Documentation

☰ NAVIGATION

A User's Guide to Bazel

Edit (https://github.com/bazelbuild/bazel/tree/master/site/docs/user-manual.html)

Bazel overview

To run Bazel, go to your base workspace (build-ref.html#workspaces) directory or any of its subdirectories and type `bazel`.

```
% bazel help
                           [Bazel release bazel-<version>]
Usage: bazel <command> <options> ...

Available commands:
  analyze-profile     Analyzes build profile data.
  build               Builds the specified targets.

  canonicalize-flags  Canonicalize Bazel flags.
  clean               Removes output files and optionally stops the server.

  dump                Dumps the internal state of the Bazel server process.

  help                Prints help for commands, or the index.
  info                Displays runtime info about the bazel server.

  fetch               Fetches all external dependencies of a target.
  mobile-install      Installs apps on mobile devices.

  query               Executes a dependency graph query.

  run                 Runs the specified target.
  shutdown            Stops the Bazel server.
  test                Builds and runs the specified test targets.
  version             Prints version information for Bazel.

Getting more help:
  bazel help <command>
                   Prints help and options for <command>.
  bazel help startup_options
                   Options for the JVM hosting Bazel.
  bazel help target-syntax
                   Explains the syntax for specifying targets.
  bazel help info-keys
                   Displays a list of keys used by the info command.
```

The `bazel` tool performs many functions, called commands; users of CVS and Subversion will be familiar with this "Swiss army knife" arrangement. The most commonly used one is of course `bazel build`. You can browse the online help messages using `bazel help`.

Client/server implementation

The Bazel system is implemented as a long-lived server process. This allows it to perform many optimizations not possible with a batch-oriented implementation, such as caching of BUILD files, dependency graphs, and other metadata from one build to the next. This improves the speed of incremental builds, and allows different commands, such as `build` and `query` to share the same cache of loaded packages, making queries very fast.

When you run `bazel`, you're running the client. The client finds the server based on the output base, which by

default is determined by the path of the base workspace directory and your userid, so if you build in multiple workspaces, you'll have multiple output bases and thus multiple Bazel server processes. Multiple users on the same workstation can build concurrently in the same workspace because their output bases will differ (different userids). If the client cannot find a running server instance, it starts a new one. The server process will stop after a period of inactivity (3 hours, by default).

For the most part, the fact that there is a server running is invisible to the user, but sometimes it helps to bear this in mind. For example, if you're running scripts that perform a lot of automated builds in different directories, it's important to ensure that you don't accumulate a lot of idle servers; you can do this by explicitly shutting them down when you're finished with them, or by specifying a short timeout period.

The name of a Bazel server process appears in the output of `ps x` or `ps -e f` as `bazel(`*`dirname`*`)`, where *dirname* is the basename of the directory enclosing the root your workspace directory. For example:

```
% ps -e f
16143 ?        Sl     3:00 bazel(src-johndoe2) -server -Djava.library.path=...
```

This makes it easier to find out which server process belongs to a given workspace. (Beware that with certain other options to `ps`, Bazel server processes may be named just `java`.) Bazel servers can be stopped using the shutdown command.

You can also run Bazel in batch mode using the `--batch` startup flag. This will immediately shut down the process after the command (build, test, etc.) has finished and not keep a server process around.

When running `bazel`, the client first checks that the server is the appropriate version; if not, the server is stopped and a new one started. This ensures that the use of a long-running server process doesn't interfere with proper versioning.

`.bazelrc`, the Bazel configuration file, the `--bazelrc=`*`file`* option, and the `--config=`*`value`* option

Bazel accepts many options. Typically, some of these are varied frequently (e.g. `--subcommands`) while others stay the same across several builds (e.g. `--package_path`). To avoid having to specify these unchanged options for every build (and other commands) Bazel allows you to specify options in a configuration file.

Where are `.bazelrc` files?

Bazel looks for an optional configuration file in the following locations, in order. It will stop searching once it has successfully found a file.

1. The path specified by the `--bazelrc=`*`file`* startup option. If specified, this option must appear *before* the command name (e.g. `build`)
2. A file named `.bazelrc` in your base workspace directory
3. A file named `.bazelrc` in your home directory

The option `--bazelrc=/dev/null` effectively disables the use of a configuration file. We strongly recommend that you use this option when performing release builds, or automated tests that invoke Bazel.

Aside from the optional configuration file described above, Bazel also looks for a master rc file named `bazel.bazelrc` next to the binary, in the workspace at `tools/bazel.rc` or system-wide at

`/etc/bazel.bazelrc`. These files are here to support installation-wide options or options shared between users. These files do not override one another; if all of these files exist, all of them will be loaded. Reading of these files can be disabled using the `--nomaster_bazelrc` option.

`.bazelrc` syntax and semantics

Like all UNIX "rc" files, the `.bazelrc` file is a text file with a line-based grammar. Lines starting `#` are considered comments and are ignored, as are blank lines. Each line contains a sequence of words, which are tokenized according to the same rules as the Bourne shell.

Imports

Lines that start with `import` are special: if Bazel encounters such a line in a `.bazelrc` file, it parses the contents of the file referenced by the import statement, too. Options specified in an imported file take precedence over options specified before the import statement. Options specified after the import statement take precedence over the options in the imported file. Options in files imported later take precedence over files imported earlier. To specify a path that is relative to the workspace root, write `import %workspace%/path/to/bazelrc`.

Option defaults

Most lines of a bazelrc define default option values. The first word on each line specifies when these defaults are applied:

1. `startup`: startup options, which go before the command, and are described in `bazel help startup_options`.
2. `common`: options that apply to all Bazel commands.
3. *`command`*: Bazel command, such as `build` or `query` to which the options apply. These options also apply to all commands that inherit from the specified command. (For example, `test` inherits from `build`.)

Each of these lines may be used more than once and the arguments that follow the first word are combined as if they had appeared on a single line. (Users of CVS, another tool with a "Swiss army knife" command-line interface, will find the syntax familiar to that of `.cvsrc`.)

Options specified in the command line always take precedence over those from a configuration file. Within the configuration file, precedence is given by specificity. This means that lines for a more specific command take precedence over lines for a less specific command, with `common` getting lowest precedence (for example, the `test` command inherits all the options from the `build` command, so the line `test --foo=bar` takes precedence over the line `build --foo=baz` line, regardless of which rc file or what order these two lines are in). Two lines specifying options for the same command at equal specificity are parsed in the order in which they appear within the file. The user-specific configuration file takes precedence over the master file.

Because this precedence rule does not match the file order, we recommend that the file follows the same order, with `common` options at the top, and most-specific commands near the bottom. This way, the order in which the options are read is the same as the order in which they are applied, which is more intuitive.

The arguments specified on line of an rc file may include arguments that are not options, such as the names of build targets, and so on. These, like the options specified in the same files, have lower precedence than their siblings on the command line, and are always prepended to the explicit list of non- option arguments.

`--config`

In addition to setting option defaults, the rc file can be used to group options and provide a shorthand for common groupings. This is done by adding a `:name` suffix to the command. These options are ignored by default, but will be included when the option `--config=`*`name`* is present, either on the command line or in a `.bazelrc` file, recursively, even inside of another config definition. Only the options specified by `command:name` for applicable commands will be expanded, in the precedence order described above.

Note that configs can be defined in any `.bazelrc` file, and that all lines of the form `command:name` (for applicable commands) will be expanded, across the different rc files. In order to avoid name conflicts, we suggest that configs defined in personal rc files start with an underscore ('_') to avoid unintentional name sharing.

The expansion behavior for these `--config=foo` options has changed. The legacy behavior, still the default, is to expand these in a fixed point expansion after all default rc options are loaded. This is unintuitive and has caused debugging difficulties in the past. The new behavior is to expand `--config=foo` to the options it expands to "in-place" so that the options specified for the config have the same precedence that the `--config=foo` option had. This is more intuitive, and can be enabled using the startup flag `--expand_configs_in_place`, which can be included in a bazelrc file on a `startup` line.

Example

Here's an example `~/.bazelrc` file:

```
# Bob's Bazel option defaults

startup --batch --host_jvm_args=-XX:-UseParallelGC
import /home/bobs_project/bazelrc
build --show_timestamps --keep_going --jobs 600
build --color=yes
query --keep_going

# Definition of --config=memcheck
build:memcheck --strip=never --test_timeout=3600
```

Building programs with Bazel

The `build` command

The most important function of Bazel is, of course, building code. Type `bazel build` followed by the name of the target you wish to build. Here's a typical session:

```
% bazel build //foo
____Loading package: foo
____Loading package: bar
____Loading package: baz
____Loading complete.  Analyzing...
____Building 1 target...
____[0 / 3] Executing Genrule //bar:helper_rule
____[1 / 3] Executing Genrule //baz:another_helper_rule
____[2 / 3] Building foo/foo.bin
Target //foo:foo up-to-date:
  bazel-bin/foo/foo.bin
  bazel-bin/foo/foo
____Elapsed time: 9.905s
```

Bazel prints the progress messages as it loads all the packages in the transitive closure of dependencies of the requested target, then analyzes them for correctness and to create the build actions, finally executing the compilers and other tools of the build.

Bazel prints progress messages during the execution phase of the build, showing the current build step (compiler, linker, etc.) that is being started, and the number of completed over total number of build actions. As the build starts the number of total actions will often increase as Bazel discovers the entire action graph, but the number will usually stabilize within a few seconds.

At the end of the build Bazel prints which targets were requested, whether or not they were successfully built, and if so, where the output files can be found. Scripts that run builds can reliably parse this output; see `--show_result` for more details.

Typing the same command again:

```
% bazel build //foo
____Loading...
____Found 1 target...
____Building complete.
Target //foo:foo up-to-date:
  bazel-bin/foo/foo.bin
  bazel-bin/foo/foo
____Elapsed time: 0.280s
```

we see a "null" build: in this case, there are no packages to re-load, since nothing has changed, and no build steps to execute. (If something had changed in "foo" or some of its dependencies, resulting in the reexecution of some build actions, we would call it an "incremental" build, not a "null" build.)

Before you can start a build, you will need a Bazel workspace. This is simply a directory tree that contains all the source files needed to build your application. Bazel allows you to perform a build from a completely read-only volume.

Setting up a `--package_path`

Bazel finds its packages by searching the package path. This is a colon separated ordered list of bazel directories, each being the root of a partial source tree.

To specify a custom package path using the `--package_path` option:

```
% bazel build --package_path %workspace%:/some/other/root
```

Package path elements may be specified in three formats:

1. If the first character is `/`, the path is absolute.
2. If the path starts with `%workspace%`, the path is taken relative to the nearest enclosing bazel directory. For instance, if your working directory is `/home/bob/clients/bob_client/bazel/foo`, then the string `%workspace%` in the package-path is expanded to `/home/bob/clients/bob_client/bazel`.
3. Anything else is taken relative to the working directory.
   This is usually not what you mean to do, and may behave unexpectedly if you use Bazel from directories below the bazel workspace. For instance, if you use the package-path element `.`, and then cd into the directory `/home/bob/clients/bob_client/bazel/foo`, packages will be resolved from the `/home/bob/clients/bob_client/bazel/foo` directory.

If you use a non-default package path, we recommend that you specify it in your Bazel configuration file for convenience.

Bazel doesn't require any packages to be in the current directory, so you can do a build from an empty bazel workspace if all the necessary packages can be found somewhere else on the package path.

Example: Building from an empty client

```
% mkdir -p foo/bazel
% cd foo/bazel
% touch WORKSPACE
% bazel build --package_path /some/other/path //foo
```

Specifying targets to build

Bazel allows a number of ways to specify the targets to be built. Collectively, these are known as *target patterns*. The on-line help displays a summary of supported patterns:

```
% bazel help target-syntax

Target pattern syntax
=====================

The BUILD file label syntax is used to specify a single target. Target
patterns generalize this syntax to sets of targets, and also support
working-directory-relative forms, recursion, subtraction and filtering.
Examples:

Specifying a single target:

  //foo/bar:wiz     The single target '//foo/bar:wiz'.
  foo/bar/wiz       Equivalent to:
                      '//foo/bar/wiz:wiz' if foo/bar/wiz is a package,
                      '//foo/bar:wiz' if foo/bar is a package,
                      '//foo:bar/wiz' otherwise.
  //foo/bar         Equivalent to '//foo/bar:bar'.

Specifying all rules in a package:

  //foo/bar:all       Matches all rules in package 'foo/bar'.

Specifying all rules recursively beneath a package:

  //foo/...:all     Matches all rules in all packages beneath directory 'foo'.
  //foo/...           (ditto)

  By default, directory symlinks are followed when performing this recursive traversal, ex
  those that point to under the output base (for example, the convenience symlinks that ar
  in the root directory of the workspace) But we understand that your workspace may intent
  contain directories with unusual symlink structures that you don't want consumed. As suc
  directory has a file named
  'DONT_FOLLOW_SYMLINKS_WHEN_TRAVERSING_THIS_DIRECTORY_VIA_A_RECURSIVE_TARGET_PATTERN' the
  in that directory won't be followed when evaluating recursive target patterns.

Working-directory relative forms:  (assume cwd = 'workspace/foo')

  Target patterns which do not begin with '//' are taken relative to
  the working directory.  Patterns which begin with '//' are always
  absolute.

  ...:all           Equivalent to  '//foo/...:all'.
  ...                 (ditto)

  bar/...:all       Equivalent to  '//foo/bar/...:all'.
  bar/...             (ditto)

  bar:wiz           Equivalent to '//foo/bar:wiz'.
  :foo              Equivalent to '//foo:foo'.
```

```
  bar               Equivalent to '//foo/bar:bar'.
  foo/bar           Equivalent to '//foo/foo/bar:bar'.

  bar:all           Equivalent to '//foo/bar:all'.
  :all              Equivalent to '//foo:all'.

Summary of target wildcards:

  :all,             Match all rules in the specified packages.
  :*, :all-targets  Match all targets (rules and files) in the specified
                      packages, including ones not built by default, such
                      as _deploy.jar files.

Subtractive patterns:

  Target patterns may be preceded by '-', meaning they should be
  subtracted from the set of targets accumulated by preceding
  patterns. (Note that this means order matters.) For example:

    % bazel build -- foo/... -foo/contrib/...

  builds everything in 'foo', except 'contrib'.  In case a target not
  under 'contrib' depends on something under 'contrib' though, in order to
  build the former bazel has to build the latter too. As usual, the '--' is
  required to prevent '-f' from being interpreted as an option.

  When running the test command, test suite expansion is applied to each target
  pattern in sequence as the set of targets is evaluated. This means that
  individual tests from a test suite can be excluded by a later target pattern.
  It also means that an exclusion target pattern which matches a test suite will
  exclude all tests which that test suite references. (Targets that would be
  matched by the list of target patterns without any test suite expansion are
  also built unless --build_tests_only is set.)
```

Whereas labels (build-ref.html#labels) are used to specify individual targets, e.g. for declaring dependencies in BUILD files, Bazel's target patterns are a syntax for specifying multiple targets: they are a generalization of the label syntax for *sets* of targets, using wildcards. In the simplest case, any valid label is also a valid target pattern, identifying a set of exactly one target.

