CodeQL code scanning for compiled languages

In this article

Understand the autobuild method CodeQL analysis uses to build code for compiled languages and learn how you can customize the build command if you need to.

> **Who can use this feature**
> If you have write permissions to a repository, you can configure code scanning for that repository.

Code scanning is available for all public repositories on GitHub.com. Code scanning is also available for private repositories owned by organizations that use GitHub Enterprise Cloud and have a license for GitHub Advanced Security. For more information, see "About GitHub Advanced Security."

About the CodeQL analysis workflow and compiled languages

Code scanning works by running queries against one or more databases. Each database contains a representation of all of the code in a single language in your repository. For the compiled languages C/C++, C#, Go, Java, and Swift, the process of populating this database involves building the code and extracting data.

CodeQL analyzes the C/C++, C#, Go, Java, and Swift source files in your repository that are built.

If you enable default setup, the `autobuild` action will be used to build your code, as part of your automatically configured CodeQL analysis workflow. If you enable advanced setup, the basic CodeQL analysis workflow uses `autobuild`. Alternatively, you can disable `autobuild` and instead specify explicit build commands to analyze only the files that are built by these custom commands.

For CodeQL code scanning, you can use default setup, which analyzes your code and automatically configures your code scanning, or advanced setup, which generates a workflow file you can edit. For more information about advanced setup, see "Configuring advanced setup for code scanning."

For information about the languages, libraries, and frameworks that are supported in the latest version of CodeQL, see "Supported languages and frameworks" in the CodeQL documentation. For information about the system requirements for running the latest version of CodeQL, see "System requirements" in the CodeQL documentation.

If your workflow uses a `language` matrix, `autobuild` attempts to build each of the compiled languages listed in the matrix. Without a matrix `autobuild` attempts to build the supported compiled language that has the most source files in the repository. With the exception of Go, analysis of other compiled languages in your repository will fail unless you supply explicit build commands.

About `autobuild` for CodeQL

CodeQL analyzes the C/C++, C#, Go, Java, and Swift source files in your repository that are built.

If you enable default setup, the `autobuild` action will be used to build your code, as part of your automatically configured CodeQL analysis workflow. If you enable advanced setup, the basic CodeQL analysis workflow uses `autobuild`. Alternatively, you can disable `autobuild` and instead specify explicit build commands to analyze only the files that are built by these custom commands.

- `autobuild` for C/C++
- `autobuild` for C#
- `autobuild` for Go
- `autobuild` for Java and Kotlin
- `autobuild` for Swift

Note: If you use self-hosted runners for GitHub Actions, you may need to install additional software to use the `autobuild` process. Additionally, if your repository requires a specific version of a build tool, you may need to install it manually. GitHub-hosted runners are always run with the software required by `autobuild`.

`autobuild` for C/C++

Supported system type	System name
Operating system	Windows, macOS, and Linux
Build system	Windows: MSbuild and build scripts Linux and macOS: Autoconf, Make, CMake, qmake, Meson, Waf, SCons, Linux Kbuild, and build scripts

The behavior of the `autobuild` step varies according to the operating system that the extraction runs on. On Windows, the `autobuild` step attempts to autodetect a suitable build method for C/C++ using the following approach:

1. Invoke `MSBuild.exe` on the solution (`.sln`) or project (`.vcxproj`) file closest to the root. If `autobuild` detects multiple solution or project files at the same (shortest) depth from the top level directory, it will attempt to build all of them.
2. Invoke a script that looks like a build script—*build.bat*, *build.cmd*, *and build.exe* (in that order).

On Linux and macOS, the `autobuild` step reviews the files present in the repository to determine the build system used:

1. Look for a build system in the root directory.
2. If none are found, search subdirectories for a unique directory with a build system for C/C++.

3. Run an appropriate command to configure the system.

For self-hosted runners, you will likely need to install the `gcc` compiler, and specific projects may also require access to `clang` or `mscv` executables. You will also need to install the build system (for example `msbuild`, `make`, `cmake`, `bazel`) and utilities (such as `python`, `perl`, `lex`, and `yacc`) that your projects depend on.

`autobuild` for C#

Supported system type	System name
Operating system	Windows, macOS, and Linux
Build system	.NET and MSbuild, as well as build scripts

The `autobuild` process attempts to autodetect a suitable build method for C# using the following approach:

1. Invoke `dotnet build` on the solution (`.sln`) or project (`.csproj`) file closest to the root.
2. Invoke `MSbuild` (Linux) or `MSBuild.exe` (Windows) on the solution or project file closest to the root. If `autobuild` detects multiple solution or project files at the same (shortest) depth from the top level directory, it will attempt to build all of them.
3. Invoke a script that looks like a build script—*build* and *build.sh* (in that order, for Linux) or *build.bat*, *build.cmd, and build.exe* (in that order, for Windows).

For .NET Core application development on self-hosted runners, the .NET SDK is required (for `dotnet`).

For .NET Framework application development, on Windows, you will need Microsoft Build Tools (for `msbuild`) and Nuget CLI (for `nuget`). On Linux and macOS, you will require Mono Runtime (to run `mono`, `msbuild`, or `nuget`).

