O'REILLY®
Report

Extending Bazel to Its Full Potential

Leveraging Cloud and Parallelization to Ship Reliable Code Faster

Brian Moakley
& Marcus Eagan

Compliments of
nativelink

Bulid faster.
Test consistently.
Deploy reliably.

Massively parallel remote execution
platform

Adaptive scheduler
optimizes costs (CPUs,GPUs)

Intelligent caching
for faster builds & retraining

Seamless integration
Open source. Bazel, Siso/Reclient, Pants. C/C++, Python, Go, Rust & more

Dependency management
ensures reproducibility

nativelink

Start free with
Nativelink Cloud
app.nativelink.com

Extending Bazel to Its Full Potential

Leveraging Cloud and Parallelization to Ship Reliable Code Faster

Brian Moakley and Marcus Eagan

O'REILLY®

Extending Bazel to Its Full Potential
by Brian Moakley and Marcus Eagan

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Megan Laddusaw
Development Editor: Gary O'Brien
Production Editor: Christopher Faucher
Copyeditor: Penelope Perkins
Proofreader: O'Reilly Media, Inc.
Cover Designer: Ellie Volckhausen
Cover Illustrator: Ellie Volckhausen
Interior Designer: David Futato
Interior Illustrator: Kate Dullea

August 2025: First Edition

Revision History for the First Edition
2025-07-25: First Release

979-8-341-62392-7

[LSI]

Table of Contents

Extending Bazel to Its Full Potential. 1
Introducing Bazel 1
Installing Bazel 4
Bazel Quick Start 8
Dependency Management 20
Executing Automated Tests 28
Extending Bazel's Capabilities 31
Building Beyond Your Local Machine 35
Conclusion 45

Extending Bazel to Its Full Potential

Modern-day build systems are just as diverse and dynamic as the many programming languages out there. While build systems are developed for a wide assortment of workflows and use cases, there are three universal expectations from end users: such a system must be fast, correct, and well-optimized.

Some build systems meet these expectations, but few are specifically built from the ground up to fulfill these expectations in a single stroke.

Introducing Bazel

Bazel, first released in 2015, is the open source version of Google's Blaze build system. Blaze started development in 2006 as a way to unify Google's growing codebase. Google famously keeps the vast majority of its code in a single repository, otherwise known as a *monorepo*. This repo contains almost every project in Google, allowing any Google engineer to build any Google project with a single command. Such a build system is required to be fast, scale with the number of developers using it, and, most importantly, produce identical builds regardless of the builder's machine.

Blaze was the answer. Google designed Blaze for fast, correct builds. By using a distributed cache, Blaze scales with the number of developers using it, condensing build times from days to hours, and from hours to minutes. While ideal for a monorepo, Blaze works just as well for projects composed of many different repos.

By 2014, after Blaze had become quite popular with its own developers, Google decided to make it available as an open source product as a way to sell its cloud service for remote builds. Google called the open source project Bazel, an anagram of Blaze.

You can run Bazel completely locally on your desktop without needing remote execution or cloud services, but you'll see later how it has also been extended to support distributed teams with remote shared caching, parallelized builds, and cloud-based storage.

What Is Bazel?

Bazel is known as an *artifact-based* build system. In an artifact-based build system, developers define a build file that includes the various "artifacts," such as the dependencies and resources required for the build. Once they are defined, Bazel analyzes the build and generates all the tasks needed to construct a build. Bazel will either generate artifacts or fetch artifacts from a shared cache or online repositories.

The central build file is declarative. When defining a build, a developer provides the inputs, and Bazel manages the process of building all the various dependencies. By analyzing the dependency graph, Bazel is aware of the available resources. This means Bazel can fully compile resources in parallel as needed. It can also include compiled artifacts that haven't changed from previous builds.

Bazel is a *deterministic* build system. Simply put, when a build file defines the same inputs, Bazel will produce the same output. This ensures each resulting build is "correct" (i.e., deterministic). Two developers on two different machines with two different configurations will be able to produce an identical build.

Why Should I Use Bazel?

You might be asking: What's in it for me? Aren't the tool's conventions too specific for my project? Does my project even meet the requirements of being buildable by Bazel? Here's what Bazel can bring to the table in a nutshell:

Declarative language

As a developer of Bazel build logic, you will use a higher-level language called Starlark, a Python derivative. Starlark introduces an abstraction to the concepts of a build and hides its implementation complexities as much as possible. As a result, you do not have to worry about low-level implementation details like compilers or linkers. Instead, you just point your build to the source code and declare dependencies. Bazel will figure out the rest. Needless to say, you can still fine-tune the compiler or linker settings if needed.

Reproducibility

When executing builds over and over again, you do not want any surprises. Nondeterministic behavior erodes trust in the correctness of build results. Bazel ensures a sandboxed build execution by explicitly enforcing the definition of all of its dependencies.

Scalability

Bazel's main focus is on projects with large codebases, predominantly for organizations that have decided to put all of their projects into a monorepo. But it's not a dealbreaker if you separate your projects into individual source code repositories. That's common practice, especially if you are working on software with a microservices architecture. Bazel can handle either code organizational structure equally well.

Parallel and distributed execution

Improvements to build performance become more apparent in larger codebases, as Bazel can execute its work in parallel and in a distributed fashion. Build execution can be performed on a single machine or distributed across multiple remote machines (e.g., located in the cloud or in a data center).

Building polyglot projects

Many build tools support building only a single language or ecosystem. That's not the case with Bazel, which can handle polyglot projects. For example, it supports the JVM (Java Virtual Machine) ecosystem, native languages, and JavaScript. Furthermore, Bazel embraces modern software development methodologies like containerization of applications with Docker and deployment to orchestration engines like Kubernetes.

Extensibility
: It's not uncommon for projects to have custom requirements. While Bazel's built-in support for languages and ecosystems is expansive, it cannot cover every possible use case. With the help of Bazel's extension mechanism, called *rules*, developers can enhance the tool's base functionality and share it across the organization or wider community.

Long-term support
: One of the biggest advantages to using Bazel is that Google is driving it, which means that the project benefits from years of in-house use and evolution at Google. Moreover, with Bazel's move to open source, it's also backed by a dedicated team of Google developers. As a result, you can expect bug fixes, new features, and long-term support. The latter was confirmed explicitly in the 1.0 release announcement.

While all the aforementioned aspects of Bazel make for a compelling build system, the true "killer feature" is the adoption of the remote build execution (RBE) protocol. This open source protocol allows for remote execution and caching, enabling you to improve performance, scalability, and resource utilization by distributing large-scale builds across compute clusters and sharing build artifacts across distributed teams. In "Building Beyond Your Local Machine," you'll learn how the Nativelink service uses this protocol to provide the infrastructure for remote builds.

I won't compare Bazel with other build tools in detail to see how they stack up—doing so would require a whole other report. Hopefully, however, the next couple of sections will give you a sense of its capabilities. You can find all the source code in a dedicated repository on GitHub if you'd like to follow along.

Installing Bazel

One of the nice things about Bazel is that it can be run in a large variety of environments. That said, Bazel only officially supports three platforms, while other platforms are community supported. You can also install Bazel from popular package managers including Homebrew, APT (Advanced Package Tool), and others. Finally, being an open source project, Bazel also provides instructions on installing from source.

Installation Options

At the time of writing, Bazel officially supports Windows, macOS, and Ubuntu. Given the variety of operating system distributions and versions, it's hard to determine 100% compatibility without trying it out. See the installation instructions for a detailed breakdown of distributions and versions.

When installing on Windows, you should ideally install Bazel on Windows 11, since Microsoft is no longer supporting Windows 10 after October 14, 2025. If you are using Windows 10, then you'll need to be running at least version 1703 (Creators Update). You'll also need to install the Microsoft Visual C++ Redistributable library.

While Bazel doesn't list any macOS requirements, it's always a good idea to use the latest version of the operating system that supports your device.

Alternatively, you can execute Bazel inside a Docker container. A Docker execution environment might be helpful if you just want to get familiar with Bazel without having to install a specific version of the runtime yourself. Docker containers are easy to stand up and can be disposed of after you've finished experimenting with them. The project provides a Docker container based on Ubuntu Linux with a preinstalled version of Bazel on the Google Cloud Marketplace. For detailed usage information, refer to the relevant section in the Bazel user manual.

Installing with Bazelisk

While Bazel provides lots of different installation options, the official documentation recommends that you install using Bazelisk, a Bazel wrapper written in Go. When running Bazel on the command line, you call Bazelisk with the same commands as Bazel.

Why use Bazelisk instead of Bazel itself? One of the key aspects of Bazel is being "correct." By providing the same inputs (source code files, libraries, etc.) you should always receive the same output. The same is true for Bazel itself. When compiling a project with Bazel, the build system itself is also an input.

When you build a project with Bazelisk, the wrapper will look for the *.bazelversion* file, which dictates the Bazel version, inside your project. When this file is executed, Bazelisk will automatically download and install the Bazel runtime and use it for the build. The

Bazel runtime for a specific version needs to be downloaded only once.

This functionality ensures that your whole team knows exactly which version of Bazel is required to build the project. Should the build fail for whatever reason, it won't be the result of an incompatible runtime version but of the build logic itself. Moreover, in a continuous integration (CI) environment, you need to ensure only that the Bazelisk runtime is installed. There's no need to maintain multiple Bazel versions in parallel independent of the CI execution environment (e.g., different CI agents or running the CI build in a Docker container).

Installing on Windows

To install Bazelisk on Windows, first make sure that you have installed the Microsoft Visual C++ Redistributable library. Next, you need to install the Chocolatey package manager.

