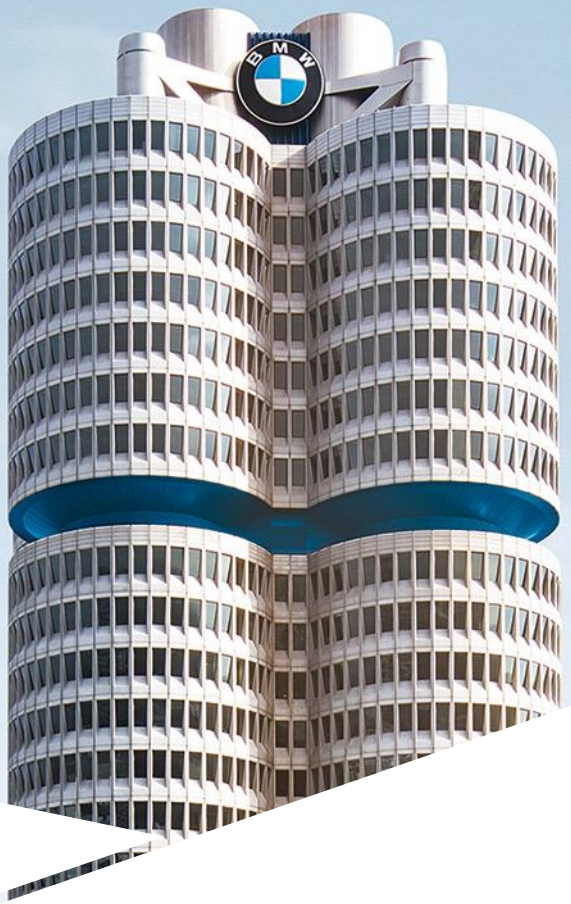
INTRODUCTION TO BAZEL TO BUILD C++ AND PYTHON



Xavier Bonaventura
code::dive 2020 - 18/11/2020

BMW GROUP THE NEXT 100 YEARS

Rolls-Royce Motor Cars Limited

ABOUT ME

- Software developer at BMW Group since 2018
- Working at the department of Full Autonomous Driving
- Working with Bazel for around 2 years
- Mainly working on C++ and Python
- GitHub: https://github.com/limdor/

Xavier Bonaventura

limdor

Software engineer working with C++, Bazel and Python

Follow ···

16 followers · 27 following · ☆ 3

Overview | Repositories 12 | Projects | Packages

Pinned

quoniam
A framework for viewpoint selection of 3D models
C++ ☆ 2

hardware-tessellation
Terrain and Ocean rendering with hardware tessellation
C++

bazel_rules_qt
Forked from justbuchanan/bazel_rules_qt
Bazel rules for Qt
Starlark

bazel-examples
The goal is to have several examples including multi language projects using Bazel build system
Starlark ☆ 1

presentations
Slides and material of the different talks that I gave
☆ 1

ABOUT BAZEL

- What is Bazel?
  - A build system, not a build generator (invokes directly the compiler)
  - With full of functionality for testing (test reports, flaky tests handling, etc.)
  - Bazel core is written in Java, rules and macros written in Starlark
- History - From Blaze to Bazel
  - Blaze is the build system at Google (development started around 2007)
  - Part of Blaze was open sourced on 2015 as Bazel
  - It moved from beta to general availability in October 2019
- Release process
  - Since the general availability release, Bazel follows semantic versioning
  - Minimum 3 months between major releases
  - A minor release every month based on GitHub HEAD
  - Long Term Support (LTS) starting from Bazel 4.0 (December 2020), a new LTS release will be provided every 9 months
    - https://blog.bazel.build/2020/11/10/long-term-support-release.html

https://docs.bazel.build/

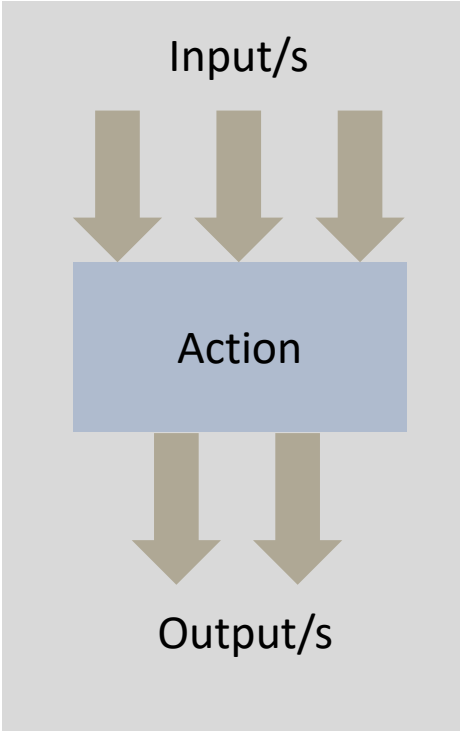
BAZEL FEATURES

- Fast and correct
  - Incremental builds and test execution
  - Parallel execution
  - Local and remote cache
  - Hermetic builds thanks to sandboxing
- Multi-language, multi-platform
  - Java, C/C++, Android, iOS, Go, Python, etc.
  - Linux, Windows, and macOS
- Scalable
  - It can handle codebases of any size
  - Multiple repositories or huge monorepo, it handles both
- Extensible
  - If a platform or language is not supported can be easily added
  - Extensions are written in Skylark, a language similar to Python

https://docs.bazel.build/versions/3.7.0/bazel-overview.html#why-should-i-use-bazel

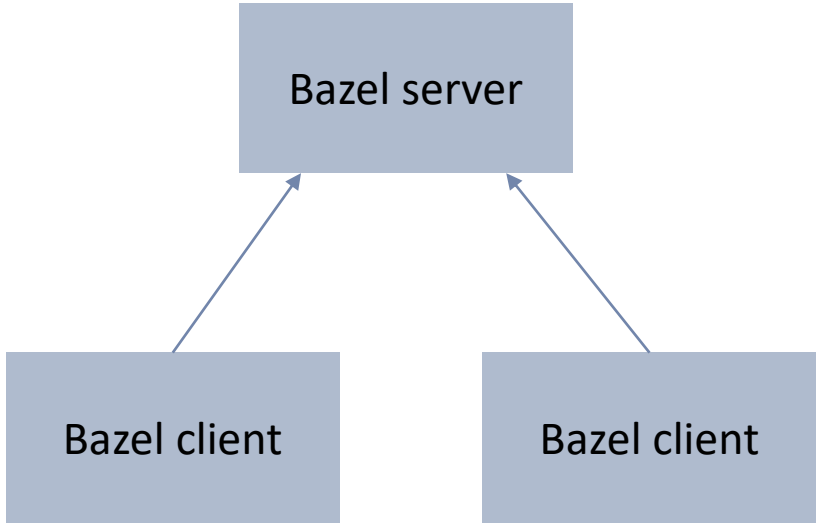
BAZEL IN A NUTSHELL

- It is an artifact based system
  - Inputs are treated as artifacts
  - Outputs are treated as artifacts
  - Actions are treated as artifacts as well
  - For every artifact a hash can be computed in advance to allow caching
- Each action runs on a sandbox
  - Improving reproducibility
  - Better detection of undeclared dependencies
- Composability
  - Outputs of an action can be used as inputs of another action
  - An action can be the output of another action