`autobuild` for Go

Supported system type	System name
Operating system	Windows, macOS, and Linux
Build system	Go modules, `dep` and Glide, as well as build scripts including Makefiles and Ninja scripts

The `autobuild` process attempts to autodetect a suitable way to install the dependencies needed by a Go repository before extracting all `.go` files:

1. Invoke `make`, `ninja`, `./build` or `./build.sh` (in that order) until one of these commands succeeds and a subsequent `go list ./...` also succeeds, indicating that the needed dependencies have been installed.
2. If none of those commands succeeded, look for `go.mod`, `Gopkg.toml` or `glide.yaml`, and run `go get` (unless vendoring is in use), `dep ensure -v` or `glide install` respectively to try to install dependencies.
3. Finally, if configurations files for these dependency managers are not found,

rearrange the repository directory structure suitable for addition to `GOPATH`, and use `go get` to install dependencies. The directory structure reverts to normal after extraction completes.

4. Extract all Go code in the repository, similar to running `go build ./...`.

> **Note:** If you use default setup, it will look for a `go.mod` file to automatically install a compatible version of the Go language.

`autobuild` for Java and Kotlin

Supported system type	System name
Operating system	Windows, macOS, and Linux (no restriction)
Build system	Gradle, Maven and Ant

The `autobuild` process tries to determine the build system for Java codebases by applying this strategy:

1. Search for a build file in the root directory. Check for Gradle then Maven then Ant build files.

2. Run the first build file found. If both Gradle and Maven files are present, the Gradle file is used.

3. Otherwise, search for build files in direct subdirectories of the root directory. If only one subdirectory contains build files, run the first file identified in that subdirectory (using the same preference as for 1). If more than one subdirectory contains build files, report an error.

If you're using self-hosted runners, the required version(s) of Java should be present:

- If the runner will be used for analyzing repositories that need a single version of Java, then the appropriate JDK version needs to be installed, and needs to be present in the PATH variable (so that `java` and `javac` can be found).

- If the runner will be used for analyzing repositories that need multiple versions of Java, then the appropriate JDK versions need to be installed, and can be specified via the `toolchains.xml` file. This is a configuration file, typically used by Apache Maven, that allows you to specify the location of the tools, the version of the tools, and any additional configuration that is required to use the tools. For more information, see "Guide to Using Toolchains" in the Apache Maven documentation.

The following executables will likely be required for a range of Java projects, and should be present in the PATH variable, but they will not be essential in all cases:

- `mvn` (Apache Maven)
- `gradle` (Gradle)
- `ant` (Apache Ant)

You will also need to install the build system (for example `make`, `cmake`, `bazel`) and utilities (such as `python`, `perl`, `lex`, and `yacc`) that your projects depend on.

`autobuild` for Swift

Supported system type	System name

Operating system	macOS
Build system	Xcode

The `autobuild` process tries to build the biggest target from an Xcode project or workspace.

Notes:

- CodeQL analysis for Swift is currently in beta. During the beta, analysis of Swift code, and the accompanying documentation, will not be as comprehensive as for other languages. Swift 5.8 is not yet supported.
- Analysis may occasionally freeze, causing jobs to time out. To limit the number of Actions minutes used by jobs that are stuck or timing out, we recommend setting a timeout of four times your normal build time.

Code scanning of Swift code uses macOS runners by default. Since GitHub-hosted macOS runners are more expensive than Linux and Windows runners, we recommend that you build only the code that you want to analyze. For more information about pricing for GitHub-hosted runners, see "About billing for GitHub Actions."

Code scanning of Swift code is not supported for runners that are part of an Actions Runner Controller (ARC), because ARC runners only use Linux and Swift requires macOS runners. However, you can have a mixture of both ARC runners and self-hosted macOS runners. For more information, see "About Actions Runner Controller."

Customizing Swift compilation in a CodeQL analysis workflow

`xcodebuild` and `swift build` are both supported for Swift builds. We recommend only targeting one architecture during the build. For example, `ARCH=arm64` for `xcodebuild`, or `--arch arm64` for `swift build`.

You can pass the `archive` and `test` options to `xcodebuild`. However, the standard `xcodebuild` command is recommended as it should be the fastest, and should be all that CodeQL requires for a successful scan.

For Swift analysis, you must always explicitly install dependencies managed via CocoaPods or Carthage before generating the CodeQL database.

Adding build steps for a compiled language

If `autobuild` fails, or you want to analyze a different set of source files from those built by the `autobuild` process, you'll need to remove the `autobuild` step from the workflow, and manually add build steps. For C/C++, C#, Go, Kotlin, Java, and Swift projects, CodeQL will analyze whatever source code is built by your specified build steps. For information on how to edit the workflow file, see "Customizing your advanced setup for code scanning."

After removing the `autobuild` step, uncomment the `run` step and add build commands that are suitable for your repository. The workflow `run` step runs command-line programs using the operating system's shell. You can modify these commands and add more commands to customize the build process.

```
- run: |
    make bootstrap
    make release
```

For more information about the `run` keyword, see "Workflow syntax for GitHub Actions."

If your repository contains multiple compiled languages, you can specify language-specific build commands. For example, if your repository contains C/C++, C# and Java, and `autobuild` correctly builds C/C++ and C# but fails to build Java, you could use the following configuration in your workflow, after the `init` step. This specifies build steps for Java while still using `autobuild` for C/C++ and C#:

```
- if: matrix.language == 'c-cpp' || matrix.language == 'csharp'
  name: Autobuild
  uses: github/codeql-action/autobuild@v2
- if: matrix.language == 'java-kotlin'
  name: Build Java
  run: |
    make bootstrap
    make release
```

For more information about the `if` conditional, see "Workflow syntax for GitHub Actions."

For more tips and tricks about why `autobuild` won't build your code, see "Automatic build failed for a compiled language."

If you added manual build steps for compiled languages and code scanning is still not working on your repository, contact us through the GitHub Support portal.