h>

namespace dir {

using ::testing::ElementsAre;

struct FooTest : ::testing::Test {
    auto GetNames(Foo& foo) {
        return foo.GetNames();
    }
};

namespace {

TEST_F(FooTest, Basics) {
    EXPECT_THAT(
        GetNames(Foo()), ElementsAre("42"));
}

} // namespace
} // namespace dir
```

CC Rules & CC test source & private code

`dir/foo.h`

```
#include <set>

namespace dir {

class Foo {
public:
    Foo() = default;

    std::set<std::string> TestOnlyNames() const;

private:
    // ...
};

} // namespace dir
```

`dir/foo_test.cpp`

```
#include "dir/foo.h"

#include <gmock/gmock.h>
#include <gtest/gtest.h>

namespace dir {
namespace {

using ::testing::Element;

struct FooTest : ::testing::Test {};

TEST_F(FooTest, Basics) {
    EXPECT_THAT(Foo().TestOnlyNames(),
                ElementsAre("42"));
}

} // namespace
} // namespace dir
```

CC Rules & CC test source & testonly code

`dir/foo.h`

```
#include <set>

namespace dir {

class Foo {
public:
    Foo() = default;

    std::set<std::string> TestOnlyNames() const;

private:
    // ...
};

} // namespace dir
```

`dir/foo_test.cpp`

```
#include "dir/foo.h"

#include <gmock/gmock.h>
#include <gtest/gtest.h>

namespace dir {
namespace {

using ::testing::Element;

struct FooTest : ::testing::Test {};

TEST_F(FooTest, Basics) {
    EXPECT_THAT(Foo().TestOnlyNames(),
                ElementsAre("42"));
}

} // namespace
} // namespace dir
```

Proto and C++

```
load("@rules_proto//proto:defs.bzl",  
     "proto_library")  
load("@rules_cc//cc:defs.bzl",  
     "cc_proto_library")
```

```
package(...)
```

```
proto_library(  
    name = "graph_op_proto",  
    srcs = ["graph_op.proto"],  
)
```

```
cc_proto_library(  
    name = "graph_op_cc_proto",  
    protos = ["graph_op_proto"],  
)
```

How to use proto rules in C++.

Proto and C++

```
load("@rules_proto//proto:defs.bzl",  
      "proto_library")  
load("@rules_cc//cc:defs.bzl",  
      "cc_proto_library")
```

```
package(...)
```

```
proto_library(  
    name = "graph_op_proto",  
    srcs = ["graph_op.proto"],  
)
```

```
cc_proto_library(  
    name = "graph_op_cc_proto",  
    protos = ["graph_op.proto"],  
)
```

How to use proto rules in C++.

Do not load `proto_library` or `cc_proto_library`
Do not use `cpp_proto_library` if you use `grpc`

Proto and C++

```
load("@rules_proto//proto:defs.bzl",  
-----  
"proto_library")  
load("@rules_cc//cc:defs.bzl",  
-----  
"cc_proto_library")
```

```
package(...)
```

```
proto_library(  
    name = "graph_op_proto",  
    srcs = ["graph_op.proto"],  
)
```

```
cc_proto_library(  
    name = "graph_op_cc_proto",  
    protos = ["graph_op_proto"],  
)
```

How to use proto rules in C++.

Do not load `proto_library` or `cc_proto_library`
Do not use `cpp_proto_library`.

All proto rules:

- Shall end in "_proto".
- Shall only have ONE proto file.

All proto language libraries:

- Shall end in "_<lang>_proto"
 - e.g. "_py_proto"
 - For C++ we use `_cc_proto`
- May depend on multiple proto rules

Proto and C++

```
load("@rules_proto//proto:defs.bzl",  
-----  
"proto_library")  
load("@rules_cc//cc:defs.bzl",  
-----  
"cc_proto_library")
```

```
package(...)
```

```
proto_library(  
    name = "graph_op_proto",  
    srcs = ["graph_op.proto"],  
)
```

```
cc_proto_library(  
    name = "graph_op_cc_proto",  
    protos = ["graph_op_proto"],  
)
```

How to use proto rules in C++.

Do not load `proto_library` or `cc_proto_library`
Do not use `cpp_proto_library`.

All proto rules:

- Shall end in "_proto".
- Shall only have ONE proto file.

All proto language libraries:

- Shall end in "_<lang>_proto"
 - e.g. "_py_proto"
 - For C++ we use `_cc_proto`
- May depend on multiple proto rules

The current set of rules is problematic though, and for most C++ protos we would need to disable `layering_check`. But behold, you're covered by patches.

If you still run into issues,...
then I fear you have to use longer names.

Proto and C++ & Grpc

```
load("@rules_proto_grpc//cpp:defs.bzl",
     "cpp_grpc_library")

package(...)

proto_library(
    name = "graph_op_proto",
    srcs = ["graph_op.proto"],
)

cpp_grpc_library(
    name = "graph_op_cc_proto",
    protos = ["graph_op_proto"],
)
```

How to use proto rules in C++ & Grpc.

All proto rules:

- Shall end in "_proto".
- Shall only have ONE proto file.

All proto language libraries:

- Shall end in "_<lang>_proto"
 - e.g. "_py_proto"
 - For C++ we use "_cc_proto"
- May depend on multiple proto rules

For proto files with services we need to use `cpp_grpc_library`.

1.2. Rule attributes

Bazel allows for a large amount of control

Rule attributes

Bazelrc files

What does attribute x do?

```
visibility = [...]
```

Controls what other rules can use the rule.

Special values that cannot be combined with others:

- `//visibility:public`
- `//visibility:private`
- These cannot be combined

Special target to allow rules in the same directory:

- `//<path...>:__pkg__`

Special target to allow rules in same and all all subdirs:

- `//<path...>:__subpackages__`

If omitted, then visibility is taken from:

- `package default_visibility.`

Do not rely on further default.

What does attribute x do?

```
visibility = [...]
```

```
testonly = True|1|False|0
```

Controls what other rules can use the rule.

Controls whether a rule can only be used in tests.

What does attribute x do?

```
visibility = [...]
```

```
testonly = True|1|False|0
```

```
copts = [  
    "-DUSE_OPENCENSUS=%d" % _USE_OPENCENSUS,  
],
```

Controls what other rules can use the rule.

Controls whether a rule can only be used in tests.

Options applied to the command line.

What does attribute x do?

```
visibility = [...]  
  
testonly = True|1|False|0  
  
copts = [  
    "-DUSE_OPENCENSUS=%d" % _USE_OPENCENSUS,  
],  
  
local_defines = [  
    "USE_OPENCENSUS=%d" % _USE_OPENCENSUS,  
],  
  
defines = [  
    "USE_OPENCENSUS=%d" % _USE_OPENCENSUS,  
],
```

Controls what other rules can use the rule.

Controls whether a rule can only be used in tests.

Options applied to the command line.

Also applied locally, prepends -D.

Compile options to this and all dependent rules.

What does attribute x do?

```
visibility = [...]
```

```
testonly = True|1|False|0
```

```
copts = [...]
```

```
local_defines = [...]
```

```
defines = [...]
```

```
linkopts = [...]
```

Controls what other rules can use the rule.

Controls whether a rule can only be used in tests.

Options applied to the command line.

Also applied locally, prepends -D.

Compile options to this and all dependent rules.

Linker options - well, I disallowed them.

What does attribute x do?