BAZEL DESIGN

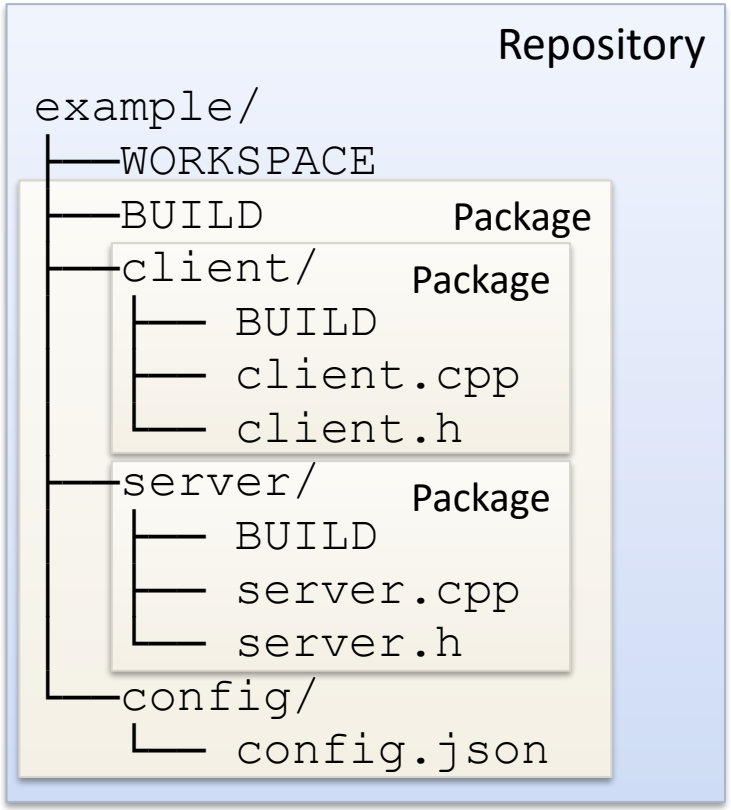
- Client/server architecture
  - The first time a Bazel command is executed, a Bazel server is started
  - After the Bazel command finishes, the server keeps running
  - The following commands executed use the already running server
  - Two Bazel clients cannot run in parallel (with exceptions)
  - The Bazel server can be stopped with `bazel shutdown`
  - This architecture allows the server to cache information



https://docs.bazel.build/versions/3.7.0/guide.html#clientserver-implementation

BAZEL FILES

- WORKSPACE
  - At the root of the source code that you want to build
  - It can be empty
  - Used to declare external dependencies
- BUILD
  - At the root of a package
  - A package is defined by all files, folders, and subfolders at the same level like the BUILD file except the ones that contain a BUILD file
  - Where targets are defined
- *.bzl
  - Use to define Bazel extensions
  - They can be loaded in a BUILD or WORKSPACE file using the load statement

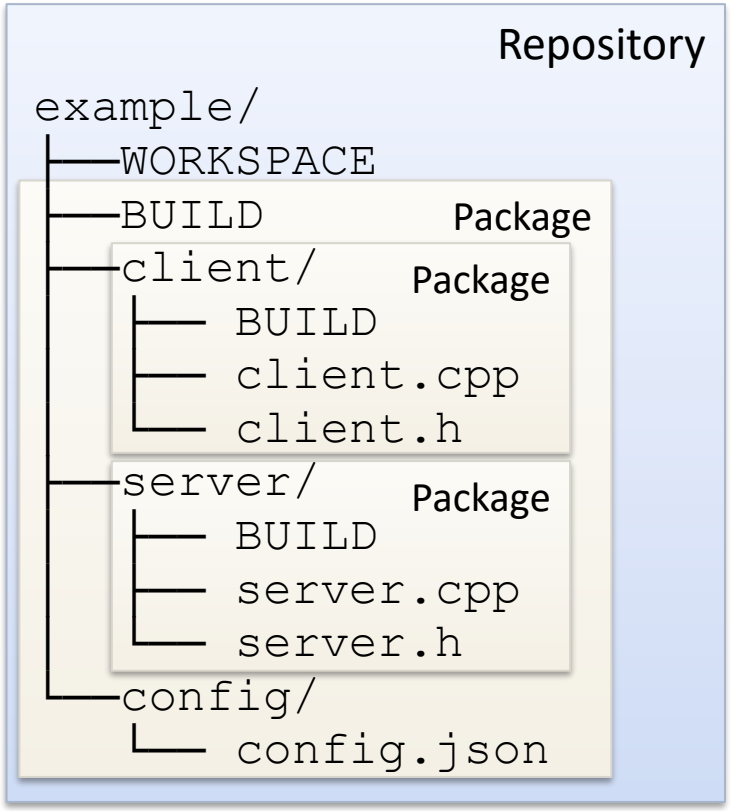


https://docs.bazel.build/versions/3.7.0/build-ref.html#concepts-and-terminology

BAZEL LABELS

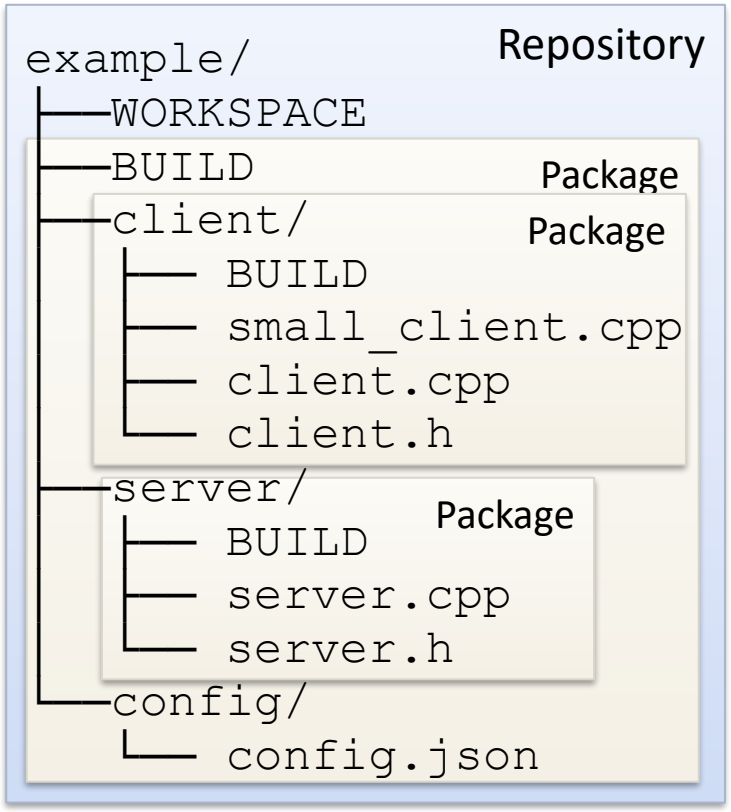
```
@repository//folder/subfolder:my_target
```

- Omitting the repository assumes the current repository:
  ```
  @repository//folder/subfolder:my_target
  //folder/subfolder:my_target
  ```
- Omitting the colon assumes the same name like the folder:
  ```
  //lib:lib
  //lib
  ```
- Starting with colon search for lib in the same BUILD file
  ```
  :lib
  ```



https://docs.bazel.build/versions/3.7.0/build-ref.html#labels

BAZEL LABELS



- From the same or another file:

```
@example//client:small_client
//client:small_client

@example//client:client
@example//client
//client:client
//client
```

- From the same BUILD file:

```
:client
:small_client
```

example/client/BUILD

```
...
cc_library(
    name = "small_client",
    ...
)
cc_library(
    name = "client",
    ...
)
```

https://docs.bazel.build/versions/3.7.0/build-ref.html#labels

TARGET VISIBILITY

```
cc_library(
    name = "my_lib",
    srcs = ["my_lib.cpp"],
    visibility = [
        "//client:__subpackages__",
    ],
    hdrs = ["my_lib.h"],
)
```