`foo/...` is a wildcard over *packages*, indicating all packages recursively beneath directory `foo` (for all roots of the package path). `:all` is a wildcard over *targets*, matching all rules within a package. These two may be combined, as in `foo/...:all`, and when both wildcards are used, this may be abbreviated to `foo/...`.

In addition, `:*` (or `:all-targets`) is a wildcard that matches *every target* in the matched packages, including files that aren't normally built by any rule, such as `_deploy.jar` files associated with `java_binary` rules.

This implies that `:*` denotes a *superset* of `:all`; while potentially confusing, this syntax does allow the familiar `:all` wildcard to be used for typical builds, where building targets like the `_deploy.jar` is not desired.

In addition, Bazel allows a slash to be used instead of the colon required by the label syntax; this is often convenient

when using Bash filename expansion. For example, `foo/bar/wiz` is equivalent to `//foo/bar:wiz` (if there is a package `foo/bar`) or to `//foo:bar/wiz` (if there is a package `foo`).

Many Bazel commands accept a list of target patterns as arguments, and they all honor the prefix negation operator `-'. This can be used to subtract a set of targets from the set specified by the preceding arguments. (Note that this means order matters.) For example,

```
bazel build foo/... bar/...
```

means "build all targets beneath `foo` *and* all targets beneath `bar`", whereas

```
bazel build -- foo/... -foo/bar/...
```

means "build all targets beneath `foo` *except* those beneath `foo/bar`". (The `--` argument is required to prevent the subsequent arguments starting with `-` from being interpreted as additional options.)

It's important to point out though that subtracting targets this way will not guarantee that they are not built, since they may be dependencies of targets that weren't subtracted. For example, if there were a target `//foo:all-apis` that among others depended on `//foo/bar:api`, then the latter would be built as part of building the former.

Targets with `tags=["manual"]` will not be included in wildcard target patterns (..., :*, :all, etc). You should specify such test targets with explicit target patterns on the command line if you want Bazel to build/test them.

Fetching external dependencies

By default, Bazel will download and symlink external dependencies during the build. However, this can be undesirable, either because you'd like to know when new external dependendencies are added or because you'd like to "prefetch" dependencies (say, before a flight where you'll be offline). If you would like to prevent new dependencies from being added during builds, you can specify the `--fetch=false` flag. Note that this flag only applies to repository rules that do not point to a directory in the local file system. Changes, for example, to `local_repository`, `new_local_repository` and Android SDK and NDK repository rules will always take effect regardless of the value `--fetch`.

If you disallow fetching during builds and Bazel finds new external dependencies, your build will fail.

You can manually fetch dependencies by running `bazel fetch`. If you disallow during-build fetching, you'll need to run `bazel fetch`:

1. Before you build for the first time.
2. After you add a new external dependency.

Once it has been run, you should not need to run it again until the WORKSPACE file changes.

`fetch` takes a list of targets to fetch dependencies for. For example, this would fetch dependencies needed to build `//foo:bar` and `//bar:baz`:

```
$ bazel fetch //foo:bar //bar:baz
```

To fetch all external dependencies for a workspace, run:

```
$ bazel fetch //...
```

You do not need to run bazel fetch at all if you have all of the tools you are using (from library jars to the JDK itself) under your workspace root. However, if you're using anything outside of the workspace directory then Bazel will automatically run `bazel fetch` before running `bazel build`.

Build configurations and cross-compilation

All the inputs that specify the behavior and result of a given build can be divided into two distinct categories. The first kind is the intrinsic information stored in the BUILD files of your project: the build rule, the values of its attributes, and the complete set of its transitive dependencies. The second kind is the external or environmental data, supplied by the user or by the build tool: the choice of target architecture, compilation and linking options, and other toolchain configuration options. We refer to a complete set of environmental data as a **configuration**.

In any given build, there may be more than one configuration. Consider a cross-compile, in which you build a `//foo:bin` executable for a 64-bit architecture, but your workstation is a 32-bit machine. Clearly, the build will require building `//foo:bin` using a toolchain capable of creating 64-bit executables, but the build system must also build various tools used during the build itself—for example tools that are built from source, then subsequently used in, say, a genrule—and these must be built to run on your workstation. Thus we can identify two configurations: the **host configuration**, which is used for building tools that run during the build, and the **target configuration** (or *request configuration*, but we say "target configuration" more often even though that word already has many meanings), which is used for building the binary you ultimately requested.

Typically, there are many libraries that are prerequisites of both the requested build target (`//foo:bin`) and one or more of the host tools, for example some base libraries. Such libraries must be built twice, once for the host configuration, and once for the target configuration.
Bazel takes care of ensuring that both variants are built, and that the derived files are kept separate to avoid interference; usually such targets can be built concurrently, since they are independent of each other. If you see progress messages indicating that a given target is being built twice, this is most likely the explanation.

Bazel uses one of two ways to select the host configuration, based on the `--distinct_host_configuration` option. This boolean option is somewhat subtle, and the setting may improve (or worsen) the speed of your builds.

`--distinct_host_configuration=false`

When this option is false, the host and request configurations are identical: all tools required during the build will be built in exactly the same way as target programs. This setting means that no libraries need to be built twice during a single build, so it keeps builds short. However, it does mean that any change to your request configuration also affects your host configuration, causing all the tools to be rebuilt, and then anything that depends on the tool output to be rebuilt too. Thus, for example, simply changing a linker option between builds might cause all tools to be re-linked, and then all actions using them reexecuted, and so on, resulting in a very large rebuild. Also, please note: if your host architecture is not capable of running your target binaries, your build will not work.

If you frequently make changes to your request configuration, such as alternating between `-c opt` and `-c dbg` builds, or between simple- and cross-compilation, we do not recommend this option, as you will typically rebuild the majority of your codebase each time you switch.

`--distinct_host_configuration=true` *(default)*

If this option is true, then instead of using the same configuration for the host and request, a completely distinct

host configuration is used. The host configuration is derived from the target configuration as follows:

- Use the same version of Crosstool (`--crosstool_top`) as specified in the request configuration, unless `--host_crosstool_top` is specified.
- Use the value of `--host_cpu` for `--cpu` (default: `k8`).
- Use the same values of these options as specified in the request configuration: `--compiler` , `--use_ijars` , If `--host_crosstool_top` is used, then the value of `--host_cpu` is used to look up a `default_toolchain` in the Crosstool (ignoring `--compiler`) for the host configuration.
- Use the value of `--host_javabase` for `--javabase`
- Use the value of `--host_java_toolchain` for `--java_toolchain`
- Use optimized builds for C++ code (`-c opt`).
- Generate no debugging information (`--copt=-g0`).
- Strip debug information from executables and shared libraries (`--strip=always`).
- Place all derived files in a special location, distinct from that used by any possible request configuration.
- Suppress stamping of binaries with build data (see `--embed_*` options).
- All other values remain at their defaults.

There are many reasons why it might be preferable to select a distinct host configuration from the request configuration. Some are too esoteric to mention here, but two of them are worth pointing out.

Firstly, by using stripped, optimized binaries, you reduce the time spent linking and executing the tools, the disk space occupied by the tools, and the network I/O time in distributed builds.

Secondly, by decoupling the host and request configurations in all builds, you avoid very expensive rebuilds that would result from minor changes to the request configuration (such as changing a linker options does), as described earlier.

That said, for certain builds, this option may be a hindrance. In particular, builds in which changes of configuration are infrequent (especially certain Java builds), and builds where the amount of code that must be built in both host and target configurations is large, may not benefit.

Correct incremental rebuilds

One of the primary goals of the Bazel project is to ensure correct incremental rebuilds. Previous build tools, especially those based on Make, make several unsound assumptions in their implementation of incremental builds.

Firstly, that timestamps of files increase monotonically. While this is the typical case, it is very easy to fall afoul of this assumption; syncing to an earlier revision of a file causes that file's modification time to decrease; Make-based systems will not rebuild.

More generally, while Make detects changes to files, it does not detect changes to commands. If you alter the options passed to the compiler in a given build step, Make will not re-run the compiler, and it is necessary to manually discard the invalid outputs of the previous build using `make clean` .

Also, Make is not robust against the unsuccessful termination of one of its subprocesses after that subprocess has started writing to its output file. While the current execution of Make will fail, the subsequent invocation of Make will blindly assume that the truncated output file is valid (because it is newer than its inputs), and it will not be rebuilt. Similarly, if the Make process is killed, a similar situation can occur.

Bazel avoids these assumptions, and others. Bazel maintains a database of all work previously done, and will only omit a build step if it finds that the set of input files (and their timestamps) to that build step, and the compilation

command for that build step, exactly match one in the database, and, that the set of output files (and their timestamps) for the database entry exactly match the timestamps of the files on disk. Any change to the input files or output files, or to the command itself, will cause re-execution of the build step.

The benefit to users of correct incremental builds is: less time wasted due to confusion. (Also, less time spent waiting for rebuilds caused by use of `make clean`, whether necessary or pre-emptive.)

Build consistency and incremental builds

Formally, we define the state of a build as *consistent* when all the expected output files exist, and their contents are correct, as specified by the steps or rules required to create them. When you edit a source file, the state of the build is said to be *inconsistent*, and remains inconsistent until you next run the build tool to successful completion. We describe this situation as *unstable inconsistency*, because it is only temporary, and consistency is restored by running the build tool.

There is another kind of inconsistency that is pernicious: *stable inconsistency*. If the build reaches a stable inconsistent state, then repeated successful invocation of the build tool does not restore consistency: the build has gotten "stuck", and the outputs remain incorrect. Stable inconsistent states are the main reason why users of Make (and other build tools) type `make clean`. Discovering that the build tool has failed in this manner (and then recovering from it) can be time consuming and very frustrating.

Conceptually, the simplest way to achieve a consistent build is to throw away all the previous build outputs and start again: make every build a clean build. This approach is obviously too time-consuming to be practical (except perhaps for release engineers), and therefore to be useful, the build tool must be able to perform incremental builds without compromising consistency.

Correct incremental dependency analysis is hard, and as described above, many other build tools do a poor job of avoiding stable inconsistent states during incremental builds. In contrast, Bazel offers the following guarantee: after a successful invocation of the build tool during which you made no edits, the build will be in a consistent state. (If you edit your source files during a build, Bazel makes no guarantee about the consistency of the result of the current build. But it does guarantee that the results of the *next* build will restore consistency.)

As with all guarantees, there comes some fine print: there are some known ways of getting into a stable inconsistent state with Bazel. We won't guarantee to investigate such problems arising from deliberate attempts to find bugs in the incremental dependency analysis, but we will investigate and do our best to fix all stable inconsistent states arising from normal or "reasonable" use of the build tool.

If you ever detect a stable inconsistent state with Bazel, please report a bug.

Sandboxed execution

Bazel uses sandboxes to guarantee that actions run hermetically[1] and correctly. Bazel runs *Spawn*s (loosely speaking: actions) in sandboxes that only contain the minimal set of files the tool requires to do its job. Currently sandboxing works on Linux 3.12 or newer with the `CONFIG_USER_NS` option enabled, and also on macOS 10.11 or newer.

Bazel will print a warning if your system does not support sandboxing to alert you to the fact that builds are not guaranteed to be hermetic and might affect the host system in unknown ways. To disable this warning you can pass the `--ignore_unsupported_sandboxing` flag to Bazel.

On some platforms such as Google Kubernetes Engine (https://cloud.google.com/kubernetes-engine/) cluster nodes or Debian, user namespaces are deactivated by default due to security concerns. This can be checked by looking at the file `/proc/sys/kernel/unprivileged_userns_clone`: if it exists and contains a 0, then user namespaces can be activated with `sudo sysctl kernel.unprivileged_userns_clone=1`.

In some cases, the Bazel sandbox fails to execute rules because of the system setup. The symptom is generally a failure that output a message similar to `namespace-sandbox.c:633: execvp(argv[0], argv): No such file or directory`. In that case, try to deactivate the sandbox for genrules with `--genrule_strategy=standalone` and for other rules with `--spawn_strategy=standalone`. Also please report a bug on our issue tracker and mention which Linux distribution you're using so that we can investigate and provide a fix in a subsequent release.

[1]: Hermeticity means that the action only uses its declared input files and no other files in the filesystem, and it only produces its declared output files.

Deleting the outputs of a build

The `clean` command

Bazel has a `clean` command, analogous to that of Make. It deletes the output directories for all build configurations performed by this Bazel instance, or the entire working tree created by this Bazel instance, and resets internal caches. If executed without any command-line options, then the output directory for all configurations will be cleaned.

Recall that each Bazel instance is associated with a single workspace, thus the `clean` command will delete all outputs from all builds you've done with that Bazel instance in that workspace.