```
visibility = [...]
```

```
testonly = True|1|False|0
```

```
copts = [...]
```

```
local_defines = [...]
```

```
defines = [...]
```

```
linkopts = [...]
```

```
tags = ["manual"]
```

Controls what other rules can use the rule.

Controls whether a rule can only be used in tests.

Options applied to the command line.

Also applied locally, prepends -D.

Compile options to this and all dependent rules.

Linker options - well, I disallowed them.

All kinds of control for target selection.

Execute all manual tests:

```
bazel query "attr(tags, '\bmanual\b', //...)"
```

What does attribute x do?

```
visibility = [...]
```

```
testonly = True|1|False|0
```

```
copts = [...]
```

```
local_defines = [...]
```

```
defines = [...]
```

```
linkopts = [...]
```

```
tags = ["manual"]
```

Controls what other rules can use the rule.

Controls whether a rule can only be used in tests.

Options applied to the command line.

Also applied locally, prepends -D.

Compile options to this and all dependent rules.

Linker options - well, I disallowed them.

All kinds of control for target selection.

Execute all manual tests:

```
bazel query "attr(tags, '\bmanual\b', //...)"
```

Some special tags are defined in:

<https://bazel.build/reference/be/common-definitions>

What are those tags?

Special Bazel tags:

- `cpu:<NUM>`
- `manual`
- `no-cache`
- `no-remote`
- `no-sandbox`
- `requires-network`

Common tags to define:

- `no_san`
- `no_asan`
- `no_msan`
- `no_tsan`
- `no_ubsan`

<https://bazel.build/reference/be/common-definitions>

- Required CPUs
 - Manual execution
 - No caching of resulting output files
 - No remote execution
 - Not running in the sandbox **Please Do Not Use**
 - Rule needs network access **Please Do Not Use**
-
- Test not run in any Sanitizer mode
 - Test not run in ASAN mode
 - Test not run in MSAN mode
 - Test not run in TSAN mode
 - Test not run in UBSAN mode

Bazelrc files

```
common --announce_rc
```

You can debug the loading phase.

Bazelrc files

```
startup <flags>  
common <flags>  
build <flags>  
test <flags>
```

A bazelrc file combines commands with flags.

That includes Bazel's startup.

Bazelrc files

```
startup <flags>
common <flags>
build <flags>
test <flags>

common:my_config <flags>
```

A bazelrc file combines commands with flags.

That includes Bazel's startup.

Custom configurations can easily be defined.

Use them as:

```
bazel test --config=my_config //...
```

Bazelrc files

```
startup <flags>  
common <flags>  
build <flags>  
test <flags>
```

```
common:my_config <flags>
```

```
common:opt -c opt
```

Do Not Do This.

A bazelrc file combines commands with flags.

That includes Bazel's startup.

Custom configurations can easily be defined.

Use them as:

```
bazel test --config=my_config //...
```

Bazel has special modes (not configs):

```
bazel test -c opt //...
```

```
bazel test -c dbg //...
```

```
bazel test -c fastdebug //...
```

Do this instead

Bazelrc files

```
import %workspace%/ .bazelrc.user  
  
try-import %workspace%/../.bazelrc.user  
try-import %workspace%/../../.bazelrc.user  
try-import %workspace%/../../../.bazelrc.user
```

Bazelrc files can have includes.

Includes can be optional.

Bazelrc files

```
import %workspace%/ .bazelrc.user

try-import %workspace%/../.bazelrc.user
try-import %workspace%/../../.bazelrc.user
try-import %workspace%/../../../.bazelrc.user
```

Bazelrc files can have includes.

Includes can be optional.

Bazel looks for RC files in:

- Workspace directory: `.bazelrc`
- User: `~/.bazelrc`
- System: `/etc/bazel.bazelrc`

Making ASAN mode test work

.bazelrc

```
common:asan --copt -fsanitize=address,undefined
common:asan --linkopt -fsanitize=address,undefined
common:asan --copt -fno-sanitize=vptr
common:asan --linkopt -fno-sanitize=vptr
common:asan --copt=-DASAN_ACTIVE
common:asan --copt=-DDISABLE_TCMALLOC
common:asan --cxxopt=-gmlt
common:asan --linkopt -ldl
common:asan --define tcmalloc=disabled
common:asan --define signal_trace=disabled
common:asan --build_tag_filters=-no_san,-no_asan # skip (a)san targets
common:asan --test_tag_filters=-no_san,-no_asan # skip (a)san targets
common:asan --define is_asan=true # Important for `//:is_asan`
common:asan --test_timeout=120,600,1800,3600 # Increase test timeouts in ASAN mode, see
# https://bazel.build/reference/be/common-definitions#test.timeout

# In order to get usable stacktraces with line numbers:
common:asan --test_env=LLVM_SYMBOLIZER_PATH=./llvm_toolchain_llvm/bin/llvm-symbolizer
common:asan --test_env=ASAN_SYMBOLIZER_PATH=./llvm_toolchain_llvm/bin/llvm-symbolizer
common:asan --test_env=ASAN_OPTIONS=symbolize=1
```

A top level BUILD file

BUILD

```
config_setting(  
    name = "is_asan",  
    define_values = {"is_asan": "true"},  
)
```

The config name that the rule creates.

What Bazel matches against.

A top level BUILD file

BUILD

```
config_setting(  
    name = "is_asan",  
    define_values = {"is_asan": "true"},  
)  
  
config_setting(  
    name = "libc_malloc",  
    define_values = {"tcmalloc": "disabled"},  
)  
  
cc_test(  
    name = "slow_test",  
    srcs = ["slow_test.cpp"],  
    malloc = select({  
        "://:libc_malloc": "@bazel_tools//tools/cpp:malloc",  
        "://conditions:default": "@com_google_tcmalloc//tcmalloc",  
    }),  
    tags = ["no_asan"],  
    deps = [ ... ]  
)
```

What does attribute x do in a test rule?

```
args = ["--flag", ...]
```

Command line arguments passed to the test.

What does attribute x do in a test rule?

```
args = ["--flag", ...]
```

```
env = {"foo": "bar", ...}
```

Command line arguments passed to the test.

Native only: Environment variable passed to the test.

What does attribute x do in a test rule?

```
args = ["--flag", ...]
```

```
env = {"foo": "bar", ...}
```

```
size = "enormous" | "large" | "medium" | "small"
```

Command line arguments passed to the test.

Native only: Environment variable passed to the test.

Control the tests usage of RAM size & time.

<https://bazel.build/reference/be/common-definitions>

What does attribute x do in a test rule?

```
args = ["--flag", ...]
```

```
env = {"foo": "bar", ...}
```

```
size = "enormous" | "large" | "medium" | "small"
```

```
timeout = "short" | "moderate" | "long" | "eternal"
```

Command line arguments passed to the test.

Native only: Environment variable passed to the test.

Control the tests usage of RAM size & time.

Control the test timeout (we set ASAN longer).
<https://bazel.build/reference/be/common-definitions>

What does attribute x do in a test rule?