To do this, open an administrative shell and run the following command:

```
> Get-ExecutionPolicy
```

If this returns "Restricted," run the following to elevate permissions so that Bazel can create symlinks before continuing with this procedure:

```
> Set-ExecutionPolicy AllSigned or
Set-ExecutionPolicy Bypass - Scope Process
```

Now, run the following command:

```
> Set-ExecutionPolicy Bypass -Scope Process -Force;
[System.Net.ServicePointManager]::SecurityProtocol =
[System.Net.ServicePointManager]::SecurityProtocol -bor 3072;
iex ((New-Object System.Net.WebClient).DownloadString(
'https://community.chocolatey.org/install.ps1'))
```

Finally, run these commands:

```
> choco install bazelisk
> bazelisk —-version
```

Installing on macOS

To install Bazelisk on macOS, your best bet is to use the Homebrew package manager. To get started, open a terminal and type the following to install Homebrew:

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/
Homebrew/install/HEAD/install.sh)"
```

Once Homebrew is installed, run the following command:

```
$ brew install bazelisk
$ bazelisk --version
```

Installing on Linux

While there may be Bazelisk packages served by package managers, it is recommended that you manually install the binary on Linux. You can access the binaries from the project release page. From there, download and install on your system. You will also need to update your `PATH` variable to point to the installation location.

Installing with npm

You can also install Bazelisk using npm, the node package manager. Run the following command:

```
> npm install -g @bazel/bazelisk
> bazelisk --version
```

Validating Your Installation

You can verify the Bazel installation by running the `help` command. This command runs the Bazel executable and renders valuable usage information on the console:

```
$ bazelisk help
WARNING: Invoking Bazel in batch mode since it is not invoked
from within a workspace (below a directory having a MODULE.bazel
file).

[bazel release 8.1.1]
Usage: bazel <command> <options>...
Available commands:
  analyze-profile     Analyzes build profile data.
  aquery              Analyzes the given targets and queries
                      the action graph.
  build               Builds the specified targets.
  canonicalize-flags  Canonicalizes a list of bazel options.
...
```

The initial warning indicates that you are accessing Bazel outside a build directory. This warning goes away once Bazel detects that a *MODULE* file is present.

Especially if you're a Bazel beginner, you'll find the `help` command invaluable as a quick reference instead of having to jump back and forth between the console and the documentation. The optional command-line interface (CLI) completion feature is an even more convenient feature.

Command-Line Completion

You might be familiar with the CLI completion functionality for other tools. Some shells propose commands when you press the Tab key or type certain letters. Bazel supports command-line completion functionality for the shells Bash and Zsh. You can install this feature as needed using the installation instructions on the Bazel website.

Bazel Quick Start

What's the best way to learn a new programming language or tool? By trying out a "Hello World" example, of course. In this section, we'll set up a C++-based project with the goal of compiling the source code and running the application. You don't need to be a C+ + expert; the concepts apply to Bazel's support for other languages as well. For a quick reference, refer to the Bazel documentation.

Basic Building Blocks

Every project in Bazel starts with a module file named (appropriately) *MODULE.bazel*. The module file resides in the root directory of your project. This is simply a text file with a *.bazel* extension that defines the various dependencies required by your build targets. It also provides metadata that other modules may use.

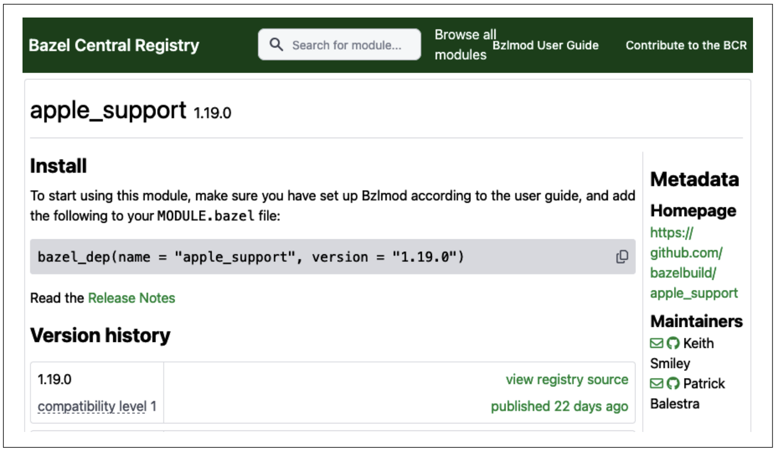
In previous versions of Bazel, projects defined a *WORKSPACE* file that was a "kitchen sink" approach for defining projects. It was where you defined dependencies, wrote extensions, and performed a variety of tasks. *WORKSPACE* files grew to be such a pain point that Bazel began work on the Bzlmod project to replace them. At the time of writing, Bazel 8 has disabled *WORKSPACE* files, and there are plans to remove *WORKSPACE* functionality in Bazel 9.

One of the big advantages of switching to a module workflow is that you no longer need to define transitive dependencies; this new system automatically fetches dependencies. Should a project's dependency graph share two libraries using different versions, Bazel will try to resolve the difference by using a library point release that can satisfy both versions. The Minimal Version Selection (MVS) algorithm is used to resolve the issue.

For example, if Project A requires SomeLibrary 1.10 and a later dependency requires SomeLibrary 1.20, Bazel will use the higher version, assuming that the point release won't break any functionality. This is a big assumption, and thankfully, in cases where point releases introduce problems, Bazel provides a means to define exact versions.

The true benefit of using this new module system is that modules can be imported into other modules. The Bazel Central Registry provides a list of hundreds of modules that you can include in your project. Each module page provides the module file code, version history, and of course, the project homepage.

The Bazel Central Registry provides instructions on publishing your own module to the registry. Submitting a module requires a few steps to validate the code and submit a pull request. Once approved, your own module will be listed in the registry as seen in Figure 1.



apple_support 1.19.0

Install

To start using this module, make sure you have set up Bzlmod according to the user guide, and add the following to your `MODULE.bazel` file:

```
bazel_dep(name = "apple_support", version = "1.19.0")
```

Read the Release Notes

Version history

1.19.0
compatibility level 1
view registry source
published 22 days ago

Metadata

Homepage
https://github.com/bazelbuild/apple_support

Maintainers
Keith Smiley
Patrick Balestra

Figure 1. Module page for the apple_support module

Using Build Files

As software projects grow in complexity over time, they're usually split up into modules. In a perfect world, modules group source code based on a dedicated function or domain responsibility. For example, you could organize a travel application by functionality for account management, reservations, and payment processing. It's very common for one module to need the functionality of another module, which requires that a dependency be defined at compile time and/or runtime.

In Bazel, a software module is called a *package*. The *BUILD* file (that is, *BUILD.bazel*) indicates that we are dealing with a package. A workspace can contain one or more packages and therefore one or more *BUILD* files. Figure 2 shows an exemplary setup of a project and its respective Bazel files.

NOTE

Now, you may say that the Bazel term *package* overlaps with what Java calls a package. You are absolutely right. It definitely makes discussing a Java project built with Bazel much harder than it needs to be. In this report, I will explicitly refer to either a Bazel package or a Java package, as they are two different concepts.

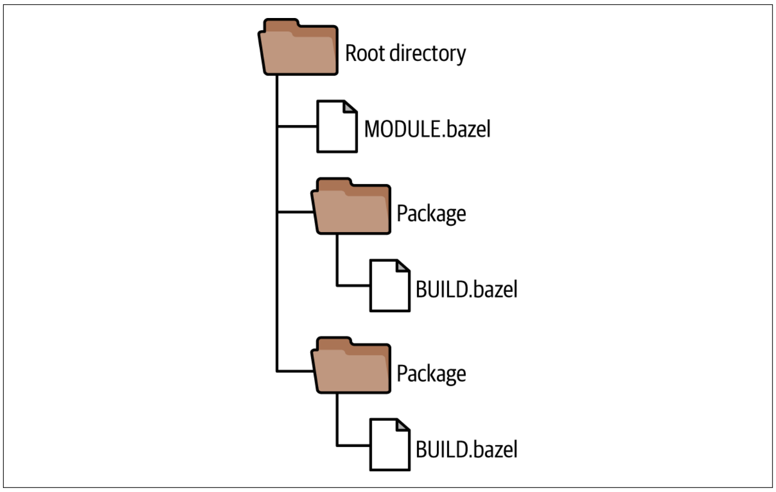


Figure 2. A sample project built with Bazel with two packages

For the purpose of building a simple C++ project, let's assume that we are just dealing with a single Bazel package. Later, we'll extend the setup of the build by breaking up the logic into a more fine-grained structure. Alongside this, we'll also talk about the pros and cons for each approach.

Building a Simple C++ Project

Out of the "box," Bazel supports the following languages and frameworks: C/C++, Java, Objective-C, Python, Android, and shell scripts. For other languages, you use community-developed rules.

In a Bazel, a *rule* is used to define custom functions that allow the build system to interact with target compilers, input files, and associated artifacts to produce an output. Each language will have its own set of rules. These rules are also used to extend Bazel to add additional functionality. While Bazel defines and supports the rules for supported languages, the vast majority of rules are community developed. Creating a custom rule is considered an advanced use of Bazel.

To get you started on your first build, we'll start with a very small and simple C++ app that prints a message to the console. Later, we'll explore some more advanced concepts by producing a Java console app with both tests and dependencies.

NOTE

While you can find this source code in the official repo for this report, this code was originally from Bazel's example repo, where you can find sample code for other programming languages as well.

Here is the simple directory structure breakdown of our C++ app:

```
├── MODULE
└── HelloWorld
    └── main
        └── BUILD
        └── hello-world.cc
```

As you can see, the root directory of the project contains the *MODULE* file. For now, the *MODULE* file is empty because our code doesn't require any external dependencies.

The *BUILD* file shown in Example 1-1 looks more interesting. We start defining our Bazel package by using our first *build rule*. As mentioned, a build rule knows how to build one or more outputs from a set of inputs. In this example, we are using the built-in `cc_binary` build rule. The sole input is represented by the source file. Notice that the source file is passed in as an array. You provide many file names or wild card characters to generate the file list. Once executed, the build rule will produce an executable file.

Example 1-1. Modeling a C++ binary by pointing to the source

```
cc_binary( ❶
    name = "hello-world", ❷
    srcs = ["hello-world.cc"], ❸
)
```

Here is the line-by-line breakdown:

❶ The build rule for generating a C++ binary file

❷ The name of the target

❸ A list of labels that represent the source code file locations on disk

We have the proper code in place, but how do we actually execute the logical steps required to compile the code? That's the purpose of a *target*. In Example 1-1, we defined a target with the name `hello-world`. Next up, we'll invoke the target from the console with the help of the Bazel runtime.

Building from the Command Line

Earlier, we ran the `help` command to verify the successful installation of Bazel. If you looked at the console output from that command, you might have noticed the `build` command. It is the primary command for executing a target. You can see the command in action:

```
$ bazelisk build //main:hello-world
```

The build command instructs Bazel to build the binary. The double slash (//) is a label that provides a reference to the root of your module. The package of the build is `main:` and `hello-world` indicates the name of your build target.

The definition of the target as part of the `bazel` command might not look as you expected. Instead of just spelling out the name, we also have to provide the path relative to the project structure. Therefore, every target belongs to exactly one package. Bazel calls the combination of package name plus target name a *label*, as shown in Figure 3.

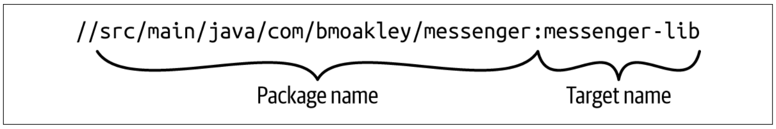


Figure 3. Composition of a label

Here is the console output that results from running that command:

```
...
INFO: Found 1 target...
Target //main:hello-world up-to-date:
  bazel-bin/main/hello-world
INFO: Elapsed time: 1.055s, Critical Path: 0.02s
INFO: 1 process: 9 action cache hit, 1 internal.
INFO: Build completed successfully, 1 total action
```

At this point, you know your build was successful and that you can immediately run `hello-world`. Any issues are printed here as well.

Exploring the Generated Artifacts

When Bazel generates your final build, it also creates a number of folders in your build directory:

bazel-bin
: A symlink to the most-written *bin* directory

bazel-<module-name>
: The working directory for all the actions that took place in your build

bazel-out
: A symlink to the output path

bazel-testlogs
: The results of all your unit tests

See the Bazel Output Directory Layout documentation for more detail.

You can view the compiled artifacts by looking in the *bazel-bin* directory:

```
$ ls bazel-bin/main
_objs
hello-world
hello-world-0.params
hello-world.cppmap
hello-world.repo_mapping
```

```
hello-world.runfiles
hello-world.runfiles_manifest
```

You can run the compiled file inside the bazel-bin subdirectory, or you can use the simple command that Bazel provides:

```
$ bazelisk run //main:hello-world
INFO: Analyzed target //main:hello-world (0 packages loaded,
0 targets configured).
INFO: Found 1 target...
Target //main:hello-world up-to-date:
  bazel-bin/main/hello-world
INFO: Elapsed time: 0.156s, Critical Path: 0.00s
INFO: 1 process: 1 internal.
INFO: Build completed successfully, 1 total action
INFO: Running command line: bazel-bin/main/hello-world
Hello world
```

Bazel also provides a shortcut command to start the build. If you are in a folder with a *BUILD* file, simply type the following to start the current build:

```
$ bazelisk run //...
```

This command runs all the targets in the *BUILD* file. You also use the same syntax to compile all the targets as well:

```
$ bazelisk build //...
```

If you need to reclaim some disk space or perform a "fresh" build, you can delete all these directories by running:

```
$ bazelisk clean
```

This resets the entire build, requiring Bazel to re-download all the various dependencies. Building after running this command may take some time.

Using bazelrc for Build Options

When calling Bazel commands, you can pass in additional build flags known as *options*. For example, to view a detailed failure log, you can pass the `--verbose_failures` option. This is useful for debugging build issues:

```
$ bazelisk build //main:hello-world --verbose_failures
```

The *.bazelrc* file is a configuration file designed to hold all your various build options. This file is just a regular text file that processes options line by line. Empty lines are ignored. If you want to add a comment, precede it with the # character.

This file can exist in many places depending on the platform. For system-wide configurations, place it in the following locations:

- macOS/Linux/Unix: */etc/bazel.bazelrc*
- Windows: *%ProgramData%\bazel.bazelrc*

For user-specific options, place the configuration file in your home directory:

- macOS/Linux/Unix: *$HOME/.bazelrc*
- Windows: *%USERPROFILE%\.bazelrc* if it exists, or else *%HOME%/.bazelrc*

You can also place a module-specific *.bazelrc* file in your *module* directory. A module-level file is an ideal way to commit your options to source control solutions.

When using multiple configuration files, Bazel will evaluate them in the following order: system, module, home, user-specific. Options are combined and in the case of a conflict, the latter option overrides the former.

Bazelisk also provides its own configuration file. Simply place a *.bazeliskrc* configuration file in the root directory of your module. This file uses specific Bazelisk options. For example, to set the Bazel version, you'd add the following:

```
USE_BAZEL_VERSION=8.2.1
```

With this file in place, Bazelisk will download the Bazel version specified. Bazelisk also provides lots of additional configuration options, which you can see in the Bazelisk official documentation.

The Lifecycle of a Bazel Build

Every Bazel build executes as part of a three-part, phased process, as illustrated in Figure 4.

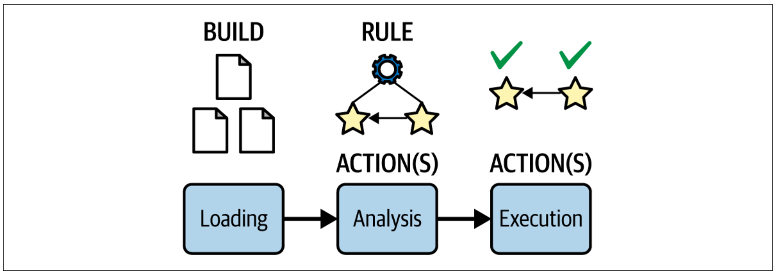


Figure 4. Phases executed for every Bazel build

The initial phase is called the *loading phase*. Its main purpose is to parse, evaluate, and cache the contents of all *BUILD* files defined for a project, including all targets and their dependencies. Any issue during this process—for example, if a *BUILD* file doesn't contain the correct notation of a rule or tries to call a function that doesn't exist—will cause the build to fail.

The next phase, the *analysis phase*, is responsible for constructing the *build execution graph*, implemented as a directed acyclic graph. The build execution graph consists of actions created by targets and formalizes their order of execution. Actions are responsible for generating the outputs we talked about before. The build can fail in this phase as well (e.g., if rule types don't match).

Lastly, the *execution phase* takes care of executing the actions. The build fails if any one of the actions cannot perform its work.

That's it from the perspective of a high-level view. There's definitely more to be said about the intricate details of each lifecycle phase; however, we won't drill any deeper here. You can read up on additional aspects in the Bazel user documentation.

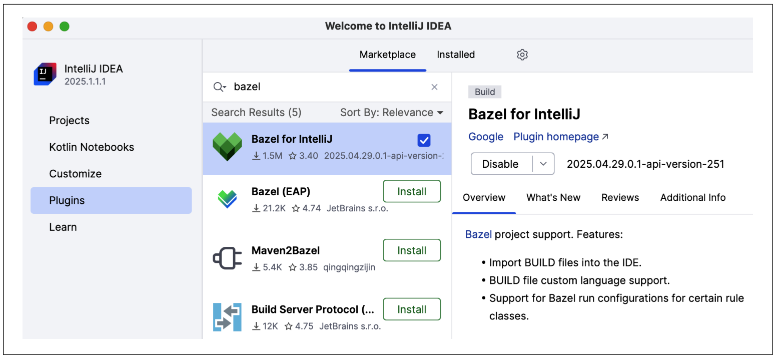
For many developers, their IDE of choice is their primary interaction with the source code and the compiler. The build usually already has all the information to perform more advanced automation processes, which can be derived from the IDE. The next section will give a short overview of Bazel's integration with popular IDEs.

Driving Bazel from the IDE

The Bazel team maintains three IDE integrations as open source projects that are used internally at Google. There is a single plug-in that supports IntelliJ, Android Studio, and CLion. There are additional integrations for Xcode, Visual Studio Code, Visual Studio, and lots of others. Bazel also provides instructions for writing your own integration.

For the purpose of demonstrating the functionality, we'll walk through opening the "Hello World" project in IntelliJ and touch on some of the features.

First things first: you'll have to install the plug-in in IntelliJ. The plug-in does not require the commercial version of IntelliJ; you can simply go with the Community Edition. Open the menu option IntelliJ IDEA > Preferences > Plugins and type in the search term "Bazel." Figure 5 shows the installed plug-in after you restart the IDE.

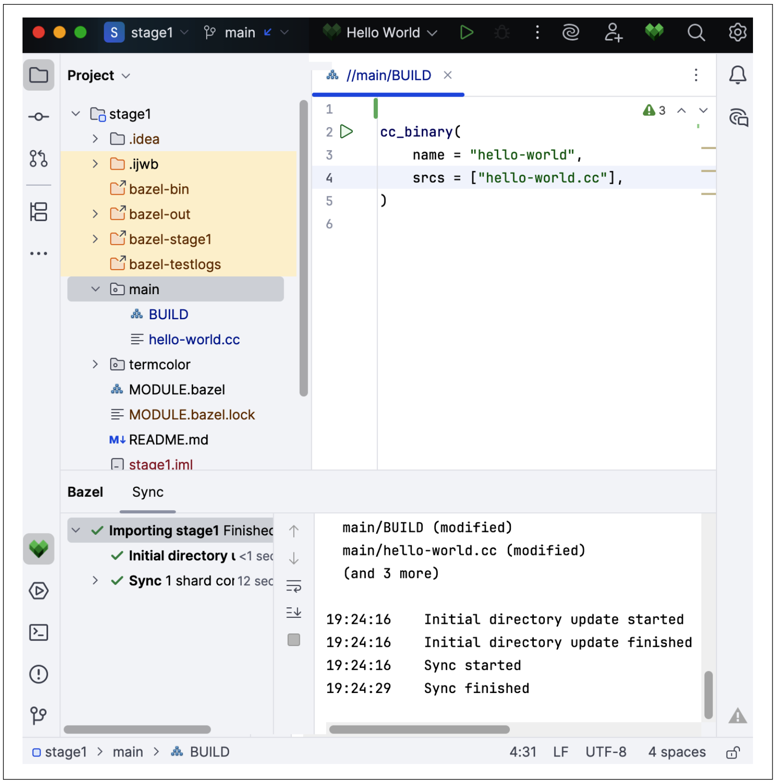


Bazel project support. Features:

- Import BUILD files into the IDE.
- BUILD file custom language support.
- Support for Bazel run configurations for certain rule classes.

Figure 5. Installation of the Bazel plug-in for IntelliJ

After installing the plug-in, you should see the option "Import Bazel project" when trying to open a new project. Select that option and point it to the root directory of your project. IntelliJ will analyze the project structure and derive all the necessary information from the build (e.g., source directories and dependencies). The imported project for our "Hello World" example is shown in Figure 6.



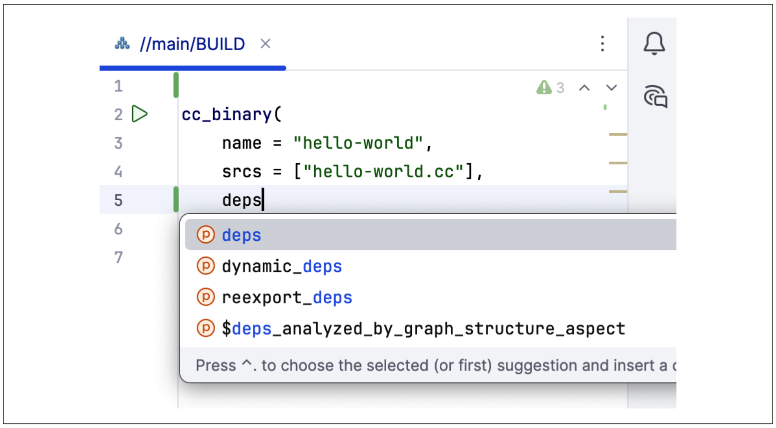
```
cc_binary(
    name = "hello-world",
    srcs = ["hello-world.cc"],
)
```