- Private: Visible only from the same BUILD file
  `//visibility:private`
- Public: Anyone can see this target
  `//visibility:public`
- Visible by a specific package and subpackages
  `//foo/bar:__subpackages__`
- Visible by a specific package but not subpackages
  `//foo/bar:__pkg__`

- Visibility can be defined per package, per target or both
- By default the target visibility is the same like the package
- If visibility is not defined, a target is only visible within the BUILD file

https://docs.bazel.build/versions/3.7.0/visibility.html

PHASES OF A BUILD

- Loading phase
  - Load extensions, BULD files, transitive dependencies
  - Duration: Several seconds the first time, faster afterwards thanks to caching
- Analysis phase
  - Semantic analysis of each rule
  - Building of the dependency graph
  - Analyze what work needs to be done
  - Duration: Several seconds the first time, faster afterwards thanks to caching
- Execution phase
  - Targets are built: compilation, linking, etc.
  - Execution of targets, tests running, etc.
  - Duration: Most of the time is spend in the execution phase, also can be speeded up thanks to caching

https://docs.bazel.build/versions/3.7.0/guide.html#phases-of-a-build

BAZEL CACHE

- Bazel provides four different levels of cache
  - In memory cache (Bazel server)
    - Lost/cleaned once the Bazel server is stopped
  - Output directory (bazel-out)
    - Local to the workspace
    - Can be removed with `bazel clean` and `bazel clean --expunge`
  - Disk cache (folder in local machine)
    - Useful to share artifacts when switching branches
    - Useful if you have multiple workspaces/checkouts of the same project
    - It can make your disk usage grow a lot
  - Remote cache (HTTP/1.1 server)
    - Useful to share artifacts between team members or with the CI

https://docs.bazel.build/versions/3.7.0/remote-caching.html

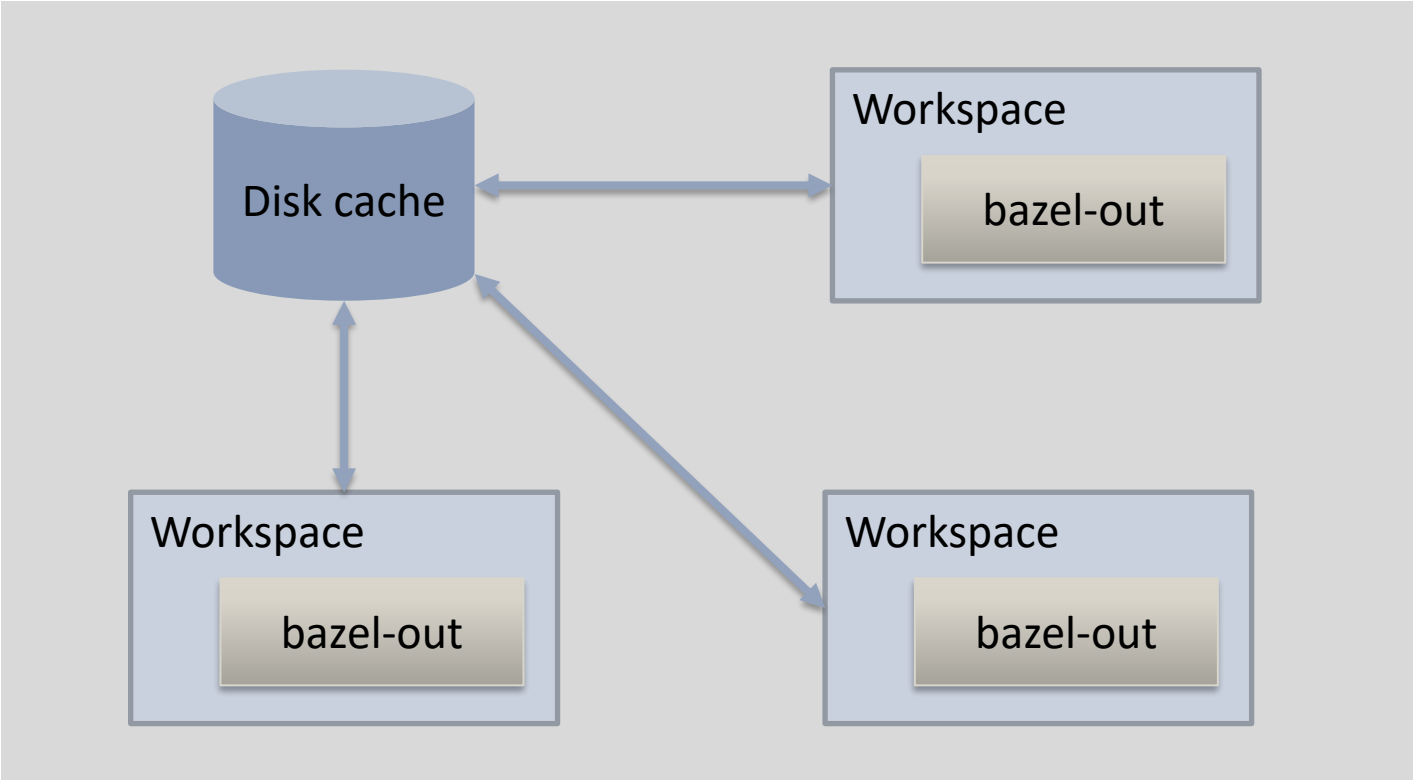
BAZEL CACHE

- Bazel cache "is always" valid and correct
  - The only reason why cache should be removed is to free space
  - Every time you solve a problem with `bazel clean`, a bug ticket should be created
    - It might be Bazel core:
      - https://github.com/bazelbuild/bazel/issues
    - It might be on some Bazel rules:
      - https://github.com/bazelbuild/rules_go/issues
      - https://github.com/bazelbuild/rules_python
      - https://github.com/bazelbuild/rules_docker
      - ...
    - It might be in one of your toolchain configurations
    - It might be in one of your Bazel extensions