```
args = ["--flag", ...]
```

```
env = {"foo": "bar", ...}
```

```
size = "enormous"|"large"|"medium"|"small"
```

```
timeout = "short"|"moderate"|"long"|"eternal"
```

```
flaky = True|1|False|0
```

Command line arguments passed to the test.

Native only: Environment variable passed to the test.

Control the tests usage of RAM size & time.

Control the test timeout (we set ASAN longer).

Whether the test is flaky.

What does attribute x do in a test rule?

```
args = ["--flag", ...]
```

```
env = {"foo": "bar", ...}
```

```
size = "enormous"|"large"|"medium"|"small"
```

```
timeout = "short"|"moderate"|"long"|"eternal"
```

```
flaky = True|1|False|0
```

```
shard_count = 42
```

Command line arguments passed to the test.

Native only: Environment variable passed to the test.

Control the tests usage of RAM size & time.

Control the test timeout (we set ASAN longer).

Whether the test is flaky.

Number of shards the test will be subdivided into.

What does attribute x do in a test rule?

```
args = ["--flag", ...]
```

```
env = {"foo": "bar", ...}
```

```
size = "enormous"|"large"|"medium"|"small"
```

```
timeout = "short"|"moderate"|"long"|"eternal"
```

```
flaky = True|1|False|0
```

```
shard_count = 42
```

```
local = True|1|False|0
```

Command line arguments passed to the test.

Native only: Environment variable passed to the test.

Control the tests usage of RAM size & time.

Control the test timeout (we set ASAN longer).

Whether the test is flaky.

Number of shards the test will be subdivided into.

Whether the test needs to run locally.

1.3. Bazel Query

Reflect on what you are BUILDing

Bazel lets you query your rules - A very powerful help.

Bazel Query

```
bazel query //dir:*
```

Shows the available rules in dir/BUILD.

Bazel Query

```
bazel query //dir:*
```

```
bazel query //dir:bar
```

Shows the available rules in dir/BUILD.

Only show **bar**.

Bazel Query

```
bazel query //dir:*
```

```
bazel query //dir:bar
```

```
bazel query //dir:foo
```

Shows the available rules in dir/BUILD.

Only show **bar**.

Only **foo**.

Bazel Query

```
bazel query //dir:*
```

```
bazel query //dir:bar
```

```
bazel query //dir:foo
```

```
bazel build //dir
```

Shows the available rules in dir/BUILD.

Only show **bar**.

Only **foo**.

Only "**dir**" if it exists.

Bazel Query

```
bazel query //dir:*
```

```
bazel query //dir:bar
```

```
bazel query //dir:foo
```

```
bazel build //dir
```

```
bazel build //dir:*
```

Shows the available rules in dir/BUILD.

Only show **bar**.

Only **foo**.

Only "**dir**" if it exists.

All (non manual) in **dir**.

Bazel Query

```
bazel query //dir:*
```

```
bazel query //dir:bar
```

```
bazel query //dir:foo
```

```
bazel build //dir
```

```
bazel build //dir:*
```

```
bazel build //dir/...
```

Shows the available rules in dir/BUILD.

Only show **bar**.

Only **foo**.

Only "**dir**" if it exists.

All (non manual) in **dir**.

All (non manual) in dir and sub-directories.

Bazel Query – What can you get?

```
bazel query //base --output label
```

```
//base:base
```

```
bazel query //base --output label_kind
```

```
cc_library rule //base:base
```

```
bazel query //base --output location
```

```
/home/marcus/repos/repo/base/BUILD:42:11:  
cc_library rule //base:base
```

```
bazel query //base --output package
```

```
base
```

Bazel Query – Complex Output

```
bazel query //base:logging -output build
```

```
# /home/marcus/repos/repo/base/BUILD:25:11
```

```
cc_library(  
  name = "logging",  
  visibility = ["//visibility:public"],  
  hdrs = ["//base:logging.h"],  
  deps = [  
    "@com_google_absl//absl/debugging:failure_signal_handler",  
    "@com_google_absl//absl/log:absl_log",  
    "@com_google_absl//absl/log:absl_check",  
    "@com_google_absl//absl/log:die_if_null",  
    "@com_google_absl//absl/log:flags",  
    "@com_google_absl//absl/log:initialize",  
    "@com_google_absl//absl/log:vlog_is_on",  
  ],  
)  
# Rule logging instantiated at (most recent call last):  
#   /home/marcus/repos/repo/base/BUILD:25:11 in <toplevel>  
# Rule cc_library defined at (most recent call last):  
#   /virtual_builtins_bzl/common/cc/cc_library.bzl:612:18 in <toplevel>
```

Bazel Query

```
bazel query 'deps(//base)'
```

```
bazel query 'deps(//base, 1)'
```

```
bazel query 'deps(//base)' | grep -E '\.h$'
```

Dependencies OF `//base`.

Show all immediate dependencies.

Show C/C++ header files... , kind of.

Bazel Query

```
bazel query 'deps(//base)'
```

```
bazel query 'deps(//base, 1)'
```

```
bazel query 'deps(//base)' | grep -E '\.h$'
```

```
bazel query 'filter("\.h$",deps(//base))'
```

Dependencies OF `//base`.

Show all immediate dependencies.

Show C/C++ header files... , kind of.

Instead of `grep` `bazel query` has a `filter` function.

Bazel Query

```
bazel query 'deps(//base)'
```

```
bazel query 'filter("\.h$",deps(//base))'
```

```
bazel query 'rdeps(//..., //base)'
```

```
bazel query 'rdeps(//..., //base, 1)'
```

Dependencies OF `//base`.

Results can be filtered.

Reverse dependencies (users) of `//base` in `'//...'`.

Show direct users of `//base` in `'//...'`.

Bazel Query

```
bazel query 'deps(//base)'
```

```
bazel query 'filter("\.h$",deps(//base))'
```

```
bazel query 'rdeps(//..., //base)'
```

```
bazel query --infer_universe_scope  
'allrdeps(//base)'
```

```
bazel query -universe_scope=//...  
'allrdeps(//base)'
```

Dependencies OF `//base`.

Results can be filtered.

Reverse dependencies (users) of `//base` in `'//...'`.

Reverse dependency, convenient but maybe avoid it.

Either specify the universe scope in the rule or flag.

Bazel Query

```
bazel query 'deps(//base)'
```

```
bazel query 'filter("\.h$",deps(//base))'
```

```
bazel query 'rdeps(//..., //base)'
```

```
bazel query 'somepath(from, to)'
```

Dependencies OF `//base`.

Results can be filtered.

Reverse dependencies (users) of `//base` in `'//...'`.

Show any path `'from'` -> `'to'` if such a path exists.

Bazel Query

```
bazel query 'deps(//base)'
```

```
bazel query 'filter("\.h$",deps(//base))'
```

```
bazel query 'rdeps(//..., //base)'
```

```
bazel query 'somepath(from, to)'
```

```
bazel query 'allpaths(from, to)'
```

Dependencies OF `//base`.

Results can be filtered.

Reverse dependencies (users) of `//base` in `'//...'`.

Show any path `'from' -> 'to'` if such a path exists.

Show ALL paths `'from' -> 'to'`.

Bazel Query

```
bazel query 'deps(//base)'
```

Dependencies OF `//base`.