```
  main/BUILD (modified)
  main/hello-world.cc (modified)
  (and 3 more)

19:24:16    Initial directory update started
19:24:16    Initial directory update finished
19:24:16    Sync started
19:24:29    Sync finished
```

Figure 6. Imported Bazel project in IntelliJ

Figure 6 reveals the main integration points with Bazel. The Bazel Sync panel shows the current status of IntelliJ, matching the IDE project structure with the Bazel structure. The Bazel Console panel shows the output of a build ccexecution, which you can trigger by clicking the little Bazel icon in the top-right corner of the window. One of the most compelling reasons for using an IDE is the autocompletion feature. Figure 7 shows an example of the autocompletion pop up in the context of a `cc_binary` rule.



```
cc_binary(
    name = "hello-world",
    srcs = ["hello-world.cc"],
    deps
```

```
deps
dynamic_deps
reexport_deps
$deps_analyzed_by_graph_structure_aspect
Press ^. to choose the selected (or first) suggestion and insert a
```

Figure 7. Autocompletion for Bazel build functionality in IntelliJ

This concludes our condensed introduction to Bazel. You should have learned everything you need to know to get started using Bazel in your own projects. It's very possible that you are not writing C++ code—maybe you need to build Java code, Go projects, or mobile applications. The Bazel website offers a wide range of hands-on tutorials to guide you. In an upcoming section, you'll see a Java project in action.

Granted, enterprise projects have far more complex requirements. In the next couple of sections, we'll dive deeper into the Bazel toolbox.

Dependency Management

Dependency management is a crucial feature of every build tool. Without it, you wouldn't be able to define compile-time dependencies on other packages, on libraries hosted in an external repository, or on projects in a different workspace. The next sections will touch on two of those dependency types in more detail. Take a look at the Bazel user documentation for a more detailed discussion of what's possible in Bazel in the realm of dependency management.

Modeling Fine-Grained Package Granularity and Dependencies

I mentioned earlier that you can break down your project source by packages in Bazel-speak. So far we've modeled only a single package, which simply pointed to all the source code found in a specific subdirectory.

One of Bazel's benefits is that you can define packages in a very fine-grained way, even to the level of a single source file per package. If they're structured properly, you'll be able to execute many of those packages in parallel or farm out the work as part of the distributed build.

In our current example, we build a binary that can be launched from the command line. We may also want to package some of our supporting code in a library. For instance, we may want to add some console coloring to the mix.

For each of the packages, we need to add a new *BUILD* file. We already have a *BUILD* file for the `HelloWorld` package. If you are following along by making these changes on your machine, you should end up with the following project structure:

```
├── MODULE
└── HelloWorld
    └── main
        └── BUILD
        └── hello-world.cc
    └── termcolor
        └── BUILD
        └── termcolor.hpp
```

You can add the `termcolor.hpp` library from the repository for this report. Update *hello-world.cc* with the bolded code as shown in Example 1-2.

Example 1-2. Modifying the hello-world.cc script

```
#include <ctime>
#include <string>
#include <iostream>
#include "termcolor/termcolor.hpp"
...
int main(int argc, char** argv) {
  std::string who = "world";
  if (argc > 1) {
    who = argv[1];
  }
  std::cout << termcolor::red << get_greet(who) << std::endl;
  print_localtime();
  return 0;
}
```

Next up, we'll edit the *BUILD* files of the packages. Example 1-3 shows the contents of the *BUILD* file in the `termcolor` package.

Example 1-3. Adding instructions to the new BUILD file

```
cc_library( ❶
  name = "termcolor", ❷
  hdrs = ["termcolor.hpp"], ❸
  visibility = ["//visibility:public",] ❹
)
```

Here is the line-by-line breakdown:

❶ This is a new rule designed for creating C++ libraries. As stated in the rule documentation, the result is an *.so*, *.lo*, or *.a* file.

❷ This is the name of the target.

❸ This passes the headers to the library.

❹ This defines the visibility of the library. Bazel requires a build author to be explicit about the visibility of targets across multiple packages. By default, a target can only "see" other targets of the same *BUILD* file

Finally, you need to update your other *BUILD* file as shown in Example 1-4.

Example 1-4. Updating your BUILD file for the previous `hello-world` target

```
cc_binary(
    name = "hello-world",
    srcs = ["hello-world.cc"],
    deps = ["//termcolor:termcolor"] ❶
)
```

❶ The `deps` parameter receives a list of dependencies. In this case, you pass in the dependency target.

Executing the build with the same command we used before will resolve the package dependency properly, compile the code, and create an executable JAR file that runs the `hello-world` application:

```
$ bazelisk build //main:hello-world
INFO: Analyzed target //main:hello-world (1 packages loaded,
3 targets configured).
INFO: Found 1 target...
Target //main:hello-world up-to-date:
  bazel-bin/main/hello-world
INFO: Elapsed time: 1.077s, Critical Path: 0.95s
INFO: 10 processes: 8 internal, 2 darwin-sandbox.
INFO: Build completed successfully, 10 total actions
```

Modeling package dependencies may seem tedious in the beginning but will feel natural after a while. There's a comfort in knowing the exact relationship between your packages, which will ultimately lead to better structured code with high cohesion and low coupling.

You do not want to write every aspect of your application yourself. For example, it's unlikely that you'll want to write code for parsing JavaScript Object Notation (JSON) or the low-level details of HTTP communication. Oftentimes, we rely on the code other people wrote and distributed as external dependencies. External dependencies usually reside in repositories, such as a Git repository containing the source code or a binary repository hosting the artifacts produced by a build (e.g., a JAR file). In the next section, we'll talk about declaring and consuming external dependencies in your build.

Declaring and Using External Dependencies

Bazel provides external dependency support through the Bazel Central Registry (BCR). The BCR is a vendor-neutral repository for a variety of modules built by Bazel. Bazel users are encouraged to submit modules to the registry. As it grows, developers can easily incorporate a third-party dependency into their project with little work.

When using a module, Bazel will analyze the dependency graph and fetch related libraries from the registry. This includes both direct dependencies and transitive dependencies.

Bazel manages the entire dependency graph. During a build action, Bazel will order the dependencies to avoid loading order issues as well as the diamond dependency problem. If two or more separate dependencies rely on the same dependency but with different versions, Bazel will try to resolve this issue by using the latest minor version that satisfies both requirements. In the case that Bazel does not, you will need to specify dependency version to Bazel by adding an override in the *MODULE* file.

The BCR provides a nice web frontend for exploring all the various dependencies that contribute to a module. When you specify a module to a project, you also provide a version number. This version declares various dependencies that are also modules. Bazel assembles the dependency graph and downloads the necessary code from the appropriate repositories.

Here's a new simple project that incorporates an external dependency. Your directory should look like the following:

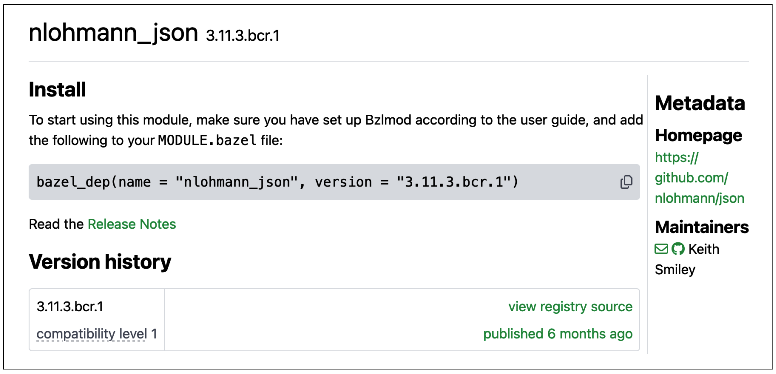
```
├── MODULE
└── JSONText
    └── src
        └── BUILD
        └── main.cpp
```

This is a simple console application that prints out JSON text to the console. Example 1-5 shows the source code for this application.

Example 1-5. A simple console application that formats JSON

