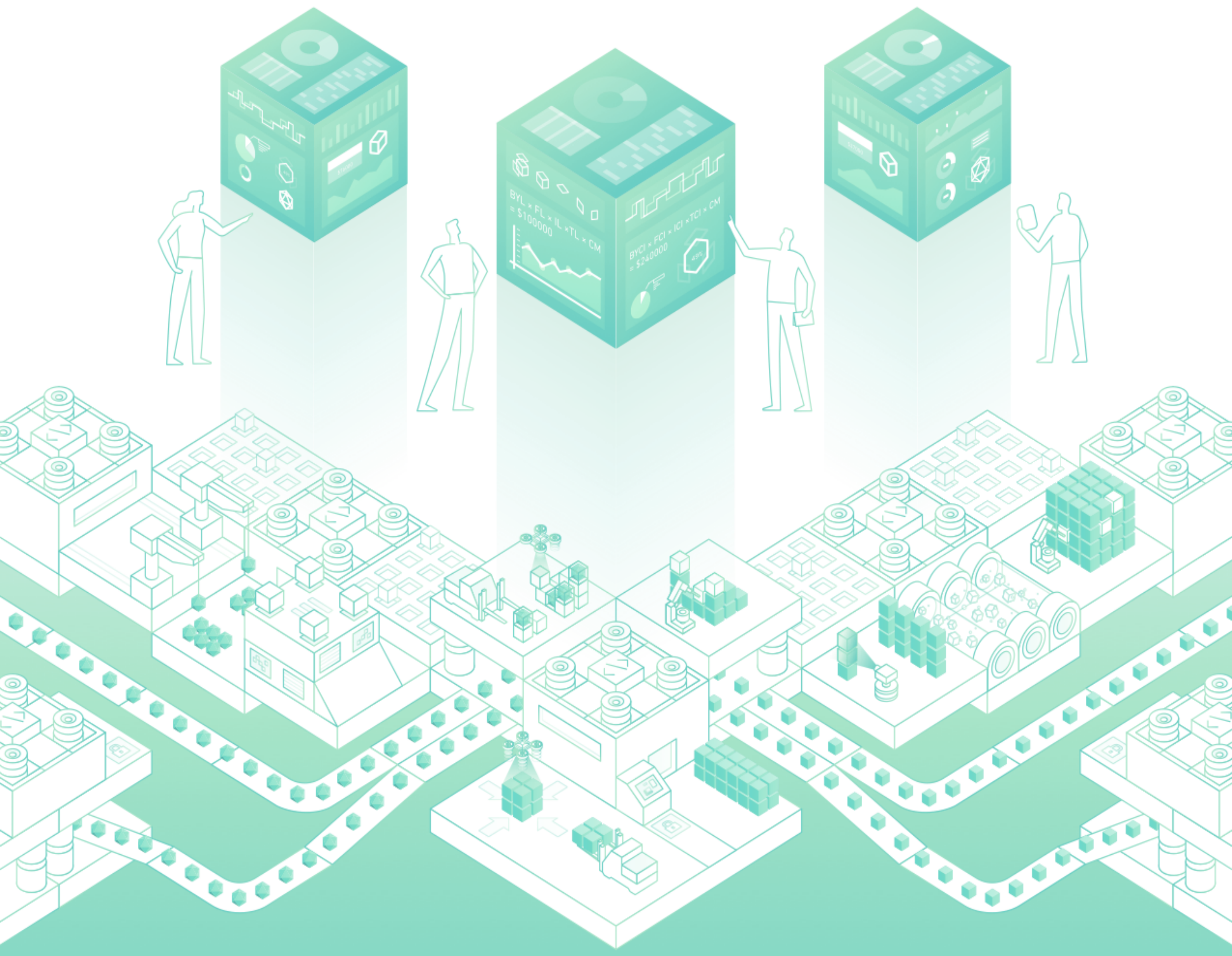
Developer Productivity Engineering Workbook


BYL × FL × IL × TL × CM = $100000
BYCI × FCI × ICI × TCI × CM = $240000

Sam Snyder

Gradle

Developer Productivity Engineering Workbook

Sam Snyder

Version 1.1, 2020-04-08

Table of Contents

Published by Gradle Inc.

Editors: Hans Dockter, Jennifer Strater
Cover and Interior Design: Charles Aweida

Preface

This is a companion to the Developer Productivity Engineering book: https://gradle.com/developer-productivity-engineering/

Where that book discusses common problems throughout the software industry at a high level, this book exists to demonstrate practical solutions to those problems. As we explore these issues, we'll be making use of Gradle Enterprise as a data source. All of the problems are general enough that with a similar data source it is possible to achieve similar results. Wherever possible we have included data from both the Gradle and Maven build tools, so that you'll be able to use whichever you prefer to complete these exercises.

It is my hope, and the hope of everyone at Gradle, that through engaging with these exercises you will come away with actionable ideas with which to improve the developer experience at your organization.

Early Access

This book is provided to you as an early access edition. You can read this book while it's being written and get the final eBook as soon as it's finished. We'll let you know when updates are available if you have registered for this book.

We hope to continue to add to this body of experience and thought leadership by inviting you to contribute to the book. You can reach out to us at devprod-eng-book@gradle.com

Chapter 1. Troubleshooting with Build Scans

In the daily routine of the developer, every code change needs to, at some point, be built to make sure the changes can be merged. Inefficient troubleshooting of failed local or CI builds is a major source of frustration and productivity loss for developers. In this section, we'll explore using build scans to efficiently troubleshoot some common build issues.

Troubleshooting Memory and Dependency Problems

1. **Finding time wasters:** Your colleagues have been complaining that the build is slow and you've taken initiative to get to the bottom of it. Figure out the source of the wasted build time in these build scans:

   - Gradle build scan: https://go.gradle.com/wb-ch1-g1
   - Maven build scan: https://go.gradle.com/wb-ch1-m1

   Hints

   Answering these questions will lead you closer to the root cause:

   - How much time is being spent on garbage collection?
   - What is the peak memory usage?

     Both of these questions can be answered on the Performance > Build section of the build scan.

   - What is the maximum JVM memory heap size allocated to the JVM running the build?

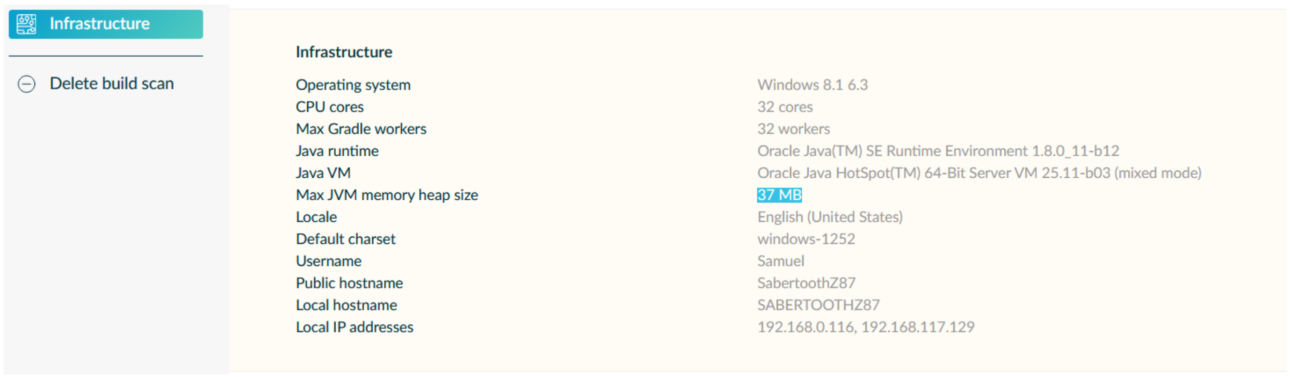
     This can be answered from the Infrastructure page

   Optional: Reproduce these build scans yourself from the repo at https://github.com/gradle/troubleshooting-with-scans. Use git (https://git-scm.com/) or download the repository as a zip file https://github.com/gradle/troubleshooting-with-scans/archive/master.zip

   Run `./gradlew build` to reproduce the Gradle build, or `./mvnw clean test` to reproduce the Maven build.

Solution - Finding time wasters: The Infrastructure page shows that the maximum heap size is quite low. The Performance page shows a significant amount of time being spent on garbage collection. The time spent on garbage collection and the high peak memory usage is compelling evidence of the garbage collector thrashing. Thrashing happens in any memory managed application when the constant need to free up memory leads to frequent pauses for garbage collection.

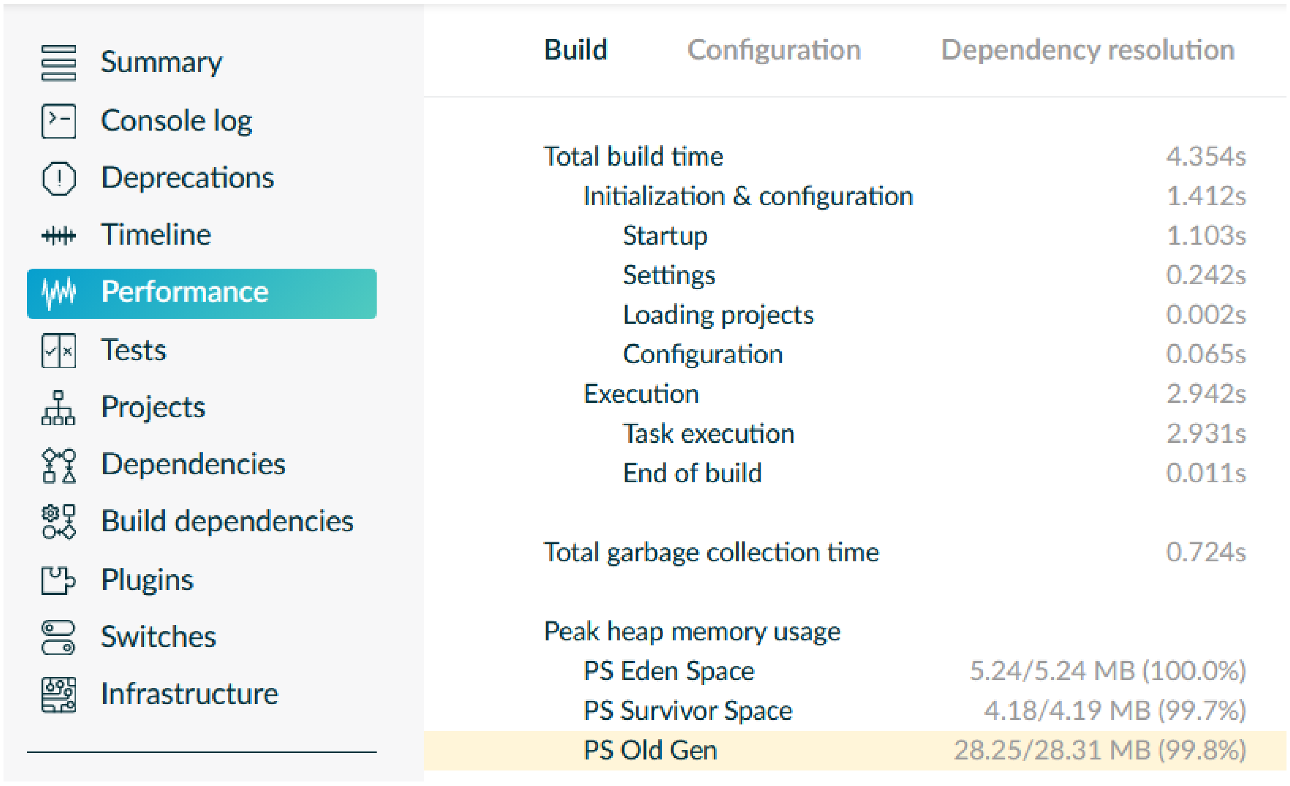
Generally, there are two ways to resolve GC thrashing: Make the program work with less memory or allocate more memory. In this case the maximum heap size has been set much lower than most configurations so raising it will immediately improve performance. Sometimes a low limit is set deliberately, particularly for CI agents that are expected to run many smaller builds simultaneously.



Infrastructure

Operating system	Windows 8.1 6.3
CPU cores	32 cores
Max Gradle workers	32 workers
Java runtime	Oracle Java(TM) SE Runtime Environment 1.8.0_11-b12
Java VM	Oracle Java HotSpot(TM) 64-Bit Server VM 25.11-b03 (mixed mode)
Max JVM memory heap size	37 MB
Locale	English (United States)
Default charset	windows-1252
Username	Samuel
Public hostname	SabertoothZ87
Local hostname	SABERTOOTHZ87
Local IP addresses	192.168.0.116, 192.168.117.129

Figure 1. Infrastructure page showing max heap size



Total build time	4.354s
Initialization & configuration	1.412s
Startup	1.103s
Settings	0.242s
Loading projects	0.002s
Configuration	0.065s
Execution	2.942s
Task execution	2.931s
End of build	0.011s
Total garbage collection time	0.724s
Peak heap memory usage	
PS Eden Space	5.24/5.24 MB (100.0%)
PS Survivor Space	4.18/4.19 MB (99.7%)
PS Old Gen	28.25/28.31 MB (99.8%)

Figure 2. Performance page showing peak memory usage and garbage collection time

2. **Verify the improvement:** Having increased the memory allocation at your recommendation, your colleagues get back to you with new build scans. Verify that these new builds have, in fact, increased the maximum heap size and that garbage collection time is down:

    - Gradle build scan: https://go.gradle.com/wb-ch1-g2
    - Maven build scan: https://go.gradle.com/wb-ch1-m2

   Optional: Change the memory settings yourself in the troubleshooting-with-scans project, publish your own build scan, and answer these same questions.