```
bazel query 'filter("\.h$",deps(//base))'
```

Results can be filtered.

```
bazel query 'rdeps(//..., //base)'
```

Reverse dependencies (users) of `//base` in `'//...'`.

```
bazel query 'somepath(from, to)'
```

Show any path `'from' -> 'to'` if such a path exists.

```
bazel query 'allpaths(from, to)'
```

Show ALL paths `'from' -> 'to'`.

```
bazel query 'kind(.*_binary, //base:*)'
```

Show rules whose KIND matches the regex.

Bazel Query

```
bazel query 'deps(//base)'
```

Dependencies OF `//base`.

```
bazel query 'filter("\.h$",deps(//base))'
```

Results can be filtered.

```
bazel query 'rdeps(//..., //base)'
```

Reverse dependencies (users) of `//base` in `'//...'`.

```
bazel query 'somepath(from, to)'
```

Show any path `'from'` -> `'to'` if such a path exists.

```
bazel query 'allpaths(from, to)'
```

Show ALL paths `'from'` -> `'to'`.

```
bazel query 'kind(.*_binary, //base:*)'
```

Show rules whose KIND matches the regex.

```
bazel query 'tests(//base:*)'
```

Show all test rules in `//base`.

Bazel Query

```
bazel query 'deps(//base)'
```

Dependencies OF `//base`.

```
bazel query 'filter("\.h$",deps(//base))'
```

Results can be filtered.

```
bazel query 'rdeps(//..., //base)'
```

Reverse dependencies (users) of `//base` in `'//...'`.

```
bazel query 'somepath(from, to)'
```

Show any path 'from' -> 'to' if such a path exists.

```
bazel query 'allpaths(from, to)'
```

Show ALL paths 'from' -> 'to'.

```
bazel query 'kind(.*_binary, //base:*)'
```

Show rules whose KIND matches the regex.

```
bazel query 'tests(//base:*)'
```

Show all test rules in `//base`.

```
bazel query '//base:* - tests(//base:*)'
```

Show all non-test rules in `//base`.

Bazel Query

```
bazel query 'deps(//base)'
```

```
bazel query 'filter("\.h$",deps(//base))'
```

```
bazel query 'rdeps(//..., //base)'
```

```
bazel query 'somepath(from, to)'
```

```
bazel query 'allpaths(from, to)'
```

```
bazel query 'kind(.*_binary, //base:*)'
```

```
bazel query 'tests(//base:*)'
```

```
bazel query '//base:* - tests(//base:*)'
```

```
bazel query '//base:* except tests(//base:*)'
```

Dependencies OF `//base`.

Results can be filtered.

Reverse dependencies (users) of `//base` in `'//...'`.

Show any path `'from'` \rightarrow `'to'` if such a path exists.

Show ALL paths `'from'` \rightarrow `'to'`.

Show rules whose KIND matches the regex.

Show all test rules in `//base`.

Show all non-test rules in `//base`.

Combining results:

- `-` \rightarrow except
- `+` \rightarrow union
- `^` \rightarrow intersect

Bazel Query

```
bazel query 'deps(//base)'
```

Dependencies OF `//base`.

```
bazel query 'filter("\.h$",deps(//base))'
```

Results can be filtered.

```
bazel query 'rdeps(//..., //base)'
```

Reverse dependencies (users) of `//base` in `'//...'`.

```
bazel query 'somepath(from, to)'
```

Show any path 'from' -> 'to' if such a path exists.

```
bazel query 'allpaths(from, to)'
```

Show ALL paths 'from' -> 'to'.

```
bazel query 'kind(.*_binary, //base:*)'
```

Show rules whose KIND matches the regex.

```
bazel query 'tests(//base:*)'
```

Show all test rules in `//base`.

```
bazel query '//base:* - tests(//base:*)'
```

Show all non-test rules in `//base`.

```
bazel query "attr(tags, '\bmanual\b', //base:*)"
```

Show rules in `//base` that are tagged 'manual'.

Bazel Query

```
bazel query 'deps(//base)'
```

Dependencies OF `//base`.

```
bazel query 'filter("\.h$",deps(//base))'
```

Results can be filtered.

```
bazel query 'rdeps(//..., //base)'
```

Reverse dependencies (users) of `//base` in `'//...'`.

```
bazel query 'somepath(from, to)'
```

Show any path 'from' -> 'to' if such a path exists.

```
bazel query 'allpaths(from, to)'
```

Show ALL paths 'from' -> 'to'.

```
bazel query 'kind(.*_binary, //base:*)'
```

Show rules whose KIND matches the regex.

```
bazel query 'tests(//base:*)'
```

Show all test rules in `//base`.

```
bazel query '//base:* - tests(//base:*)'
```

Show all non-test rules in `//base`.

```
bazel query "attr(tags, '\bmanual\b', //base:*)"
```

Show rules in `//base` that are tagged 'manual'.

```
bazel query "attr(srcs, '\[\]', //base:*)"
```

Show rules in `//base` whose `srcs` attribute is empty.

Bazel Query

```
bazel query 'deps(//base)'
```

Dependencies OF `//base`.

```
bazel query 'filter("\.h$",deps(//base))'
```

Results can be filtered.

```
bazel query 'rdeps(//..., //base)'
```

Reverse dependencies (users) of `//base` in `'//...'`.

```
bazel query 'somepath(from, to)'
```

Show any path `'from' -> 'to'` if such a path exists.

```
bazel query 'allpaths(from, to)'
```

Show ALL paths `'from' -> 'to'`.

```
bazel query 'kind(.*_binary, //base:*)'
```

Show rules whose KIND matches the regex.

```
bazel query 'tests(//base:*)'
```

Show all test rules in `//base`.

```
bazel query '//base:* - tests(//base:*)'
```

Show all non-test rules in `//base`.

```
bazel query "attr(tags, '\bmanual\b', //base:*)"
```

Show rules in `//base` that are tagged `'manual'`.

```
bazel query "attr(srcs, '\[\]', //base:*)"
```

Show rules in `//base` whose `srcs` attribute is empty.

```
bazel query 'labels(srcs, //base::*)'
```

Show elements of attribute `srcs` of all rules in `//base`.

Bazel Query – All immediate source files

```
bazel query: ask for srcs + hdrs but just in the given targets
```

Bazel Query – All immediate source files

```
bazel query: ask for srcs + hdrs but just in the given targets
```

```
bazel query '(...hdrs...) + (...srcs...)' # easy?
```

```
bazel query deps... # gives us all information
```

```
bazel query 'kind(source file, <rules>)' # all transitive (srcs, hdrs, data)
```

```
bazel query 'filter("//base:.*", kind("source file", deps("//base")))' # BAD!
```

```
bazel query 'kind("source file", deps("//base", 1))' # YAY - But has `BUILD` and data :-)
```

```
bazel query 'labels(srcs, //base) + labels(hdrs, //base)' # YAY!
```

Bazel Query – All immediate source files

```
bazel query: ask for srcs + hdrs but just in the given targets
```

```
bazel query '(...hdrs...) + (...srcs...)' # easy?
```

```
bazel query deps... # gives us all information
```

```
bazel query 'kind(source file, <rules>)' # all transitive (srcs, hdrs, data)
```