To completely remove the entire working tree created by a Bazel instance, you can specify the `--expunge` option. When executed with `--expunge`, the clean command simply removes the entire output base tree which, in addition to the build output, contains all temp files created by Bazel. It also stops the Bazel server after the clean, equivalent to the `shutdown` command. For example, to clean up all disk and memory traces of a Bazel instance, you could specify:

```
% bazel clean --expunge
```

Alternatively, you can expunge in the background by using `--expunge_async`. It is safe to invoke a Bazel command in the same client while the asynchronous expunge continues to run. Note, however, that this may introduce IO contention.

The `clean` command is provided primarily as a means of reclaiming disk space for workspaces that are no longer needed. However, we recognize that Bazel's incremental rebuilds might not be perfect; `clean` may be used to recover a consistent state when problems arise.

Bazel's design is such that these problems are fixable; we consider such bugs a high priority, and will do our best fix them. If you ever find an incorrect incremental build, please file a bug report. We encourage developers to get out of the habit of using `clean` and into that of reporting bugs in the tools.

Phases of a build

In Bazel, a build occurs in three distinct phases; as a user, understanding the difference between them provides insight into the options which control a build (see below).

Loading phase

The first is **loading** during which all the necessary BUILD files for the initial targets, and their transitive closure of dependencies, are loaded, parsed, evaluated and cached.

For the first build after a Bazel server is started, the loading phase typically takes many seconds as many BUILD files are loaded from the file system. In subsequent builds, especially if no BUILD files have changed, loading occurs very quickly.

Errors reported during this phase include: package not found, target not found, lexical and grammatical errors in a BUILD file, and evaluation errors.

Analysis phase

The second phase, **analysis**, involves the semantic analysis and validation of each build rule, the construction of a build dependency graph, and the determination of exactly what work is to be done in each step of the build.

Like loading, analysis also takes several seconds when computed in its entirety. However, Bazel caches the dependency graph from one build to the next and only reanalyzes what it has to, which can make incremental builds extremely fast in the case where the packages haven't changed since the previous build.

Errors reported at this stage include: inappropriate dependencies, invalid inputs to a rule, and all rule-specific error messages.

The loading and analysis phases are fast because Bazel avoids unnecessary file I/O at this stage, reading only BUILD files in order to determine the work to be done. This is by design, and makes Bazel a good foundation for analysis tools, such as Bazel's query command, which is implemented atop the loading phase.

Execution phase

The third and final phase of the build is **execution**. This phase ensures that the outputs of each step in the build are consistent with its inputs, re-running compilation/linking/etc. tools as necessary. This step is where the build spends the majority of its time, ranging from a few seconds to over an hour for a large build. Errors reported during this phase include: missing source files, errors in a tool executed by some build action, or failure of a tool to produce the expected set of outputs.

Options

The following sections describe the options available during a build. When `--long` is used on a help command, the on-line help messages provide summary information about the meaning, type and default value for each option.

Most options can only be specified once. When specified multiple times, the last instance wins. Options that can be specified multiple times are identified in the on-line help with the text 'may be used multiple times'.

Options that affect how packages are located

See also the `--show_package_location` option.

`--package_path`

This option specifies the set of directories that are searched to find the BUILD file for a given package.

`--deleted_packages`

This option specifies a comma-separated list of packages which Bazel should consider deleted, and not attempt to load from any directory on the package path. This can be used to simulate the deletion of packages without actually deleting them.

Error checking options

These options control Bazel's error-checking and/or warnings.

`--check_constraint` *constraint*

This option takes an argument that specifies which constraint should be checked.

Bazel performs special checks on each rule that is annotated with the given constraint.

The supported constraints and their checks are as follows:

- `public` : Verify that all java_libraries marked with `constraints = ['public']` only depend on java_libraries that are marked as `constraints = ['public']` too. If bazel finds a dependency that does not conform to this rule, bazel will issue an error.

`--[no]check_visibility`

If this option is set to false, visibility checks are demoted to warnings. The default value of this option is true, so that by default, visibility checking is done.

`--output_filter` *regex*

The `--output_filter` option will only show build and compilation warnings for targets that match the regular expression. If a target does not match the given regular expression and its execution succeeds, its standard output and standard error are thrown away.

Here are some typical values for this option:

`--output_filter='^//(first/project|second/project):'`	Show the output for the specified packages.
`--output_filter='^//((?!(first/bad_project|second/bad_project):).)*$'`	Don't show output for the specified packages.
`--output_filter=`	Show everything.
`--output_filter=DONT_MATCH_ANYTHING`	Show nothing.

Flags options

These options control which options Bazel will pass to other tools.

`--copt` *gcc-option*

This option takes an argument which is to be passed to gcc. The argument will be passed to gcc whenever gcc is invoked for preprocessing, compiling, and/or assembling C, C++, or assembler code. It will not be passed when linking.

This option can be used multiple times. For example:

```
% bazel build --copt="-g0" --copt="-fpic" //foo
```

will compile the `foo` library without debug tables, generating position-independent code.

Note that changing `--copt` settings will force a recompilation of all affected object files. Also note that copts values listed in specific cc_library or cc_binary build rules will be placed on the gcc command line *after* these options.

Warning: C++-specific options (such as `-fno-implicit-templates`) should be specified in `--cxxopt`, not in `--copt`. Likewise, C-specific options (such as -Wstrict-prototypes) should be specified in `--conlyopt`, not in `copt`. Similarly, gcc options that only have an effect at link time (such as `-l`) should be specified in `--linkopt`, not in `--copt`.

`--host_copt` *gcc-option*

This option takes an argument which is to be passed to gcc for source files that are compiled in the host configuration. This is analogous to the `--copt` option, but applies only to the host configuration.

`--host_cxxopt` *gcc-option*

This option takes an argument which is to be passed to gcc for source files that are compiled in the host configuration. This is analogous to the `--cxxopt` option, but applies only to the host configuration.

`--conlyopt` *gcc-option*

This option takes an argument which is to be passed to gcc when compiling C source files.

This is similar to `--copt`, but only applies to C compilation, not to C++ compilation or linking. So you can pass C-specific options (such as `-Wno-pointer-sign`) using `--conlyopt`.

Note that copts parameters listed in specific cc_library or cc_binary build rules will be placed on the gcc command line *after* these options.

`--cxxopt` *gcc-option*

This option takes an argument which is to be passed to gcc when compiling C++ source files.

This is similar to `--copt`, but only applies to C++ compilation, not to C compilation or linking. So you can pass C++-

specific options (such as `-fpermissive` or `-fno-implicit-templates`) using `--cxxopt`. For example:

```
% bazel build --cxxopt="-fpermissive" --cxxopt="-Wno-error" //foo/cruddy_code
```

Note that copts parameters listed in specific cc_library or cc_binary build rules will be placed on the gcc command line *after* these options.

`--linkopt` *linker-option*

This option takes an argument which is to be passed to gcc when linking.

This is similar to `--copt`, but only applies to linking, not to compilation. So you can pass gcc options that only make sense at link time (such as `-lssp` or `-Wl,--wrap,abort`) using `--linkopt`. For example:

```
% bazel build --copt="-fmudflap" --linkopt="-lmudflap" //foo/buggy_code
```

Build rules can also specify link options in their attributes. This option's settings always take precedence. Also see cc_library.linkopts (be/c-cpp.html#cc_library.linkopts).

`--strip (always|never|sometimes)`

This option determines whether Bazel will strip debugging information from all binaries and shared libraries, by invoking the linker with the `-Wl,--strip-debug` option. `--strip=always` means always strip debugging information. `--strip=never` means never strip debugging information. The default value of `--strip=sometimes` means strip iff the `--compilation_mode` is `fastbuild`.

```
% bazel build --strip=always //foo:bar
```

will compile the target while stripping debugging information from all generated binaries.

Note that if you want debugging information, it's not enough to disable stripping; you also need to make sure that the debugging information was generated by the compiler, which you can do by using either `-c dbg` or `--copt -g`.

Note also that Bazel's `--strip` option corresponds with ld's `--strip-debug` option: it only strips debugging information. If for some reason you want to strip *all* symbols, not just *debug* symbols, you would need to use ld's `--strip-all` option, which you can do by passing `--linkopt=-Wl,--strip-all` to Bazel.

`--stripopt` *strip-option*

An additional option to pass to the `strip` command when generating a `*.stripped` binary (be/c-cpp.html#cc_binary_implicit_outputs). The default is `-S -p`. This option can be used multiple times.

Note that `--stripopt` does not apply to the stripping of the main binary with `--strip=(always|sometimes)`.

`--fdo_instrument` *profile-output-dir*

The `--fdo_instrument` option enables the generation of FDO (feedback directed optimization) profile output when the built C/C++ binary is executed. For GCC, the argument provided is used as a directory prefix for a per-

object file directory tree of .gcda files containing profile information for each .o file.

Once the profile data tree has been generated, the profile tree should be zipped up, and provided to the `--fdo_optimize=profile-zip` Bazel option to enable the FDO optimized compilation.

For the LLVM compiler the argument is also the directory under which the raw LLVM profile data file(s) is dumped, e.g. `--fdo_instrument=/path/to/rawprof/dir/`.

The options `--fdo_instrument` and `--fdo_optimize` cannot be used at the same time.

`--fdo_optimize profile-zip`

The `--fdo_optimize` option enables the use of the per-object file profile information to perform FDO (feedback directed optimization) optimizations when compiling. For GCC, the argument provided is the zip file containing the previously-generated file tree of .gcda files containing profile information for each .o file.

Alternatively, the argument provided can point to an auto profile identified by the extension .afdo.

Note that this option also accepts labels that resolve to source files. You may need to add an `exports_files` directive to the corresponding package to make the file visible to Bazel.

For the LLVM compiler the argument provided should point to the indexed LLVM profile output file prepared by the llvm-profdata tool, and should have a .profdata extension.

The options `--fdo_instrument` and `--fdo_optimize` cannot be used at the same time.

`--lipo (off|binary)`

The `--lipo=binary` option enables LIPO (Lightweight Inter-Procedural Optimization). LIPO is an extended C/C++ optimization technique that optimizes code across different object files. It involves compiling each C/C++ source file differently for every binary. This is in contrast to normal compilation where compilation outputs are reused. This means that LIPO is more expensive than normal compilation.

This option only has an effect when FDO is also enabled (see the --fdo_instrument and --fdo_options). Currently LIPO is only supported when building a single `cc_binary` rule.

Setting `--lipo=binary` implicitly sets `--dynamic_mode=off`.

`--lipo_context context-binary`

Specifies the label of a `cc_binary` rule that was used to generate the profile information for LIPO that was given to the `--fdo_optimize` option.

Specifying the context is mandatory when `--lipo=binary` is set. Using this option implicitly also sets `--linkopt=-Wl,--warn-unresolved-symbols`.

`--[no]output_symbol_counts`

If enabled, each gold-invoked link of a C++ executable binary will output a *symbol counts* file (via the `--print-symbol-counts` gold option). For each linker input, the file logs the number of symbols that were defined and the number of symbols that were used in the binary. This information can be used to track unnecessary link

dependencies. The symbol counts file is written to the binary's output path with the name `[targetname].sc`.

This option is disabled by default.

`--jvmopt` *jvm-option*

This option allows option arguments to be passed to the Java VM. It can be used with one big argument, or multiple times with individual arguments. For example:

```
% bazel build --jvmopt="-server -Xms256m" java/com/example/common/foo:all
```

will use the server VM for launching all Java binaries and set the startup heap size for the VM to 256 MB.

`--javacopt` *javac-option*

This option allows option arguments to be passed to javac. It can be used with one big argument, or multiple times with individual arguments. For example:

```
% bazel build --javacopt="-g:source,lines" //myprojects:prog
```

will rebuild a java_binary with the javac default debug info (instead of the bazel default).

The option is passed to javac after the Bazel built-in default options for javac and before the per-rule options. The last specification of any option to javac wins. The default options for javac are:

```
-source 8 -target 8 -encoding UTF-8
```

Note that changing `--javacopt` settings will force a recompilation of all affected classes. Also note that javacopts parameters listed in specific java_library or java_binary build rules will be placed on the javac command line *after* these options.

```
-extra_checks[:(off|on)]
```

This javac option enables extra correctness checks. Any problems found will be presented as errors. Either `-extra_checks` or `-extra_checks:on` may be used to force the checks to be turned on. `-extra_checks:off` completely disables the analysis. When this option is not specified, the default behavior is used.

`--strict_java_deps (default|strict|off|warn|error)`

This option controls whether javac checks for missing direct dependencies. Java targets must explicitly declare all directly used targets as dependencies. This flag instructs javac to determine the jars actually used for type checking each java file, and warn/error if they are not the output of a direct dependency of the current target.

- `off` means checking is disabled.
- `warn` means javac will generate standard java warnings of type `[strict]` for each missing direct dependency.
- `default`, `strict` and `error` all mean javac will generate errors instead of warnings, causing the current target to fail to build if any missing direct dependencies are found. This is also the default behavior when the flag is unspecified.

Semantics options

These options affect the build commands and/or the output file contents.

`--compilation_mode (fastbuild|opt|dbg) (-c)`

This option takes an argument of `fastbuild`, `dbg` or `opt`, and affects various C/C++ code-generation options, such as the level of optimization and the completeness of debug tables. Bazel uses a different output directory for each different compilation mode, so you can switch between modes without needing to do a full rebuild *every* time.

- `fastbuild` means build as fast as possible: generate minimal debugging information (`-gmlt -Wl,-S`), and don't optimize. This is the default. Note: `-DNDEBUG` will **not** be set.
- `dbg` means build with debugging enabled (`-g`), so that you can use gdb (or another debugger).
- `opt` means build with optimization enabled and with `assert()` calls disabled (`-O2 -DNDEBUG`). Debugging information will not be generated in `opt` mode unless you also pass `--copt -g` .

`--cpu` *cpu*

This option specifies the target CPU architecture to be used for the compilation of binaries during the build.