https://docs.bazel.build/versions/3.7.0/remote-caching.html

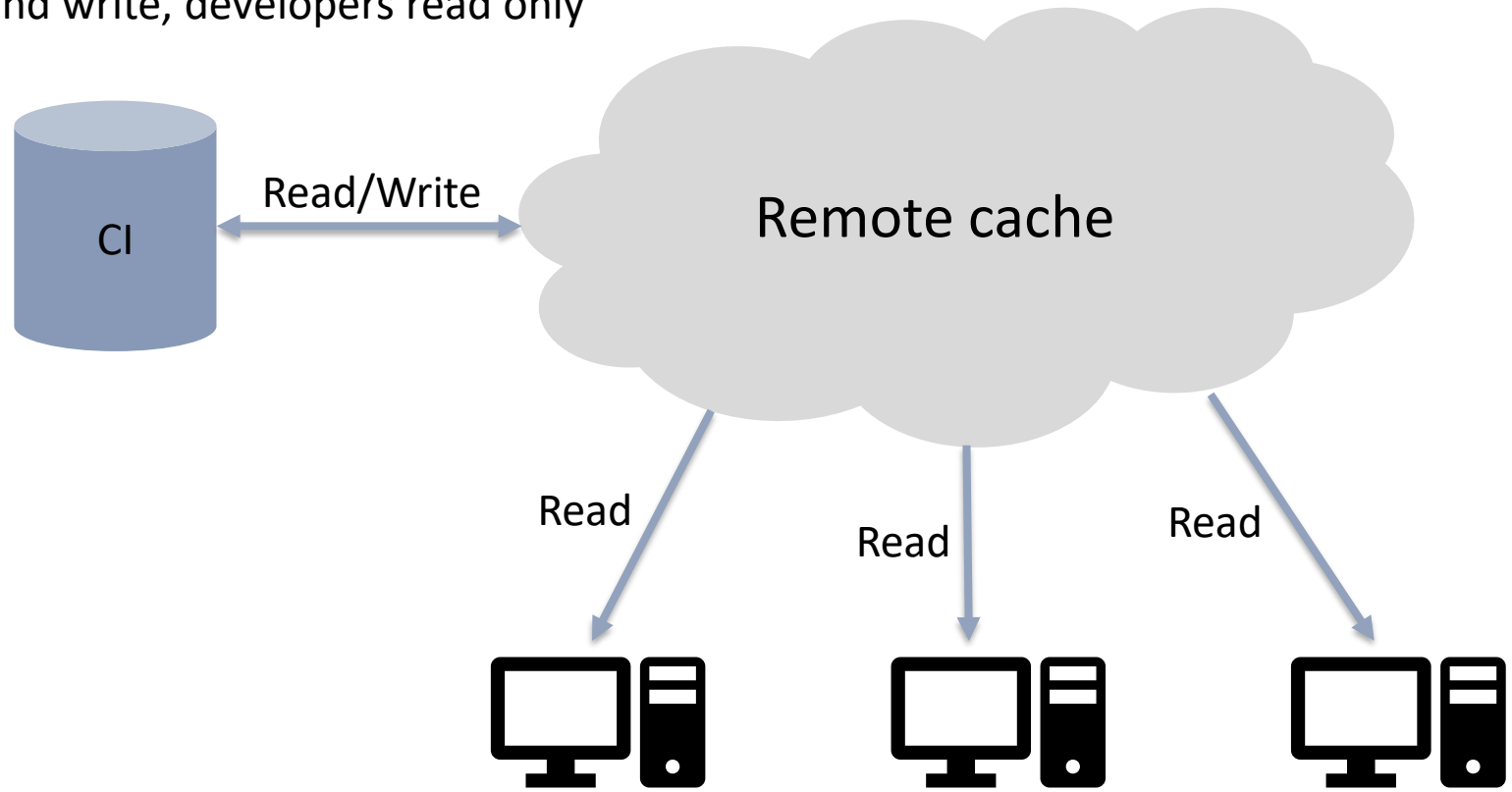
BAZEL LOCAL CACHE SETUP

- Common setup:



BAZEL REMOTE CACHE SETUP

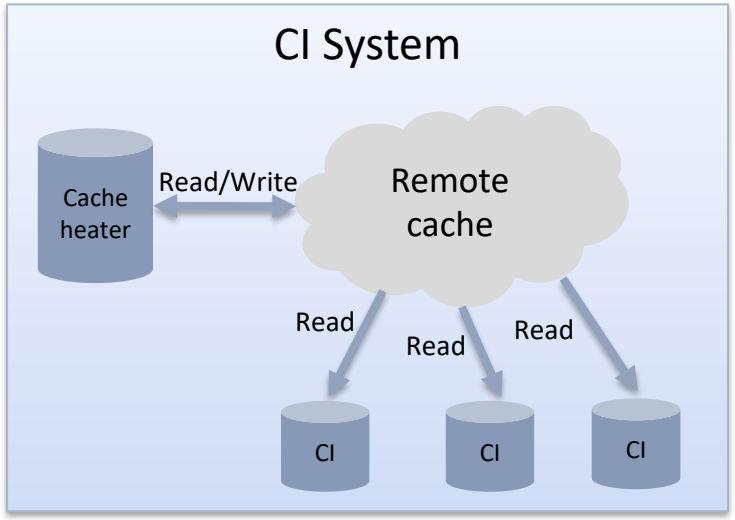
- Recommended by Bazel:
  - CI read and write, developers read only

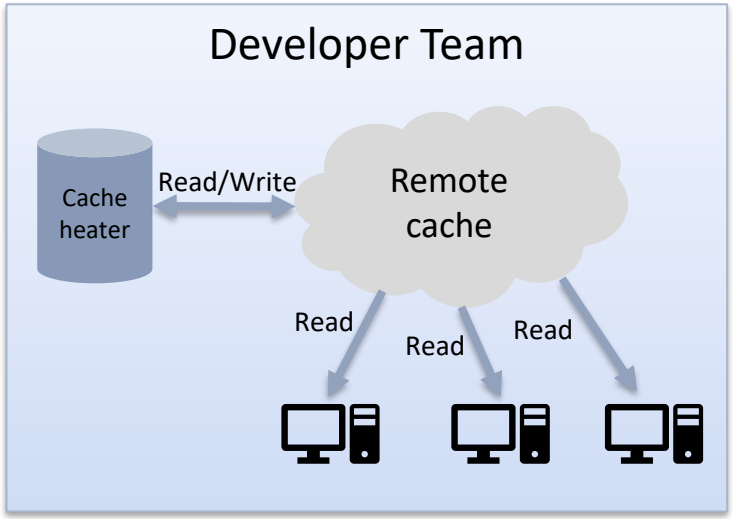


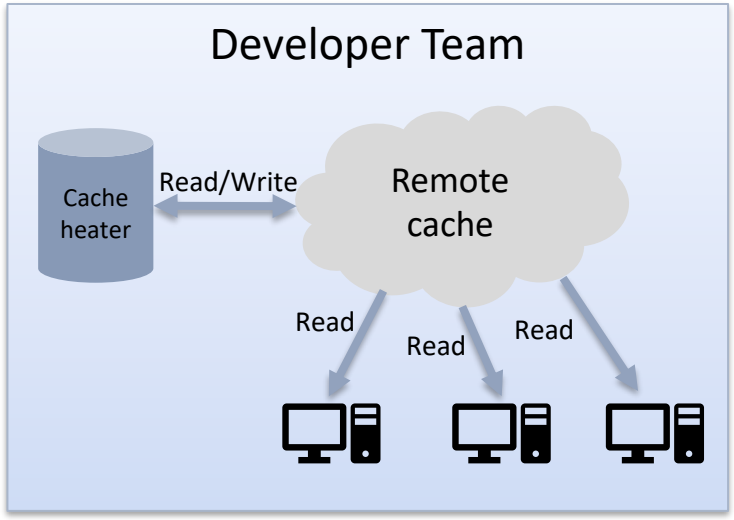
https://archive.fosdem.org/2018/schedule/event/datacenter_build/

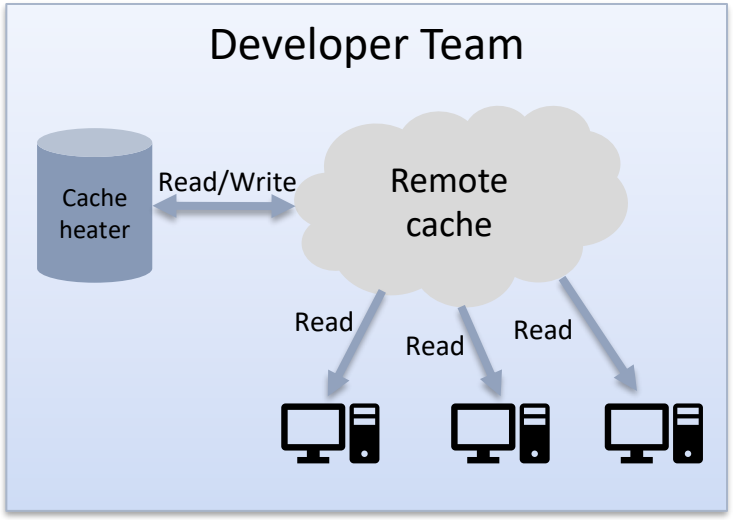
BAZEL REMOTE CACHE SETUP

- A more distributed approach:
  - One remote cache per developer team
  - One remote cache for CI only
  - On each remote cache one single entity writes to it
  - In addition each developer uses disk cache