3. **Investigate classpath confusion:** You've verified that the build performance issues have been fixed, but there is an issue with this project's dependencies. Check the console log and dependencies sections of the original build scans and see if you can identify the problem.

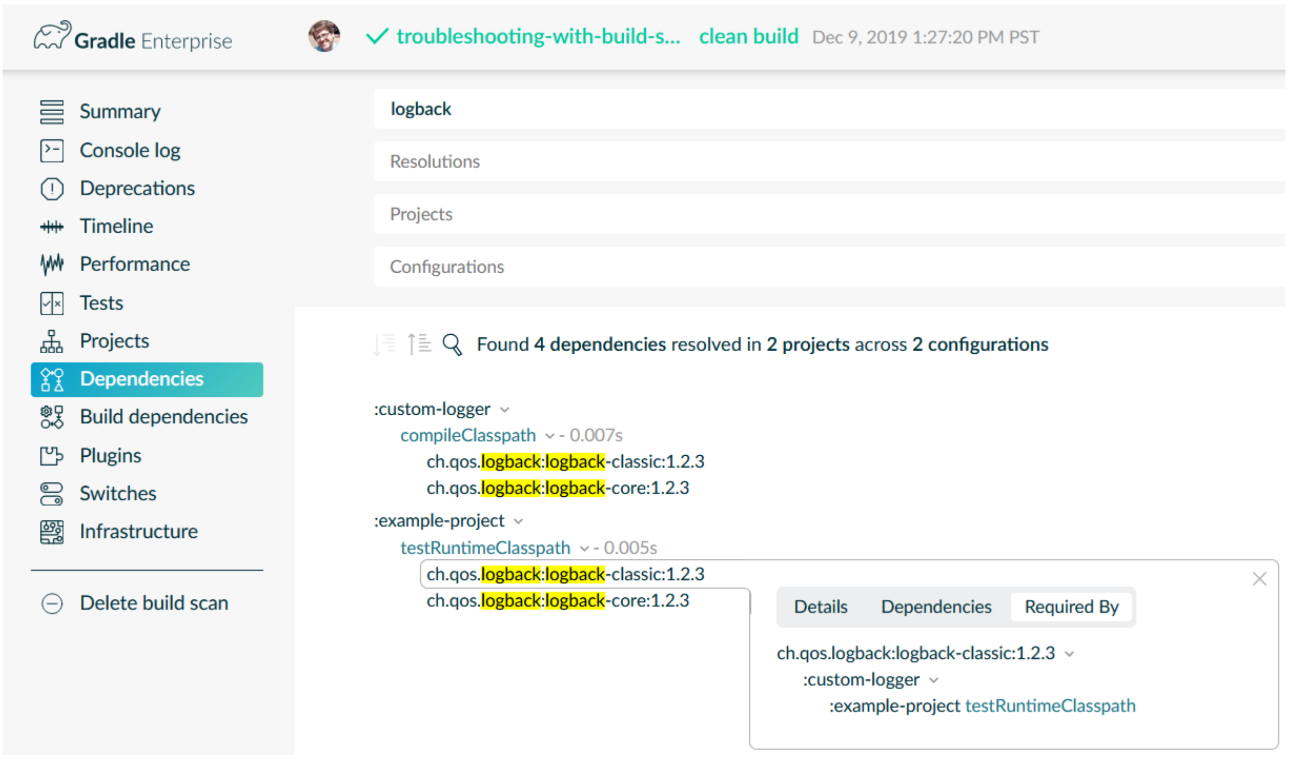
Solution - Investigate classpath confusion: This is the relevant excerpt from the console logs:

```
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in
[jar:file:/C:/Users/Samuel/.m2/repository/ch/qos/logback/logback-
classic/1.2.3/logback-classic-1.2.3.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/C:/Users/Samuel/.m2/repository/org/slf4j/slf4j-
simple/1.7.25/slf4j-simple-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type
[ch.qos.logback.classic.util.ContextSelectorStaticBinder]
```

This indicates that multiple jars implementing slf4j's logging interface are present on the classpath during test execution. We can use the dependencies view to investigate where these conflicting dependencies are coming from. Search the Dependencies section of the build scan for "logback" to identify which sub-project within this build is including this unwanted dependency:

- Gradle build scan: https://go.gradle.com/wb-ch1-g3
- Maven build scan: https://go.gradle.com/wb-ch1-m3

This build scan shows that logback-classic is ending up on example-project's runtime classpath via the custom-logger project:



Found 4 dependencies resolved in 2 projects across 2 configurations

```
:custom-logger
    compileClasspath - 0.007s
        ch.qos.logback:logback-classic:1.2.3
        ch.qos.logback:logback-core:1.2.3
:example-project
    testRuntimeClasspath - 0.005s
        ch.qos.logback:logback-classic:1.2.3
        ch.qos.logback:logback-core:1.2.3
```

```
ch.qos.logback:logback-classic:1.2.3
    :custom-logger
        :example-project testRuntimeClasspath
```

Figure 3. Searching for "logback" in the dependencies view

4. **Classpath cleaning:** Having identified multiple slf4j bindings as the problem, we've edited the build.gradle and pom.xml to use only slfj-simple and gotten new build scans. Use the new build scan to validate that slfj-simple is used and logback-classic is not:

    - Gradle build scan: https://go.gradle.com/wb-ch1-g4
    - Maven build scan: https://go.gradle.com/wb-ch1-m4

  For information on more powerful dependency management techniques, see: https://docs.gradle.org/current/dsl/org.gradle.api.artifacts.DependencySubstitutions.html

  Optional: Edit the troubleshooting-with-scans project to use only a single slf4j implementation and publish your own build scans.

  For an example solution, see the `solution` branch: https://github.com/gradle/troubleshooting-with-scans/compare/solution

Using Build Scans On Your Project

Are all of your builds using optimal JVM memory settings and are using the correct dependencies? Try publishing a build scan to scans.gradle.com to see for sure.

For quick-start instructions for Gradle and Maven, see: https://scans.gradle.com/

For full details on how to configure & customize build scans in your own build see:

- Gradle: https://docs.gradle.com/enterprise/gradle-plugin/
- Maven: https://docs.gradle.com/enterprise/maven-extension/

To learn more about the benefits of monitoring & optimizing your development toolchain, see this chapter from our book Developer Productivity Engineering: https://go.gradle.com/dpe-toolchain-monitoring

Chapter 2. Answering Cross-Domain Questions

Once you have a data source for your builds you'll want to ask questions of that data. What's fast? What's slow? What's passing? What's failing? These are great questions (and ones that the dashboards built into Gradle Enterprise can help you answer!), but they lead to more complex questions. Questions such as "What's the duration between a new developer cloning the repo and getting their first change committed, and how many failed builds did they have along the way?". Or "How reliable are local developer builds compared to CI builds?".

More complex questions span domains, not just the build alone but source control, CI, IDEs, and all of the other tools we use to deliver software. In order to answer these questions we need the equivalent of a database's foreign keys to tie data from multiple systems together. Gradle's answer to establishing these relationships is attaching custom values, tags, and links to build scans.

Adding custom metadata to builds, such as the source control state on a developer's local machine, can be particularly useful for debugging.

Helping Jenn Fix Her Build

Jenn is a Software Engineer who is trying to add new functionality to a task called "ciDiagnostics". She has been assured that the task is passing in its current, unmodified form on CI. To make sure that it also works on her machine, Jenn is attempting to run it locally before applying her changes. Frustratingly, she hasn't been able to run it successfully and has asked for your help. Her build fails with the error message `Task 'ciDiagnostics' not found in root project 'gradle'.`

Jenn has asked for your help.

Follow this link https://go.gradle.com/wb-ch2-g1 to access the build scans list, pre-filtered to show the relevant time range. Use these build scans to identify why the ciDiagnostics task isn't working for Jenn.

NOTE All of the build scans in this exercise are from Gradle builds. Maven build scans for this exercise are coming in a subsequent edition of this book.

Hints

- Jenn has been working on the "sam/jenn-scenario" git branch. The build scans list can be filtered to show only builds from this branch by filling in the custom values filter with:

  `git branch name=sam/jenn-scenario`

- Jenn's build has been instrumented with extra custom values about the state of her workspace

Solution

Here's the custom values section of one of Jenn's failed builds: https://go.gradle.com/wb-ch2-g2. The git diff custom value includes this snippet:

```
// Within build.gradle.kts
// This an example of a task that is only created conditionally
// Conditional creation of tasks is sometimes used to avoid expensive configuration
slowing down builds that don't need it
// This technique has been supplanted by the TaskContainer.register() API which
accomplishes the time savings without the pitfalls
// See this guide on task configuration avoidance for more information:
//      https://docs.gradle.org/current/userguide/task_configuration_avoidance.html
if(System.getenv("CI") != null) {
    tasks.register("ciDiagnostics") {
        // This is admittedly contrived, so let's make-believe this actually does
something useful
        doLast {
            println("Doing some expensive calculations...")
            Thread.sleep(1000L)
            println("Everything is A-Okay")
        }
    }
}
```

In the code snippet above, can you see why the task executes on CI but cannot be found when running locally?

If you figured it out or want to skip to the answer: This code registers the `ciDiagnostics` tasks conditionally based on an environment variable being present. The CI system used at Jenn's company, like most CI systems, sets an environment variable on its runners to identify them. So this task isn't even added to Gradle's task graph when the environment variable isn't set. To successfully execute this task, Jenn should add that environment variable to her local environment, as the original developer of this task presumably did.

Adding Custom Metadata to Your Build Scans

The git diff included in Jenn's build incidentally includes the code used to record the diff itself:

```
// Within settings.gradle.kts
fun execute(vararg command: String): String {
    val baos = ByteArrayOutputStream()
    exec {
        commandLine(*command)
        standardOutput = baos
        workingDir = rootDir
    }
    return baos.toString()
}
gradleEnterprise {
    buildScan {
        publishAlways()
        server = "https://enterprise-samples.gradle.com"
        // Run expensive operations on a background thread to avoid slowing down
configuration time unnecessarily
        background {
            val currentBranch = execute("git", "rev-parse", "--abbrev-ref",
"HEAD").trim()
            val diffWithMastter = execute("git", "diff", "master...$currentBranch")
            value("Git Diff with 'master'", diffWithMastter)
        }
    }
}
```

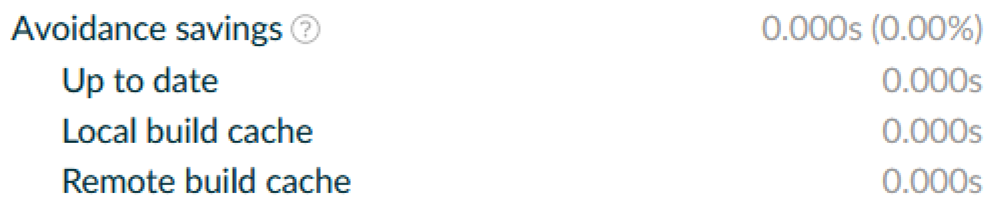
For more details on adding custom metadata to Gradle build scans, see: https://docs.gradle.com/enterprise/gradle-plugin/#extending_build_scans

For more details on adding custom metadata to Maven build scans, see: https://docs.gradle.com/enterprise/maven-extension/#extending_build_scans

Chapter 3. Accelerating Builds with a Local Cache

Caches accelerate builds by reusing the results of previous builds. To provide performance benefits, a cache must first be populated with these previous results. To optimize the benefits and avoid losing them to regressions, we have to be able to measure them. Helpfully, build scans track how builds interact with the cache in great detail.

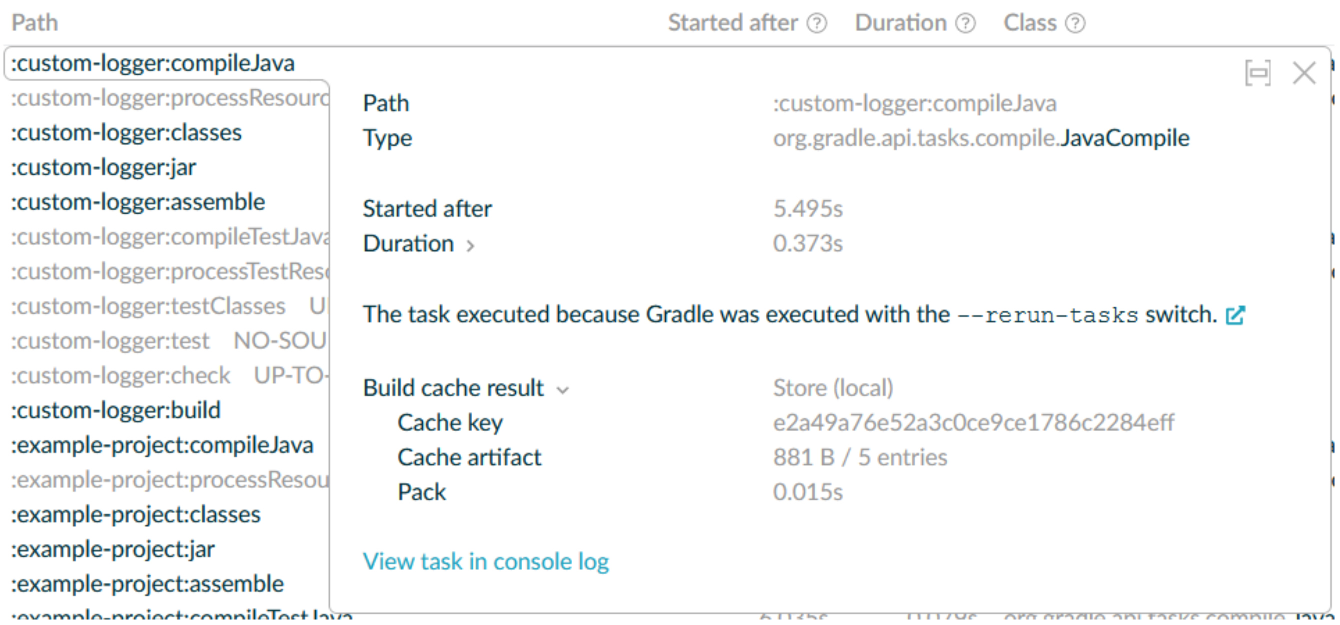
Scans include the "Avoidance savings" section which displays the estimated time savings. **For Gradle builds**, the "Avoidance savings" figures can be found under `Performance > Task execution`. **For Maven builds**, these figures can be found under `Performance > Goal execution`.