`--experimental_action_listener=`*label*

The `experimental_action_listener` option instructs Bazel to use details from the `action_listener` (be/extra-actions.html#action_listener) rule specified by *label* to insert `extra_actions` (be/extra-actions.html#extra_action) into the build graph.

`--[no]experimental_extra_action_top_level_only`

If this option is set to true, extra actions specified by the `--experimental_action_listener` command line option will only be scheduled for top level targets.

`--experimental_extra_action_filter=`*regex*

The `experimental_extra_action_filter` option instructs Bazel to filter the set of targets to schedule `extra_actions` for.

This flag is only applicable in combination with the `--experimental_action_listener` flag.

By default all `extra_actions` in the transitive closure of the requested targets-to-build get scheduled for execution. `--experimental_extra_action_filter` will restrict scheduling to `extra_actions` of which the owner's label matches the specified regular expression.

The following example will limit scheduling of `extra_actions` to only apply to actions of which the owner's label contains '/bar/':

```
% bazel build --experimental_action_listener=//test:al //foo/... \
  --experimental_extra_action_filter=.*/bar/.*
```

Build strategy options

Note that a particular combination of crosstool version, compiler version, libc version, and target CPU is allowed only if it has been specified in the currently used CROSSTOOL file.

`--host_cpu` *cpu*

This option specifies the name of the CPU architecture that should be used to build host tools.

`--fat_apk_cpu` *cpu[,cpu]**

The CPUs to build C/C++ libraries for in the transitive `deps` of `android_binary` rules. Other C/C++ rules are not affected. For example, if a `cc_library` appears in the transitive `deps` of an `android_binary` rule and a `cc_binary` rule, the `cc_library` will be built at least twice: once for each CPU specified with `--fat_apk_cpu` for the `android_binary` rule, and once for the CPU specified with `--cpu` for the `cc_binary` rule.

The default is `armeabi-v7a`.

One `.so` file will be created and packaged in the APK for each CPU specified with `--fat_apk_cpu`. The name of the `.so` file will be the name of the `android_binary` rule prefixed with "lib", e.g., if the name of the `android_binary` is "foo", then the file will be `libfoo.so`.

Note that an Android-compatible crosstool must be selected. If an `android_ndk_repository` rule is defined in the WORKSPACE file, an Android-compatible crosstool is automatically selected. Otherwise, the crostool can be selected using the `--android_crosstool_top` or `--crosstool_top` flags.

`--per_file_copt` *[+-]regex[,[+-]regex]...@option[,option]...*

When present, any C++ file with a label or an execution path matching one of the inclusion regex expressions and not matching any of the exclusion expressions will be built with the given options. The label matching uses the canonical form of the label (i.e `//package:label_name`). The execution path is the relative path to your workspace directory including the base name (including extension) of the C++ file. It also includes any platform dependent prefixes. Note, that if only one of the label or the execution path matches the options will be used.

Notes: To match the generated files (e.g. genrule outputs) Bazel can only use the execution path. In this case the regexp shouldn't start with '//' since that doesn't match any execution paths. Package names can be used like this: `--per_file_copt=base/.*\.pb\.cc@-g0`. This will match every `.pb.cc` file under a directory called `base`.

This option can be used multiple times.

The option is applied regardless of the compilation mode used. I.e. it is possible to compile with `--compilation_mode=opt` and selectively compile some files with stronger optimization turned on, or with optimization disabled.

Caveat: If some files are selectively compiled with debug symbols the symbols might be stripped during linking. This can be prevented by setting `--strip=never`.

Syntax: `[+-]regex[,[+-]regex]...@option[,option]...` Where `regex` stands for a regular expression that can be prefixed with a `+` to identify include patterns and with `-` to identify exclude patterns. `option` stands for an arbitrary option that is passed to the C++ compiler. If an option contains a `,` it has to be quoted like so `\,`. Options

can also contain `@`, since only the first `@` is used to separate regular expressions from options.

Example: `--per_file_copt=//foo:.*\.cc,-//foo:file\.cc@-O0,-fprofile-arcs` adds the `-O0` and the `-fprofile-arcs` options to the command line of the C++ compiler for all `.cc` files in `//foo/` except `file.cc`.

`--dynamic_mode` *mode*

Determines whether C++ binaries will be linked dynamically, interacting with the linkstatic attribute (be/c-cpp.html#cc_binary.linkstatic) on build rules.

Modes:

- `auto`: Translates to a platform-dependent mode; `default` for linux and `off` for cygwin.
- `default`: Allows bazel to choose whether to link dynamically. See linkstatic (be/c-cpp.html#cc_binary.linkstatic) for more information.
- `fully`: Links all targets dynamically. This will speed up linking time, and reduce the size of the resulting binaries.
- `off`: Links all targets in mostly static (be/c-cpp.html#cc_binary.linkstatic) mode. If `-static` is set in linkopts, targets will change to fully static.

`--fission (yes|no|[dbg][,opt][,fastbuild])`

Enables Fission (https://gcc.gnu.org/wiki/DebugFission), which writes C++ debug information to dedicated .dwo files instead of .o files, where it would otherwise go. This substantially reduces the input size to links and can reduce link times.

When set to `[dbg][,opt][,fastbuild]` (example: `--fission=dbg,fastbuild`), Fission is enabled only for the specified set of compilation modes. This is useful for bazelrc settings. When set to `yes`, Fission is enabled universally. When set to `no`, Fission is disabled universally. Default is `dbg`.

`--force_ignore_dash_static`

If this flag is set, any `-static` options in linkopts of `cc_*` rules BUILD files are ignored. This is only intended as a workaround for C++ hardening builds.

`--[no]force_pic`

If enabled, all C++ compilations produce position-independent code ("-fPIC"), links prefer PIC pre-built libraries over non-PIC libraries, and links produce position-independent executables ("-pie"). Default is disabled.

Note that dynamically linked binaries (i.e. `--dynamic_mode fully`) generate PIC code regardless of this flag's setting. So this flag is for cases where users want PIC code explicitly generated for static links.

`--android_resource_shrinking`

Selects whether to perform resource shrinking for android_binary rules. Sets the default for the shrink_resources attribute (be/android.html#android_binary.shrink_resources) on android_binary rules; see the documentation for that rule for further details. Defaults to off.

`--custom_malloc` *malloc-library-target*

When specified, always use the given malloc implementation, overriding all `malloc="target"` attributes, including in those targets that use the default (by not specifying any `malloc`).

`--crosstool_top` *label*

This option specifies the location of the crosstool compiler suite to be used for all C++ compilation during a build. Bazel will look in that location for a CROSSTOOL file and uses that to automatically determine settings for `--compiler`.

`--host_crosstool_top` *label*

If not specified, bazel uses the value of `--crosstool_top` to compile code in the host configuration, i.e., tools run during the build. The main purpose of this flag is to enable cross-compilation.

`--apple_crosstool_top` *label*

The crosstool to use for compiling C/C++ rules in the transitive `deps` of objc_*, ios__*, and apple_* rules. For those targets, this flag overwrites `--crosstool_top`.

`--android_crosstool_top` *label*

The crosstool to use for compiling C/C++ rules in the transitive `deps` of `android_binary` rules. This is useful if other targets in the build require a different crosstool. The default is to use the crosstool generated by the `android_ndk_repository` rule in the WORKSPACE file. See also `--fat_apk_cpu`.

`--compiler` *version*

This option specifies the C/C++ compiler version (e.g. `gcc-4.1.0`) to be used for the compilation of binaries during the build. If you want to build with a custom crosstool, you should use a CROSSTOOL file instead of specifying this flag.

Note that only certain combinations of crosstool version, compiler version, libc version, and target CPU are allowed.

`--glibc` *version*

This option specifies the version of glibc that the target should be linked against. If you want to build with a custom crosstool, you should use a CROSSTOOL file instead of specifying this flag. In that case, Bazel will use the CROSSTOOL file and the following options where appropriate:

- `--cpu`

Note that only certain combinations of crosstool version, compiler version, glibc version, and target CPU are allowed.

`--android_sdk` *label*

This option specifies the Android SDK/platform toolchain and Android runtime library that will be used to build any Android-related rule. The Android SDK will be automatically selected if an `android_sdk_repository` rule is defined in the WORKSPACE file.

`--java_toolchain` *`label`*

This option specifies the label of the java_toolchain used to compile Java source files.

`--host_java_toolchain` *`label`*

If not specified, bazel uses the value of `--java_toolchain` to compile code in the host configuration, i.e., tools run during the build. The main purpose of this flag is to enable cross-compilation.

`--javabase` (*`label`*)

This option sets the *label* of the base Java installation to use for running JavaBuilder, SingleJar, for *bazel run* and *bazel test*, and for Java binaries built by `java_binary` and `java_test` rules. The various "Make" variables (be/make-variables.html) for Java (`JAVABASE` , `JAVA` , `JAVAC` and `JAR`) are derived from this option.

`--host_javabase` *`label`*

If not specified, bazel uses the value of `--javabase` in the host configuration, i.e., for Java-based tools that run during the build. The main purpose of this flag is to enable cross-compilation.

This does not select the Java compiler that is used to compile Java source files. The compiler can be selected by settings the `--java_toolchain` option.

This option should not be confused with the startup option --host_javabase.

Build strategy options

These options affect how Bazel will execute the build. They should not have any significant effect on the output files generated by the build. Typically their main effect is on the speed on the build.

`--spawn_strategy` *`strategy`*

This option controls where and how commands are executed.

- `standalone` causes commands to be executed as local subprocesses.
- `sandboxed` causes commands to be executed inside a sandbox on the local machine. This requires that all input files, data dependencies and tools are listed as direct dependencies in the `srcs` , `data` and `tools` attributes. This is the default on systems that support sandboxed execution.

`--genrule_strategy` *`strategy`*

This option controls where and how genrules are executed.

- `standalone` causes genrules to run as local subprocesses.

- `sandboxed` causes genrules to run inside a sandbox on the local machine. This requires that all input files are listed as direct dependencies in the `srcs` attribute, and the program(s) executed are listed in the `tools` attribute. This is the default for Bazel on systems that support sandboxed execution.

`--jobs` *n* (-j)

This option, which takes an integer argument, specifies a limit on the number of jobs that should be executed concurrently during the execution phase of the build.

Note that the number of concurrent jobs that Bazel will run is determined not only by the `--jobs` setting, but also by Bazel's scheduler, which tries to avoid running concurrent jobs that will use up more resources (RAM or CPU) than are available, based on some (very crude) estimates of the resource consumption of each job. The behavior of the scheduler can be controlled by the `--ram_utilization_factor` option.

`--progress_report_interval` *n*

Bazel periodically prints a progress report on jobs that are not finished yet (e.g. long running tests). This option sets the reporting frequency, progress will be printed every `n` seconds.

The default is 0, that means an incremental algorithm: the first report will be printed after 10 seconds, then 30 seconds and after that progress is reported once every minute.

`--ram_utilization_factor` *percentage*

This option, which takes an integer argument, specifies what percentage of the system's RAM Bazel should try to use for its subprocesses. This option affects how many processes Bazel will try to run in parallel. The default value is 67. If you run several Bazel builds in parallel, using a lower value for this option may avoid thrashing and thus improve overall throughput. Using a value higher than the default is NOT recommended. Note that Bazel's estimates are very coarse, so the actual RAM usage may be much higher or much lower than specified. Note also that this option does not affect the amount of memory that the Bazel server itself will use.

`--local_resources` *availableRAM,availableCPU,availableIO*

This option, which takes three comma-separated floating point arguments, specifies the amount of local resources that Bazel can take into consideration when scheduling build and test activities. Option expects amount of available RAM (in MB), number of CPU cores (with 1.0 representing single full core) and workstation I/O capability (with 1.0 representing average workstation). By default Bazel will estimate amount of RAM and number of CPU cores directly from system configuration and will assume 1.0 I/O resource.

If this option is used, Bazel will ignore --ram_utilization_factor.

`--[no]build_runfile_links`

This option, which is enabled by default, specifies whether the runfiles symlinks for tests and binaries should be built in the output directory. Using `--nobuild_runfile_links` can be useful to validate if all targets compile without incurring the overhead for building the runfiles trees.

When tests (or applications) are executed, their run-time data dependencies are gathered together in one place.

Within Bazel's output tree, this "runfiles" tree is typically rooted as a sibling of the corresponding binary or test. During test execution, runfiles may be accessed using paths of the form `$TEST_SRCDIR/workspace/packagename/filename`. The runfiles tree ensures that tests have access to all the files upon which they have a declared dependence, and nothing more. By default, the runfiles tree is implemented by constructing a set of symbolic links to the required files. As the set of links grows, so does the cost of this operation, and for some large builds it can contribute significantly to overall build time, particularly because each individual test (or application) requires its own runfiles tree.

--[no]build_runfile_manifests

This option, which is enabled by default, specifies whether runfiles manifests should be written to the output tree. Disabling it implies `--nobuild_runfile_links`. It can be disabled when executing tests on Forge, as runfiles trees will be created remotely from in-memory manifests.

--[no]discard_analysis_cache

When this option is enabled, Bazel will discard the analysis cache right before execution starts, thus freeing up additional memory (around 10%) for the execution phase. The drawback is that further incremental builds will be slower.

--[no]keep_going (-k)

As in GNU Make, the execution phase of a build stops when the first error is encountered. Sometimes it is useful to try to build as much as possible even in the face of errors. This option enables that behavior, and when it is specified, the build will attempt to build every target whose prerequisites were successfully built, but will ignore errors.

While this option is usually associated with the execution phase of a build, it also effects the analysis phase: if several targets are specified in a build command, but only some of them can be successfully analyzed, the build will stop with an error unless `--keep_going` is specified, in which case the build will proceed to the execution phase, but only for the targets that were successfully analyzed.

--[no]use_ijars

This option changes the way `java_library` targets are compiled by Bazel. Instead of using the output of a `java_library` for compiling dependent `java_library` targets, Bazel will create interface jars that contain only the signatures of non-private members (public, protected, and default (package) access methods and fields) and use the interface jars to compile the dependent targets. This makes it possible to avoid recompilation when changes are only made to method bodies or private members of a class.