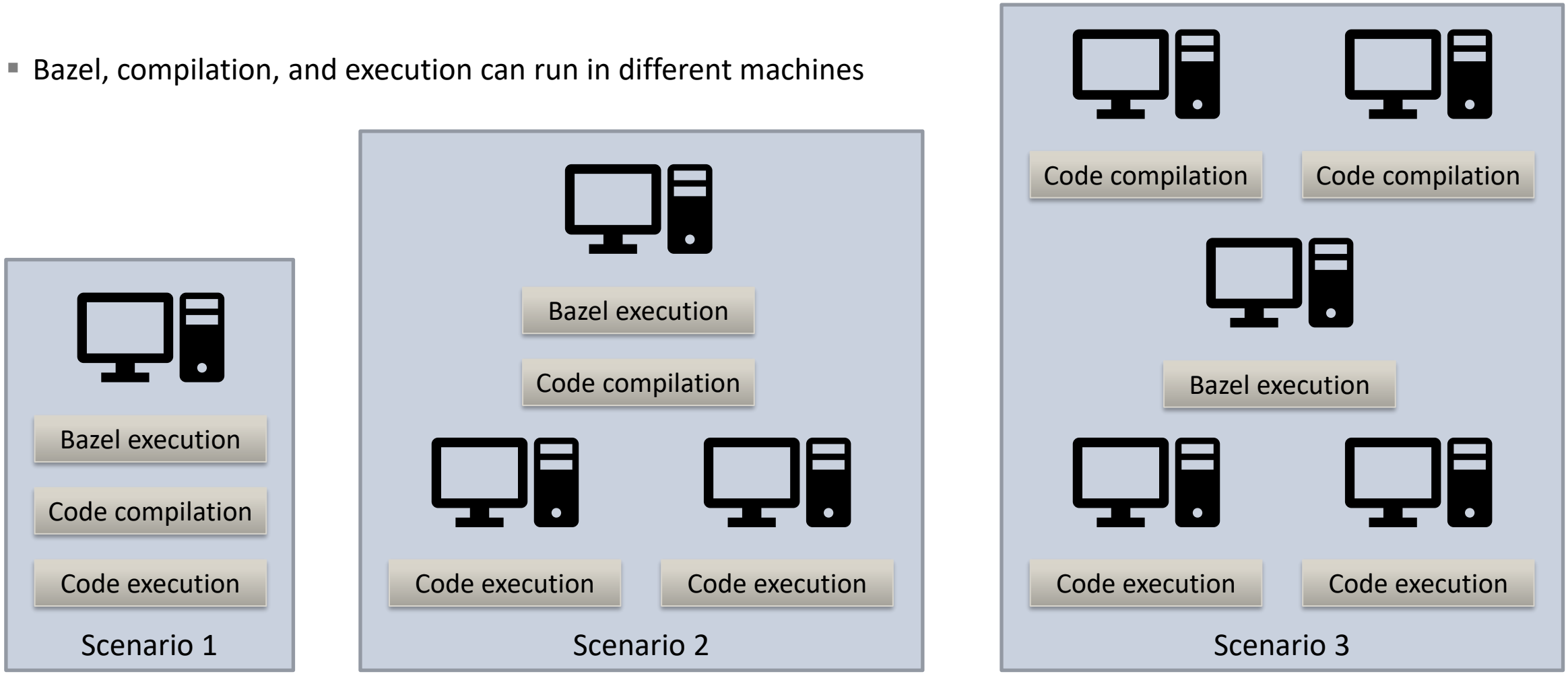






BAZEL REMOTE EXECUTION

- Bazel, compilation, and execution can run in different machines



https://docs.bazel.build/versions/3.7.0/remote-execution.html

BAZEL FOR C++ AND PYTHON

- Basic rules for C++

```
load("@rules_cc//cc:defs.bzl", "cc_binary", "cc_library", "cc_test")
```

- Basic rules for Python

```
load("@rules_python//python:defs.bzl", "py_binary", "py_library", "py_test")
```

- Invoking Bazel:

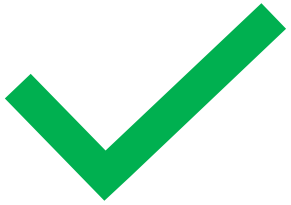
```
> bazel run //:my_binary
> bazel build //:my_binary_library_or_test
> bazel test //:my_test
```

- Also with wildcards:

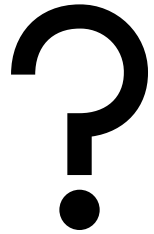
```
> bazel build //...
> bazel build //folder/subfolder/...
> bazel test //...
> bazel test //folder/subfolder/...
```

BUILD SYSTEM FOR PYTHON?

Compiled language



Interpreted language



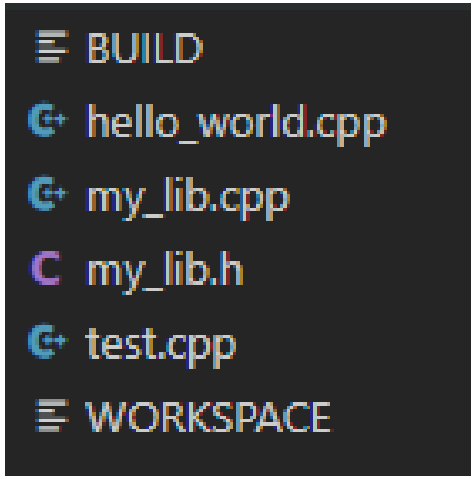
- More difficult to leak dependencies with sandboxing
- Unified way to run your targets across languages
- You benefit from Bazel test utilities
- Apart from `bazel build`, you still need all the rest

C++ EXAMPLE WITH BAZEL

```
load("@rules_cc//cc:defs.bzl", "cc_binary", "cc_library")

cc_binary(
    name = "hello_world",
    srcs = ["hello_world.cpp"],
    deps = [":my_lib"],
)

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




```
> bazel run //:hello_world
> bazel build //:hello_world
> bazel build //:my_lib
```

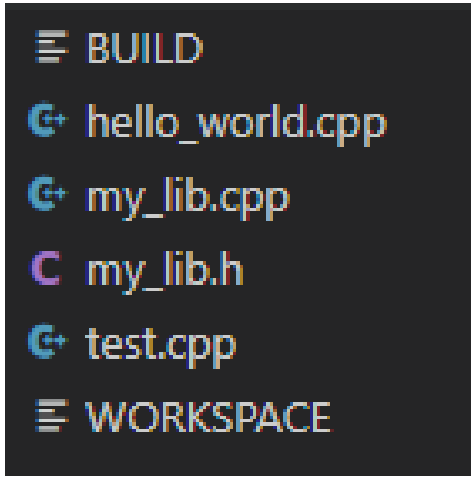
https://github.com/limdor/bazel-examples/tree/master/cpp

C++ EXAMPLE WITH BAZEL