```
bazel query 'filter("//base:.*", kind("source file", deps("//base")))' # BAD!
```

```
bazel query 'kind("source file", deps("//base", 1))' # YAY - But has `BUILD` and data :-)
```

```
bazel query 'labels(srcs, //base) + labels(hdrs, //base)' # YAY!
```

```
bazel query 'let X="//base" in labels(srcs, $X) + labels(hdrs, $X)'
```

```
# Note: attribute 'srcs' is local, 'data' is always recursive and 'hdrs' something in-between.
```

Bazel Query – Other local source files

```
bazel query: ask for srcs + hdrs in same package but not in rule...
```

Bazel Query – Other local source files

```
bazel query: ask for srcs + hdrs in same package but not in rule...
```

```
bazel query 'kind("source file", deps("//base:*", 1))' # All files in package //base
```

```
bazel query 'kind("source file", deps("//base:*", 1)) - deps("//base)'
```

```
bazel query 'kind("source file", deps("//base:*", 1)) - deps("//base)' | grep ...
```

Bazel Query – Other local source files

```
bazel query: ask for srcs + hdrs in same package but not in rule...
```

```
bazel query 'kind("source file", deps("//base:*", 1))' # All files in package //base
```

```
bazel query 'kind("source file", deps("//base:*", 1)) - deps("//base)'
```

```
bazel query 'kind("source file", deps("//base:*", 1)) - deps("//base)' | grep ...
```

```
bazel query 'let D="//base:*" in (labels(srcs, $D) + labels(hdrs, $D)) - deps("//base)'
```

Bazel Query – The Manual Virus

```
bazel query: check whether reverse deps of manual rules are manual
```

```
bazel query 'attr(tags, "\bmanual\b", //...)' # All manual rules
```

Bazel Query – The Manual Virus

```
bazel query: check whether reverse deps of manual rules are manual
```

```
bazel query 'attr(tags, "\bmanual\b", //...)' # All manual rules
```

```
bazel query 'rdeps(//..., attr(tags, "\bmanual\b", //...))' # Far too many we don't care about
```

Bazel Query – The Manual Virus

```
bazel query: check whether reverse deps of manual rules are manual
```

```
bazel query 'attr(tags,"\bmanual\b",//...)' # All manual rules
```

```
bazel query 'rdeps(//...,attr(tags,"\bmanual\b",//...))' # Far too many we don't care about
```

```
bazel query 'rdeps(//...,attr(tags,"\bmanual\b",//...),1)' # But... no need for recursion
```

Bazel Query – The Manual Virus

```
bazel query: check whether reverse deps of manual rules are manual
```

```
bazel query 'attr(tags, "\bmanual\b", //...)' # All manual rules  
bazel query 'rdeps(//..., attr(tags, "\bmanual\b", //...))' # Far too many we don't care about  
bazel query 'rdeps(//..., attr(tags, "\bmanual\b", //...), 1)' # But... no need for recursion  
bazel query 'kind(".*(cc|py).*", //...)' # Restrict to C++ & Python rules
```

Bazel Query – The Manual Virus

```
bazel query: check whether reverse deps of manual rules are manual
```

```
bazel query 'attr(tags, "\bmanual\b", //...)' # All manual rules
bazel query 'rdeps(//..., attr(tags, "\bmanual\b", //...))' # Far too many we don't care about
bazel query 'rdeps(//..., attr(tags, "\bmanual\b", //...), 1)' # But... no need for recursion
bazel query 'kind(".*(cc|py).*", //...)' # Restrict to C++ & Python rules
bazel query 'let M=attr(tags, "\bmanual\b", //...) in
             rdeps(//..., kind(".*(cc|py).*", $M), 1)' # Direct deps may be manual
```

Bazel Query – The Manual Virus

```
bazel query: check whether reverse deps of manual rules are manual
```

```
bazel query 'attr(tags,"\bmanual\b",//...)' # All manual rules

bazel query 'rdeps(//...,attr(tags,"\bmanual\b",//...))' # Far too many we don't care about

bazel query 'rdeps(//...,attr(tags,"\bmanual\b",//...),1)' # But... no need for recursion

bazel query 'kind(".*(cc|py).*",//...)' # Restrict to C++ & Python rules

bazel query 'let M=attr(tags,"\bmanual\b",//...) in
  rdeps(//...,kind(".*(cc|py).*", $M),1)' # Direct deps may be manual

bazel query 'let M=attr(tags,"\bmanual\b",//...) in
  rdeps(//...,kind(".*(cc|py).*", $M),1)
- $M' # Now we get mypy_type rules.
```

Bazel Query – The Manual Virus

```
bazel query: check whether reverse deps of manual rules are manual
```

```
bazel query 'attr(tags, "\bmanual\b", //...)' # All manual rules

bazel query 'rdeps(//..., attr(tags, "\bmanual\b", //...))' # Far too many we don't care about

bazel query 'rdeps(//..., attr(tags, "\bmanual\b", //...), 1)' # But... no need for recursion

bazel query 'kind(".*(cc|py).*", //...)' # Restrict to C++ & Python rules

bazel query 'let M=attr(tags, "\bmanual\b", //...) in
  rdeps(//..., kind(".*(cc|py).*", $M), 1)' # Direct deps may be manual

bazel query 'let M=attr(tags, "\bmanual\b", //...) in
  rdeps(//..., kind(".*(cc|py).*", $M), 1)
  - $M' # Now we get mypy_type rules.

bazel query 'let M=attr(tags, "\bmanual\b", //...) in
  rdeps(//..., kind(".*(cc|py).*", $M), 1)
  - $M - kind("mypy_type_.*", //...)' # Finally!
```

Bazel Query – The Manual Virus

```
bazel query: check whether reverse deps of manual rules are manual
```

```
bazel query 'attr(tags,"\bmanual\b",//...)' # All manual rules

bazel query 'rdeps(//...,attr(tags,"\bmanual\b",//...))' # Far too many we don't care about

bazel query 'rdeps(//...,attr(tags,"\bmanual\b",//...),1)' # But... no need for recursion

bazel query 'kind(".*(cc|py).*",//...)' # Restrict to C++ & Python rules

bazel query 'let M=attr(tags,"\bmanual\b",//...) in
  rdeps(//...,kind(".*(cc|py).*", $M),1)' # Direct deps may be manual

bazel query 'let M=attr(tags,"\bmanual\b",//...) in
  rdeps(//...,kind(".*(cc|py).*", $M),1)
  - $M' # Now we get mypy_type rules.

bazel query 'let M=attr(tags,"\bmanual\b",//...) in
  rdeps(//...,kind(".*(cc|py).*", $M),1)
  - $M - kind("mypy_type_.*",//...)' # Finally!

