Gradle Release Notes

The Gradle team is excited to announce a new major version of Gradle, 7.0.

This release enables file system watching by default to make your incremental builds faster, expands support for building projects with Java 16, and adds support for building on Macs using Apple Silicon processors (such as M1).

This release also introduces a feature preview for centralized dependency versions, enables build validation errors to make your builds more reliable, and makes it easier to create convention plugins for settings files. Many incubating features have been promoted to stable.

These release notes only list what's new since Gradle 6.8. You can also see the highlights of all the changes between Gradle 6.0 to 7.0.

We would like to thank the following community members for their contributions to this release of Gradle:

Matthew Haughton, Leon Linhart, Sebastian Schuberth, Aidar Nugmanoff, Martin d'Anjou, Till Krullmann, Andreas Axelsson, Pedro Tôrres, Stefan Oehme, Jeff, Rene Groeschke, Niels Doucet, Tobias Hermann, Rishaba-Jain, Jerome Dochez, Vitaly Polonetsky, Naoki Ando, Ståle Undheim.

Table Of Contents

❯ Upgrade instructions
❯ Performance improvements

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Reject Accept

❯ Known issues
❯ External contributions
❯ Reporting Problems

Upgrade instructions

Switch your build to use Gradle 7.0 by updating your wrapper:

```
./gradlew wrapper --gradle-version=7.0
```

See the Gradle upgrade guide to learn about deprecations, breaking changes and other considerations when upgrading to Gradle 7.0.

For Java, Groovy, Kotlin and Android compatibility, see the full compatibility notes.

Performance improvements

This release contains further improvements for incremental development — the part of the software development process where you make frequent small changes.

Faster incremental builds

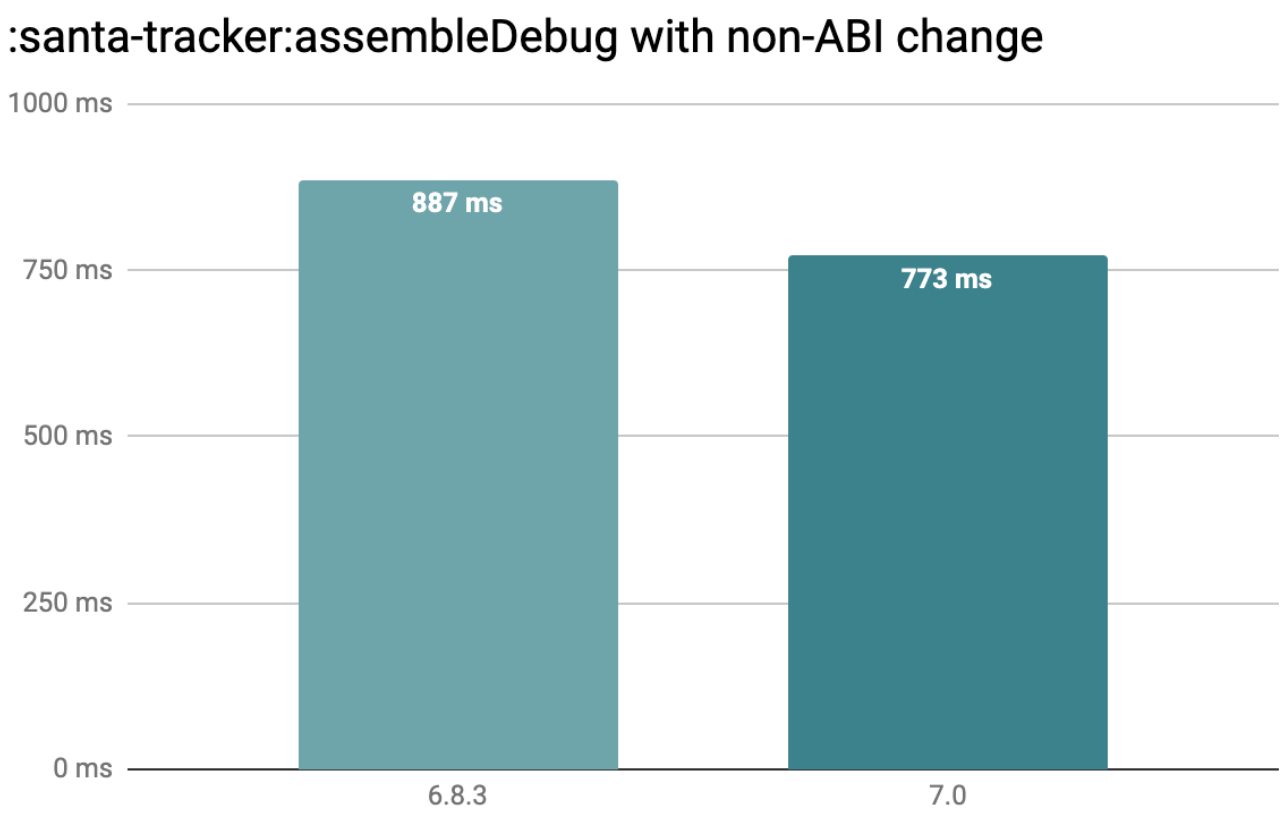
In an incremental build, input and output files are checked to determine what needs to be rebuilt. This feature typically saves a lot of time; however, it adds some I/O overhead, which can be noticeable in large projects when not much has changed since the previous build.