Avoidance savings	0.000s (0.00%)
Up to date	0.000s
Local build cache	0.000s
Remote build cache	0.000s

Figure 4. Example Avoidance savings block from a Gradle build

To identify exactly how a particular task interacted with the cache, mouse over its entry in the Timeline view and click the button. It will expand into a window like this one, providing details about how the task did (or did not) interact with the cache.



Path | Started after | Duration | Class

:custom-logger:compileJava
:custom-logger:processResourc
:custom-logger:classes
:custom-logger:jar
:custom-logger:assemble
:custom-logger:compileTestJava
:custom-logger:processTestRes
:custom-logger:testClasses U
:custom-logger:test NO-SOU
:custom-logger:check UP-TO-
:custom-logger:build
:example-project:compileJava
:example-project:processResou
:example-project:classes
:example-project:jar
:example-project:assemble

Path	:custom-logger:compileJava
Type	org.gradle.api.tasks.compile.JavaCompile
Started after	5.495s
Duration ›	0.373s

The task executed because Gradle was executed with the `--rerun-tasks` switch.

Build cache result ⌄	Store (local)
Cache key	e2a49a76e52a3c0ce9ce1786c2284eff
Cache artifact	881 B / 5 entries
Pack	0.015s

View task in console log

Figure 5. Example expanded task information window

Seeing the Local Cache in Action

In this exercise we'll accelerate the troubleshooting-with-scans repository with the local build cache: https://github.com/gradle/troubleshooting-with-scans

The `local-cache` branch has been pre-configured to have caching enabled for Gradle and Maven builds.

1. **Building with an empty cache:** Using these build scans taken from building the `local-cache` branch with an empty cache:

    - Gradle build scan: https://go.gradle.com/wb-ch3-g1
    - Maven build scan: https://go.gradle.com/wb-ch3-m1

    Answer these questions:

    - Were the results of any cacheable tasks stored within the cache?
    - Were there any timesavings due to the cache in these builds?

    Optional: Publish your own build scans from the `local-cache` and use those build scans to answer the questions. Gradle build scans can be reproduced by running `./gradlew clean build`. The local cache for Maven is a paid feature of Gradle Enterprise. Reproducing the Maven build scans in this chapter with the free scans.gradle.com service will not result in any cache hits.

2. **Building with a populated cache:** Now that the local cache has been populated with entries, use these build scans from subsequent builds:

    - Gradle build scan: https://go.gradle.com/wb-ch3-g2
    - Maven build scan: https://go.gradle.com/wb-ch3-m2

    To answer the questions:

    - Which tasks were resolved from the cache?
    - Were there any cacheable tasks that were not resolved from the cache?
    - How much faster is the fully cached build than the uncached build?

Hints

To search for tasks resolved from the cache, click the magnifying glass

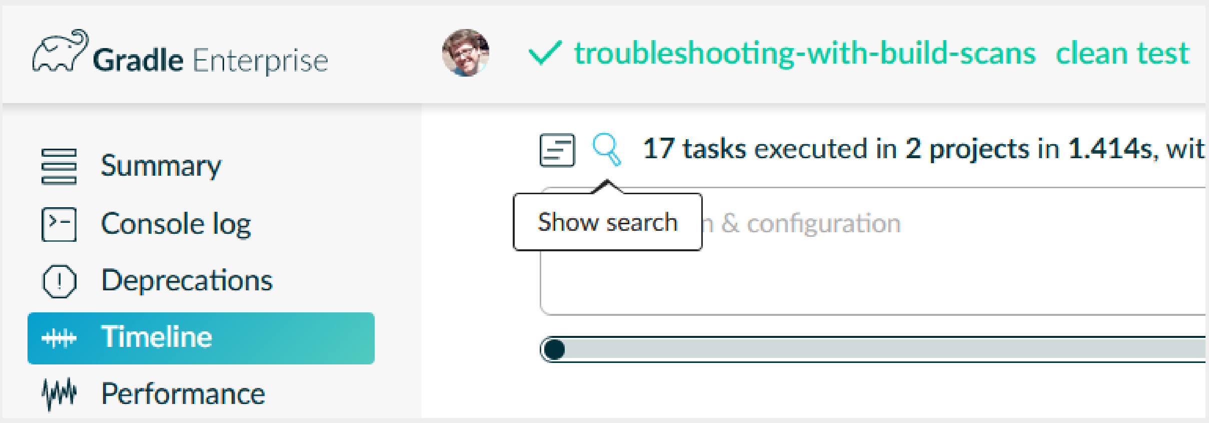


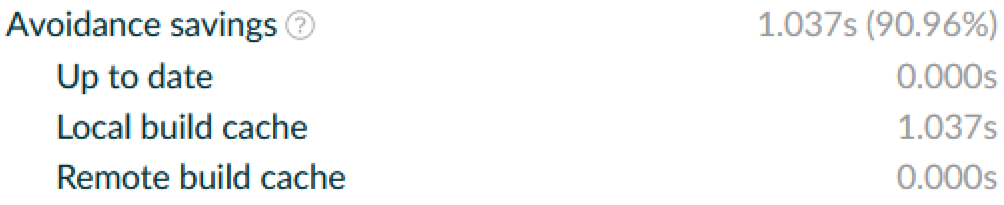
Figure 6. Search button on the Timeline portion of a build scan

Then set "Task output cacheability" to "Cacheable".

Solution - Building with a populated cache: Filtered by cacheability the timeline shows that all of the cacheable tasks did, in fact, come from the cache. Compare to the previous build scan you published to see that cache hits on these tasks were faster than executing the tasks was the first time.

View the "Avoidance savings" section of the Performance tab to observe in this, admittedly ideal case, using this cache has led to quite substantial time savings:

Gradle build avoidance savings due to the local cache



Avoidance savings	1.037s (90.96%)
Up to date	0.000s
Local build cache	1.037s
Remote build cache	0.000s

NOTE Maven users should note that the cache is only enabled on `clean` builds. So if you run just `./mvnw test` the task will be executed even if the cache is populated. This behavior differs from the Gradle build tool which uses the cache on both clean and incremental builds.

Enabling the Local Cache on your build

For Gradle, enable the local cache by adding the following line to the gradle.properties file:

```
org.gradle.caching=true
```

That’s all there is to turning it on as a Gradle user! If you don’t already have this enabled on your Gradle build, what are you waiting for?

For Maven, the local cache is a commercial feature that comes with Gradle Enterprise. See gradle.com for more information on starting a trial. Assuming you have a Gradle Enterprise server, begin enabling the cache by applying the gradle-enterprise-maven-extension in the .mvn/extensions.xml file in your project:

```
<?xml version="1.0" encoding="UTF-8"?>
<extensions>
  <extension>
    <groupId>com.gradle</groupId>
    <artifactId>gradle-enterprise-maven-extension</artifactId>
    <version>see https://scans.gradle.com for latest version</version>
  </extension>
</extensions>
```

Then enable the local cache in .mvn/gradle-enterprise.xml, noting that you have to supply the URL of a Gradle Enterprise sever run within your organization:

```
<gradleEnterprise
  xmlns="https://www.gradle.com/gradle-enterprise-maven"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://www.gradle.com/gradle-enterprise-maven/
https://www.gradle.com/schema/gradle-enterprise-maven.xsd">
  <server>
    <url>https://your-gradle-enterprise-instance/</url>
  </server>
  <buildScan>
    <publish>ALWAYS</publish>
  </buildScan>
  <buildCache>
    <local>
      <enabled>true</enabled>
    </local>
  </buildCache>
</gradleEnterprise>
```

Of course, just enabling a cache doesn’t mean you’ll automatically be getting the maximum possible benefits from it. In subsequent exercises we’ll cover monitoring, optimizing, and debugging cache misses in more depth.

To learn more about how a build cache works, see this chapter from our book Developer Productivity Engineering: https://go.gradle.com/dpe-faster-caching

Chapter 4. Getting the Most Out of a Cache

The mere existence of a build cache doesn't guarantee you'll get any real benefits from it. Without monitoring the cache hit rate, throughput, and overhead, it is overwhelmingly likely that the cache will dramatically underperform and provide only a fraction of its theoretical peak value. I once made a change that, unintentionally, ceased all publishing of new artifacts to our remote cache. It wasn't obvious at first - cache hit rates didn't immediately drop to 0 since portions of the codebase are stable. Over the course of a few weeks our cache hit rate steadily degraded from ~60%, down to ~30% when I could no longer ascribe the decline to any transient circumstance.

Without monitoring the data it's likely no one would have noticed for… who knows how long. The build just would have been slower, the cache idly twiddling its thumbs waiting to serve up a hit that would never come again. That poor, lonely strangely-personified cache. Save a poor, personified build cache from eternal loneliness today - monitor and optimize!

Help Luke Accelerate His Build

Luke is a Software Engineer who has been frustrated with how slow his builds have been. So he was excited to hear that the Developer Productivity team at his company was setting up a remote build cache. Once setup, his coworkers were happy with how much faster their builds had gotten, but Luke was frustrated to discover that his builds are still as slow as ever. He's asked you to compare data from his builds to builds published on CI and to figure out why the cache is doing nothing for him.

Follow this link https://go.gradle.com/wb-ch4-g1 to access the build scans list, pre-filtered to show the relevant time range. Use these build scans to figure out how Luke's builds could be accelerated.

NOTE All of the build scans in this exercise are from Gradle builds. Maven build scans for this exercise are coming in a subsequent edition of this book.

Hints

- Luke has been working on the "sam/build-data-generation" git branch. The build scan list can be filtered to show only builds from this branch by filling in the custom values filter with:

  `git branch name=sam/build-data-generation`

- You may have to expand the time range, or remove the time range filter, to find the builds where this happened
- The performance dashboard shows avoidance savings, are Luke’s builds benefiting?
- Cache misses can be caused by different file inputs/outputs
- Cache misses can be caused by different JVM versions between producer & consumer
- The Compare build scan button in a build scan can be used to compare two builds
- Builds from CI machines are tagged with the “CI” tag

Solution

Here’s a comparison between one of Luke’s builds and one that ran on CI: https://go.gradle.com/wb-ch4-cmp

Note that Luke’s build is running on a version 1.8.0 JDK, and CI is running on a version 11.0.4 JDK. For the purposes of Java related tasks like compilation, test execution, and checkstyle the major and minor version of the JDK are included in the calculation of the cache key. So Luke isn’t getting any cache hits because he’s using an older JDK - not to mention occasionally having different runtime behavior than on CI! If you’ve ever been in the position of struggling to replicate a problem on CI, or have something work locally that doesn’t work on CI, this might be something you want to double check yourself!

To learn more about the importance of having highly efficient builds, see this chapter from our book Developer Productivity Engineering: https://go.gradle.com/dpe-every-second

Chapter 5. The Sneaky Saboteurs Undermining your Cache

A build cache is, conceptually, quite simple: Hashed inputs form a key, compressed outputs form a value. Taken together, you get entries in the store we call a cache. But even simple concepts can have complex behavior and surprising interactions with other systems. Of particular interest to those who want blazing fast builds are the tricky, elusive phenomena that can cripple cache effectiveness:

The first saboteur appears at first glance to be completely innocuous: The humble timestamp. If a timestamp appears in the inputs of a cacheable task everything will appear perfectly normal. The task will run as normal and its result will be stored in the cache without incident. The next time that task is invoked, even if nothing else has changed, that timestamp will have changed. And so every execution will add a new entry into the cache that will never, ever be used in any subsequent build.