bazel query 'let M=attr(tags,"\bmanual\b",//...) in
  rdeps(//...,kind(".*(cc|py).*", $M),1)
  - $M - kind("mypy_type_.*",//...)' | wc -l # TEST: Line Count = 0?
```

More Bazel Query

Query:

- <https://bazel.build/query/guide>
- <https://bazel.build/query/language>

CQuery:

- Mostly the same, but it understands configuration (select).

AQuery

- Gives you access to the Bazel execution graph.
- Used by tools, like: `bazel run //:refresh_clang`

Query results are often post-processed using:

- `awk`, `grep`, `sed`, `xargs`

2. Bazel Tools

A Long list of tools

Buildifier

Gazelle

rules_foreign_cc

toolchains_llvm

Bazel Central Registry

VSCoDe plugins

Buildfarm

Bazel Playground

Github Action Setup-Bazel

Buildifier

Buildifier is your friendly formatter.

It is used by many tools since it can do more :-)

Gazelle

```
load("@bazel_gazelle//:def.bzl", "gazelle")  
  
# gazelle:prefix github.com/example/project  
gazelle(name = "gazelle")
```

Gazelle automagically generates & updates BUILD files.

Simple Bazelmod install.

After updates run: `bazel run //:gazelle`

There's more at:

<https://github.com/bazel-contrib/bazel-gazelle>

Not everything has Bazel support

WORKSPACE

```
git_repository(  
  name = "zlib",  
  build_file = "//third_party/zlib.BUILD",  
  remote = "https://github.com/madler/zlib",  
  # sha256 = ...  
  tag = "v1.3",  
)
```

third_party/zlib.BUILD

```
cc_library(  
  name = "zlib",  
  srcs = glob(["*.c"]),  
  hdrs = glob(["*.h"]),  
  copts = ["-w", "-DZ_HAVE_UNISTD_H"],  
  includes = ["."],  
  visibility = ["//visibility:public"],  
)
```

No Bazel support? rules_foreign_cc

WORKSPACE

```
git_repository(  
  name = "z3",  
  build_file = "@//third_party:z3.BUILD",  
  remote = "https://github.com/Z3Prover/z3",  
  # sha256 = ...  
  tag = "z3-4.13.4",  
)
```

third_party/z3.BUILD

```
load(  
  "@rules_foreign_cc//foreign_cc:defs.bzl",  
  "cmake"  
)  
filegroup(  
  name = "all_srcs",  
  srcs = glob(["**"]),  
)  
  
cmake(  
  name = "z3",  
  build_args = ["--", "-j 40"],  
  cache_entries = {  
    "CMAKE_CXX_FLAGS": "-Wno-error",  
    "CMAKE_INSTALL_INCLUDEDIR": "include/z3",  
    "Z3_BUILD_EXECUTABLE": "OFF",  
    "Z3_BUILD_LIBZ3_SHARED": "OFF",  
    "Z3_BUILD_TEST_EXECUTABLES": "OFF",  
    "Z3_ENABLE_EXAMPLE_TARGETS": "OFF",  
  },  
  lib_source = ":all_srcs",  
  out_static_libs = ["libz3.a"],  
)
```

No Bazel support? rules_foreign_cc

WORKSPACE

```
_ALL_CONTENT = ""
filegroup(
    name = "all",
    srcs = glob(["**"]),
    visibility = [
        "@//deps:__subpackages__",
    ],
)
""

wrap_git_repository(
    name = "libmicrohttpd",
    build_file_content = _ALL_CONTENT,
    remote =
    "https://github.com/Karlson2k/libmicrohttpd",
    tag = "v1.0.1",
    # sha256 = ...
),
```

third_party/microhttpd/BUILD

```
load(
    "@rules_foreign_cc//foreign_cc:defs.bzl",
    "configure_make"
)

configure_make(
    name = "libmicrohttpd",
    autogen = True,
    configure_in_place = True,
    configure_options = [
        "--disable-cookie",
        "--disable-curl",
        # MANY MORE :-),
        "--without-libcurl",
    ],
    env = {
        "TSAN_OPTIONS": "",
    },
    lib_source = "@libmicrohttpd//:all",
    visibility = ["//visibility:private"],
)
```

toolchains_llvm

MODULE.bazel

```
bazel_dep(name = "toolchains_llvm", version = "1.3.0")

git_override(
    module_name = "toolchains_llvm",
    commit = "e831f94a8b7f3a39391f5822adcab8e4d443c03b",
    # Add more tools by default (#463)
    remote = "https://github.com/bazel-contrib/toolchains_llvm",
)

llvm = use_extension(
    "@toolchains_llvm//toolchain/extensions:llvm.bzl", "llvm", dev_dependency = True)

llvm.toolchain(
    name = "llvm_toolchain_llvm",
    llvm_version = "19.1.6",
)

use_repo(llvm, "llvm_toolchain_llvm")

register_toolchains("@llvm_toolchain_llvm//:all", dev_dependency = True)
```

toolchains_llvm

MODULE.bazel

```
bazel_dep(name = "toolchains_llvm", version = "1.3.0")

git_override(
    module_name = "toolchains_llvm",
    commit = "e831f94a8b7f3a39391f5822adcab8e4d443c03b",
    # Add more tools by default (#463)
    remote = "https://github.com/bazel-contrib/toolchains_llvm",
)

llvm = use_extension(
    "@toolchains_llvm//toolchain/extensions:llvm.bzl", "llvm", dev_dependency = True)

llvm.toolchain(
    name = "llvm_toolchain_llvm",
    llvm_version = "19.1.6",
)

use_repo(llvm, "llvm_toolchain_llvm")

register_toolchains("@llvm_toolchain_llvm//:all", dev_dependency = True)
```

toolchains_llvm

<https://github.com/helly25/mbo/blob/main/.bazelrc>

```
common --consistent_labels
common --enable_bzlmod
common --no incompatible_enable_cc_toolchain_resolution
common --incompatible_disallow_empty_glob
common --nolegacy_external_runfiles
common --features=layering_check

common --apple_platform_type=macos
common --enable_platform_specific_config

common:macos --features=-supports_dynamic_linker
common:macos --linkopt=-framework --linkopt=CoreFoundation
common:macos --host_linkopt=-framework --host_linkopt=CoreFoundation

common:clang --incompatible_enable_cc_toolchain_resolution
common:clang --cxxopt=-gmlt
common:clang --host_cxxopt=-gmlt
common:clang --linkopt=-fuse-ld=lld
common:clang --host_linkopt=-fuse-ld=lld

common:cpp23 --cxxopt=-std=c++23 --host_cxxopt=-std=c++23
```

toolchains_llvm

<https://github.com/helly25/mbo/blob/main/.bazelrc>

```
common --consistent_labels
common --enable_bzlmod
common --no incompatible_enable_cc_toolchain_resolution
common --incompatible_disallow_empty_glob
common --nolegacy_external_runfiles
common --features=layering_check

common --apple_platform_type=macos
common --enable_platform_specific_config

common:macos --features=-supports_dynamic_linker
common:macos --linkopt=-framework --linkopt=CoreFoundation
common:macos --host_linkopt=-framework --host_linkopt=CoreFoundation

common:clang --incompatible_enable_cc_toolchain_resolution
common:clang --cxxopt=-gmlt
common:clang --host_cxxopt=-gmlt
common:clang --linkopt=-fuse-ld=lld
common:clang --host_linkopt=-fuse-ld=lld

common:cpp23 --cxxopt=-std=c++23 --host_cxxopt=-std=c++23
```

toolchains_llvm

<https://github.com/helly25/mbo/blob/main/.bazelrc>