Note that using `--use_ijars` might give you a different error message when you are accidentally referring to a non visible member of another class: Instead of getting an error that the member is not visible you will get an error that the member does not exist.

Note that changing the `--use_ijars` setting will force a recompilation of all affected classes.

--[no]interface_shared_objects

This option enables *interface shared objects*, which makes binaries and other shared libraries depend on the *interface* of a shared object, rather than its implementation. When only the implementation changes, Bazel can avoid rebuilding targets that depend on the changed shared library unnecessarily.

Output selection options

These options determine what to build or test.

`--[no]build`

This option causes the execution phase of the build to occur; it is on by default. When it is switched off, the execution phase is skipped, and only the first two phases, loading and analysis, occur.

This option can be useful for validating BUILD files and detecting errors in the inputs, without actually building anything.

`--[no]build_tests_only`

If specified, Bazel will build only what is necessary to run the *_test and test_suite rules that were not filtered due to their size, timeout, tag, or language. If specified, Bazel will ignore other targets specified on the command line. By default, this option is disabled and Bazel will build everything requested, including *_test and test_suite rules that are filtered out from testing. This is useful because running `bazel test --build_tests_only foo/...` may not detect all build breakages in the `foo` tree.

`--[no]check_up_to_date`

This option causes Bazel not to perform a build, but merely check whether all specified targets are up-to-date. If so, the build completes successfully, as usual. However, if any files are out of date, instead of being built, an error is reported and the build fails. This option may be useful to determine whether a build has been performed more recently than a source edit (e.g. for pre-submit checks) without incurring the cost of a build.

See also `--check_tests_up_to_date` .

`--[no]compile_one_dependency`

Compile a single dependency of the argument files. This is useful for syntax checking source files in IDEs, for example, by rebuilding a single target that depends on the source file to detect errors as early as possible in the edit/build/test cycle. This argument affects the way all non-flag arguments are interpreted: for each source filename, one rule that depends on it will be built. For C++ and Java sources, rules in the same language space are preferentially chosen. For multiple rules with the same preference, the one that appears first in the BUILD file is chosen. An explicitly named target pattern which does not reference a source file results in an error.

`--save_temps`

The `--save_temps` option causes temporary outputs from gcc to be saved. These include .s files (assembler code), .i (preprocessed C) and .ii (preprocessed C++) files. These outputs are often useful for debugging. Temps will only be generated for the set of targets specified on the command line.

Note that our implementation of `--save_temps` does not use gcc's `-save-temps` flag. Instead, we do two passes, one with `-S` and one with `-E`. A consequence of this is that if your build fails, Bazel may not yet have produced the ".i" or ".ii" and ".s" files. If you're trying to use `--save_temps` to debug a failed compilation, you may need to also use `--keep_going` so that Bazel will still try to produce the preprocessed files after the compilation fails.

The `--save_temps` flag currently works only for cc_* rules.

To ensure that Bazel prints the location of the additional output files, check that your `--show_result` *n* setting is high enough.

`--build_tag_filters` *tag[,tag]**

If specified, Bazel will build only targets that have at least one required tag (if any of them are specified) and does not have any excluded tags. Build tag filter is specified as comma delimited list of tag keywords, optionally preceded with '-' sign used to denote excluded tags. Required tags may also have a preceding '+' sign.

`--test_size_filters` *size[,size]**

If specified, Bazel will test (or build if `--build_tests_only` is also specified) only test targets with the given size. Test size filter is specified as comma delimited list of allowed test size values (small, medium, large or enormous), optionally preceded with '-' sign used to denote excluded test sizes. For example,

```
% bazel test --test_size_filters=small,medium //foo:all
```

and

```
% bazel test --test_size_filters=-large,-enormous //foo:all
```

will test only small and medium tests inside //foo.

By default, test size filtering is not applied.

`--test_timeout_filters` *timeout[,timeout]**

If specified, Bazel will test (or build if `--build_tests_only` is also specified) only test targets with the given timeout. Test timeout filter is specified as comma delimited list of allowed test timeout values (short, moderate, long or eternal), optionally preceded with '-' sign used to denote excluded test timeouts. See --test_size_filters for example syntax.

By default, test timeout filtering is not applied.

`--test_tag_filters` *tag[,tag]**

If specified, Bazel will test (or build if `--build_tests_only` is also specified) only test targets that have at least one required tag (if any of them are specified) and does not have any excluded tags. Test tag filter is specified as comma delimited list of tag keywords, optionally preceded with '-' sign used to denote excluded tags. Required tags may also have a preceding '+' sign.

For example,

```
% bazel test --test_tag_filters=performance,stress,-flaky //myproject:all
```

will test targets that are tagged with either `performance` or `stress` tag but are **not** tagged with the `flaky` tag.

By default, test tag filtering is not applied. Note that you can also filter on test's `size` and `local` tags in this manner.

`--test_lang_filters` *lang[,lang]**

Specifies a comma-separated list of test languages for languages with an official `*_test` rule the (see build encyclopedia (be/overview.html) for a full list of these). Each language can be optionally preceded with '-' to specify excluded languages. The name used for each language should be the same as the language prefix in the `*_test` rule, for example, `cc`, `java` or `sh`.

If specified, Bazel will test (or build if `--build_tests_only` is also specified) only test targets of the specified language(s).

For example,

```
% bazel test --test_lang_filters=cc,java foo/...
```

will test only the C/C++ and Java tests (defined using `cc_test` and `java_test` rules, respectively) in `foo/...`, while

```
% bazel test --test_lang_filters=-sh,-java foo/...
```

will run all of the tests in `foo/...` except for the `sh_test` and `java_test` tests.

By default, test language filtering is not applied.

`--test_filter=`*filter-expression*

Specifies a filter that the test runner may use to pick a subset of tests for running. All targets specified in the invocation are built, but depending on the expression only some of them may be executed; in some cases, only certain test methods are run.

The particular interpretation of *filter-expression* is up to the test framework responsible for running the test. It may be a glob, substring, or regexp. `--test_filter` is a convenience over passing different `--test_arg` filter arguments, but not all frameworks support it.

Verbosity options: options that control what Bazel prints

These options control the verbosity of Bazel's output, either to the terminal, or to additional log files.

`--explain` *logfile*

This option, which requires a filename argument, causes the dependency checker in `bazel build`'s execution phase to explain, for each build step, either why it is being executed, or that it is up-to-date. The explanation is written to *logfile*.

If you are encountering unexpected rebuilds, this option can help to understand the reason. Add it to your `.bazelrc` so that logging occurs for all subsequent builds, and then inspect the log when you see an execution step executed unexpectedly. This option may carry a small performance penalty, so you might want to remove it when it is no longer needed.

`--verbose_explanations`

This option increases the verbosity of the explanations generated when the --explain option is enabled.

In particular, if verbose explanations are enabled, and an output file is rebuilt because the command used to build it has changed, then the output in the explanation file will include the full details of the new command (at least for most commands).

Using this option may significantly increase the length of the generated explanation file and the performance penalty of using `--explain`.

If `--explain` is not enabled, then `--verbose_explanations` has no effect.

`--profile` *file*

This option, which takes a filename argument, causes Bazel to write profiling data into a file. The data then can be analyzed or parsed using the `bazel analyze-profile` command. The Build profile can be useful in understanding where Bazel's `build` command is spending its time.

`--[no]show_loading_progress`

This option causes Bazel to output package-loading progress messages. If it is disabled, the messages won't be shown.

`--[no]show_progress`

This option causes progress messages to be displayed; it is on by default. When disabled, progress messages are suppressed.

`--show_progress_rate_limit` *n*

This option causes bazel to display only one progress message per `n` seconds, where *n* is a real number. If `n` is -1, all progress messages will be displayed. The default value for this option is 0.03, meaning bazel will limit the progress messages to one per every 0.03 seconds.

`--show_result` *n*

This option controls the printing of result information at the end of a `bazel build` command. By default, if a single build target was specified, Bazel prints a message stating whether or not the target was successfully brought up-to-date, and if so, the list of output files that the target created. If multiple targets were specified, result information is not displayed.

While the result information may be useful for builds of a single target or a few targets, for large builds (e.g. an entire

top-level project tree), this information can be overwhelming and distracting; this option allows it to be controlled. `--show_result` takes an integer argument, which is the maximum number of targets for which full result information should be printed. By default, the value is 1. Above this threshold, no result information is shown for individual targets. Thus zero causes the result information to be suppressed always, and a very large value causes the result to be printed always.

Users may wish to choose a value in-between if they regularly alternate between building a small group of targets (for example, during the compile-edit-test cycle) and a large group of targets (for example, when establishing a new workspace or running regression tests). In the former case, the result information is very useful whereas in the latter case it is less so. As with all options, this can be specified implicitly via the `.bazelrc` file.

The files are printed so as to make it easy to copy and paste the filename to the shell, to run built executables. The "up-to-date" or "failed" messages for each target can be easily parsed by scripts which drive a build.

`--subcommands (-s)`

This option causes Bazel's execution phase to print the full command line for each command prior to executing it.

```
>>>>> # //examples/cpp:hello-world [action 'Linking examples/cpp/hello-world']
(cd /home/johndoe/.cache/bazel/_bazel_johndoe/4c084335afceb392cfbe7c31afee3a9f/bazel && '
  exec env - \
  /usr/bin/gcc -o bazel-out/local-fastbuild/bin/examples/cpp/hello-world -B/usr/bin/ -Wl
```

Where possible, commands are printed in a Bourne shell compatible syntax, so that they can be easily copied and pasted to a shell command prompt. (The surrounding parentheses are provided to protect your shell from the `cd` and `exec` calls; be sure to copy them!) However some commands are implemented internally within Bazel, such as creating symlink trees. For these there's no command line to display.

See also --verbose_failures, below.

`--verbose_failures`

This option causes Bazel's execution phase to print the full command line for commands that failed. This can be invaluable for debugging a failing build.

Failing commands are printed in a Bourne shell compatible syntax, suitable for copying and pasting to a shell prompt.

Options that control how Bazel embeds workspace status information into binaries ("stamping")

Use these options to "stamp" Bazel-built binaries: to embed additional information into the binaries, such as the source control revision or other workspace-related information. You can use this mechanism with rules that support the `stamp` attribute, such as `genrule`, `cc_binary`, and more.

`--workspace_status_command` *program*

This flag lets you specify a binary that Bazel runs before each build. The program can report information about the status of the workspace, such as the current source control revision.

The flag's value must be a path to a native program. On Linux/macOS this may be any executable. On Windows this must be a native binary, typically an ".exe", ".bat", or a ".cmd" file.

The program should print zero or more key/value pairs to standard output, one entry on each line, then exit with zero (otherwise the build fails). The key names can be anything but they may only use upper case letters and underscores. The first space after the key name separates it from the value. The value is the rest of the line (including additional whitespaces).

Bazel partitions the keys into two buckets: "stable" and "volatile". (The names "stable" and "volatile" are a bit counter-intuitive, so don't think much about them.)

Bazel then writes the key-value pairs into two files:

- `bazel-out/stable-status.txt` contains all keys and values where the key's name starts with `STABLE_`
- `bazel-out/volatile-status.txt` contains the rest of the keys and their values

The contract is:

- "stable" keys' values should change rarely, if possible. If the contents of `stable-status.txt` change, it invalidates the actions that depend on them. In other words, if a stable key's value changes, it'll make Bazel rebuild stamped actions. Therefore the stable status should not contain things like timestamps, because they change all the time, and would make Bazel rebuild the stamped actions with each build.

  Bazel always outputs the following stable keys:

  - `BUILD_EMBED_LABEL` : value of `--embed_label`
  - `BUILD_HOST` : the name of the host machine that Bazel is running on
  - `BUILD_USER` : the name of the user that Bazel is running as
- "volatile" keys' values may change often. Bazel expects them to change all the time, like timestamps do, and duly updates the `volatile-status.txt` file. In order to avoid rebuilding stamped actions all the time though, **Bazel pretends that the volatile file never changes**. In other words, if the volatile status file is the only one whose contents changed, that will not invalidate actions that depend on it. If other inputs of the actions have changed, then Bazel rebuilds that action, and the action will use the updated volatile status, but just the volatile status changing alone will not invalidate the action.

  Bazel always outputs the following volatile keys:

  - `BUILD_TIMESTAMP` : time of the build in milliseconds since the Unix Epoch (the value of `System.currentTimeMillis()`)

On Linux/macOS you can pass `--workspace_status_command=/bin/true` to disable retrieving workspace status, because `true` does nothing successfully (exits with zero) and prints no output. On Windows you can pass the path of MSYS's `true.exe` for the same effect.

If the workspace status command fails (exits non-zero) for any reason, the build will fail.

Example program on Linux using Git:

```
#!/bin/bash
echo "CURRENT_TIME $(date +%s)"
echo "RANDOM_HASH $(cat /dev/urandom | head -c16 | md5sum 2>/dev/null | cut -f1 -d' ')"
echo "STABLE_GIT_COMMIT $(git rev-parse HEAD)"
echo "STABLE_USER_NAME $USER"
```

Pass this program's path with `--workspace_status_command`, and the stable status file will include the STABLE lines and the volatile status file will include the rest of the lines.

`--[no]stamp`

This option controls whether stamping is enabled for rule types that support it. For most of the supported rule types stamping is enabled by default (e.g. `cc_binary`). By default, stamping is disabled for all tests. Specifying `--stamp` does not force affected targets to be rebuilt, if their dependencies have not changed.

Stamping can be enabled or disabled explicitly in BUILD using the `stamp` attribute of certain rule types, please refer to the build encyclopedia (be/overview.html) for details. For rules that are neither explicitly or implicitly configured as `stamp = 0` or `stamp = 1`, the `--[no]stamp` option selects whether stamping is enabled. Bazel never stamps binaries that are built for the host configuration, regardless of the stamp attribute.