File system watching was introduced as an opt-in feature in Gradle 6.5 and marked as production-ready in Gradle 6.7. When enabled, Gradle keeps what it has learned about the file system in memory between builds and skips reading from the file system on each build. This significantly reduces the amount of disk I/O needed to determine what has changed since the previous build.

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

This release contains performance improvements for incremental changes in Android projects, especially those that use Jettifier.

For example, `assembleDebug` for a non-ABI change on the Santa Tracker Android project improved by 12% compared to Gradle 6.8:



NOTE

File system watching and configuration caching is enabled for the comparison.

Ignore empty `buildSrc` project

In earlier Gradle versions, the mere presence of a `buildSrc` directory was enough to trigger

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

- `:buildSrc:*` tasks will not be needlessly executed.
- The empty `buildSrc.jar` will not be added to the buildscript class path, avoiding cache misses that this can cause.

New features and usability improvements

Native support for Apple Silicon

Previous Gradle versions were able to run on new Macs with Apple Silicon processors with some disadvantages:

- With a native ARM JDK, Gradle features like the rich console and file system watching would be disabled.
- With an Intel JDK, Gradle would run at about half speed through the Rosetta2 compatibility layer.

With this release, every feature is now supported using a native ARM JDK. If you're using a new Mac with Apple Silicon, you should use Gradle with a native ARM JDK for optimal performance.

Support for Java 16

Gradle now supports running on and building with Java 16.

In previous Gradle versions, running Gradle itself on Java 16 resulted in an error. JVM projects could have been built with Java 16 using toolchains but only with incremental compilation disabled.

As of Gradle 7.0, both running Gradle itself and building JVM projects with Java 16 is fully supported.

Centralized dependency versions

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

With this release, Gradle introduces *version catalogs* as an experimental feature. A version catalog enables build authors to centralize the dependency coordinates (group, artifact, version) of their third party dependencies in a conventional configuration file and declare the actual dependencies in a type-safe way.

In order to enable this experimental feature, add the following line in your `settings.gradle(.kts)` file:

```
enableFeaturePreview("VERSION_CATALOGS")
```

Then, you can use the `gradle/libs.versions.toml` file from the root of your project to declare the catalog contents:

```
[versions]
groovy = "3.0.5"

[libraries]
groovy-core = { module = "org.codehaus.groovy:groovy",
                version.ref = "groovy" }
groovy-json = { module = "org.codehaus.groovy:groovy-json",
                version.ref = "groovy" }
groovy-nio = { module = "org.codehaus.groovy:groovy-nio",
               version.ref = "groovy" }
commons-lang3 = { group = "org.apache.commons",
                  name = "commons-lang3",
                  version = { strictly = "[3.8,4.0[", prefer="3.9" } }

[bundles]
groovy = ["groovy-core", "groovy-json", "groovy-nio"]
```

After that, in any build script of any subproject, you can then use type-safe accessors to declare

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Declaring dependencies this way allows easy reuse of dependency coordinates in all modules, provides content assist in the IDEs and reduces risk of typos. More importantly, it also provides a single location to change versions when upgrading a library.

In addition, dependencies that are frequently used together can be organized in bundles and declared in relevant projects with a single line of code. Catalogs can also be shared between different builds.

For plugin authors, or more advanced use cases, an API is available on `Settings` to declare version catalogs.

NOTE

This feature is aimed at centralizing declarations and has no impact on the dependency resolution process. This means that the resolved version can still be different from the ones declared in the version catalogs. For example, the resolved version of a dependency can be higher than declared in the catalog if a higher version is required by a transitive dependency.

Refer to the documentation for more details.

Type-safe project accessors

Before Gradle 7.0, the only way to declare dependencies between projects in a multi-project build was by using string notation, for example `project(":some:path")`.

This release of Gradle adds an experimental feature for project accessors which provides type safety and enables code completion in IDEs.

In order to enable this experimental feature, add this line to your `settings.gradle(.kts)` file:

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

```
  implementation projects.client
}
```

A project with more levels of nesting like `:commons:utils:numbers` will be accessible using the following notation:

```
dependencies {
  // type-safe alternative to project(":commons:utils:numbers")
  implementation projects.commons.utils.numbers
}
```

Please refer to the documentation for more details.

Groovy 3 upgrade

In order to support JDK 16 and keep up to date with the latest Groovy release, Gradle has been upgraded to use Groovy 3 in Groovy DSL build scripts. Groovy 3 comes with a new parser and host of other new features and capabilities that make interoperability with new Java features easier.