```
common --consistent_labels
common --enable_bzlmod
common --no incompatible_enable_cc_toolchain_resolution
common --incompatible_disallow_empty_glob
common --nolegacy_external_runfiles
common --features=layering_check

common --apple_platform_type=macos
common --enable_platform_specific_config

common:macos --features=-supports_dynamic_linker
common:macos --linkopt=-framework --linkopt=CoreFoundation
common:macos --host_linkopt=-framework --host_linkopt=CoreFoundation

common:clang --incompatible_enable_cc_toolchain_resolution
common:clang --cxxopt=-gmlt
common:clang --host_cxxopt=-gmlt
common:clang --linkopt=-fuse-ld=lld
common:clang --host_linkopt=-fuse-ld=lld

common:cpp23 --cxxopt=-std=c++23 --host_cxxopt=-std=c++23
```

toolchains_llvm

<https://github.com/helly25/mbo/blob/main/.bazelrc>

```
common --consistent_labels
common --enable_bzlmod
common --no incompatible_enable_cc_toolchain_resolution
common --incompatible_disallow_empty_glob
common --nolegacy_external_runfiles
common --features=layering_check

common --apple_platform_type=macos
common --enable_platform_specific_config

common:macos --features=-supports_dynamic_linker
common:macos --linkopt=-framework --linkopt=CoreFoundation
common:macos --host_linkopt=-framework --host_linkopt=CoreFoundation

common:clang --incompatible_enable_cc_toolchain_resolution
common:clang --cxxopt=-gmlt
common:clang --host_cxxopt=-gmlt
common:clang --linkopt=-fuse-ld=lld
common:clang --host_linkopt=-fuse-ld=lld

common:cpp23 --cxxopt=-std=c++23 --host_cxxopt=-std=c++23
```

<https://registry.bazel.build/>

Central Bazel registry - central to all new Bazel projects :-)

Classic UI - but it has everything you need - search and links.

It is the package manager that was missing.

VSCode



Bazel v0.11.0

The Bazel Team [bazel.build](#) | 643,076 | ★★★★★ (8)

Bazel BUILD integration

Auto Update

Extension is enabled on 'SSH: marcus'



Blue Bazel v1.0.6

NVIDIA [nvidia.com](#) | 16,816 | ★★★★★ (2)

Bazel vscode UI integration to build, debug, and test targets

Auto Update

Extension is enabled on 'SSH: marcus'

- Syntax highlighting
- Bazel Targets tree display
- CodeLens links in BUILD
- Buildifier integration
- Bazel Task definitions for tasks.json
- Coverage Support showing
- Debug Starlark code in your .bzl files

```
2
3  import "testing"
4  import "fmt"
5
6  bluebazel test | bluebazel debug | run test | debug test
7  func TestIt(t *testing.T) {
8      one := 1
9      two := 2
10     fmt.Println(one + two)
11 }
```

compile commands extraction

There are a few ways to extract `compile_commands.json`.

- My favorite: <https://github.com/hedronvision/bazel-compile-commands-extractor/>
- Development currently on hold :-)

compile commands extraction

There are a few ways to extract compile_commands.json.

- My favorite: <https://github.com/hedronvision/bazel-compile-commands-extractor/>
- Development currently on hold :-(
- Taken from <http://github.com/helly25/mbo>

- Workspace

```
github_archive(  
    name = "hedron_compile_commands",  
    commit = "6dd21b47db481a70c61698742438230e2399b639",  
    repo = "https://github.com/helly25/bazel-compile-commands-extractor",  
    sha256 = "348a643defa9ab34ed9cb2ed1dc54b1c4ffef1282240aa24c457ebd8385ff2d5",  
)
```

- Bzelmod