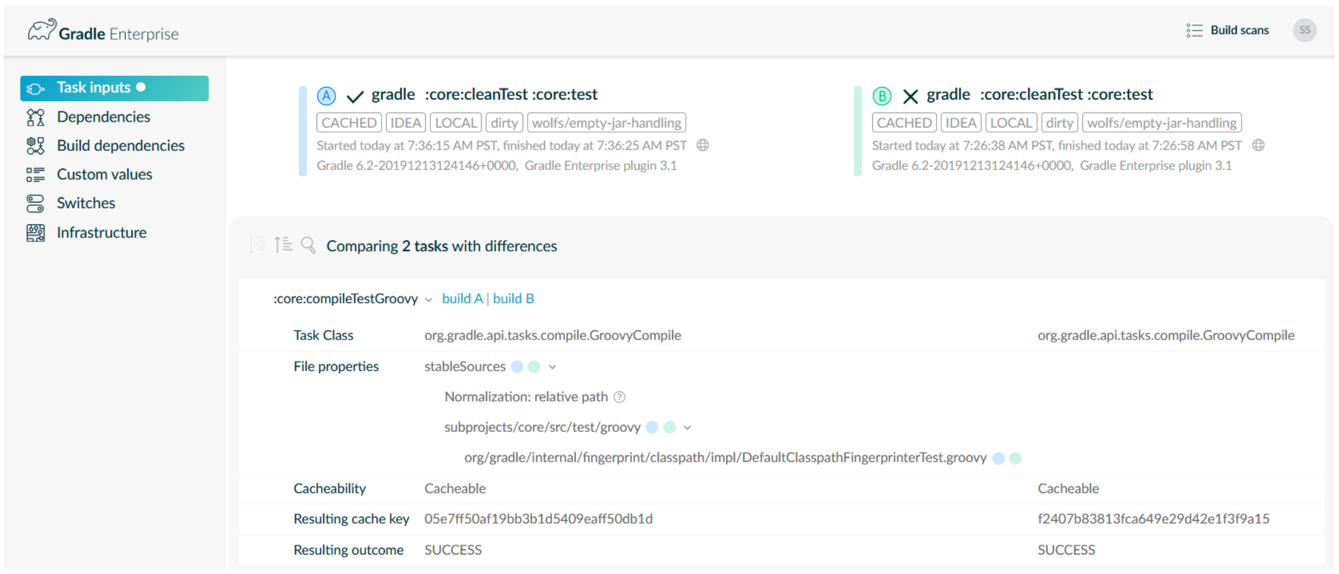
The second subtle compromiser of caches are absolute file paths. These aren't an issue for a local cache that only you use. But for remote caches shared throughout an organization - unless everyone (including CI machines) have standardized on the exact some hard drive layout - an absolute file path on my machine is unlikely to be exactly match one on yours.

The third cache underminer is non-deterministic ordering of collection inputs. The mathematical formulation of a `Set` is unordered, but the practical implementations we use can be iterated over. Iterating over unordered collections may or may not yield elements in the same order. This is particularly likely when implementations of the collection differ. Such as when you and your colleague are using different versions of the JDK. Even patch versions of the JDK can result in unordered collections returning their elements in different order.

There are other rogues out there, but these three are among the most common.

Root Causing Cache Misses with Build Comparison

The Build Comparison view enables various aspects of two different builds to be compared side-by-side. One advanced, optional feature of this view is capturing detailed information about task/goal inputs and outputs. By comparing a build you expect to have populated the cache with the build that should have gotten a cache hit you can see exactly which input is volatile.


Gradle Enterprise

Build scans SS

- Task inputs
- Dependencies
- Build dependencies
- Custom values
- Switches
- Infrastructure

A ✓ gradle :core:cleanTest :core:test
CACHED IDEA LOCAL dirty wolfs/empty-jar-handling
Started today at 7:36:15 AM PST, finished today at 7:36:25 AM PST
Gradle 6.2-20191213124146+0000, Gradle Enterprise plugin 3.1

B ✗ gradle :core:cleanTest :core:test
CACHED IDEA LOCAL dirty wolfs/empty-jar-handling
Started today at 7:26:38 AM PST, finished today at 7:26:58 AM PST
Gradle 6.2-20191213124146+0000, Gradle Enterprise plugin 3.1

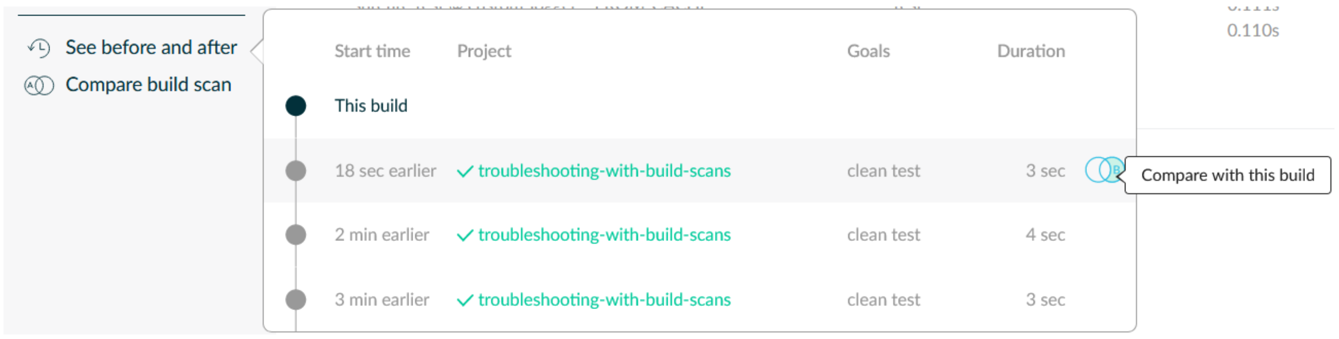
Comparing 2 tasks with differences

:core:compileTestGroovy build A | build B

Task Class	org.gradle.api.tasks.compile.GroovyCompile	org.gradle.api.tasks.compile.GroovyCompile
File properties	stableSources ● ● Normalization: relative path subprojects/core/src/test/groovy ● ● org/gradle/internal/fingerprint/classpath/impl/DefaultClasspathFingerprinterTest.groovy ● ●	
Cacheability	Cacheable	Cacheable
Resulting cache key	05e7ff50af19bb3b1d5409eaff50db1d	f2407b83813fca649e29d42e1f3f9a15
Resulting outcome	SUCCESS	SUCCESS

Figure 7. Task input comparison showing which task inputs differed between two builds

If the builds you wish to compare were run from the same workspace, there’s a handy shortcut for getting to this comparison view. From one of the build scans, click the "See before and after" link and mouse over the build you wish to compare to reveal the "Compare with this build" button:



	Start time	Project	Goals	Duration
●	This build			
●	18 sec earlier	✓ troubleshooting-with-build-scans	clean test	3 sec
●	2 min earlier	✓ troubleshooting-with-build-scans	clean test	4 sec
●	3 min earlier	✓ troubleshooting-with-build-scans	clean test	3 sec

Figure 8. Using the "See before and after" button to initiate comparison of two builds from the same workspace

Volatility vs Cache

1. **Locating volatility:** A simple way to discover volatility undermining the effectiveness of your cache is to run the same build twice with no changes. Based on these build scans, run in sequence in the same workspace, identify any cacheable tasks that executed instead of being resolved from the cache:
   - Gradle build scans produced by running `./gradlew clean build` twice:
     - https://go.gradle.com/wb-ch5-g1
     - https://go.gradle.com/wb-ch5-g2
   - Maven build scans produced by running `./mvnw clean test` twice:
     - https://go.gradle.com/wb-ch5-m1
     - https://go.gradle.com/wb-ch5-m2

   Optional: Reproduce these build scans yourself by checking out the `volatile-inputs` branch of the troubleshooting-with-scans repository. This branch has been pre-configured to have caching and detailed task/goal input capturing enabled for Gradle and Maven. Check out that branch with git https://github.com/gradle/troubleshooting-with-scans/tree/volatile-inputs, or download it as a zip file: https://github.com/gradle/troubleshooting-with-scans/archive/volatile-inputs.zip. The local cache for Maven is a paid feature of Gradle Enterprise. Reproducing the Maven build scans in this chapter with the free scans.gradle.com service will not result in any cache hits.

2. **Comparing build scans:** Use the Compare build scan button to compare these two builds and identify the inputs impacting task cacheability:
   - Example Gradle Comparison: https://go.gradle.com/wb-ch5-g3
   - Example Maven Comparison: https://go.gradle.com/wb-ch5-m3

Solution - Comparing build scans: This comparison indicates the existence of a timestamp file being generated in the `example-project` project. This timestamp is placed in src/main/resources, a location that Gradle and Maven alike treat as containing files that should be available at runtime. For Gradle builds, example-project/build.gradle defines the custom task which creates this file. For Maven builds, example-project/pom.xml adds a plugin which runs the timestamp.groovy script which creates this file. In both cases, the most cache-friendly solution is to decide you don't *really* need that timestamp inside the jar and remove the logic that creates it entirely.

3. **Verifying volatility removal:** With this volatility removed, rebuilding the project twice will allow you to verify the impact of these changes. Answer the questions:

   ◦ Are all cacheable tasks being resolved from the cache or does some volatility remain?

   ◦ Are the avoidance savings for these builds better or worse than the builds that included timestamp files?

   Using these build scans:

   ◦ Gradle build scan: https://go.gradle.com/wb-ch5-g4

   ◦ Maven build scan: https://go.gradle.com/wb-ch5-m4

   Optional: Remove the timestamp generation and verify the impact on cacheable tasks with build scans of your own. The "See before and after" button and Build Comparison features are not enabled on scans.gradle.com.

Living with Volatility

If the volatility in your Gradle builds turns out to be something that can't be removed completely, then there are techniques that can mitigate the performance impact. See the "Normalization" section of the documentation to learn more about these techniques and their caveats.

- Gradle: https://docs.gradle.org/current/userguide/more_about_tasks.html#sec:configure_input_normalization
- Maven: https://docs.gradle.com/enterprise/maven-extension/#normalization

Chapter 6. Catching Performance Regressions Before They Catch You

Creating performant software is difficult. Keeping that performance as the software grows & evolves is very difficult. Keeping performance without measuring, analyzing, and acting is impossible. Part of what makes maintaining this all so challenging is the ease and invisibility with which regressions are introduced.

In this exercise we'll explore using performance analytics to detect, understand, and fix a serious performance regression.

Hiding in Plain Sight: Data Reveals Regressions

1. Follow this link https://go.gradle.com/wb-ch7-g1 to access the Performance Trends Dashboard, pre-filtered to the relevant time range.
2. **Locating the regression:** Using the information presented on the dashboard, answer these questions:
   - Across the entire time period, what was the average build time?
   - How much time was cumulatively saved because of the remote cache and local caches?
   - On what date did build performance regress and by approximately how much?

Hints

- Hovering the mouse over a data point on any chart will update the percentile/mean/median display.

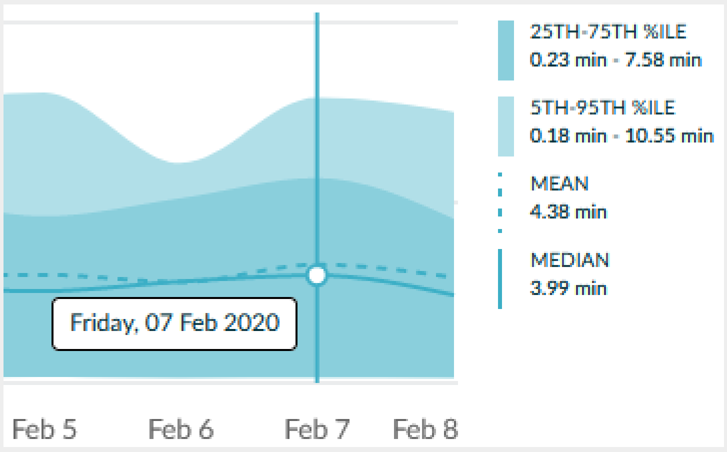


Figure 9. The legend showing percentile/mean/median for a particular day

- If day-to-day variance is making it hard to see broader trends, try changing the time resolution from "Day" to "Week"

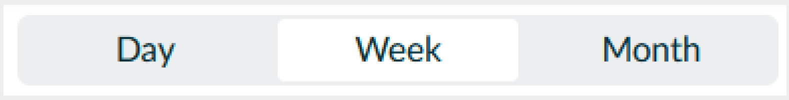
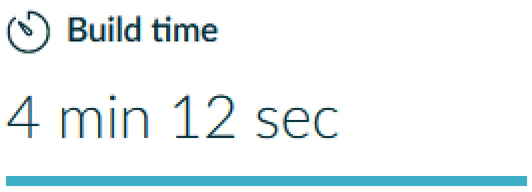


Figure 10. The time resolution selector set to "Week"

Solution - Locating the regression:

- Average Build Time is displayed on the "Build time" chart:



Build time

4 min 12 sec

- Multiplying the build count by the cache time savings displayed in the "Avoidance Savings" chart shows the caches saved this organization ~40 person-hours this month.

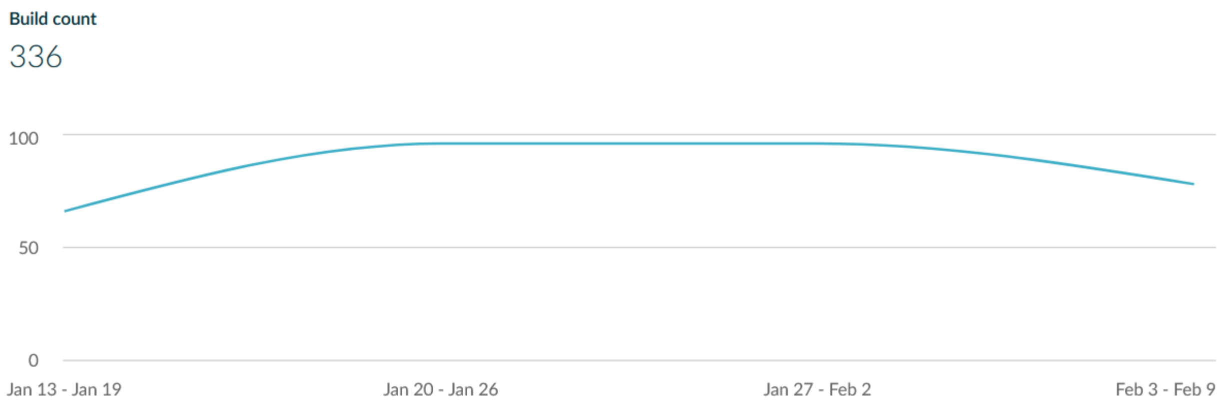
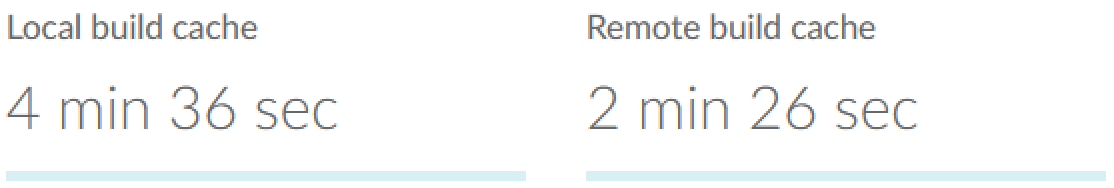


Figure 11. Build count chart showing total number of builds in the time period



Local build cache

4 min 36 sec

Remote build cache

2 min 26 sec

Figure 12. Average per-build avoidance savings due to the local and remote caches

- The build time chart shows that over a single day average build time increased by over a third.