There are some incompatibilities between Groovy 2 and 3, that may cause issues when upgrading to Gradle 7.0.

You may be affected by the Groovy upgrade if:

- You are using the Groovy DSL for your build scripts (`.gradle` files)
- You are using a Gradle plugin implemented in Groovy
- You are building a Gradle plugin implemented in Groovy

You should not be impacted by the upgrade if:

- You are only using the Kotlin DSL (`.gradle.kts` files)

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

In order to learn more about the improvements and new features in Groovy 3.0, refer to the Groovy project's release notes.

Dependency locking improvements

Dependency locking is a mechanism for ensuring reproducible builds also when using dynamic dependency versions.

This release defaults to the improved dependency locking file format that results in fewer lock files in most projects that use this feature. Specifically, it results in one file per project instead of one file per configuration per project.

Gradle will automatically clean up previous lock files when migrating them over to the new file format.

In addition, when using the new format, the lock file name and location can be configured.

Using dynamic versions in the plugins block

Until now, the `plugins { }` block only supported fixed versions for community plugins. All version string notations Gradle supports are now accepted, including `+` or `latest.release`.

We recommend using the `plugins {}` block for applying plugins using Gradle 7. The old `apply plugin:` mechanism will be deprecated in the future.

Note that dynamic versions will introduce non-deterministic behavior to your build process and should be used judiciously. You can use dependency locking to save the set of dependencies resolved when using dynamic versions.

Build reliability improvements

Gradle employs a number of optimizations to ensure that builds are executed as fast as possible. These optimizations rely on the inputs and outputs of tasks to be well-defined. Gradle already

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy. By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Validate missing dependencies between tasks

One of the potential problems now flagged is a task that consumes the output produced by another without declaring an explicit or inferred task dependency. Gradle now detects the missing dependency between the consumer and the producer and emits a warning in that case. For more information see the user manual on input and output validation.

Dependency management improvements

Correct handling of libraries with standard JVM and Android variants

The Java plugins now recognizes the `org.gradle.jvm.environment` attribute during dependency resolution.

This allows libraries, like Guava, to clearly distinguish variants optimized for `standard-jvm` and `android`. Gradle then automatically chooses the best variant based on the current project type (Java or Android),

Plugin development improvements

Support for plugins with multiple variants

Gradle 7 looks for the new org.gradle.plugin.api-version attribute during plugin resolution. This allows plugin authors to publish different variants of their plugins for different Gradle versions. This user manual section describes how the new attribute can be used with feature variants to add additional variants to a plugin.

By default, plugins do not publish metadata with this attribute yet.

Using included builds for local plugins

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

```
pluginManagement {
    includeBuild("../my-settings-plugin")
}
plugins {
    id("my.settings-plugin")
}
```

The above example assumes that the included build defines a settings plugin with the id `my.settings-plugin`.

Library components produced by builds included though the `pluginManagement` block are not automatically visible to the including build. However, the same build can be included as plugin build and normal library build:

```
pluginManagement {
    // contributes plugins
    includeBuild("../project-with-plugin-and-library")
}
// contributes libraries
includeBuild("../project-with-plugin-and-library")
```

This distinction reflects what Gradle offers for repository declarations - repositories are specified separately for plugin dependencies and for production dependencies.

Security advisories

This release of Gradle contains fixes for the following security advisories:

- Local privilege escalation through system temporary directory
- Repository content filters do not work in Settings pluginManagement

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy. By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Lifecycle" for more information.

Java Module Support

Compiling, testing and executing Java modules is now a stable feature.

It is no longer required to activate the functionality using `java.modularity.inferModulePath.set(true)`.

Dependency Verification

Dependency verification is promoted to a stable feature.

Java Toolchain

Java Toolchains is promoted to a stable feature.

Changing the priority of the daemon process

Changing the priority of the daemon process with `--priority` is now a stable feature.

Promoted APIs

In Gradle 7.0 we moved the following classes or methods out of the incubation phase.

- Injectable services
- Configurable properties/Provider API
- Worker API
- Composite builds
- Build Caching
- Reporting
- Kotlin DSL
- Dependency notations
- Capabilities resolution

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy. By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

- Publishing
- IDE
- Tooling API
- Java Ecosystem
- Other APIs

Fixed issues

Known issues

Known issues are problems that were discovered post release that are directly related to changes made in this release.

External contributions

We love getting contributions from the Gradle community. For information on contributing, please see gradle.org/contribute.

Reporting Problems

If you find a problem with this release, please file a bug on GitHub Issues adhering to our issue guidelines. If you're not sure you're encountering a bug, please use the forum.

We hope you will build happiness with Gradle, and we look forward to your feedback via Twitter or on GitHub.

© 2026 Gradle, Inc. Gradle®, Develocity®,

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.