```
load("@rules_cc//cc:defs.bzl", "cc_library", "cc_test")

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

cc_test(
    name = "my_test",
    srcs = ["test.cpp"],
    deps = [":my_lib"],
)
```



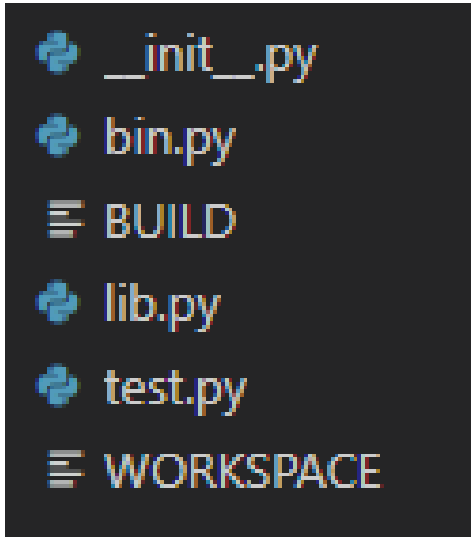

```
> bazel test //:my_test
//:my_test      PASSED in 0.0s
```

https://github.com/limdor/bazel-examples/tree/master/cpp

PYTHON EXAMPLE WITH BAZEL

```
load("@rules_python//python:defs.bzl", "py_binary", "py_library")

py_binary(
    name = "bin",
    srcs = ["bin.py"],
    deps = [":lib"],
)
py_library(
    name = "lib",
    srcs = [
        "__init__.py",
        "lib.py",
    ],
)
```



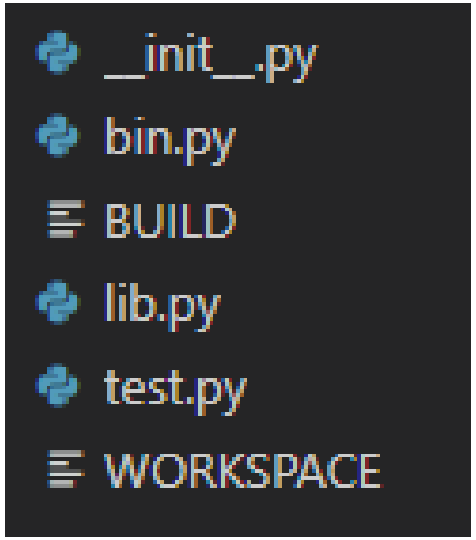

```
> bazel run //:bin
> bazel build //:my_lib
Target //:lib up-to-date (nothing to build)
> bazel run //:my_lib
ERROR: Cannot run target //:lib: Not executable
```

https://github.com/limdor/bazel-examples/tree/master/python

PYTHON EXAMPLE WITH BAZEL

```
load("@rules_python//python:defs.bzl", "py_library", "py_test")

py_library(
    name = "lib",
    srcs = [
        "__init__.py",
        "lib.py",
    ],
)
py_test(
    name = "test",
    srcs = ["test.py"],
    deps = [":lib"],
)
```

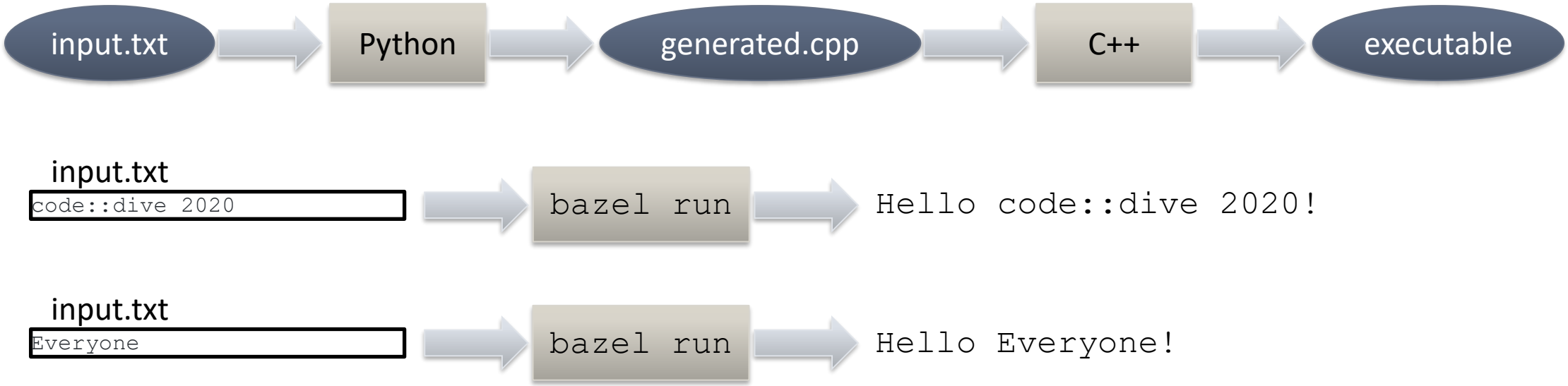



```
> bazel test //:test
//:test        PASSED in 0.0s
```

https://github.com/limdor/bazel-examples/tree/master/python

COMPILE C++ CODE GENERATED WITH PYTHON

- Bazel can be extended using macros and/or rules
  - https://docs.bazel.build/versions/3.7.0/skylark/macros.html
  - https://docs.bazel.build/versions/3.7.0/skylark/rules.html
- Example:



https://github.com/limdor/bazel-examples/tree/master/cpp_and_python

COMPILE C++ CODE GENERATED WITH PYTHON

- The generator:
  - generator.py and BUILD file

```python
import argparse
import os

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("input_file")
    parser.add_argument("output_file")
    args = parser.parse_args()
    hello_world_message = "World"
    with open(args.input_file, 'r') as message_file:
        hello_world_message = message_file.readline()
    with open(args.output_file, 'w') as output_file:
        output_file.write('#include <iostream>\n')
        output_file.write('\n')
        output_file.write('int main()\n')
        output_file.write('{\n')
        output_file.write(f'    std::cout << "Hello {hello_world_message}!" << std::endl;\n')
        output_file.write('}\n')

if __name__ == "__main__":
    main()
```

```
load("@rules_python//python:defs.bzl", "py_binary")

py_binary(
    name = "generator",
    srcs = ["generator.py"],
    visibility = ["//visibility:public"],
)
```

https://github.com/limdor/bazel-examples/tree/master/cpp_and_python

COMPILE C++ CODE GENERATED WITH PYTHON

- The macro definition:
  - generator.bzl

```
"""
Macro to generate a hello world cpp file
"""

def hello_world(name, visibility = None):
    native.genrule(
        name = name,
        srcs = [name + ".txt"],
        outs = [name + ".cpp"],
        cmd = "$(location //hello_world:generator) $< $@",
        tools = ["//hello_world:generator"],
        visibility = visibility,
    )
```

- The macro invocation:
  - BUILD