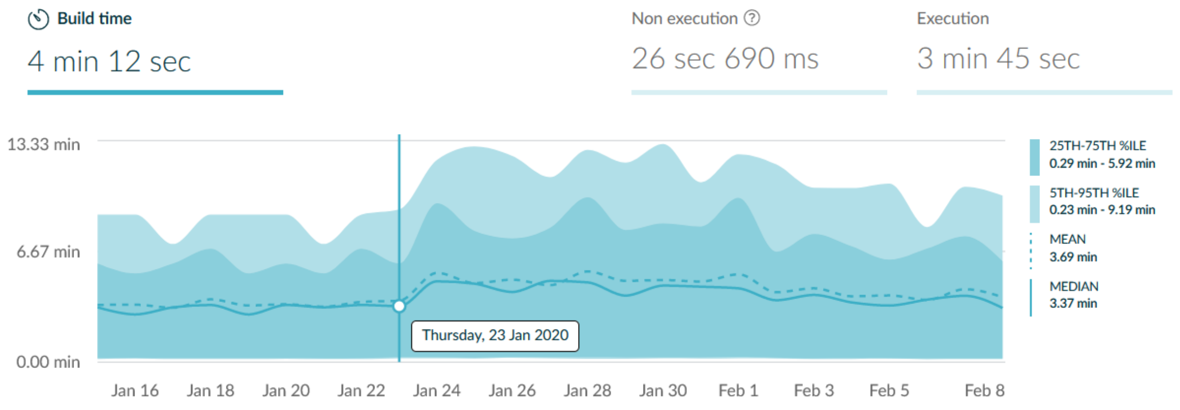


Figure 13. Build Time chart before the performance regression

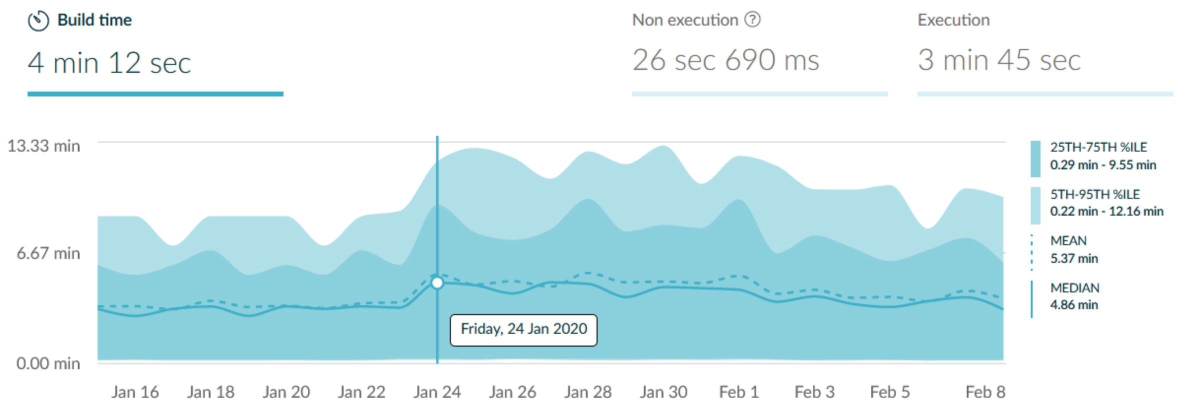
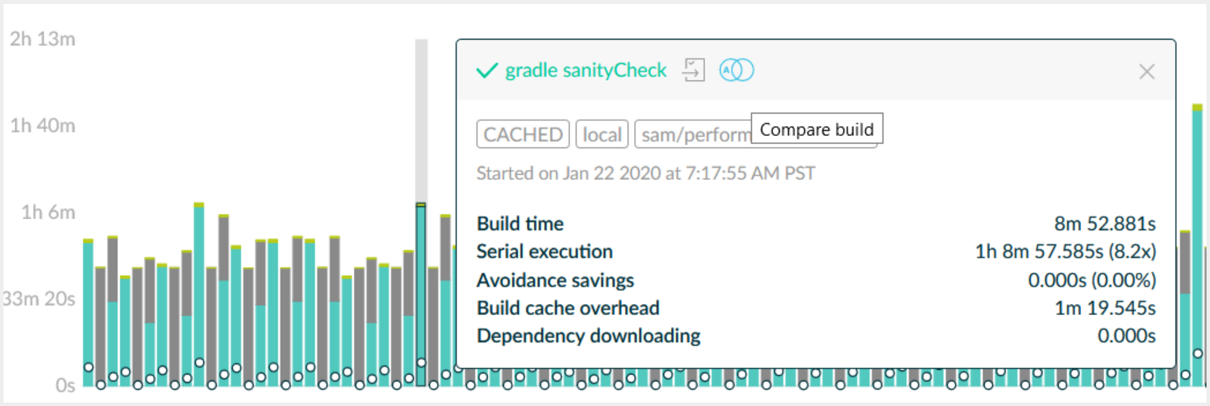


Figure 14. Build Time chart after the performance regression

3. **Root causing the regression:** Compare a build from before and from after the regression to determine what caused it.

Hints

You can initiate comparison of two builds directly from the Performance Dashboard. Click one of the columns representing a build, click the Compare build button, repeat for a second build, and click the "Compare" button to initiate comparison.

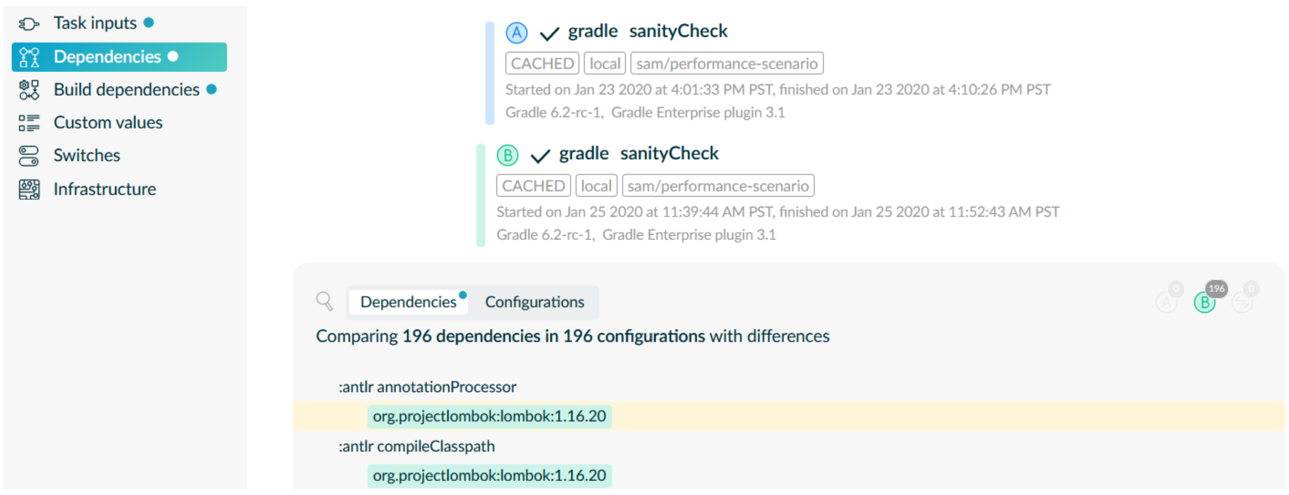


Started on Jan 22 2020 at 7:17:55 AM PST

Build time	8m 52.881s
Serial execution	1h 8m 57.585s (8.2x)
Avoidance savings	0.000s (0.00%)
Build cache overhead	1m 19.545s
Dependency downloading	0.000s

Figure 15. Selecting a build for comparison from the Performance Dashboard

Solution - Root causing the regression: Comparing builds before and after the regression and viewing the Dependencies tab shows that an old version of the Lombok annotation processor has been added to the build.



A gradle sanityCheck
CACHED local sam/performance-scenario
Started on Jan 23 2020 at 4:01:33 PM PST, finished on Jan 23 2020 at 4:10:26 PM PST
Gradle 6.2-rc-1, Gradle Enterprise plugin 3.1

B gradle sanityCheck
CACHED local sam/performance-scenario
Started on Jan 25 2020 at 11:39:44 AM PST, finished on Jan 25 2020 at 11:52:43 AM PST
Gradle 6.2-rc-1, Gradle Enterprise plugin 3.1

Dependencies Configurations

Comparing 196 dependencies in 196 configurations with differences

:antlr annotationProcessor
org.projectlombok:lombok:1.16.20
:antlr compileClasspath
org.projectlombok:lombok:1.16.20

Figure 16. A dependency comparison showing that Lombok is used in the later build (B), and but not in the earlier build (A)

Annotation processors are powerful tools, but asking for more work to be done during compilation isn't free. Particularly when it's an older version from before Lombok supported incremental compilation. Lombok added support for incremental compilation when building with Gradle in version 1.16.22.

4. **Understanding the improvement:** Using the same techniques we used to locate and root cause the regression, answer these questions:
   - When did build time speed back up?
   - How much faster is the build post-speedup?
   - Is the build now faster or slower than it was before the original regression?
   - What changed to cause the performance improvement?

Solution - Understanding the improvement: On the Build time chart we can see when the improvement took place.

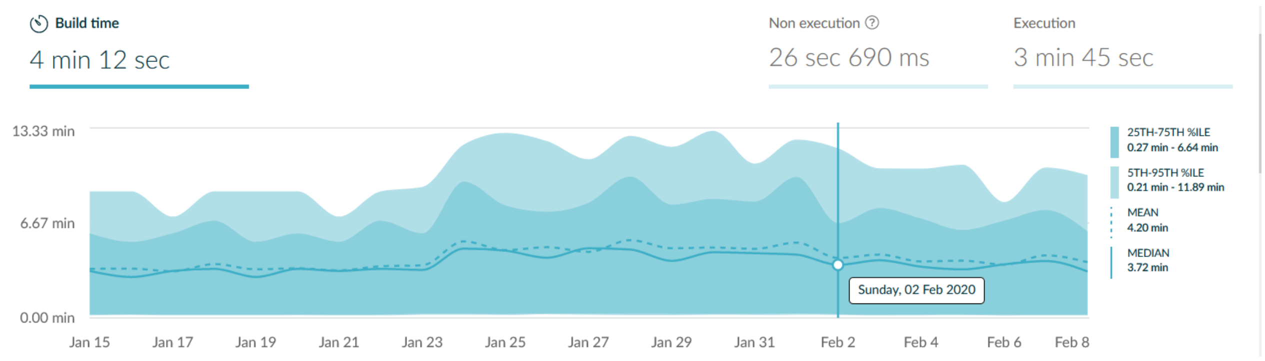
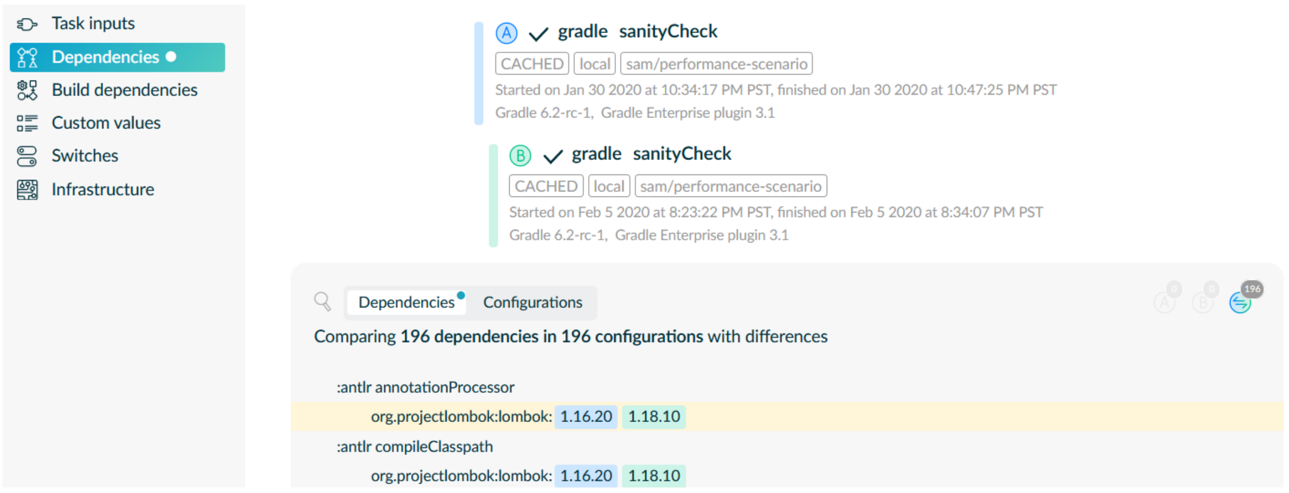


Figure 17. Build time chart showing the date the performance improvement took effect.