```
#include <iostream>
#include <string>
#include <vector>
#include <nlohmann/json.hpp>
using json = nlohmann::json;
int main() {
  json jsonText;
  jsonText["name"] = "My Bazel App (Bzlmod)";
  jsonText["version"] = 1.0;
  jsonText["features"] = json::array({"Simple", "Uses JSON",
    "Console Output", "Bzlmod"});
  std::string pretty_json_string = jsonText.dump(4);
  std::cout << "Successfully created JSON object:" << std::endl;
  std::cout << pretty_json_string << std::endl;
  std::cout << "\nAccessing a value: App Name = " <<
    jsonText["name"].get<std::string>() << std::endl;
  return 0;
}
```

This dependency uses a compiled header for all the imported code. Now you need to add the dependency to your *MODULE* file. The project page in the BCR shows how to do this (see Figure 8).


nlohmann_json 3.11.3.bcr.1

Install

To start using this module, make sure you have set up Bzlmod according to the user guide, and add the following to your `MODULE.bazel` file:

```
bazel_dep(name = "nlohmann_json", version = "3.11.3.bcr.1")
```

Read the Release Notes

Version history

3.11.3.bcr.1	view registry source
compatibility level 1	published 6 months ago

Metadata

Homepage
https://github.com/nlohmann/json

Maintainers
Keith Smiley

Figure 8. The official BCR page for the `nlohmann_json` module

You simply add the highlighted code to your *MODULE* file. It should look as follows:

```
bazel_dep(name = "nlohmann_json", version = "3.11.3.bcr.1")
```

The `bazel_dep` method takes a name and a version number. The name is the critical piece of information because there are times you'll want to refer to this name in your build files. Bazel lets us use this name as a label: by adding an @ before the name, you can refer to this repository. This is known as an *apparent name*.

You can also specify the full name that combines the repository name, package, and target. This is known as the *canonical name*, and it starts with double @s. For example: *@@myrepo//my/app/main:app_binary*

Here is the *BUILD* file that uses this dependency:

```
cc_binary(
  name = "app",
  srcs = ["main.cpp"],
  deps = ["@nlohmann_json//:json",], ❶
)
```

❶ The `deps` parameter now refers to an external repository.

The BCR makes it easy to incorporate third-party dependencies into your project. You can also incorporate your own project in the BCR. The Bazel team provides instructions on how to add your project. There are many steps, but once your project has been approved, it can be added as a dependency for other projects.

Unfortunately, not every dependency can be found in the BCR. In these cases, you need to write your own custom extension code, as you'll see later in this report.

Incorporating Third-Party Package Managers like Maven

While it is clear that Bazel would prefer that the BCR be the primary means for dependency management, Bazel works well with other package managers. Each set of rules provides functions for using an associated package manager. For example, `rules_swift` declares functions for interacting with the Swift Package Manager.

To follow along, we've included a new Java project in the example repository, which includes two external Maven dependencies. Here is the structure of the Java example:

```
├── MODULE
└── src
    ├── main
    │   └── java
    │        └── com
    │          └── example
    │                ├──messenger
    │                │    ├── BUILD
    │                │    └── Messenger.java
    │                ├── BUILD
    │                └── HelloWorld.java
    └── test
        └── java
            └── com
                └── example
                    └── messenger
                        ├── BUILD
                        └── MessengerTest.java
```

`Messenger.java` is a Java library that prints out some simple text to the console. `HelloWorld.java` puts the library to use. Finally, `MessengerTest.java` is a simple unit test. We will handle the unit test in the following section.

First, add the following code to the *MODULE* file:

```
bazel_dep(name = "rules_jvm_external", version = "6.7")
maven = use_extension("@rules_jvm_external//:extensions.bzl",
  "maven") ❶
maven.install( ❷
  artifacts = [
    "Org.apache.commons:commons-lang3:3.9",
    "junit:junit:4.12"
  ],
  repositories = [
    "https://repo1.maven.org/maven2",
  ]
)
use_repo(maven, "maven") ❸
```

❶ The `use_extension()` function is a built-in Bazel function. In this case, it loads all the extensions provided by the JVM rules.

❷ The `maven_install()` function is used to fetch the dependencies from repositories. In this case, Maven is fetching two dependencies from the Maven central repository.

❸ The `use_repo()` function allows the repository to be used in the current module scope.

As you can see, our code requires the dependency Apache Commons Lang version 3.9. For Bazel to resolve the dependency, you need to provide its *group*, *artifact ID*, and *version* (GAV). If you are a JVM developer, you've probably used this notation before. Each portion of the dependency declaration `org.apache.commons:commons-lang3:3.9` is separated by a colon.

At runtime, Bazel reads the dependency information from the *MODULE* file, looks up artifacts in the list of declared repositories, downloads the artifacts, and then uses them in the build for specific tasks (e.g., compilation or test execution).

At this point, you can incorporate your Maven dependencies into your *BUILD* files. Here is the *BUILD* file for the `Messenger` library:

```
java_library( ❶
  name = "messenger-lib",
  srcs = ["Messenger.java"],
  visibility = [
    "//src/main/java/com/example:__pkg__",
  ],
  deps = [
    "@maven//:org_apache_commons_commons_lang3" ❷
  ],
)
```

❶ Creating a Java library is almost the same as creating a C++ library.

❷ The dependency is now referencing the Maven dependency declared in the *MODULE* file.

That's it. Now you just have to run the `build` target. Bazel will automatically resolve and download external dependencies. Of course, if the dependency has been downloaded before, it will simply be reused from the local cache.

I hope you are writing tests alongside your application code and running them to verify its correct behavior. Bazel can execute tests from the build. You'll find that we are building upon the knowledge from the previous sections to make that happen.

Executing Automated Tests

The Java rule set also includes a rule for compiling and executing test source code named `java_test`. The way you model the package is very similar to what we've done before. First, we'll create the Java test code in the source directory *src/test/java*. The test class `MessengerTest.java` uses the API of the test framework JUnit 4. Next, we'll create a BUILD file for that package. Your project structure should end up as follows:

```
├── MODULE
└── src
    ├── main
    │   └── java
    │       └── ...
    └── test
        └── java
            └── com
                └── example
                    └── messenger
                        ├── BUILD
                        └── MessengerTest.java
```

Let's also populate the contents of the new BUILD file. Example 1-6 creates the test rule with the name `messenger-test`.

Example 1-6. Defining a unit test that includes library dependency

```
java_test(
    name = "messenger-test",
    srcs = [
        "MessengerTest.java"
    ],
    test_class = "com.example.messenger.MessengerTest",
    deps = [
        "//src/main/java/com/example/messenger:messenger-lib",
    ],
)
```

The test function looks similar to existing functions. The big difference is both the name and the `test_class` parameter.

To run the test, use the test command and specify the target's name:

```
$ bazelisk test messenger-test
INFO: Analyzed target //src/test/java/com/example/messenger:
messenger-test.
INFO: Found 1 test target...
...
INFO: Build completed successfully, 8 total actions
//src/test/java/com/example/messenger:messenger-test
```

This command runs the `messenger-test` unit test. If the test fails, Bazel will provide a link to the appropriate log file.

By default, Bazel uses JUnit 4. At this point, JUnit 4 is somewhat old. Thankfully, you can use JUnit 5 instead. Example 1-7 shows the *MODULE* file updated to take advantage of JUnit 5.

Example 1-7. A MODULE file that uses JUnit 5 functionality

```
JUNIT_JUPITER_VERSION = "5.13.0-M2" ❶
JUNIT_PLATFORM_VERSION = "1.13.0-M2"

bazel_dep(name = "rules_jvm_external", version = "6.7")
bazel_dep(name = "contrib_rules_jvm", version = "0.28.0")
maven = use_extension("@rules_jvm_external//:extensions.bzl",
  "maven")
maven.install(
  artifacts = [
    "org.apache.commons:commons-lang3:3.9",
    "org.junit.platform:junit-platform-launcher:%s"
      % JUNIT_PLATFORM_VERSION, ❷
    "org.junit.platform:junit-platform-reporting:%s"
      % JUNIT_PLATFORM_VERSION,
    "org.junit.jupiter:junit-jupiter-api:%s"
```