```
bazel_dep(name = "hedron_compile_commands", dev_dependency = True)  
git_override(  
    module_name = "hedron_compile_commands",  
    commit = "6dd21b47db481a70c61698742438230e2399b639",  
    remote = "https://github.com/hedronvision/bazel-compile-commands-extractor",  
)
```

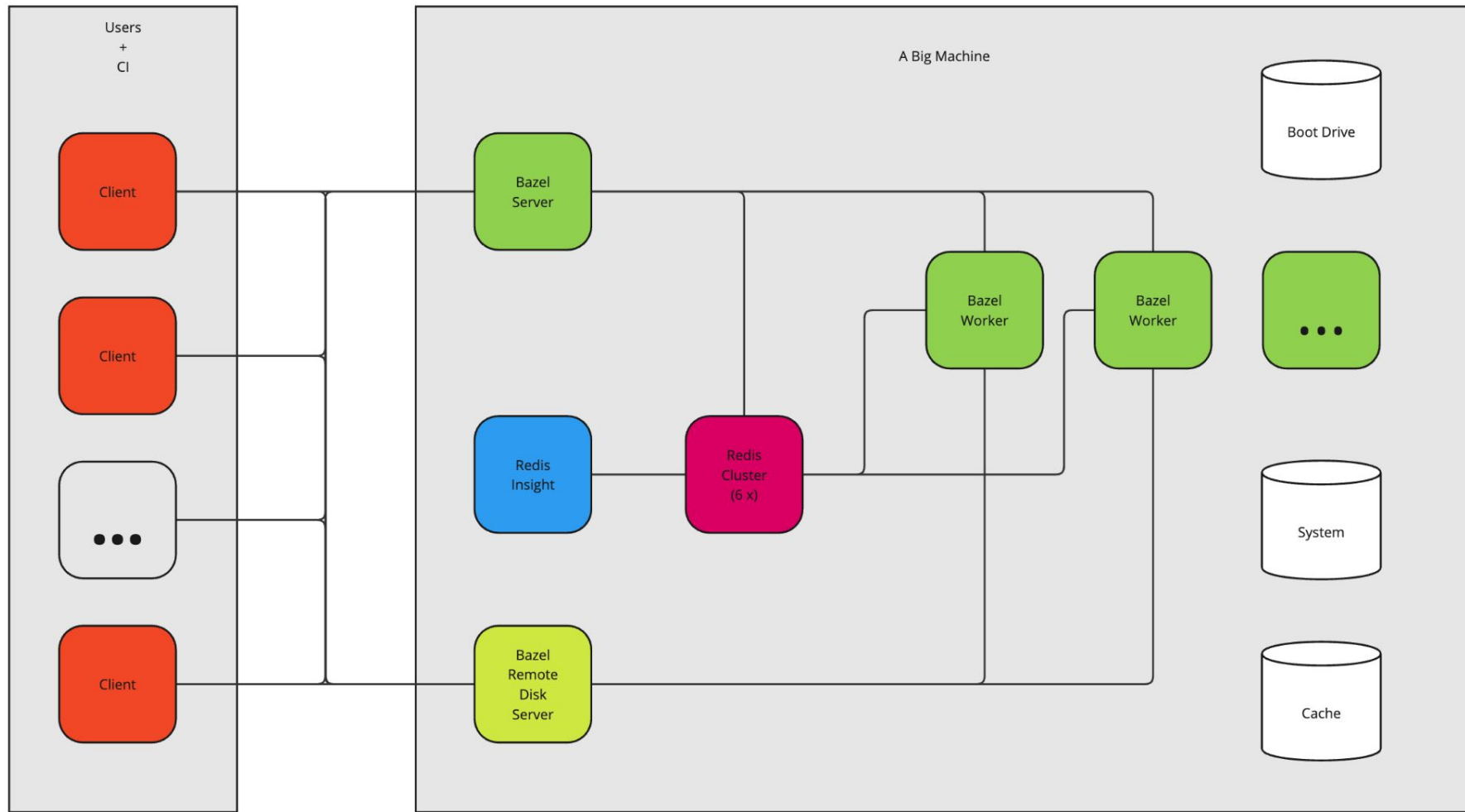
Buildfarm

There are a number of Buildfarm solutions available.

They come as OpenSource projects and as commercial solutions.

- Primary OpenSource: <https://github.com/buildfarm/buildfarm>
 - Supports many backends, again OS as well as commercial.
 - Easy to setup.
 - They have a Slack channel :-)
 - Repo comes with helm charts.
 - Comes with a server and workers
 - Needs storage and something like Redis
 - Supports Prometheus
 - We used it together with <https://github.com/buildfarm/buildfarm>
 - There is a helpful manager <https://github.com/buildfarm/bfmgr>

A Simple Setup



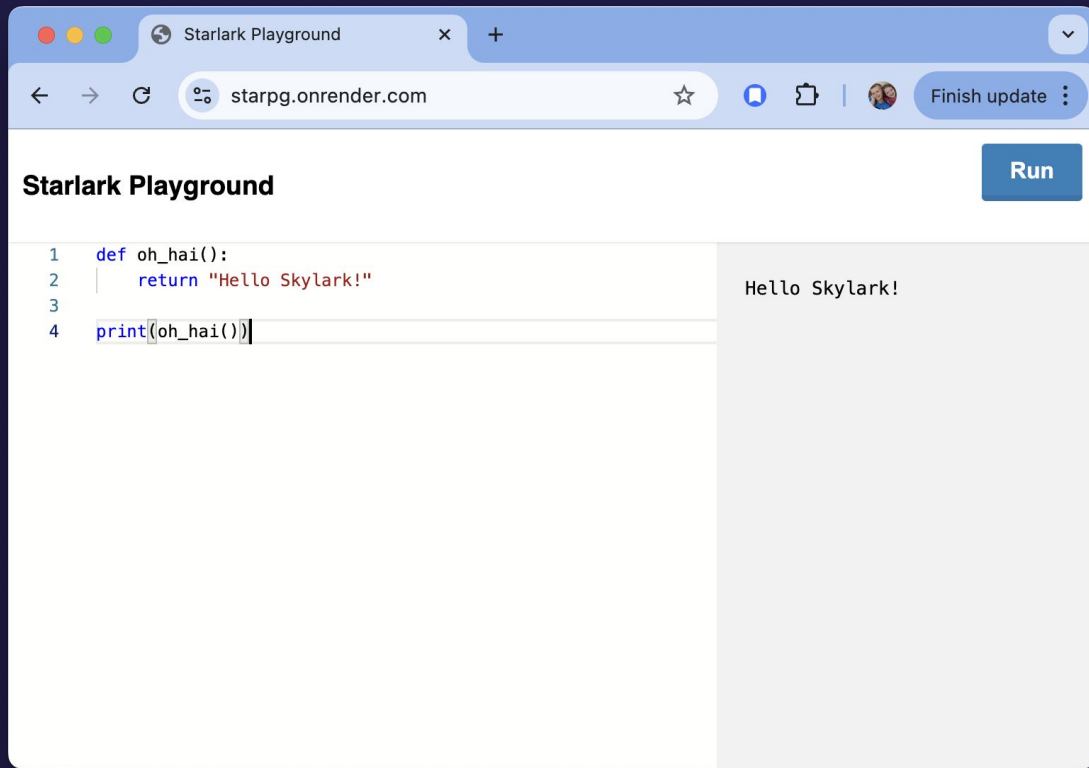
Bazel Playground

Technically a Starlark Playground.

There are a few of them.

I sometimes use:

<https://starpq.onrender.com/>



Github Action Setup-Bazel

<https://github.com/bazel-contrib/setup-bazel>

Easy way to set up Bazel based actions with caching in Github.

Github Action Setup-Bazel

```
name: Test
on: [push]
```

```
jobs:
```

```
  test:
```

```
    runs-on: ubuntu-latest
```

```
    steps:
```

```
      - uses: actions/checkout@v4
```

```
      - uses: bazel-contrib/setup-bazel@0.14.0
```

```
        with:
```

```
          bazelisk-cache: true
```

```
          disk-cache: ${{ github.workflow }}
```

```
          repository-cache: true
```

```
      - name: Built & Test
```

```
        run: |
```

```
          bazel test //...
```

Github Action Setup-Bazel

name: Test

on: [push]

jobs:

test:

needs: pre-commit

strategy:

matrix:

os: [ubuntu-latest, macos-latest, windows-latest]

mode: [bzlmod, workspace, both]

runs-on: \${{ matrix.os }}

steps:

- uses: actions/checkout@v4

- uses: bazel-contrib/setup-bazel@0.14.0

with:

bazelisk-cache: true

disk-cache: \${{ github.workflow }}

repository-cache: true

- name: Built & Test

env:

BZLMOD: \${{ matrix.mode != 'workspace' && '--enable_bzlmod' || '--noenable_bzlmod' }}

WORKSPACE: \${{ matrix.mode != 'bzlmod' && '--enable_workspace' || '--noenable_workspace' }}

run: |

bazel test \${{env.BZLMOD}} \${{env.WORKSPACE}} //...

Github Action Setup-Bazel

name: Test

on: [push]

jobs:

test:

needs: pre-commit

strategy:

matrix:

os: [ubuntu-latest, macos-latest, windows-latest]

mode: [bzlmod, workspace, both]

runs-on: \${{ matrix.os }}

steps:

- uses: actions/checkout@v4

- uses: bazel-contrib/setup-bazel@0.14.0

with:

bazelisk-cache: true

disk-cache: \${{ github.workflow }}

repository-cache: true

- name: Built & Test

env:

BZLMOD: \${{ matrix.mode != 'workspace' && '--enable_bzlmod' || '--noenable_bzlmod' }}

WORKSPACE: \${{ matrix.mode != 'bzlmod' && '--enable_workspace' || '--noenable_workspace' }}

run: |

bazel test \${{env.BZLMOD}} \${{env.WORKSPACE}} //...

accu
conference
2025

Bazel

Thanks

Marcus Boerger



ACCU
conference
2025

Bazel

Thanks

Questions

Marcus Boerger



accu
conference
2025

Bazel

Thanks

Questions

Marcus Boerger

Test, Analyze & Stop Debugging!

Links

<https://bazel.build/>

<https://registry.bazel.build/>

<https://marketplace.visualstudio.com/items?itemName=BazelBuild.vscode-bazel>

<https://marketplace.visualstudio.com/items?itemName=NVIDIA.bluebazel>

<https://github.com/hedronvision/bazel-compile-commands-extractor/>

https://github.com/bazel-contrib/rules_foreign_cc

https://github.com/bazel-contrib/toolchains_llvm

<https://bazel.build/query/guide>

<https://bazel.build/query/language>