Comparing builds from before and after the improvement we can see that the faster builds use a newer version of the annotation processor.



A ✓ gradle sanityCheck
CACHED local sam/performance-scenario
Started on Jan 30 2020 at 10:34:17 PM PST, finished on Jan 30 2020 at 10:47:25 PM PST
Gradle 6.2-rc-1, Gradle Enterprise plugin 3.1

B ✓ gradle sanityCheck
CACHED local sam/performance-scenario
Started on Feb 5 2020 at 8:23:22 PM PST, finished on Feb 5 2020 at 8:34:07 PM PST
Gradle 6.2-rc-1, Gradle Enterprise plugin 3.1

Comparing 196 dependencies in 196 configurations with differences

:antlr annotationProcessor
    org.projectlombok:lombok: 1.16.20 1.18.10
:antlr compileClasspath
    org.projectlombok:lombok: 1.16.20 1.18.10

Figure 18. Dependency comparison of builds from before and after the upgrade.

This newer version fully supports incremental compilation and so it has a minimal, but non-zero, impact on build performance.

5. **Optional bonus exercise:** One developer is experiencing significantly longer builds than any of their colleagues. Use the Performance Dashboard to determine who is experiencing slow builds and why.

Hints

Use the 'Tags' filter to only show 'local' builds to avoid seeing any CI builds. Try comparing a longer-duration build to a similar, but shorter-duration build.

Solution - Optional bonus exercise: It turns out that all of the longest-duration builds are being run by Eric. The Infrastructure section of Eric’s builds show that he’s running on a 4-core, Linux system named "erics-pocket-calculator". His builds take substantially longer than those of any of his peers with faster computers. Over the course of a year, Eric will have wasted days waiting on slow builds. In this case, the most effective remediation is to buy Eric a new computer!

Chapter 7. Flaky Builds: Bane of Developer Productivity

Nothing quite ruins my day like a build failure that has nothing to do with my changes. Complex builds can be rife with issues that recur with varying frequency and severity. This variability can make it difficult to identify the highest priority issue. Even verifying that a fix worked can be difficult when not everything is known about the original bug.

But comprehensive data, when properly analyzed, can shed light on all aspects of even the trickiest bugs. In this exercise we'll use the Failures Dashboard to identify the most severe problem afflicting a build and fix it.

Investigating a Recurring Build Problem

1. Follow this link https://go.gradle.com/wb-ch6-g1 to access the Failures Dashboard, pre-filtered to show the relevant time range.

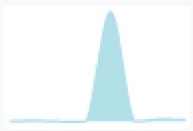
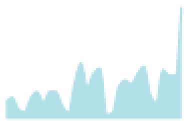
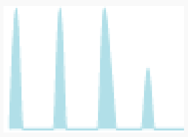
   Note that the failure type filter is set to "Non-Verification" by default. We define "Non-Verification" failures as build failures that are not the fault of the changes being built. This is in contrast to "Verification" failures where the changes being built have failed a quality gate. This allows us to more easily focus on issues likely to be related to the correctness and reliability of the build. For more information on test failure categorization, see: https://docs.gradle.com/enterprise/failure-classification/.

2. **Finding build bugs:** Using information presented on the dashboard, answer these questions:

   - How many builds failed with Non-Verification failures within the time range?
   - Are there any issues that appear, apparently get fixed, then reoccur again later?
   - What is the most frequent failure?
   - If the most frequent failure were fixed, by how much would it reduce the total number of Non-Verification build failures?

Hints

Each group of failures listed on the dashboard comes with a small preview of its "Failures over time" chart. So before even clicking the failure to get more details you can get a sense of when it has been happening and if the problem is ongoing.

Failures over Time Preview	Possible Interpretation
	Transient issue. Already fixed or circumstance hasn't yet reoccurred.
	Ongoing issue.
	Recurring issue. May be something easily regressed, or the issue occurs in a regularly scheduled CI job.

This data is heavily influenced by patterns in human behavior. A dip in the rate of appearance of a particular failure may indicate any (or none) of the following:

- The issue has been fixed and no longer fails.
- Developers have gotten frustrated with the faulty workflow and are avoiding it.
- It's the weekend or a holiday and developers aren't running builds.

Data visualizations that appear to unambiguously show a strong trend can immediately tempt you toward a particular narrative. Always consider whether the trend you're investigating may be best explained by human context, rather than anything technical.

Note that charts in the Failures Dashboard are subject to all of the normal filtering criteria as well as the `Non-Verification / Verification / All Failures` selector. Depending on how you've setup the filters you may see slightly different numbers but the overall result should be the same.

Solution - Finding build bugs:

> How many builds failed with Non-Verification failures within the time range?

The "Failure Count" ring chart compares the number of builds with failures to the total number of builds.

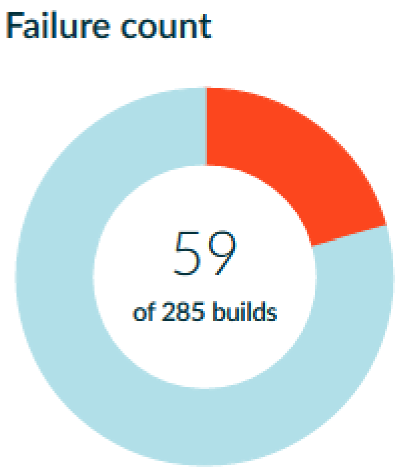
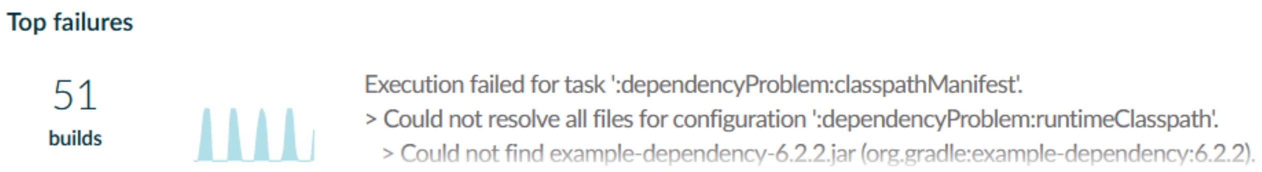


Figure 19. The failure count chart

> Are there any issues that appear, apparently get fixed, then reoccur again later? What is the most frequent failure? If the most frequent failure were fixed, by how much would it reduce the total number of Non-Verification build failures?

The data shows that this recurring issue is the most frequent problem in this build, representing ~85% of all Non-Verification failures:



```
Execution failed for task ':dependencyProblem:classpathManifest'.
> Could not resolve all files for configuration ':dependencyProblem:runtimeClasspath'.
  > Could not find example-dependency-6.2.2.jar (org.gradle:example-dependency:6.2.2).
```

Figure 20. A recurring failure shown on the Failures Dashboard

Having found the worst build issue, now it's time to fix it.

3. **Contextualizing the bug** Click on the recurring failure to go to its details page and use the information there to answer these questions:
   - Does this issue affect local builds, CI builds, or both?
   - Suppose you wanted to remotely access a CI runner to investigate further. What is a host

name of a CI machine that has exhibited this issue?

- Does this affect only a single developer, or does it affect multiple developers?
- Who could you ask for help reproducing this issue?

Hints

- You can filter by the `CI` tag to see only data from CI builds.
- You can filter by the `local` tag to see only data from local builds.

Solution - Contextualizing the bug: This issue affects both local and CI builds. To determine the host name of a CI machine to remote into, filter by the `CI` tag and look at the "Affected hosts" chart. dev87.gradle.org is one such host.

The "Affected users" chart shows us that this issue affects multiple different developers. This tells us that the issue is probably not the result of a single developer messing up their working environment. To find the name of someone to ask for help reproducing the issue, filter by the `local` tag and look at the "Affected users" chart. Luke has run into this issue a number of times, he might be able to provide more information.

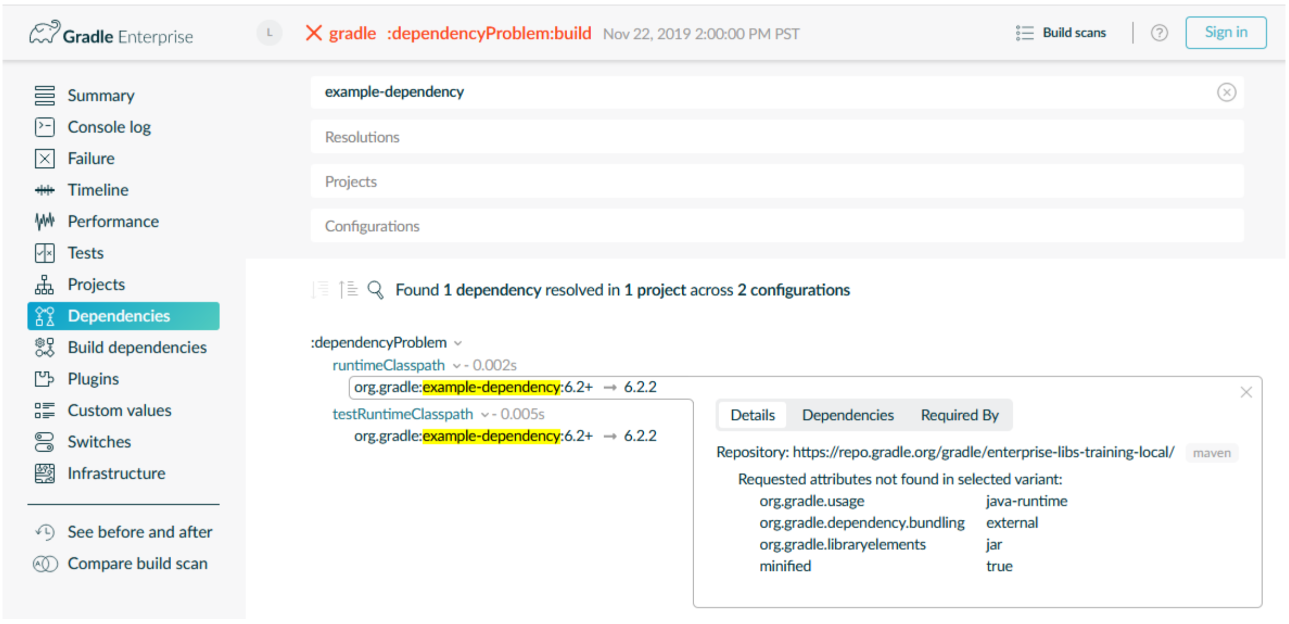
4. **Revealing the root cause:** Investigate a build scan from one of the builds that exhibited this issue to determine:

   - Which project is requesting the dependency that cannot be resolved?
   - What version of the dependency is being requested?
   - Which repository is the problematic dependency being resolved from?

Hints

- You can access individual build scans from builds exhibiting the failure from the "Failed builds" table below the charts.
- Access the Dependencies section of a build scan and use the magnifying glass button to search for "example-dependency"
- Mouse over a dependency on the list to reveal the more info button for full details on that dependency

Solution - Revealing the root cause: The project, helpfully named "dependencyProblem," is the one requesting org.gradle.example-dependency as part of its `runtimeCLasspath`. The version being requested is `6.2+`. The `+` instructs Gradle to search all the repositories known to this build for the latest available version of this dependency. The more info window shows that version 6.2.2 of this dependency is being resolved from a maven repository, located at https://repo.gradle.org/gradle/enterprise-libs-training-local/.



Found 1 dependency resolved in 1 project across 2 configurations

```
:dependencyProblem
  runtimeClasspath - 0.002s
    org.gradle:example-dependency:6.2+ → 6.2.2
  testRuntimeClasspath - 0.005s
    org.gradle:example-dependency:6.2+ → 6.2.2
```

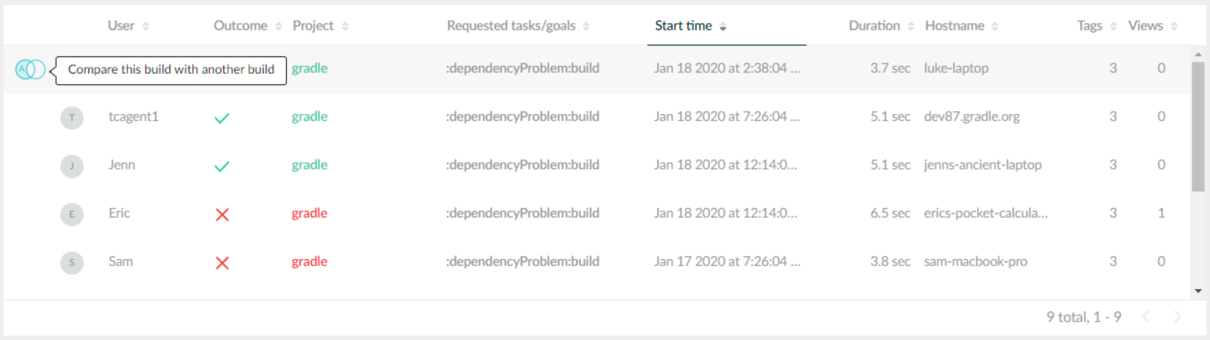
```
Repository: https://repo.gradle.org/gradle/enterprise-libs-training-local/  maven
  Requested attributes not found in selected variant:
    org.gradle.usage                  java-runtime
    org.gradle.dependency.bundling    external
    org.gradle.libraryelements        jar
    minified                          true
```

Figure 21. The details for the dependency related to this failure

5. **After and before:** From that build scan, click compare build scan and find a passing build for the same project and git branch. What are the differences in how this dependency is resolved?