```
      % JUNIT_JUPITER_VERSION,
    "org.junit.jupiter:junit-jupiter-engine:%s"
      % JUNIT_JUPITER_VERSION,
  ],
  repositories = [
    "https://repo1.maven.org/maven2",
  ],
)
use_repo(maven, "maven")
```

❶ Defines two constants for the JUnit 5 versions

❷ The Maven dependencies using the defined constants

Naturally, you'll need to update your *BUILD* file to incorporate JUnit 5. Example 1-8 shows the updated file using `junit_test_suite`.

Example 1-8. A modified BUILD file that incorporates JUnit 5

```
load("@rules_jvm_external//:defs.bzl", "artifact") ❶
load("@contrib_rules_jvm//java:defs.bzl", "JUNIT5_DEPS",
  "java_test_suite")
java_junit5_test( ❷
  name = "messenger-test",
  srcs = [
    "MessengerTest.java"
  ],
  test_class = "com.example.messenger.MessengerTest",
  deps = [
    "//src/main/java/com/example/messenger:messenger-lib",
    artifact("org.junit.jupiter:junit-jupiter-api"),
    artifact("org.junit.jupiter:junit-jupiter-engine"),
    artifact("org.junit.platform:junit-platform-launcher"),
    artifact("org.junit.platform:junit-platform-reporting")
  ],
)
```

❶ This incorporates a few dependencies that you'll use in your BUILD file.

❷ `java_junit5_test()` is a JUnit 5 unit test. It's meant to be a drop-in replacement for `java_test()`.

The test case is launched using the same command as the previous unit test only it will now use JUnit 5.

Extending Bazel's Capabilities

Extensibility is one of Bazel's core capabilities. In this section, I'll give you a first taste of the functionality, its possibilities, and some code examples to demonstrate the concepts in action. We'll start by talking about two concepts in theory—rules and macros.

Extension Concepts

To extend Bazel's capabilities, we start by creating a new file with the extension *.bzl*. This file can live anywhere in your project directory or can be hosted on an HTTP server for wider exposure.

A rule represents the most powerful extension point in Bazel. It has full control over Bazel's internals, can configure other rules, and introduces elaborate features that are complex in nature. You will want to write a rule for nontrivial functionality. Think of it as a plug-in for the Bazel ecosystem. As an example, the fully fledged Go language support in Bazel has been written as a rule.

The other extension option is a *macro*. A macro is a good fit for externalizing common functionality into a new, reusable function. It's a means to better organize your build or to call a rule with parameters you want to set by default. You will want to write a macro if your build logic becomes too complex to maintain or to avoid the copy-paste antipattern.

Rules are considered an advanced use case for Bazel and beyond the needs of most users. If you need custom functionality, the Bazel development team recommends that you first start with macros, which should provide all the functionality you need.

When are macros and rules evaluated and executed during the lifecycle phases of a Bazel build? Let's revisit the lifecycle phases we discussed earlier: macros are evaluated during the loading phase, and rules are executed during the analysis phase. Consequently, you cannot modify a macro once the build has left the loading phase. Figure 9 shows how both concepts fit into the lifecycle.

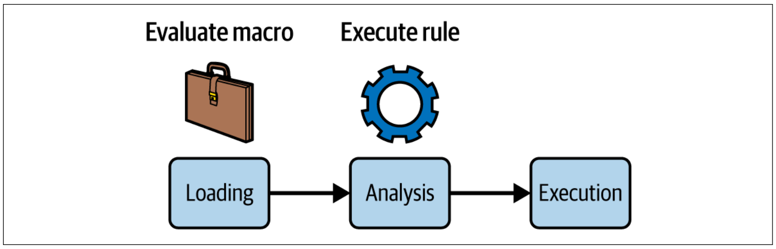


Figure 9. Evaluation of macros and rules in the lifecycle of a Bazel build

Now you have a basic understanding of the extension concepts in Bazel, but what language or syntax do you actually use to implement them? To express build script logic as well as extension implementations, Bazel uses the language Starlark. Let's take a closer look at it.

The Starlark Build Language

Technically, the Starlark language is based on Python 3. If you're familiar with the Python language, you should be able to read and write a Bazel build script and any of its extensions on a syntax level.

There are differences between Starlark and pure Python, though. Starlark is more of a custom runtime and dialect of Python because it introduces specific restrictions. For example, it cannot access the filesystem, network, or system clock. Moreover, mutability and access to the standard Python library is limited. For instance, an array is mutable in the current file's execution scope, but beyond that file, the array is immutable.

The main reason for limiting the available functionality is to achieve optimal build execution performance by supporting parallel and remote execution and to allow multithreaded processing of build logic.

In practice, you will interact with a custom API when implementing macros or rules. We'll learn how to use Starlark in the following sections by writing a custom macro and rule.

Writing a Macro by Example

In the previous section, you saw how to incorporate JUnit 5 into a Java project. This was a custom function designed to complement existing rules. The java_junit5_test was developed by the Bazel community. In this upcoming example, you'll define your own JUnit 5 unit test by way of a macro.

Macros are best organized into their own package, somewhat separate from the actual application source code. For that purpose, we'll create a new directory named *macros*. We'll place a *BUILD* file to model a package and the macro file, which we'll name *junit5.bzl*, in there. The result should look as follows:

```
├── MODULE
├── macros
│   ├── BUILD
│   └── junit5.bzl
└── src
    └── ...
```

For macros, it's not required to populate the *BUILD* file with instructions, so we'll just leave it empty. It merely acts as an indicator that we're modeling a package here. A macro is basically a function that can instantiate and configure rules, which is exactly what we are planning to do here. Example 1-9 implements such a function; it ingests a list of parameters, massages them, and then creates a `java_test` rule with the appropriate parameters. You can also see that it sets up the JUnit 5 dependencies without having to declare them repeatedly for every single test package.

Example 1-9. A macro to make it easier to include JUnit 5 unit test cases

```
def java_junit5_test(name, srcs, test_package, deps = [],
                     runtime_deps = [], **kwargs): ❶
    FILTER_KWARGS = [
        "main_class",
        "use_testrunner",
        "args",
    ]

    for arg in FILTER_KWARGS:
        if arg in kwargs.keys():
            kwargs.pop(arg)

    junit_console_args = []
    if test_package:
        junit_console_args += ["--select-package", test_package]
    else:
        fail("must specify 'test_package'")

    native.java_test( ❷
        name = name,
        srcs = srcs,
        use_testrunner = False,
        main_class = "org.junit.platform.console.ConsoleLauncher",
        args = junit_console_args,
        deps = deps + [
            "@maven//:org_junit_jupiter_junit_jupiter_api",
            "@maven//:org_junit_jupiter_junit_jupiter_engine"
        ],
        runtime_deps = runtime_deps + [
            "@maven//:org_junit_platform_junit_platform_console"
        ],
        **kwargs
    )
```

❶ Defines the macro, including the expected parameters and default values

❷ Calls the built-in, “native” rule named `java_test` and configures it

With the goal of reusability and encapsulation achieved, we can move on to loading and using the macro. The existing *BUILD* file of our test package can simply reference the macro and use it as if it were a built-in function provided by the Bazel runtime. Example 1-10 shows the revised *BUILD* file.

Example 1-10. Calling the custom macro in the BUILD file