Miscellaneous options

`--symlink_prefix` *`string`*

Changes the prefix of the generated convenience symlinks. The default value for the symlink prefix is `bazel-` which will create the symlinks `bazel-bin`, `bazel-testlogs`, and `bazel-genfiles`.

If the symbolic links cannot be created for any reason, a warning is issued but the build is still considered a success. In particular, this allows you to build in a read-only directory or one that you have no permission to write into. Any paths printed in informational messages at the conclusion of a build will only use the symlink-relative short form if the symlinks point to the expected location; in other words, you can rely on the correctness of those paths, even if you cannot rely on the symlinks being created.

Some common values of this option:

- **Suppress symlink creation:** `--symlink_prefix=/` will cause Bazel to not create or update any symlinks, including the `bazel-out` and `bazel-<workspace>` symlinks. Use this option to suppress symlink creation entirely.
- **Reduce clutter:** `--symlink_prefix=.bazel/` will cause Bazel to create symlinks called `bin` (etc) inside a hidden directory `.bazel`.

`--platform_suffix` *`string`*

Adds a suffix to the configuration short name, which is used to determine the output directory. Setting this option to different values puts the files into different directories, for example to improve cache hit rates for builds that otherwise clobber each others output files, or to keep the output files around for comparisons.

`--default_visibility=`*`(private|public)`*

Temporary flag for testing bazel default visibility changes. Not intended for general use but documented for completeness' sake.

`--[no]use_action_cache`

This option is enabled by default. If disabled, Bazel will not use its local action cache. Disabling the local action cache saves memory and disk space for clean builds, but will make incremental builds slower.

Using Bazel for releases

Bazel is used both by software engineers during the development cycle, and by release engineers when preparing binaries for deployment to production. This section provides a list of tips for release engineers using Bazel.

Significant options

When using Bazel for release builds, the same issues arise as for other scripts that perform a build, so you should read the scripting section of this manual. In particular, the following options are strongly recommended:

- `--bazelrc=/dev/null`
- `--batch`

These options (q.v.) are also important:

- `--package_path`
- `--symlink_prefix` : for managing builds for multiple configurations, it may be convenient to distinguish each build with a distinct identifier, e.g. "64bit" vs. "32bit". This option differentiates the `bazel-bin` (etc.) symlinks.

Running tests with Bazel

To build and run tests with bazel, type `bazel test` followed by the name of the test targets.

By default, this command performs simultaneous build and test activity, building all specified targets (including any non-test targets specified on the command line) and testing `*_test` and `test_suite` targets as soon as their prerequisites are built, meaning that test execution is interleaved with building. Doing so usually results in significant speed gains.

Options for `bazel test`

`--cache_test_results=(yes|no|auto) (-t)`

If this option is set to 'auto' (the default) then Bazel will only rerun a test if any of the following conditions applies:

- Bazel detects changes in the test or its dependencies
- the test is marked as `external`
- multiple test runs were requested with `--runs_per_test`
- the test failed.

If 'no', all tests will be executed unconditionally.

If 'yes', the caching behavior will be the same as auto except that it may cache test failures and test runs with `--runs_per_test` .

Note that test results are *always* saved in Bazel's output tree, regardless of whether this option is enabled, so you

needn't have used `--cache_test_results` on the prior run(s) of `bazel test` in order to get cache hits. The option only affects whether Bazel will *use* previously saved results, not whether it will save results of the current run.

Users who have enabled this option by default in their `.bazelrc` file may find the abbreviations `-t` (on) or `-t-` (off) convenient for overriding the default on a particular run.

`--check_tests_up_to_date`

This option tells Bazel not to run the tests, but to merely check and report the cached test results. If there are any tests which have not been previously built and run, or whose tests results are out-of-date (e.g. because the source code or the build options have changed), then Bazel will report an error message ("test result is not up-to-date"), will record the test's status as "NO STATUS" (in red, if color output is enabled), and will return a non-zero exit code.

This option also implies `--check_up_to_date` behavior.

This option may be useful for pre-submit checks.

`--test_verbose_timeout_warnings`

This option tells Bazel to explicitly warn the user if a test's timeout is significantly longer then the test's actual execution time. While a test's timeout should be set such that it is not flaky, a test that has a highly over-generous timeout can hide real problems that crop up unexpectedly.

For instance, a test that normally executes in a minute or two should not have a timeout of ETERNAL or LONG as these are much, much too generous. This option is useful to help users decide on a good timeout value or sanity check existing timeout values.

Note that each test shard is allotted the timeout of the entire `XX_test` target. Using this option does not affect a test's timeout value, merely warns if Bazel thinks the timeout could be restricted further.

`--[no]test_keep_going`

By default, all tests are run to completion. If this flag is disabled, however, the build is aborted on any non-passing test. Subsequent build steps and test invocations are not run, and in-flight invocations are canceled. Do not specify both `--notest_keep_going` and `--keep_going`.

`--flaky_test_attempts` *attempts*

This option specifies the maximum number of times a test should be attempted if it fails for any reason. A test that initially fails but eventually succeeds is reported as `FLAKY` on the test summary. It is, however, considered to be passed when it comes to identifying Bazel exit code or total number of passed tests. Tests that fail all allowed attempts are considered to be failed.

By default (when this option is not specified, or when it is set to "default"), only a single attempt is allowed for regular tests, and 3 for test rules with the `flaky` attribute set. You can specify an integer value to override the maximum limit of test attempts. Bazel allows a maximum of 10 test attempts in order to prevent abuse of the system.

`--runs_per_test` *[regex@]number*

This option specifies the number of times each test should be executed. All test executions are treated as separate tests (e.g. fallback functionality will apply to each of them independently).

The status of a target with failing runs depends on the value of the `--runs_per_test_detects_flakes` flag:

- If absent, any failing run causes the entire test to fail.
- If present and two runs from the same shard return PASS and FAIL, the test will receive a status of flaky (unless other failing runs cause it to fail).

If a single number is specified, all tests will run that many times. Alternatively, a regular expression may be specified using the syntax regex@number. This constrains the effect of --runs_per_test to targets which match the regex (e.g. "--runs_per_test=^//pizza:.*@4" runs all tests under //pizza/ 4 times). This form of --runs_per_test may be specified more than once.

`--[no]runs_per_test_detects_flakes`

If this option is specified (by default it is not), Bazel will detect flaky test shards through --runs_per_test. If one or more runs for a single shard fail and one or more runs for the same shard pass, the target will be considered flaky with the flag. If unspecified, the target will report a failing status.

`--test_summary` *`output_style`*

Specifies how the test result summary should be displayed.

- `short` prints the results of each test along with the name of the file containing the test output if the test failed. This is the default value.
- `terse` like `short`, but even shorter: only print information about tests which did not pass.
- `detailed` prints each individual test case that failed, not only each test. The names of test output files are omitted.
- `none` does not print test summary.

`--test_output` *`output_style`*

Specifies how test output should be displayed:

- `summary` shows a summary of whether each test passed or failed. Also shows the output log file name for failed tests. The summary will be printed at the end of the build (during the build, one would see just simple progress messages when tests start, pass or fail). This is the default behavior.
- `errors` sends combined stdout/stderr output from failed tests only into the stdout immediately after test is completed, ensuring that test output from simultaneous tests is not interleaved with each other. Prints a summary at the build as per summary output above.
- `all` is similar to `errors` but prints output for all tests, including those which passed.
- `streamed` streams stdout/stderr output from each test in real-time.

`--java_debug`

This option causes the Java virtual machine of a java test to wait for a connection from a JDWP-compliant debugger before starting the test. This option implies --test_output=streamed.

`--[no]verbose_test_summary`

By default this option is enabled, causing test times and other additional information (such as test attempts) to be printed to the test summary. If `--noverbose_test_summary` is specified, test summary will include only test name, test status and cached test indicator and will be formatted to stay within 80 characters when possible.

`--test_tmpdir` *path*

Specifies temporary directory for tests executed locally. Each test will be executed in a separate subdirectory inside this directory. The directory will be cleaned at the beginning of the each `bazel test` command. By default, bazel will place this directory under Bazel output base directory. Note that this is a directory for running tests, not storing test results (those are always stored under the `bazel-out` directory).

`--test_timeout` *seconds* OR `--test_timeout` *seconds*,*seconds*,*seconds*,*seconds*

Overrides the timeout value for all tests by using specified number of seconds as a new timeout value. If only one value is provided, then it will be used for all test timeout categories.

Alternatively, four comma-separated values may be provided, specifying individual timeouts for short, moderate, long and eternal tests (in that order). In either form, zero or a negative value for any of the test sizes will be substituted by the default timeout for the given timeout categories as defined by the page Writing Tests (test-encyclopedia.html). By default, Bazel will use these timeouts for all tests by inferring the timeout limit from the test's size whether the size is implicitly or explicitly set.

Tests which explicitly state their timeout category as distinct from their size will receive the same value as if that timeout had been implicitly set by the size tag. So a test of size 'small' which declares a 'long' timeout will have the same effective timeout that a 'large' tests has with no explicit timeout.

`--test_arg` *arg*

Passes command-line options/flags/arguments to each test process. This option can be used multiple times to pass several arguments, e.g. `--test_arg=--logtostderr --test_arg=--v=3` .

`--test_env` *variable*=*value* OR `--test_env` *variable*

Specifies additional variables that must be injected into the test environment for each test. If *value* is not specified it will be inherited from the shell environment used to start the `bazel test` command.

The environment can be accessed from within a test by using `System.getenv("var")` (Java), `getenv("var")` (C or C++),

`--run_under=`*command-prefix*

This specifies a prefix that the test runner will insert in front of the test command before running it. The *command-prefix* is split into words using Bourne shell tokenization rules, and then the list of words is prepended to the command that will be executed.

If the first word is a fully qualified label (i.e. starts with `//`) it is built. Then the label is substituted by the corresponding executable location that is prepended to the command that will be executed along with the other words.

Some caveats apply:

- The PATH used for running tests may be different than the PATH in your environment, so you may need to use an **absolute path** for the `--run_under` command (the first word in *command-prefix*).
- **`stdin` is not connected**, so `--run_under` can't be used for interactive commands.

Examples:

```
--run_under=/usr/bin/valgrind
--run_under=/usr/bin/strace
--run_under='/usr/bin/strace -c'
--run_under='/usr/bin/valgrind --quiet --num-callers=20'
```

Test selection

As documented under Output selection options, you can filter tests by size, timeout, tag, or language. A convenience general name filter can forward particular filter args to the test runner.

Other options for `bazel test`

The syntax and the remaining options are exactly like bazel build.

Running executables with Bazel

The `bazel run` command is similar to `bazel build`, except it is used to build and run a single target. Here is a typical session:

```
% bazel run -- java/myapp:myapp --arg1 --arg2
Welcome to Bazel
INFO: Loading package: java/myapp
INFO: Loading package: foo/bar
INFO: Loading complete.  Analyzing...
INFO: Found 1 target...
...
Target //java/myapp:myapp up-to-date:
  bazel-bin/java/myapp:myapp
INFO: Elapsed time: 0.638s, Critical Path: 0.34s

INFO: Running command line: bazel-bin/java/myapp:myapp --arg1 --arg2
Hello there
$EXEC_ROOT/java/myapp/myapp
--arg1
--arg2
```

Bazel closes stdin, so you can't use `bazel run` if you want to start an interactive program or pipe data to it.

Note the use of the `--`. This is needed so that Bazel does not interpret `--arg1` and `--arg2` as Bazel options, but rather as part of the command line for running the binary. (The program being run simply says hello and prints out its args.)

Options for `bazel run`

`--run_under=`*command-prefix*

This has the same effect as the `--run_under` option for `bazel test` (see above), except that it applies to the command being run by `bazel run` rather than to the tests being run by `bazel test` and cannot run under label.

Executing tests

`bazel run` can also execute test binaries, which has the effect of running the test, but without the setup documented on the page Writing Tests (test-encyclopedia.html), so that the test runs in an environment closer to the current shell environment. Note that none of the --test_* arguments have an effect when running a test in this manner.

Querying the dependency graph with Bazel

Bazel includes a query language for asking questions about the dependency graph used during the build. The query tool is an invaluable aid to many software engineering tasks.

The query language is based on the idea of algebraic operations over graphs; it is documented in detail in Bazel Query Reference (query.html). Please refer to that document for reference, for examples, and for query-specific command-line options.

The query tool accepts several command-line option. `--output` selects the output format. `--[no]keep_going` (disabled by default) causes the query tool to continue to make progress upon errors; this behavior may be disabled if an incomplete result is not acceptable in case of errors.

The `--[no]host_deps` option, enabled by default, causes dependencies on "host configuration" targets to be included in the dependency graph over which the query operates.

The `--[no]implicit_deps` option, enabled by default, causes implicit dependencies to be included in the dependency graph over which the query operates. An implicit dependency is one that is not explicitly specified in the BUILD file but added by bazel.

Example: "Show the locations of the definitions (in BUILD files) of all genrules required to build all the tests in the PEBL tree."

```
bazel query --output location 'kind(genrule, deps(kind(".*_test rule", foo/bar/pebl/...)
```

Miscellaneous Bazel commands and options

The `help` command

The `help` command provides on-line help. By default, it shows a summary of available commands and help topics, as shown in the *Bazel overview* section above. Specifying an argument displays detailed help for a particular topic. Most topics are Bazel commands, e.g. `build` or `query`, but there are some additional help topics that do not correspond to commands.

`--[no]long (-l)`

By default, `bazel help [topic]` prints only a summary of the relevant options for a topic. If the `--long` option is specified, the type, default value and full description of each option is also printed.

The `shutdown` command

Bazel server processes (see Client/server implementation) may be stopped by using the `shutdown` command. This command causes the Bazel server to exit as soon as it becomes idle (i.e. after the completion of any builds or other commands that are currently in progress). Bazel servers stop themselves after an idle timeout, so this command is rarely necessary; however, it can be useful in scripts when it is known that no further builds will occur in a given workspace.