Hints

- In one of the builds failing with this issue, look in the Custom values section and note the "Git Branch Name". Use the "Custom values" field on the build scans list to filter to only build scans relevant to this issue
- From the build scans list mouse over the space to the left of an entry to reveal the "Compare this build with another build" button



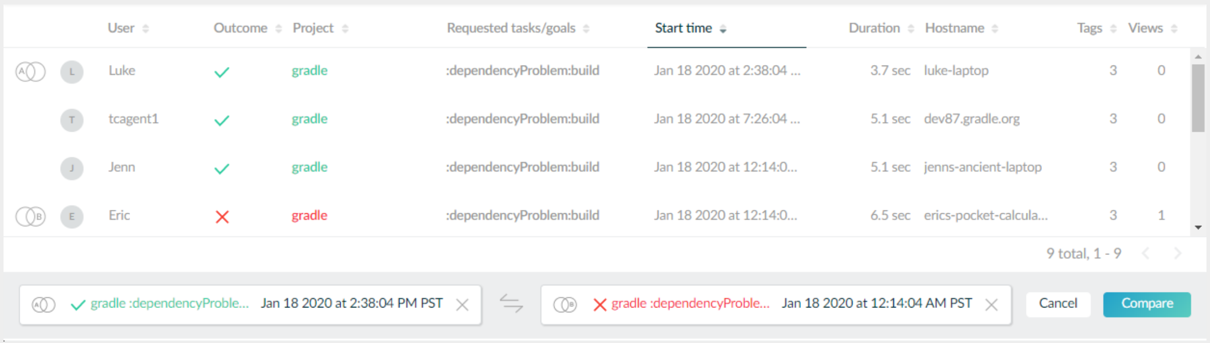
Compare this build with another build

User	Outcome	Project	Requested tasks/goals	Start time	Duration	Hostname	Tags	Views
		gradle	:dependencyProblem:build	Jan 18 2020 at 2:38:04 ...	3.7 sec	luke-laptop	3	0
tcagent1	✓	gradle	:dependencyProblem:build	Jan 18 2020 at 7:26:04 ...	5.1 sec	dev87.gradle.org	3	0
Jenn	✓	gradle	:dependencyProblem:build	Jan 18 2020 at 12:14:0...	5.1 sec	jenns-ancient-laptop	3	0
Eric	✗	gradle	:dependencyProblem:build	Jan 18 2020 at 12:14:0...	6.5 sec	erics-pocket-calcula...	3	1
Sam	✗	gradle	:dependencyProblem:build	Jan 17 2020 at 7:26:04 ...	3.8 sec	sam-macbook-pro	3	0

9 total, 1 - 9

Figure 22. First step of comparing two build scans from the build scans list

Repeating with a second build will all you to compare any two build scans.



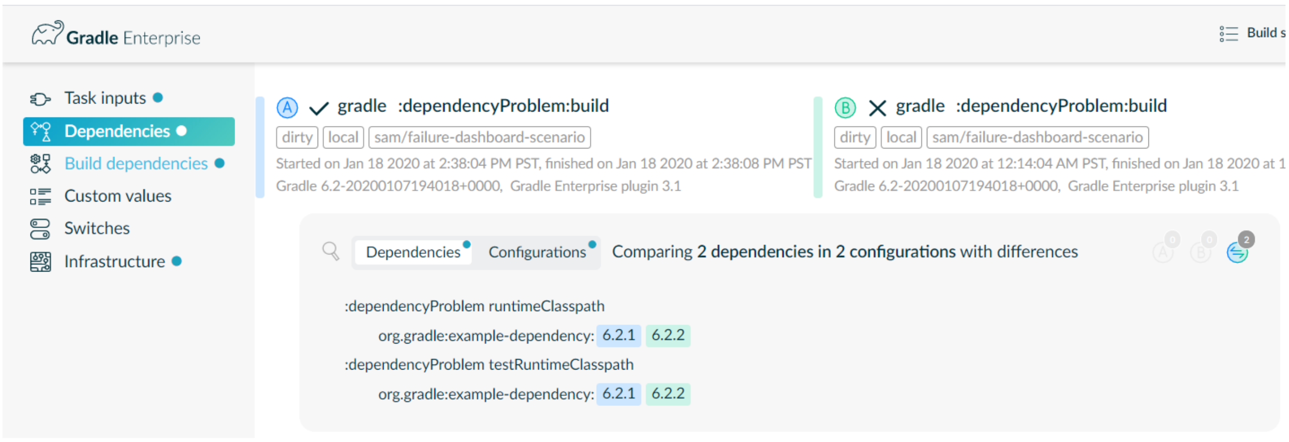
User	Outcome	Project	Requested tasks/goals	Start time	Duration	Hostname	Tags	Views
Luke	✓	gradle	:dependencyProblem:build	Jan 18 2020 at 2:38:04 ...	3.7 sec	luke-laptop	3	0
tcagent1	✓	gradle	:dependencyProblem:build	Jan 18 2020 at 7:26:04 ...	5.1 sec	dev87.gradle.org	3	0
Jenn	✓	gradle	:dependencyProblem:build	Jan 18 2020 at 12:14:0...	5.1 sec	jenns-ancient-laptop	3	0
Eric	✗	gradle	:dependencyProblem:build	Jan 18 2020 at 12:14:0...	6.5 sec	erics-pocket-calcula...	3	1

9 total, 1 - 9

✓ gradle :dependencyProble... Jan 18 2020 at 2:38:04 PM PST ✗ gradle :dependencyProble... Jan 18 2020 at 12:14:04 AM PST

Figure 23. Second step of comparing two build scans from the build scan list

Solution - After and before: Comparing a passing and failing build from the same `sam/failure-dashboard-scenario` branch and visiting the Dependencies section will show the key difference. The passing build is using example-dependency version 6.2.1 and the failing build is trying and failing to use example-dependency 6.2.2.



A ✓ gradle :dependencyProblem:build
dirty | local | sam/failure-dashboard-scenario
Started on Jan 18 2020 at 2:38:04 PM PST, finished on Jan 18 2020 at 2:38:08 PM PST
Gradle 6.2-20200107194018+0000, Gradle Enterprise plugin 3.1

B ✕ gradle :dependencyProblem:build
dirty | local | sam/failure-dashboard-scenario
Started on Jan 18 2020 at 12:14:04 AM PST, finished on Jan 18 2020 at 1
Gradle 6.2-20200107194018+0000, Gradle Enterprise plugin 3.1

Dependencies | Configurations | Comparing 2 dependencies in 2 configurations with differences

```
:dependencyProblem runtimeClasspath
    org.gradle:example-dependency: 6.2.1  6.2.2
:dependencyProblem testRuntimeClasspath
    org.gradle:example-dependency: 6.2.1  6.2.2
```

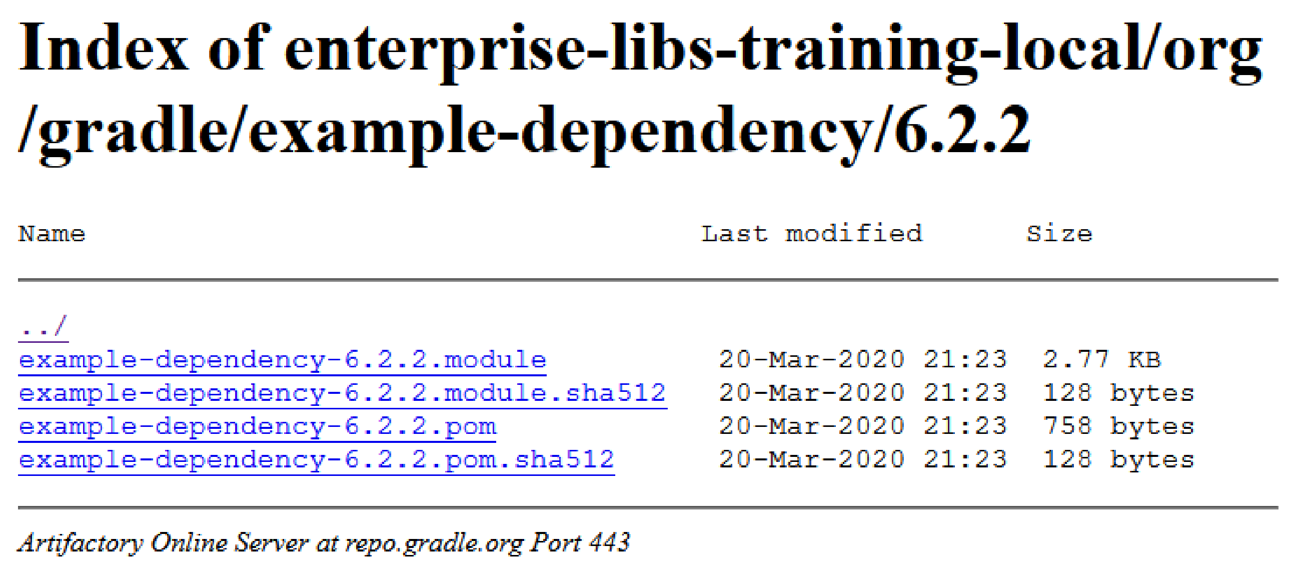
Figure 24. Comparison of dependencies between a passing and a failing build

See https://go.gradle.com/wb-ch6-g2 to explore the build comparison view pictured above.

So through this investigation we've learned that:

- This issue recurs periodically, affecting developers' local builds as well as CI builds.
- The "dependencyProblem" project depends on org.gradle.example-dependency.
- It is requesting the latest available version of that dependency.
- That dependency is being resolved from a repository on the local hard drive.
- When version 6.2.2 is found in the repository, the build fails.

If we look inside that artifact repository, we see that example-dependency version 6.2.2 has a pom file but no jar.



Index of enterprise-libs-training-local/org/gradle/example-dependency/6.2.2

```
Name                                     Last modified      Size
-------------------------------------------------------------------
../
example-dependency-6.2.2.module           20-Mar-2020 21:23  2.77 KB
example-dependency-6.2.2.module.sha512    20-Mar-2020 21:23  128 bytes
example-dependency-6.2.2.pom              20-Mar-2020 21:23  758 bytes
example-dependency-6.2.2.pom.sha512       20-Mar-2020 21:23  128 bytes
-------------------------------------------------------------------
Artifactory Online Server at repo.gradle.org Port 443
```

Figure 25. The contents of the repository, showing a pom file but no jar

Because Gradle can find the metadata, it thinks it has found the dependency, but ultimately the jar itself doesn’t exist so dependency resolution fails. This can happen when publishing an artifact fails partway or is otherwise interrupted. This can also happen as a result of mis-configured or buggy cleanup actions running on the repository.

This specific problem can be addressed by deleting the corrupted entry from the repository and successfully publishing the full entry for 6.2.2. This problem was much more disruptive than it needed to be! Because the build was configured to always take the latest, as soon as the incomplete dependency was published all new builds broke! This category of problem can be mitigated by removing dynamic dependencies or using lock files. See this article on dependency locking: https://docs.gradle.org/current/userguide/dependency_locking.html

Build Breaks are Expensive

Build breaks waste CI resources and CPU time. More importantly, build breaks waste the time and focus of developers who would rather be getting their work done, free of distraction. If you’d like to read more about improving the developer experience through reliable builds, see this chapter from our book Developer Productivity Engineering: https://go.gradle.com/dpe-reliable-builds

Chapter 8. Better Development through Reliable Testing

Automated testing is a powerful technique for keeping an ever evolving codebase from ossifying. Many companies that have been around long enough have at least one dreaded "legacy code" codebase everyone is loath to touch. We write tests to measure and protect the correctness of our product code, but test suites themselves can degrade in usefulness over time. The most common way that test suites degrade is flakiness. A test is said to be flaky when it may return both passing and failing results without the code under test having changed.

In this exercise we'll explore using the Test dashboard to improve and maintain test reliability.

Find and Fix Flakes

Developers like Luke, Jenn, and Eric have been complaining that flaky tests are getting in the way of their work. You've decided to investigate and see what can be done to help.

1. Follow this link https://go.gradle.com/wb-ch8-g1 to access the Test Dashboard, pre-filtered to local builds in the relevant time range.
2. **Finding failures:** Using the information presented on the dashboard, answer these questions:
    - What percentage of all builds have failed tests?
    - Which test class fails most often?
    - Consider: Is this enough information to decide which test classes are problematic?