```
load("@rules_cc//cc:defs.bzl", "cc_binary")
load("//hello_world:generator.bzl", "hello_world")

hello_world(
    name = "code_dive",
)

hello_world(
    name = "everyone",
)

cc_binary(
    name = "hello_world_code_dive",
    srcs = [":code_dive"],
)

cc_binary(
    name = "hello_world_everyone",
    srcs = [":everyone"],
)
```

https://github.com/limdor/bazel-examples/tree/master/cpp_and_python

BAZEL TOOLCHAINS

- Wait! If it is so hermetic, why we did not have to specify our compiler?
  - For practical reasons Bazel provide some predefined toolchains that can be used if some compilers are installed in your machine
  - The option `--toolchain_resolution_debug` can be used to see what toolchains are being used
    `Selected toolchain @local_config_cc//:cc-compiler-k8`
  - Still for any production project, you should be defining an hermetic toolchain
    - Defining a toolchain is not enough if it points to a locally installed compiler
    - Compilers and linkers should be provided like any input artifact
    - Python interpreter should be provided like any input artifact
    - If a user needs to install something in his machine apart from Bazel, you are doing something wrong
  - Bazel provide a lot of documentation on how to define toolchains
    - https://docs.bazel.build/versions/3.7.0/tutorial/cc-toolchain-config.html
    - https://docs.bazel.build/versions/3.7.0/toolchains.html

https://docs.bazel.build/versions/master/tutorial/cc-toolchain-config.html

BAZEL TEST RUNNER

- All testing infrastructure provided by Bazel is language agnostic
- When using wildcards it runs all rules *_test (cc_test, py_test, etc.)
  `bazel test //...`
- Except the ones that have `manual` in the `tags` parameter (deactivated)
- The ones with `flaky` flag set to `True`, will be rerun automatically if they fail
- Tests will timeout depending on the value in `timeout` and `size` parameters
- Tests run in parallel unless `exclusive` is specified in the `tags`
- The output of the test is stored to a file and displayed to the console in case that it fails
- The output can be showed interactively specifying `--test_output=streamed`, but then the tests do not run in parallel
- If a library sets the `testonly` parameter, that library can only be used by testonly targets
- If a test or its dependencies did not changed, the test will not be executed
  `//:my_test (cached) PASSED in 0.0s`

```
cc_test(
    name = "deactivated_test",
    tags = ["manual"],
    ...
)

py_test(
    name = "flaky_test",
    flaky = True,
    ...
)

sh_test(
    name = "flaky_test",
    size = True,
    timeout = "short",
    size = "small",
    ...
)

cc_library(
    name = "my_test_lib",
    srcs = ["my_lib.cpp"],
    testonly = True,
    hdrs = ["my_lib.h"],
)
```

https://docs.bazel.build/versions/3.7.0/test-encyclopedia.html

BAZEL TEST RUNNER

- In the end Bazel prints a summary of all executed tests

```
//:my_cpp_test                (cached) PASSED in 0.0s
//:my_python_test                      PASSED in 0.0s
//:my_flaky_test                       FLAKY, failed in 2 out of 3 in 2.0s
  Stats over 3 runs: max = 2.0s, min = 2.0s, avg = 2.0s, dev = 0.0s
  /path/to/the/teslogs/folder/my_flaky_test/test_attempts/attempt_1.log
  /path/to/the/teslogs/folder/my_flaky_test/test_attempts/attempt_2.log
//:my_fail_test                        FAILED in 2.0s
  /path/to/the/teslogs/folder/my_fail_test/test.log
//:my_build_errors_test                FAILED TO BUILD
//:my_too_long_test                    TIMEOUT in 60.0s
  /path/to/the/teslogs/folder/my_too_long_test/test.log
```

https://docs.bazel.build/versions/3.7.0/test-encyclopedia.html

BAZEL COVERAGE

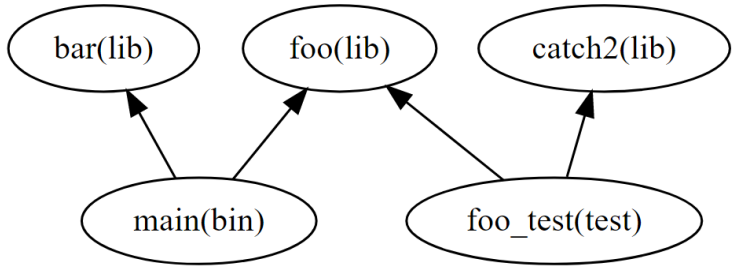
- Bazel provides a command line to compute code coverage
  `bazel coverage //:foo_test`
- Runs the tests and collects coverage information
- Afterwards, a report can be generated
  `genhtml --output-directory coverage-report bazel-testlogs/foo_test/coverage.dat`

LCOV - code coverage report

		Hit	Total	Coverage
Current view: top level - cpp_coverage				
Test: coverage.dat	Lines:	4	5	80.0 %
Date: 2020-10-12 20:02:51	Functions:	1	1	100.0 %

Filename	Line Coverage		Functions	
foo.cpp	80.0 %	4 / 5	100.0 %	1 / 1

Generated by: LCOV version 1.14



- Baseline coverage is not supported at the moment out of the box
  - https://github.com/bazelbuild/bazel/issues/5716

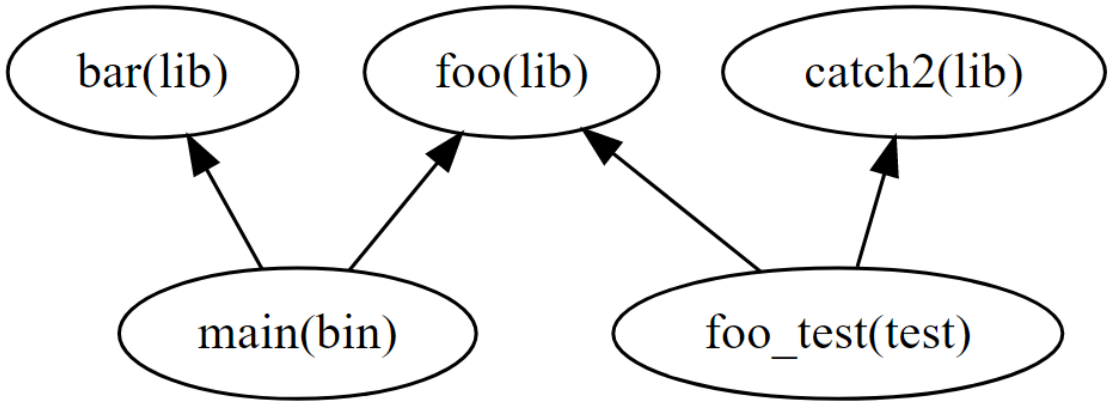
https://github.com/limdor/bazel-examples/tree/master/cpp_coverage

BAZEL QUERY

- Bazel provides three commands to understand the build graph

```
bazel query //:foo_test
bazel cquery //:foo_test
bazel aquery //:foo_test
```

-
```
> bazel query "deps(//:foo_test)" --notool_deps --noimplicit_deps
//:foo_test
@catch2//:catch2
@catch2//:single_include/catch2/catch.hpp
//:foo_test.cpp
//:foo
//:foo.h
//:foo.cpp
```



https://docs.bazel.build/versions/3.7.0/query-how-to.html

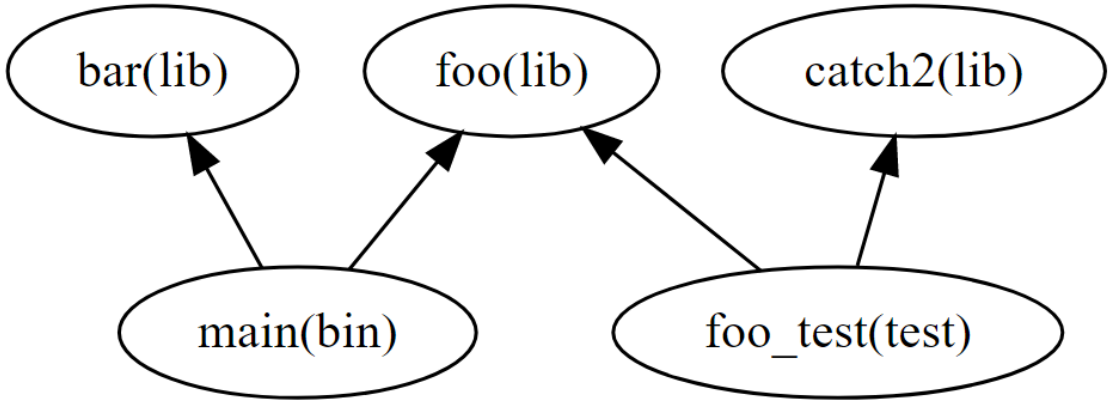
BAZEL QUERY

- How foo_test depends on foo.h?