`shutdown` accepts one option, `--iff_heap_size_greater_than n`, which requires an integer argument (in MB). If specified, this makes the shutdown conditional on the amount of memory already consumed. This is useful for scripts that initiate a lot of builds, as any memory leaks in the Bazel server could cause it to crash spuriously on occasion; performing a conditional restart preempts this condition.

The `info` command

The `info` command prints various values associated with the Bazel server instance, or with a specific build configuration. (These may be used by scripts that drive a build.)

The `info` command also permits a single (optional) argument, which is the name of one of the keys in the list below. In this case, `bazel info key` will print only the value for that one key. (This is especially convenient when scripting Bazel, as it avoids the need to pipe the result through `sed -ne /key:/s/key://p`:

Configuration-independent data

- `release`: the release label for this Bazel instance, or "development version" if this is not a released binary.
- `workspace` the absolute path to the base workspace directory.
- `install_base`: the absolute path to the installation directory used by this Bazel instance for the current user. Bazel installs its internally required executables below this directory.
- `output_base`: the absolute path to the base output directory used by this Bazel instance for the current user and workspace combination. Bazel puts all of its scratch and build output below this directory.
- `execution_root`: the absolute path to the execution root directory under output_base. This directory is the root for all files accessible to commands executed during the build, and is the working directory for those commands. If the workspace directory is writable, a symlink named `bazel-<workspace>` is placed there pointing to this directory.
- `output_path`: the absolute path to the output directory beneath the execution root used for all files actually

generated as a result of build commands. If the workspace directory is writable, a symlink named `bazel-out` is placed there pointing to this directory.
- `server_pid` : the process ID of the Bazel server process.
- `command_log` : the absolute path to the command log file; this contains the interleaved stdout and stderr streams of the most recent Bazel command. Note that running `bazel info` will overwrite the contents of this file, since it then becomes the most recent Bazel command. However, the location of the command log file will not change unless you change the setting of the `--output_base` or `--output_user_root` options.
- `used-heap-size` , `committed-size` , `max-heap-size` : reports various JVM heap size parameters. Respectively: memory currently used, memory currently guaranteed to be available to the JVM from the system, maximum possible allocation.
- `gc-count` , `gc-time` : The cumulative count of garbage collections since the start of this Bazel server and the time spent to perform them. Note that these values are not reset at the start of every build.
- `package_path` : A colon-separated list of paths which would be searched for packages by bazel. Has the same format as the `--package_path` build command line argument.

Example: the process ID of the Bazel server.

```
% bazel info server_pid
1285
```

Configuration-specific data

These data may be affected by the configuration options passed to `bazel info` , for example `--cpu` , `--compilation_mode` , etc. The `info` command accepts all the options that control dependency analysis, since some of these determine the location of the output directory of a build, the choice of compiler, etc.

- `bazel-bin` , `bazel-testlogs` , `bazel-genfiles` : reports the absolute path to the `bazel-*` directories in which programs generated by the build are located. This is usually, though not always, the same as the `bazel-*` symlinks created in the base workspace directory after a successful build. However, if the workspace directory is read-only, no `bazel-*` symlinks can be created. Scripts that use the value reported by `bazel info` , instead of assuming the existence of the symlink, will be more robust.
- The complete "Make" environment (be/make-variables.html). If the `--show_make_env` flag is specified, all variables in the current configuration's "Make" environment are also displayed (e.g. `CC` , `GLIBC_VERSION` , etc). These are the variables accessed using the `$(CC)` or `varref("CC")` syntax inside BUILD files.

Example: the C++ compiler for the current configuration. This is the `$(CC)` variable in the "Make" environment, so the `--show_make_env` flag is needed.

```
% bazel info --show_make_env -c opt COMPILATION_MODE
opt
```

Example: the `bazel-bin` output directory for the current configuration. This is guaranteed to be correct even in cases where the `bazel-bin` symlink cannot be created for some reason (e.g. you are building from a read-only directory).

The `version` command

The version command prints version details about the built Bazel binary, including the changelist at which it was

built and the date. These are particularly useful in determining if you have the latest Bazel, or if you are reporting bugs. Some of the interesting values are:

- `changelist` : the changelist at which this version of Bazel was released.
- `label` : the release label for this Bazel instance, or "development version" if this is not a released binary. Very useful when reporting bugs.

The `mobile-install` command

The `mobile-install` command installs apps to mobile devices. Currently only Android devices running ART are supported. See bazel mobile-install (mobile-install.html) for more information.

Note that this command does not install the same thing that `bazel build` produces: Bazel tweaks the app so that it can be built, installed and re-installed quickly. This should, however, be mostly transparent to the app.

The following options are supported:

`--incremental`

If set, Bazel tries to install the app incrementally, that is, only those parts that have changed since the last build. This cannot update resources referenced from `AndroidManifest.xml` , native code or Java resources (i.e. ones referenced by `Class.getResource()`). If these things change, this option must be omitted. Contrary to the spirit of Bazel and due to limitations of the Android platform, it is the **responsibility of the user** to know when this command is good enough and when a full install is needed. If you are using a device with Marshmallow or later, consider the `--split_apks` flag.

`--split_apks`

Whether to use split apks to install and update the application on the device. Works only with devices with Marshmallow or later. Note that the `--incremental` flag is not necessary when using `--split_apks` .

`--start_app`

Starts the app in a clean state after installing. Equivalent to `--start=COLD` .

`--debug_app`

Waits for debugger to be attached before starting the app in a clean state after installing. Equivalent to `--start=DEBUG` .

`--start=`*`start_type`*

How the app should be started after installing it. Supported *start_type*s are:

- `NO` Does not start the app. This is the default.
- `COLD` Starts the app from a clean state after install.
- `WARM` Preserves and restores the application state on incremental installs.
- `DEBUG` Waits for the debugger before starting the app in a clean state after install.

Note that if more than one of `--start=start_type`, `--start_app` or `--debug_app` is set, the last value will be used.

`--adb path`

Indicates the `adb` binary to be used. The default is to use the adb in the Android SDK specified by `--android_sdk`.

`--adb_arg arg`

Extra arguments to `adb`. These come before the subcommand in the command line and are typically used to specify which device to install to. For example, to select the Android device or emulator to use:

```
% bazel mobile-install --adb_arg=-s --adb_arg=deadbeef
```

will invoke `adb` as

```
adb -s deadbeef install ...
```

`--incremental_install_verbosity number`

The verbosity for incremental install. Set to 1 for debug logging to be printed to the console.

The `dump` command

The `dump` command prints to stdout a dump of the internal state of the Bazel server. This command is intended primarily for use by Bazel developers, so the output of this command is not specified, and is subject to change.

By default, command will just print help message outlining possible options to dump specific areas of the Bazel state. In order to dump internal state, at least one of the options must be specified.

Following options are supported:

- `--action_cache` dumps action cache content.
- `--packages` dumps package cache content.
- `--skyframe` dumps state of internal Bazel dependency graph.
- `--rules` dumps rule summary for each rule and aspect class, including counts and action counts. This includes both native and Skylark rules. If memory tracking is enabled, then the rules' memory consumption is also printed.
- `--skylark_memory` dumps a pprof compatible .gz file to the specified path. You must enable memory tracking for this to work.
- `--action_graph=/path/to/file` dumps the state of the internal Bazel action graph in proto format to `/path/to/file`. You have to run (at least) the analysis phase for the targets you are interested in (for example, `bazel build --nobuild //foo:bar`). Note that this feature is still experimental, subject to change and will probably be integrated into `cquery` in the future.
- `--action_graph:targets=target1,target2,...` filters the actions to the comma-separated list of targets when dumping the action graph.

Memory tracking

Some `dump` commands require memory tracking. To turn this on, you have to pass startup flags to Bazel:

- `--host_jvm_args=-javaagent:$BAZEL/third_party/allocation_instrumenter/java-allocation-instrumenter-3.0.1.jar`
- `--host_jvm_args=-DRULE_MEMORY_TRACKER=1`

The java-agent is checked into bazel at third_party/allocation_instrumenter/java-allocation-instrumenter-3.0.1.jar, so make sure you adjust `$BAZEL` for where you keep your bazel repository. Do not forget to keep passing these options to Bazel for every command or the server will restart.

Example:

```
% bazel --host_jvm_args=-javaagent:$BAZEL/third_party/allocation_instrumenter/java-all
--host_jvm_args=-DRULE_MEMORY_TRACKER=1 \
build --nobuild <targets>

# Dump rules
% bazel --host_jvm_args=-javaagent:$BAZEL/third_party/allocation_instrumenter/java-all
--host_jvm_args=-DRULE_MEMORY_TRACKER=1 \
dump --rules

# Dump Skylark heap and analyze it with pprof
% bazel --host_jvm_args=-javaagent:$BAZEL/third_party/allocation_instrumenter/java-all
--host_jvm_args=-DRULE_MEMORY_TRACKER=1 \
dump --skylark_memory=$HOME/prof.gz
% pprof -flame $HOME/prof.gz
```

The `analyze-profile` command

The `analyze-profile` command analyzes data previously gathered during the build using `--profile` option. It provides several options to either perform analysis of the build execution or export data in the specified format.

The following options are supported:

- `--dump=text` displays all gathered data in a human-readable format
- `--dump=raw` displays all gathered data in a script-friendly format
- `--html` generates an HTML file visualizing the actions and rules executed in the build, as well as summary statistics for the build
  - `--html_details` adds more fine-grained information on actions and rules to the HTML visualization
    - `--html_histograms` adds histograms for Skylark functions clicked in the statistics table. This will increase file size massively
    - `--nochart` hides the task chart from generated HTML
- `--combine` combines multiple profile data files into a single report. Does not generate HTML task charts
- `--task_tree` prints the tree of tasks matching the given regular expression
  - `--task_tree_threshold` skip tasks with duration less than threshhold, in milliseconds. Default is 50ms

See the section on Troubleshooting performance by profiling for format details and usage help.

The `canonicalize-flags` command

The `canonicalize-flags` command, which takes a list of options for a Bazel command and returns a list of options that has the same effect. The new list of options is canonical, i.e., two lists of options with the same effect are canonicalized to the same new list.

The `--for_command` option can be used to select between different commands. At this time, only `build` and `test` are supported. Options that the given command does not support cause an error.

Note that a small number of options cannot be reordered, because Bazel cannot ensure that the effect is identical.

Bazel startup options

The options described in this section affect the startup of the Java virtual machine used by Bazel server process, and they apply to all subsequent commands handled by that server. If there is an already running Bazel server and the startup options do not match, it will be restarted.

All of the options described in this section must be specified using the `--key=value` or `--key value` syntax. Also, these options must appear *before* the name of the Bazel command.

`--output_base=`*`dir`*

This option requires a path argument, which must specify a writable directory. Bazel will use this location to write all its output. The output base is also the key by which the client locates the Bazel server. By changing the output base, you change the server which will handle the command.

By default, the output base is derived from the user's login name, and the name of the workspace directory (actually, its MD5 digest), so a typical value looks like: `/var/tmp/google/_bazel_johndoe/d41d8cd98f00b204e9800998ecf8427e`. Note that the client uses the output base to find the Bazel server instance, so if you specify a different output base in a Bazel command, a different server will be found (or started) to handle the request. It's possible to perform two concurrent builds in the same workspace directory by varying the output base.

For example:

```
% bazel --output_base /tmp/1 build //foo  &  bazel --output_base /tmp/2 build //bar
```

In this command, the two Bazel commands run concurrently (because of the shell `&` operator), each using a different Bazel server instance (because of the different output bases). In contrast, if the default output base was used in both commands, then both requests would be sent to the same server, which would handle them sequentially: building `//foo` first, followed by an incremental build of `//bar`.

We recommend you do not use NFS locations for the output base, as the higher access latency of NFS will cause noticeably slower builds.

`--output_user_root=`*`dir`*

By default, the `output_base` value is chosen to as to avoid conflicts between multiple users building in the same workspace directory. In some situations, though, it is desirable to build from a directory shared between multiple users; release engineers often do this. In those cases it may be useful to deliberately override the default so as to ensure "conflicts" (i.e., sharing) between multiple users. Use the `--output_user_root` option to achieve this: the output base is placed in a subdirectory of the output user root, with a unique name based on the workspace, so the result of using an output user root that is not a function of `$USER` is sharing. Of course, it is important to ensure (via umask and group membership) that all the cooperating users can read/write each others files.

If the `--output_base` option is specified, it overrides using `--output_user_root` to calculate the output base.

The install base location is also calculated based on `--output_user_root`, plus the MD5 identity of the Bazel embedded binaries.

You can also use the `--output_user_root` option to choose an alternate base location for all of Bazel's output (install base and output base) if there is a better location in your filesystem layout.

--host_javabase=*dir*

Specifies the Java virtual machine in which *Bazel itself* runs. The value must be a path to the directory containing a JDK or JRE. It should not be a label. This option should appear before any bazel command, and not be confused with the build option --host_javabase, for example:

```
% bazel --host_javabase=/usr/local/buildtools/java/jdk9 build //foo
```

This flag does *not* affect the JVMs used by Bazel subprocesses such as applications, tests, tools, and so on. Use build options --javabase or --host_javabase instead.

--host_jvm_args=*string*

Specifies a startup option to be passed to the Java virtual machine in which *Bazel itself* runs. This can be used to set the stack size, for example:

```
% bazel --host_jvm_args="-Xss256K" build //foo
```

This option can be used multiple times with individual arguments. Note that setting this flag should rarely be needed. You can also pass a space-separated list of strings, each of which will be interpreted as a separate JVM argument, but this feature will soon be deprecated.

That this does *not* affect any JVMs used by subprocesses of Bazel: applications, tests, tools, etc. To pass JVM options to executable Java programs, whether run by `bazel run` or on the command-line, you should use the `--jvm_flags` argument which all `java_binary` and `java_test` programs support. Alternatively for tests, use `bazel test --test_arg=--jvm_flags=foo ...`.

--host_jvm_debug

This option causes the Java virtual machine to wait for a connection from a JDWP-compliant debugger before calling the main method of *Bazel itself*. This is primarily intended for use by Bazel developers.