```
load("//macros:junit5.bzl", "java_junit5_test") ❶

java_junit5_test( ❷
    name = "messenger-test",
    srcs = [
        "MessengerTest.java"
    ],
    test_package = "com.bmuschko.messenger",
    deps = [
        "//src/main/java/com/bmuschko/messenger:messenger-lib"
    ],
)
```

❶ Loads the macro with the appropriate package

❷ Calls the macro and configures it

While we didn't inspect every single implementation detail, I think it's clear that macros can help quite a bit with code maintenance.

Building Beyond Your Local Machine

As you've seen, Bazel provides a way to build software that's correct, fast, and reproducible. You truly unlock Bazel's flexibility and power once you incorporate it in a network or cloud-based environment. Using the RBE protocol built into Bazel, multiple developers can share the same build target as well as other artifacts produced by other developers. This can dramatically speed up your build times.

While Bazel provides the protocol for multiple build machines, it is up to the build or platform team to provide the backend infrastructure and configure the machines to communicate with each other. This takes time, people, expertise, and patience. Thankfully, the Bazel community has many solutions that do the hard work for you.

Introducing Nativelink

Trace Machina released its open source Nativelink platform as a way to provide remote execution, caching, and building over a scalable cloud-based service or distributed on-premises network instead of a local machine.

Nativelink gives you the power of Bazel to efficiently support large and complex builds, without the headaches of configuring and maintaining your own custom build infrastructure.

In particular, Nativelink handles the complexity of orchestrating builds and tests, scheduling the most cost-effective resources on which to run them, and updating the shared cache. Developers access the platform from their IDEs using any client-side build tool that supports the RBE protocol, including Bazel, Buck2, Pantsbuild, and Siso. This allows Nativelink to intercept tasks and route them to the optimal resources. For example, Bazel may route resources to various cores on a machine, like an air traffic controller assigning runways. Nativelink manages resources among different machines on a network or in a cloud environment. You can think of this as a regional controller directing flights between different airports based on current usage and priority.

Nativelink also reduces computing costs by automatically running compiler tasks on standard CPUs while scheduling specialized ML (machine learning) training tasks for GPUs, for example. You can even take advantage of AWS (Amazon Web Services) spot instances, which allow you to purchase unused EC2 (Elastic Cloud Compute) capacity at significantly reduced prices—up to 90% off the on-demand rate.

Nativelink is written entirely in the Rust programming language, leveraging its memory safety model and taking advantage of Rust's latest concurrency features of `async-await`. This allows simulation tests to run fast, reliably, and *natively*—hence the name. Tests can be run on a diverse range of target platforms, which may have limited resources, such as embedded and AI-enabled devices, without requiring virtualization or runtime layers that can add latency, producing inconsistent or unpredictable results.

Trace Machina also offers a cloud service called Nativelink Cloud, which provides an auto-scaling infrastructure, built-in security and observability, and an intuitive web interface to track build history, cache hits, and generated artifacts. Figure 10 is a screenshot of the Nativelink dashboard. It can also be installed on premises, and is installable with a Helm chart.

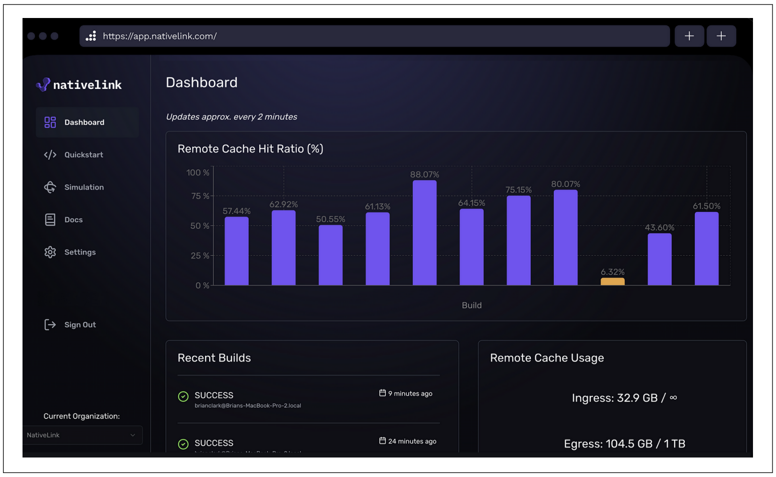


Figure 10. The Nativelink Cloud dashboard

Nativelink Cloud takes build performance to a new level. For instance, a typical build of Chromium may take one or more hours of compilation time. With Nativelink Cloud, you can expand the number of CPU resources, shrinking the entire build from hours to minutes. All this takes is some additional configuration to use Nativelink as described in the official documentation.

Nativelink Cloud is free for individuals, open source projects, and cloud production environments, with support for unlimited team members.

Executing Bazel Projects on CI/CD Platforms

In the spirit of continuous delivery, organizations need to ship software fast and frequently without sacrificing quality, safety, or security. In practice, that means building and testing the code multiple times a day, not only on developer machines but as part of a CI environment. This is especially important for safety-critical devices, such as autonomous vehicles and robots, which require multiple levels of testing on a more frequent basis, including unit tests, subsystem tests, and system-wide tests.

Slow builds, failed builds, inconsistent test results, and lack of predictability impact your team's ability to deliver the latest features or bug fixes to your customers, but they also negatively impact the developer experience.

Bazel uses a remote cache of generated artifacts and builds only changed packages, so builds take a shorter amount of time than with traditional build tools. Bazel will also cache test results. This may or may not be ideal, especially in the case of fickle results, so Bazel provides a `--cache_test_results` flag that enables you to fine tune the caching behavior. There is also a `--runs_per_test flag` that will rerun a series of tests a specific number of times: if any of those repeated tests fail, the entire test case is considered a failure.

Nativelink provides out-of-the-box integration with all popular CI/CD platforms, including GitHub Actions, and allows you to provision your own infrastructure, whether it's located on premises or in the cloud. See the documentation for instructions on how to incorporate Nativelink Cloud into your CI/CD workflow.

Remote Execution and Remote Caching

Bazel offers two solutions that can help with achieving the goals of build avoidance and build scalability, while at the same time keeping the promise of consistency and correctness (see Figure 11):

Remote execution
: Remote execution offloads build execution to high-performance computing in a datacenter or cloud provider, then uses those results on the originating build machine.

Remote caching
: Remote caching involves sharing and reusing build results across multiple, physically separate machines (e.g., developer machines and CI infrastructure).

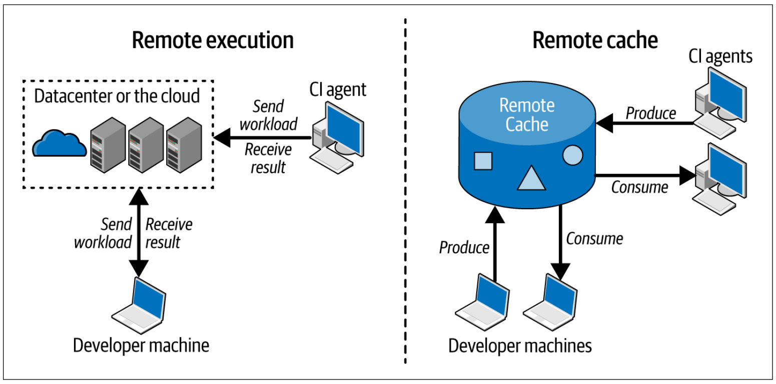


Figure 11. Moving parts of remote execution and remote caching with Bazel

With those basic definitions out of the way, let's dive into a high-level discussion of both concepts. While we cannot walk through all the intricate details, I'll provide pointers on how to incorporate remote execution and remote caching into your own Bazel projects.

Builds can be very demanding when it comes to consuming hardware resources. It's not uncommon to end up with a completely overloaded machine while compiling code or executing tests. What can we do in the meantime? Check X on our phones? Wouldn't it be great if you could instead offload the work to other machines? That's where remote build execution comes into play.

Bazel can be configured to execute build or test actions in the cloud. Depending on the available hardware resources on the receiving end, you will end up with a much faster and potentially more reliable build because you don't need to depend on the sandbox of your local machine.

Various remote execution services have evolved over time. Among them are open source, self-hosted solutions like Buildbarn, Buildfarm, BuildGrid, and Scoop, as well as Google's commercial offering, Cloud Build. This report won't go into depth on creating and using a multinode build farm to avoid going into all the details specific to a particular solution.

The idea of remote caching is to share build outputs across multiple machines that invoke a Bazel build. If a build output has already been produced for certain inputs of an action, then you can simply reuse that output without having to actually execute its actions.

Let's illustrate this functionality with the help of an example. Say you have two different teams working on the same project across geographically separate locations. One team resides in the US, the other in Asia. Considering the difference in time zones, the Asia team starts its day before the US team. In the course of the day, the Asia team executes a Bazel build and produces build outputs. Hours later, the US team comes online. As soon as it starts executing its builds, the US team will likely be able to reuse some of the existing build outputs, which leads to faster builds in most situations.

The central piece of the architecture is a server that acts as an entry point and storage facility for build outputs. Bazel offers various options, some of which are one-stop solutions:

- An nginx server that acts as a cache but requires manual configuration
- Bazel-remote, the open source remote build cache built by Google
- A fully managed build cache and remote execution system via Nativelink Cloud

It's up to your organization to pick the solution that's best suited to your needs.

Parallelism

Bazel takes advantage of large infrastructure. It is built from the ground up to not just take advantage of multi-core CPUs, but to utilize clusters of compute resources to accelerate builds. During the analysis stage of a build, Bazel will examine the dependency graph to determine what needs to be built, along with all the resources for the build. Build tasks are then distributed throughout the entire network, taking advantage of any cache.

Bazel is also ideally suited for a cloud-based infrastructure, allowing you to scale up your build to the number of machines available to you.

Containers

Bazel works hand in hand with Docker. To demonstrate the build cache functionality, we'll set up bazel-remote as a Docker container. You can retrieve the Docker image for bazel-remote from Docker Hub. Open your terminal and enter the following:

```
$ docker pull buchgr/bazel-remote-cache
```

To ensure that the container can persist the cached data, we'll mount a volume represented by a path on our local disk. For now, the volume mount path is */Users/YOUR_USER_NAME/example/dev/bazel-cache*. Make sure to create the directory before starting the container. The following command starts the build cache server in a container, maps port 8080 to 9090, and mounts the volume:

```
$ docker run -v /Users/bmoakley/example/dev/bazel-cache:/data
-p 9090:8080 buchgr/bazel-remote-cache –max_size 109
2025/04/24 17:48:16 Loaded 0 existing disk cache items.
```

Initially, the cache directory will be empty. You'll have to use the command-line option `--remote-cache` to tell Bazel about the existence of the remote cache. The following Bazel invocation executes the tests of our example project and populates the cache with build outputs:

```
$ bazel test --remote_cache=http://localhost:9090
//src/test/java/com/example/messenger:messenger-test
...
//src/test/java/com/example/messenger:messenger-test
                        PASSED in 0.4s
Executed 1 out of 1 test: 1 test passes.
```

Earlier, we started the build cache container in foreground mode, which is the default. This mode is convenient for inspecting log messages; however, if you plan to set up the build cache in a production environment, you should run in *detached mode* (with the `--detach` or `-d` flag) instead:

```
$ docker run -d -v /Users/bmoakley/example/dev/bazel-cache:
/data -p 9090:8080 buchgr/bazel-remote-cache -max_size 109
```

The log messages now indicate that Bazel is looking for existing build outputs but can't find them (indicated by the HTTP GET 404 status code). As a result, Bazel populates the cache with the result we just produced (indicated by the PUT 200 status code):