```
> bazel query "somepath(//:foo_test, //:foo.h)" --notool_deps --noimplicit_deps
//:foo_test
//:foo
//:foo.h
```

- How foo_test depends on bar library?

```
> bazel query "somepath(//:foo_test, //:bar)" --notool_deps --noimplicit_deps
INFO: Empty results
```



https://docs.bazel.build/versions/3.7.0/query-how-to.html

BAZELISK

- It is a launcher for Bazel
- It allows to use multiple Bazel versions in one machine/workspace
- Developers do not need to care about upgrading Bazel
- Some ways that can be used:
  - Define an environment variable called USE_BAZEL_VERSION specifying the version to be used
  - Check in a file with the name .bazelrc to your repository containing the version to be used
  - If no version is specified, it will always use the latest Bazel version released
- It can be run like Bazel

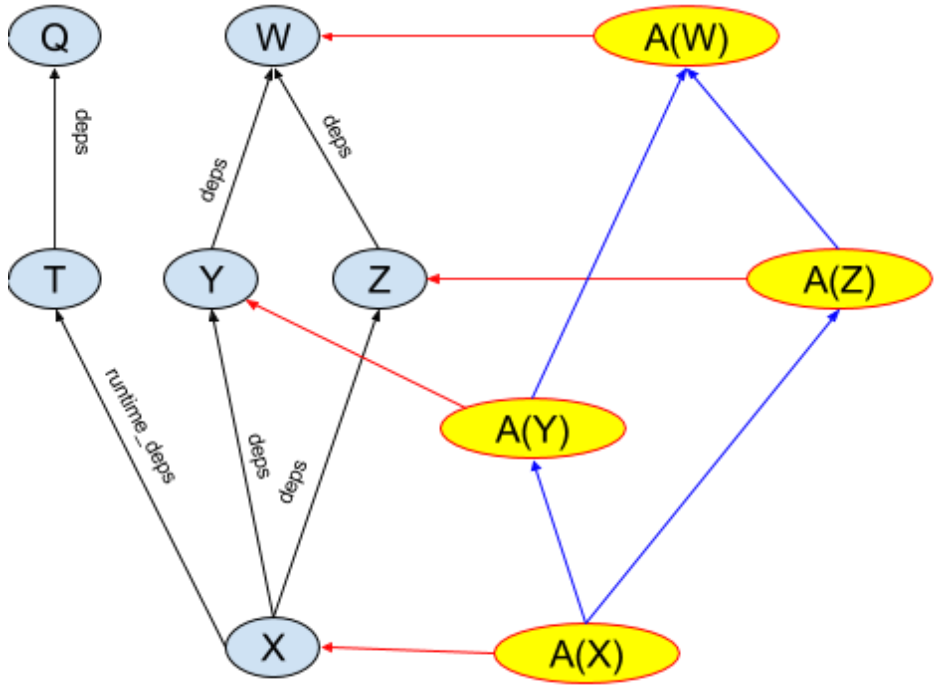
```
> bazelisk run //:hello_world
```

- Or you can put it to your binary path named as `bazel`
- When running it with `--strict` and `--migrate` can help in the migration process to a newer Bazel version

https://github.com/bazelbuild/bazelisk

FOOD FOR THOUGHT

- Bazel provide also aspects
  - Allows to add additional information to the dependency graph
  - For what could be used?
    - IDE integration
    - Static analysis
    - Code coverage
    - Compiler warnings
    - …



https://docs.bazel.build/versions/3.7.0/skylark/aspects.html

BAZEL COMMANDS

- `bazel help` Show all commands, can be used with a specific command (`bazel help build`)
- `bazel version` Show Bazel version, can be different per workspace if using bazelisk
- `bazel build` Build targets, provides the option to run only loading and analysis phase
- `bazel run` Should be used to run the targets, takes care of runtime dependencies
- `bazel test` Basic command to execute your tests with a lot of helpful options
- `bazel coverage` Should be used to compute code coverage, not fully supported for all languages
- `bazel query` Retrieve information from the build graph without running the analysis phase
- `bazel cquery` Retrieve information from the build graph after running the analysis phase
- `bazel aquery` Allow you to query information regarding the actions in the build graph
- `bazel clean` If you have to run it, there is a bug somewhere

https://docs.bazel.build/versions/master/command-line-reference.html#commands

ADDITIONAL REFERENCES

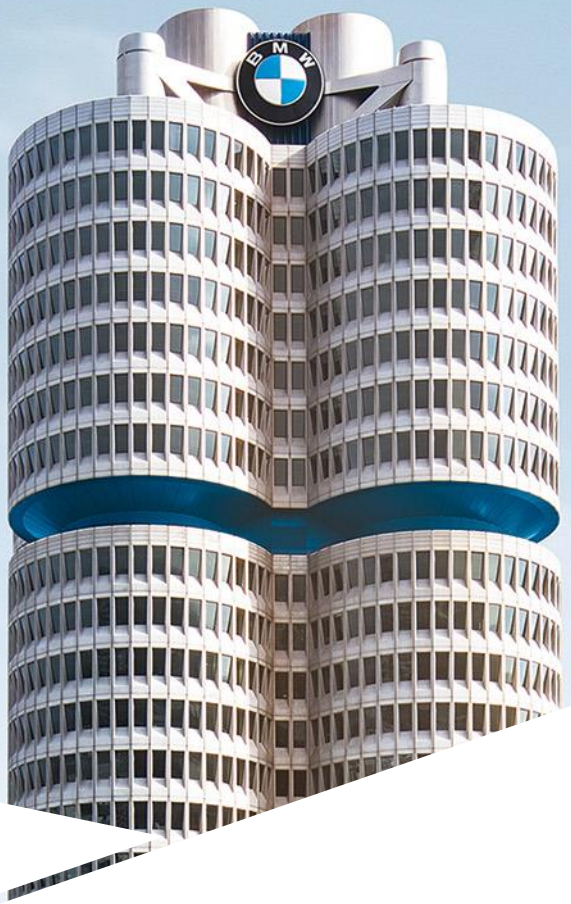
- Bazel – CppCast
  Lukács Berki and Julio Merino (Google - Bazel team)
  https://cppcast.com/bazel/

- Building Self Driving Cars with Bazel – BazelCon 2019
  Axel Uhlig and Patrick Ziegler (BMW Group)
  https://youtu.be/Gh4SJuYUoQI

- Collecting Code Coverage with Bazel – BazelCon 2018
  Irina Iancu (Google - Bazel team)
  https://youtu.be/P51Rgcbxhyk

INTRODUCTION TO BAZEL TO BUILD C++ AND PYTHON

THANKS FOR LISTENING!



Xavier Bonaventura
code::dive 2020 - 18/11/2020

BMW GROUP THE NEXT 100 YEARS

Rolls-Royce Motor Cars Limited