(Please note that this does *not* affect any JVMs used by subprocesses of Bazel: applications, tests, tools, etc.)

`--batch`

This switch will cause bazel to be run in batch mode, instead of the standard client/server mode described above. Doing so provides more predictable semantics with respect to signal handling, job control, and environment variable inheritance, and is necessary for running bazel in a chroot jail.

Batch mode retains proper queueing semantics within the same output_base. That is, simultaneous invocations will be processed in order, without overlap. If a batch mode bazel is run on a client with a running server, it first kills the server before processing the command.

Bazel will run slower in batch mode, compared to client/server mode. Among other things, the build file cache is memory-resident, so it is not preserved between sequential batch invocations. Therefore, using batch mode often makes more sense in cases where performance is less critical, such as continuous builds.

`--max_idle_secs` *n*

This option specifies how long, in seconds, the Bazel server process should wait after the last client request, before it exits. The default value is 10800 (3 hours).

This option may be used by scripts that invoke Bazel to ensure that they do not leave Bazel server processes on a user's machine when they would not be running otherwise. For example, a presubmit script might wish to invoke `bazel query` to ensure that a user's pending change does not introduce unwanted dependencies. However, if the user has not done a recent build in that workspace, it would be undesirable for the presubmit script to start a Bazel server just for it to remain idle for the rest of the day. By specifying a small value of `--max_idle_secs` in the query request, the script can ensure that *if* it caused a new server to start, that server will exit promptly, but if instead there was already a server running, that server will continue to run until it has been idle for the usual time. Of course, the existing server's idle timer will be reset.

`--[no]block_for_lock`

If enabled, Bazel will wait for other Bazel commands holding the server lock to complete before progressing. If disabled, Bazel will exit in error if it cannot immediately acquire the lock and proceed. Developers might use this in presubmit checks to avoid long waits caused by another Bazel command in the same client.

`--io_nice_level` *n*

Sets a level from 0-7 for best-effort IO scheduling. 0 is highest priority, 7 is lowest. The anticipatory scheduler may only honor up to priority 4. Negative values are ignored.

`--batch_cpu_scheduling`

Use `batch` CPU scheduling for Bazel. This policy is useful for workloads that are non-interactive, but do not want to lower their nice value. See 'man 2 sched_setscheduler'. This policy may provide for better system interactivity at the expense of Bazel throughput.

Miscellaneous options

`--[no]announce_rc`

Controls whether Bazel announces command options read from the bazelrc file when starting up. (Startup options are unconditionally announced.)

`--color (yes|no|auto)`

This option determines whether Bazel will use colors to highlight its output on the screen.

If this option is set to `yes`, color output is enabled. If this option is set to `auto`, Bazel will use color output only if the output is being sent to a terminal and the TERM environment variable is set to a value other than `dumb`, `emacs`, or `xterm-mono`. If this option is set to `no`, color output is disabled, regardless of whether the output is going to a terminal and regardless of the setting of the TERM environment variable.

`--config` *name*

Selects additional config section from the rc files; for the current `command`, it also pulls in the options from `command:name` if such a section exists. Can be specified multiple times to add flags from several config sections. Expansions can refer to other definitions (i.e. expansions can be chained).

`--curses (yes|no|auto)`

This option determines whether Bazel will use cursor controls in its screen output. This results in less scrolling data, and a more compact, easy-to-read stream of output from Bazel. This works well with `--color`.

If this option is set to `yes`, use of cursor controls is enabled. If this option is set to `no`, use of cursor controls is disabled. If this option is set to `auto`, use of cursor controls will be enabled under the same conditions as for `--color=auto`.

`--[no]show_timestamps`

If specified, a timestamp is added to each message generated by Bazel specifying the time at which the message was displayed.

Calling Bazel from scripts

Bazel can be called from scripts in order to perform a build, run tests or query the dependency graph. Bazel has been designed to enable effective scripting, but this section lists some details to bear in mind to make your scripts more robust.

Choosing the output base

The `--output_base` option controls where the Bazel process should write the outputs of a build to, as well as various working files used internally by Bazel, one of which is a lock that guards against concurrent mutation of the output base by multiple Bazel processes.

Choosing the correct output base directory for your script depends on several factors. If you need to put the build

outputs in a specific location, this will dictate the output base you need to use. If you are making a "read only" call to Bazel (e.g. `bazel query`), the locking factors will be more important. In particular, if you need to run multiple instances of your script concurrently, you will need to give each one a different (or random) output base.

If you use the default output base value, you will be contending for the same lock used by the user's interactive Bazel commands. If the user issues long-running commands such as builds, your script will have to wait for those commands to complete before it can continue.

Server or no server?

By default, Bazel uses a long-running server process as an optimization; this behavior can be disabled using the `--batch` option. There's no hard and fast rule about whether or not your script should use a server, but in general, the trade-off is between performance and reliability. The server mode makes a sequence of builds, especially incremental builds, faster, but its behavior is more complex and more likely to fail. We recommend in most cases that you use batch mode unless the performance advantage is critical.

If you do use the server, don't forget to call `shutdown` when you're finished with it, or, specify `--max_idle_secs=5` so that idle servers shut themselves down promptly.

What exit code will I get?

Bazel attempts to differentiate failures due to the source code under consideration from external errors that prevent Bazel from executing properly. Bazel execution can result in following exit codes:

Exit Codes common to all commands:

- `0` - Success
- `2` - Command Line Problem, Bad or Illegal flags or command combination, or Bad Environment Variables. Your command line must be modified.
- `8` - Build Interrupted but we terminated with an orderly shutdown.
- `32` - External Environment Failure not on this machine.
- `33` - OOM failure. You need to modify your command line.
- `34` - Reserved for Google-internal use.
- `35` - Reserved for Google-internal use.
- `36` - Local Environmental Issue, suspected permanent.
- `37` - Unhandled Exception / Internal Bazel Error.
- `38` - Reserved for Google-internal use.
- `40-44` - Reserved for errors in Bazel's command line launcher, `bazel.cc` that are not command line related. Typically these are related to bazel server being unable to launch itself.

Return codes for commands `bazel build`, `bazel test`.

- `1` - Build failed.
- `3` - Build OK, but some tests failed or timed out.
- `4` - Build successful but no tests were found even though testing was requested.

For `bazel run`:

- `1` - Build failed.
- `6` - Run command failure. The executed subprocess returned a non-zero exit code. The actual subprocess exit code is given in stderr.

For `bazel query`:

- `3` - Partial success, but the query encountered 1 or more errors in the input BUILD file set and therefore the results of the operation are not 100% reliable. This is likely due to a `--keep_going` option on the command line.
- `7` - Command failure.

Future Bazel versions may add additional exit codes, replacing generic failure exit code `1` with a different non-zero value with a particular meaning. However, all non-zero exit values will always constitute an error.

Reading the .bazelrc file

By default, Bazel will read the `.bazelrc` file from the base workspace directory or the user's home directory. Whether or not this is desirable is a choice for your script; if your script needs to be perfectly hermetic (e.g. when doing release builds), you should disable reading the .bazelrc file by using the option `--bazelrc=/dev/null`. If you want to perform a build using the user's preferred settings, the default behavior is better.

Command log

The Bazel output is also available in a command log file which you can find with the following command:

```
% bazel info command_log
```

The command log file contains the interleaved stdout and stderr streams of the most recent Bazel command. Note that running `bazel info` will overwrite the contents of this file, since it then becomes the most recent Bazel command. However, the location of the command log file will not change unless you change the setting of the `--output_base` or `--output_user_root` options.

Parsing output

The Bazel output is quite easy to parse for many purposes. Two options that may be helpful for your script are `--noshow_progress` which suppresses progress messages, and `--show_result n`, which controls whether or not "build up-to-date" messages are printed; these messages may be parsed to discover which targets were successfully built, and the location of the output files they created. Be sure to specify a very large value of *n* if you rely on these messages.

Troubleshooting performance by profiling

The first step in analyzing the performance of your build is to profile your build with the `--profile` option.

The file generated by the `--profile` command is a binary file. Once you have generated this binary profile, you can analyze it using Bazel's `analyze-profile` command. By default, it will print out summary analysis information for each of the specified profile datafiles. This includes cumulative statistics for different task types for each build phase and an analysis of the critical execution path.

The first section of the default output describes an overview of the time spent on the different build phases:

```
=== PHASE SUMMARY INFORMATION ===

Total launch phase time         6.00 ms    0.01%
Total init phase time            864 ms    1.11%
Total loading phase time       21.841 s   28.05%
Total analysis phase time       5.444 s    6.99%
Total preparation phase time     155 ms    0.20%
Total execution phase time     49.473 s   63.54%
Total finish phase time         83.9 ms    0.11%
Total run time                 77.866 s  100.00%
```

The following sections show the execution time of different tasks happening during a particular phase:

```
=== INIT PHASE INFORMATION ===

Total init phase time                      864 ms

Total time (across all threads) spent on:
               Type    Total    Count     Average
          VFS_STAT    2.72%        1     23.5 ms
      VFS_READLINK   32.19%        1      278 ms

=== LOADING PHASE INFORMATION ===

Total loading phase time                21.841 s

Total time (across all threads) spent on:
               Type    Total    Count     Average
              SPAWN    3.26%      154      475 ms
          VFS_STAT   10.81%    65416     3.71 ms
[...]
SKYLARK_BUILTIN_FN   13.12%    45138     6.52 ms

=== ANALYSIS PHASE INFORMATION ===

Total analysis phase time                5.444 s

Total time (across all threads) spent on:
               Type    Total    Count     Average
     SKYFRAME_EVAL    9.35%        1     4.782 s
       SKYFUNCTION   89.36%    43332     1.06 ms

=== EXECUTION PHASE INFORMATION ===

Total preparation time                    155 ms
Total execution phase time              49.473 s
Total time finalizing build              83.9 ms

Action dependency map creation           0.00 ms
Actual execution time                   49.473 s

Total time (across all threads) spent on:
               Type    Total    Count     Average
             ACTION    2.25%    12229     10.2 ms
[...]
       SKYFUNCTION    1.87%   236131     0.44 ms
```

The last section shows the critical path:

```
Critical path (32.078 s):
    Id        Time Percentage   Description
1109746     5.171 s   16.12%   Building [...]
1109745      164 ms    0.51%   Extracting interface [...]
1109744     4.615 s   14.39%   Building [...]
[...]
1109639     2.202 s    6.86%   Executing genrule [...]
1109637     2.00 ms    0.01%   Symlinking [...]
1109636      163 ms    0.51%   Executing genrule [...]
          4.00 ms    0.01%   [3 middleman actions]
```

You can use the following options to display more detailed information:

- `--dump=text`
  This option prints all recorded tasks in the order they occurred. Nested tasks are indented relative to the parent. For each task, output includes the following information:

  ```
  [task type] [task description]
  Thread: [thread id]    Id: [task id]     Parent: [parent task id or 0 for top-level t
  Start time: [time elapsed from the profiling session start]       Duration: [task dur
  [aggregated statistic for nested tasks, including count and total duration for each n
  ```

- `--dump=raw`
  This option is most useful for automated analysis with scripts. It outputs each task record on a single line using '|' delimiter between fields. Fields are printed in the following order:

  1. thread id - integer positive number, identifies owner thread for the task
  2. task id - integer positive number, identifies specific task
  3. parent task id for nested tasks or 0 for root tasks
  4. task start time in ns, relative to the start of the profiling session
  5. task duration in ns. Please note that this will include duration of all subtasks.
  6. aggregated statistic for immediate subtasks per type. This will include type name (lower case), number of subtasks for that type and their cumulative duration. Types are space-delimited and information for single type is comma-delimited.
  7. task type (upper case)
  8. task description

  Example:

  ```
  1|1|0|0|0||PHASE|Launch Bazel
  1|2|0|6000000|0||PHASE|Initialize command
  1|3|0|168963053|278111411||VFS_READLINK|/[...]
  1|4|0|571055781|23495512||VFS_STAT|/[...]
  1|5|0|869955040|0||PHASE|Load packages
  [...]
  ```

- `--html`
  This option writes a file called `<profile-file>.html` in the directory of the profile file. Open it in your browser to see the visualization of the actions in your build. Note that the file can be quite large and may push the capabilities of your browser – please wait for the file to load.

In most cases, the HTML output from `--html` is easier to read than the `--dump` output. It includes a Gantt chart that displays time on the horizontal axis and threads of execution along the vertical axis. If you click on the Statistics link in the top right corner of the page, you will jump to a section that lists summary analysis information from your build.

- `--html_details`
  Additionally passing this option will render a more detailed execution chart and additional tables on the performance of built-in and user-defined Skylark functions. Beware that this increases the file size and the load on the browser considerably.

If Bazel appears to be hung, you can hit **`ctrl`** + `\` or send Bazel a `SIGQUIT` signal (`kill -3 $(bazel info server_pid)`) to get a thread dump in the file `$(bazel info output_base)/server/jvm.out` .

Since you may not be able to run `bazel info` if bazel is hung, the `output_base` directory is usually the parent of the `bazel-<workspace>` symlink in your workspace directory.

About

Who's using Bazel (https://github.com/bazelbuild/bazel/wiki/Bazel-Users)
Roadmap (https://www.bazel.build/roadmap.html)
Contribute (https://www.bazel.build/contributing.html)
Governance Plan (https://www.bazel.build/governance.html)

Support

Stack Overflow (http://stackoverflow.com/questions/tagged/bazel)
Issue Tracker (https://github.com/bazelbuild/bazel/issues)
Documentation (https://docs.bazel.build)
FAQ (https://www.bazel.build/faq.html)
Support Policy (https://www.bazel.build/support.html)

Stay Connected

Twitter (https://twitter.com/bazelbuild)
Blog (https://blog.bazel.build)
GitHub (https://github.com/bazelbuild/bazel)
Discussion group (https://groups.google.com/forum/#!forum/bazel-discuss)

© 2018 Google