```
2025/04/24 17:48:55  GET 404      172.17.0.1 /ac/06f08c1047e4
a6c5ae5202724b7abe8cd3a633463de28d66544aaa77292cda70
2025/04/24 17:48:55  GET 404      172.17.0.1 /ac/0a317cb909b6
273627720a96fd954080ddfe6e0052ce525551e60fa4bd37b5a0
2025/04/24 17:48:55  GET 404      172.17.0.1 /ac/46af73a1541c
d9322acd2f8b6fd3a76634c0e67d448636a7540810542bfde1c4
2025/04/24 17:48:55  GET 404      172.17.0.1 /ac/efccd560dcca
7935b5f9e15aab4dabb536653324a741dee2a7f5364a942bdff6
2025/04/24 17:48:55  GET 404      172.17.0.1 /ac/66005e46cb37
ac52b6bac4f8654b4ef7b3c9aab38391ab9db9f96969df8d5d90
2025/04/24 17:48:55  PUT 200      172.17.0.1 /cas/e74e18475fa
864bf5aa5b60512c86bbb08619cf3ae8443394105c012ad523fd2
2025/04/24 17:48:55  PUT 200      172.17.0.1 /cas/e3b0c44298f
c1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
2025/04/24 17:48:55  PUT 200      172.17.0.1 /cas/bd19f26dc7b
acb77e659b82e4582097d15fb27f20fbd0545fe11396490728c7a
2025/04/24 17:48:55  PUT 200      172.17.0.1 /cas/0a317cb909b
6273627720a96fd954080ddfe6e0052ce525551e60fa4bd37b5a0
2025/04/24 17:48:55  PUT 200      172.17.0.1 /cas/00afd170b3e
c8a4408292d894af328d212705a0180575a3fb61172091137d816
...
```

Now, let's run the build with the same command again and observe the difference. In the output, you can see that the build outputs for the tests were reused from the cache. Bazel flags the target with the `cached` marker:

```
$ bazel test --remote_cache=http://localhost:9090
//src/test/java/com/example/messenger:messenger-test
...
//src/test/java/com/example/messenger:messenger-test
           (cached) PASSED in 0.4s
Executed 0 out of 1 test: 1 test passes.
```

To cross-check the expected behavior, you can also inspect the new log messages on the build cache container. The GET 200 status code indicates that existing build outputs could be reused:

```
2025/04/24 17:52:24  GET 200      172.17.0.1 /ac/06f08c1047e4
a6c5ae5202724b7abe8cd3a633463de28d66544aaa77292cda70
2025/04/24 17:52:24  GET 200      172.17.0.1 /ac/66005e46cb37
ac52b6bac4f8654b4ef7b3c9aab38391ab9db9f96969df8d5d90
2025/04/24 17:52:24  GET 200      172.17.0.1 /ac/0a317cb909b6
273627720a96fd954080ddfe6e0052ce525551e60fa4bd37b5a0
2025/04/24 17:52:24  GET 200      172.17.0.1 /ac/46af73a1541c
d9322acd2f8b6fd3a76634c0e67d448636a7540810542bfde1c4
2025/04/24 17:52:24  GET 200      172.17.0.1 /ac/efccd560dcca
7935b5f9e15aab4dabb536653324a741dee2a7f5364a942bdff6
2025/04/24 17:52:24  GET 200      172.17.0.1 /cas/e74e18475fa
864bf5aa5b60512c86bbb08619cf3ae8443394105c012ad523fd2
2025/04/24 17:52:24  GET 200      172.17.0.1 /cas/9d20c5302dd
7b4a2166180fbda5e5656fb49ec7bc9d3da61759db0336c2c9feb
```

The build cache is tremendously helpful for implementing build avoidance. It offers a number of operational modes and command-line options to control the runtime behavior. At the time of writing, the build cache does not offer a user interface or visualization option to monitor performance trends over time. To use the build cache for your own project, see the user documentation for more details.

Executing Simulations for AI-Powered Chips and Devices

Google found other uses for Bazel's sandbox execution environment. Beyond compiling code, Bazel plays a foundational role in building the complex hardware and software stacks required for modern AI chips and embedded systems. In Google's internal semiconductor workflows—especially for projects like Tensor Processing Units (TPUs) and edge AI chips—Bazel serves as the client interface for managing source code, modeling dependencies, and driving builds across a large and heterogeneous infrastructure.

Unlike traditional software builds, chip development requires orchestrating simulation flows, hardware description language (HDL) compilation, firmware generation, and cross-compilation across different architectures. To meet these needs, Google extended Bazel's capabilities through a powerful remote execution backend called Forge, a high-performance, largely C++-based server built internally to execute Bazel actions in a secure, scalable, and distributed environment.

In this model:

- Bazel operates purely as a deterministic client—managing local build metadata and orchestrating build actions.
- Forge performs those actions remotely, including invoking vendor-specific toolchains (e.g., Synopsys, Cadence, Ansys, or custom GCC [GNU Compiler Collection] cross-compilers), simulating hardware modules, and running verification tests across large compute farms.
- These actions are sandboxed, fully reproducible, and performance-optimized to scale with chip complexity.

For developers outside Google, Nativelink's Rust implementation is the closest open source equivalent to Forge. Nativelink provides remote execution and caching features compatible with Bazel's Remote Execution API, allowing chip developers to offload builds and simulations across data center infrastructure or cloud environments. When paired with a custom toolchain (e.g., a Nordic SDK or ARM cross-compiler), this setup enables the same kind of structured and scalable development used internally at Google.

Advanced features in Nativelink, such as building for simulation tools or building for exotic hardware, use Bazel as a frontend. The *BUILD* file in Example 1-11 shows a rule targeting embedded firmware.

Example 1-11. A BUILD rule for an ARM Cortex-M chip with vendor-specific simulation tools

```
cc_binary(
    name = "firmware",
    srcs = ["main.c", "peripherals.c"],
    copts = ["-mcpu=cortex-m4", "-mthumb"],
    linkopts = ["-Tlinker_script.ld"],
    deps = ["@nordic_sdk//drivers:gpio"]
)
```

With remote execution enabled, Bazel delegates this build to the execution service (e.g., Forge or Nativelink), which invokes the correct cross-compiler, verifies the sandbox environment, and returns the binary artifact. This approach makes it easy to test different builds, maintain reproducibility, and scale across development teams—all essential for fast-paced chip design cycles.

Bazel's platform abstraction and support for cross-compilation also help developers switch between hardware targets without rewriting build logic. The *BUILD* file in Example 1-12 shows rules for switching between microcontroller units (MCUs) or silicon revisions. The platform rule defines a new platform, accepting constraints such as a CPU or OS for a specific platform. The toolchain rule defines the constraints for a given toolset.

Example 1-12. BUILD rules for switching between microcontroller units (MCUs) or silicon revisions

```
platform(
    name = "arm_cortex_m4",
    constraint_values = [
            "@platforms//cpu:arm",
            "@platforms//os:none"
    ],
)
toolchain(
    name = "arm_gcc_toolchain",
    toolchain_type = "@bazel_tools//tools/cpp:toolchain_type",
    toolchain = ":arm_gcc_impl",
    target_compatible_with = [":arm_cortex_m4"]
)
```

By externalizing the complexity into these abstractions, Bazel enables clean separation of build logic from environment-specific configurations—a core requirement in hardware-software codesign workflows.

In summary, Bazel has become a key enabler for chip development at scale—not by being a full toolchain itself, but by offering a declarative, cache-friendly client that integrates with powerful remote execution systems like Forge. As the open source community builds equivalents like Nativelink, more hardware teams can benefit from the same modularity, scalability, and reproducibility that Bazel brought to software engineering.

Conclusion

Bazel is a feature-rich and versatile polyglot build tool that we've explored with the help of typical use cases applicable to Java-based projects. We identified the building blocks in Bazel responsible for compiling, testing, and packaging code. You learned that Bazel takes a strong, opinionated view on how to model a project as a direct result of years of internal use at Google. While Bazel requires an explicit definition of fine-grained modules and dependencies, it rewards the user with fast, incremental, and parallel build execution. Bazel's remote caching and execution capabilities form the foundation for scalable, performant enterprise projects, especially those that reside in a monorepo. If needed, Bazel can be extended with the help of macros and rules. These extension features enable Bazel to easily adapt to the ever-changing landscape of new languages, frameworks, and platforms.

Bazel's feature set is quite impressive. It covers most aspects required for building modern polyglot enterprise applications, small and large. A first look at Bazel reveals that it can handle typical requirements with ease. Additionally, new open source platforms like Nativelink extend Bazel with a scalable backend infrastructure optimized for distributed teams, large and complex code bases, and diverse runtime environments.

Upon closer inspection, you'll find that the Bazel ecosystem still has to catch up with some features that end users have come to love from other prominent build tools and now expect. Over the course of this report, we've touched on some of those aspects—for example, refined and powerful dependency management capabilities, and support for a standard way to publish process artifacts to binary repositories. While all of these capabilities can be implemented as a rule, the effort would be quite significant for a team trying to switch to Bazel, ultimately leading to a less refined and polished experience. As Bazel gains popularity, I have no doubt that the ecosystem will catch up with other build tools. In the user documentation you can find a list of available rules, some of which have been contributed by the community.

It's hard to give a generalized personal recommendation on a build tool. It always depends on the needs of the organization, team, or project. For some teams, flexibility and build language syntax is important; for others, build execution performance and scalability is paramount. If you are evaluating Bazel to see whether it's a good fit for your project, I'd recommend implementing a prototype that can live alongside your current automation logic. Measure and compare aspects that are important to you and then make your decision. Very soon, you'll determine whether Bazel can adapt to your needs.

About the Authors

Brian Moakley is the author of the Building with Bazel course, *Unity Games by Tutorials*, and *Amazon Sumerian by Tutorials*. He has also produced many video courses, articles, and screencasts on all aspects of mobile development. You learn more about his work at *www.jezner.com*.

Marcus Eagan is the CEO and founder of Trace Machina, creators of Nativelink.