Solution - Finding failures: The "Builds with failed tests" chart gives the percentage of builds with failing tests:

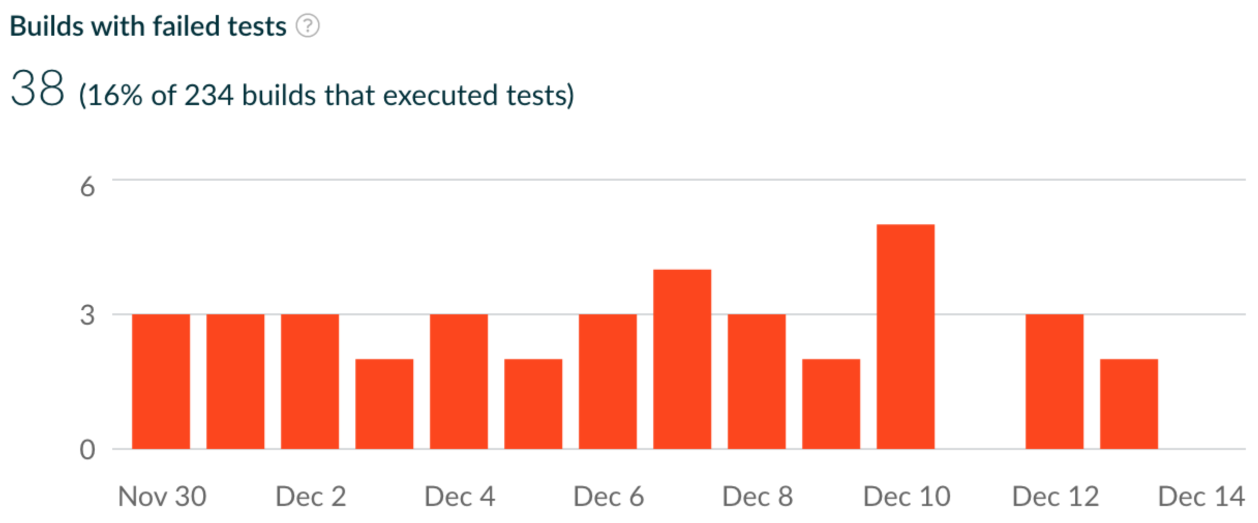
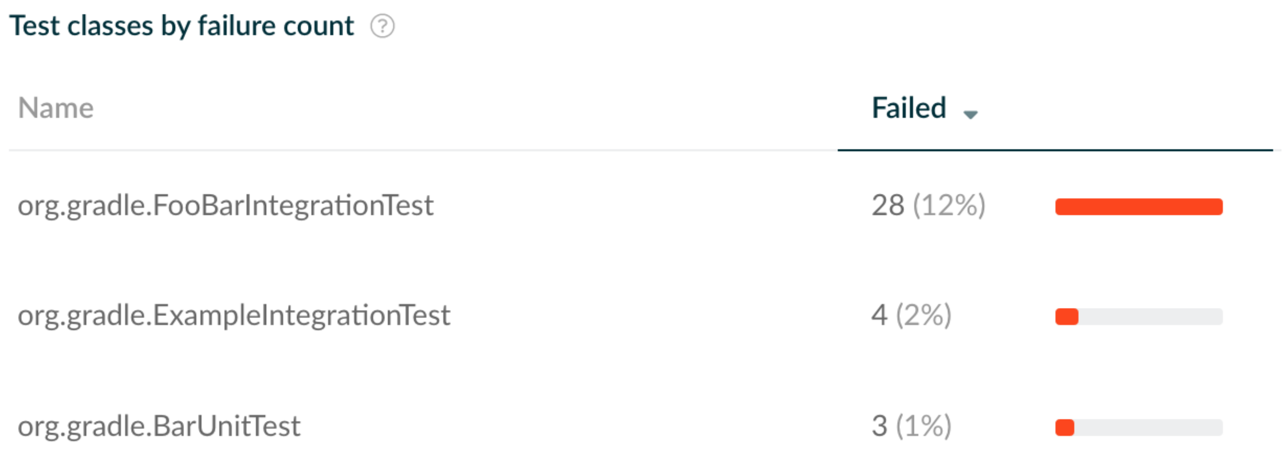


Figure 26. The Builds with failed tests chart

The "Test classes" table sorted by "Failed" lists the test class with the most failures at the top:

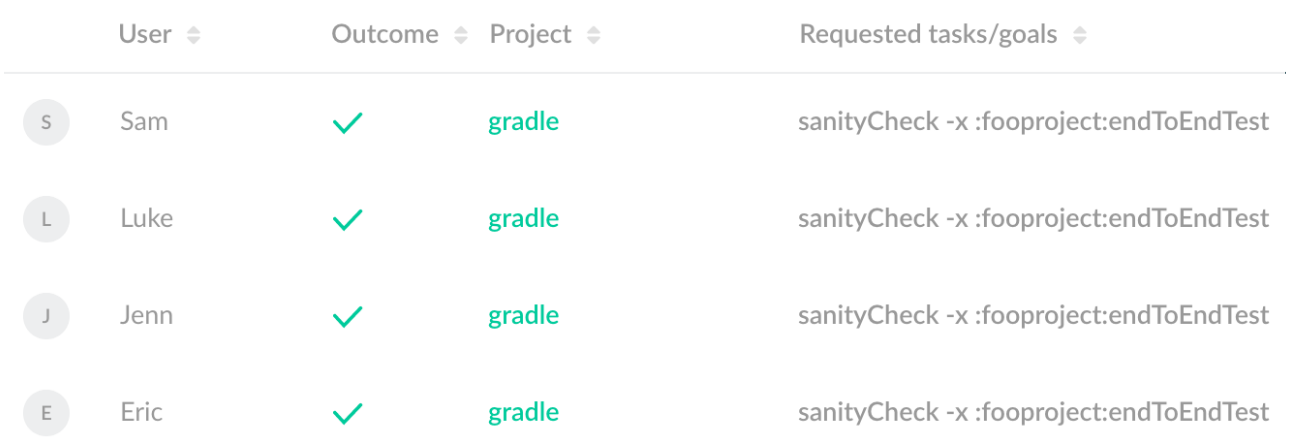


Test classes by failure count

Name	Failed
org.gradle.FooBarIntegrationTest	28 (12%)
org.gradle.ExampleIntegrationTest	4 (2%)
org.gradle.BarUnitTest	3 (1%)

Figure 27. The Test classes table showing that org.gradle.FooBarIntegrationTest frequently fails

The `org.gradle.FooBarIntegrationTest` is where the most failures are happening. Are these the tests we should focus on improving? Deferring answering that question for *just* a moment longer: If you click the "Build scans" button and view the build scan list, you may observe something slightly peculiar:



	User	Outcome	Project	Requested tasks/goals
S	Sam	✓	gradle	sanityCheck -x :fooproject:endToEndTest
L	Luke	✓	gradle	sanityCheck -x :fooproject:endToEndTest
J	Jenn	✓	gradle	sanityCheck -x :fooproject:endToEndTest
E	Eric	✓	gradle	sanityCheck -x :fooproject:endToEndTest

Figure 28. The build scan list showing local builds with a particular task excluded

All of these different developers are excluding a particular task from executing on their local builds. This `:fooproject:endToEndTest` is being excluded from execution on the command line. These engineers have, independently of your investigation, decided that they'd rather lose this test coverage than deal with the frustration of running these flaky tests.

The behavior of these engineers has incidentally caused the data we're analyzing to omit the very thing we're searching for. These engineers - who clearly know how to skip running tests - are still running `org.gradle.FooBarIntegrationTest`. This suggests that it *is* a useful group of tests and likely not the source of the complaints.

This is a useful reminder that forgetting the human aspect of the data you're analyzing can lead your analysis astray. Fortunately, the Test Retry Gradle Plugin has been enabled on CI to help detect flaky tests.

3. **Finding flaky failures:** Remove the `local` entry from the tags filter so that Flaky test results detected on CI may be displayed. Then use the data visualized on the Test dashboard to answer these questions:
   - Which test class is contributing the most flaky results?
   - Which test method within the class is the flakiest of them all?

Hints

- The "Test classes" table can be sorted by "Flaky" to show the flakiest test classes first.
- Clicking a row in this table will display the history of the executions of that test class, with method-level granularity

Solution - Finding flaky failures: The flakiest class is `org.gradle.EndToEndTests` and the method `acceptanceTest1` is the flakiest method within this class. Confident that you've found the place where your efforts will produce the best results, you fix all of the flakiness in this test class. Possibly discovering & fixing flakiness in the product code under test along the way. After all, flakiness isn't always the fault of the test - often the test is faithfully reporting inconsistent product behavior. If flaky tests are merely ignored then it might well be your users who suffer the consequences.

4. **Verifying the improvement:** Expand the time range an additional two weeks into the future to see the results of your efforts. Use the data to visualized on the Test dashboard to answer these questions:

   - Has the overall count of builds with failed tests increased or decreased?
   - Have developers resumed running the `EndToEndTests` locally?
   - Has all flakiness been fixed?

Solution - Verifying the improvement: Perhaps surprisingly, the overall number of build failures hasn't decreased since you fixed the flakiness.

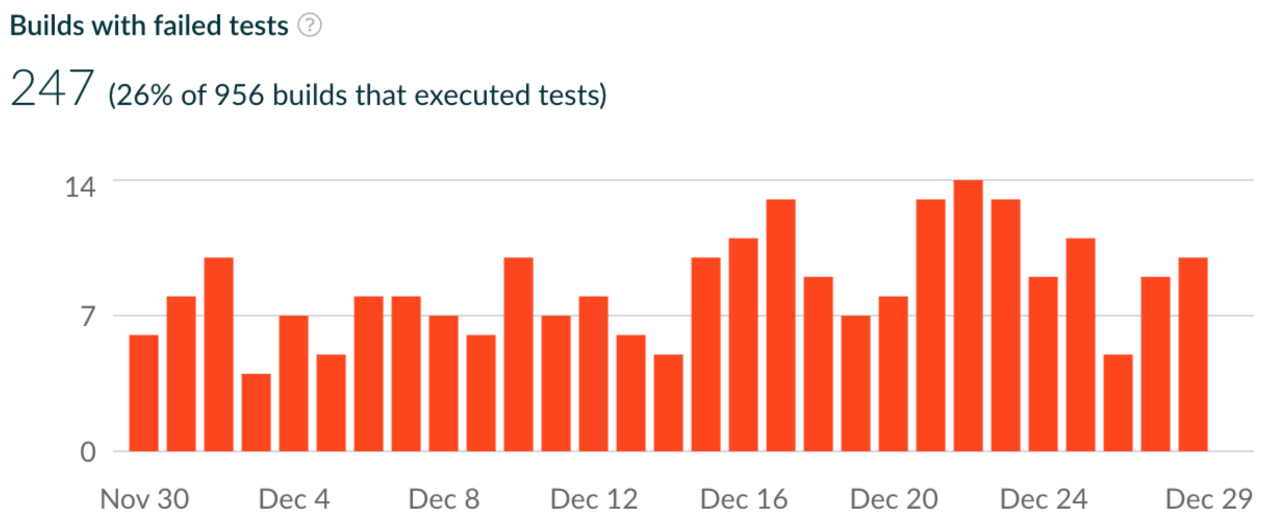


Figure 29. The Builds with failed tests chart for the entire time range

Were your efforts to investigate & de-flake tests ultimately wasted? There are actually **more** test failures now than there were before you started!

Looking at the class-level view, we can see that `org.gradle.EndToEndTests` usage has increased markedly once it ceased to be burdensome to run.

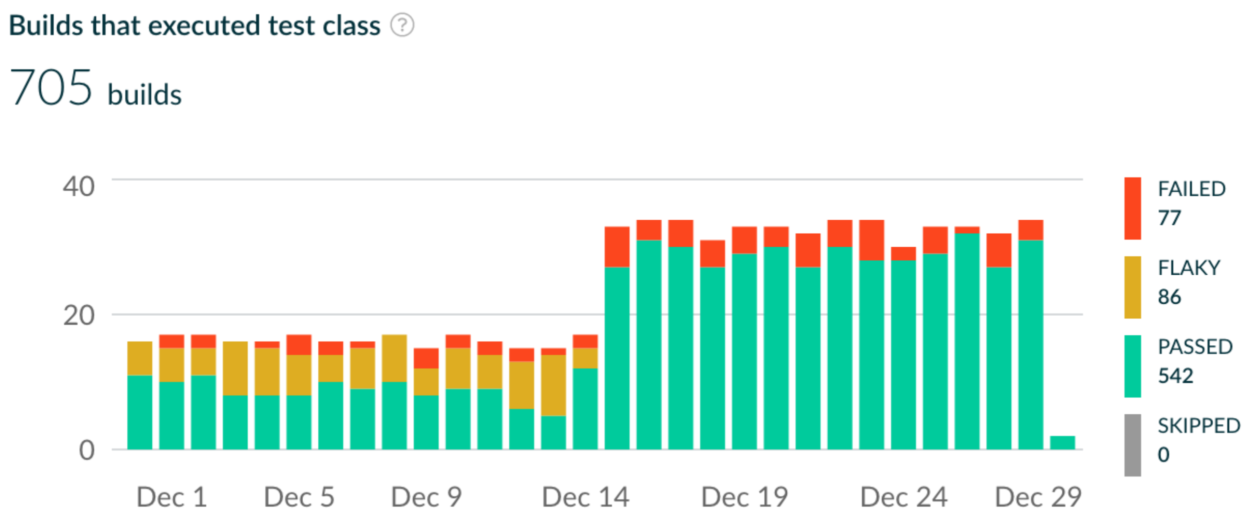


Figure 30. The class-level execution history of org.gradle.EndToEndTests before and after the fix.

These failures are again trusted by developers to legitimately signal the quality of the product code. So the increase in the overall number of failing tests actually reflects *recovery*. Recovery of developer trust that the suite is useful. Recovery of the coverage that started to be lost when the signal/frustration ratio dropped too low. Filtering by the `local` tag confirms that this increase is due to developers no longer excluding these tests from their local builds.

So have you completely solved the flakiness problem? The "Builds with flaky tests" chart shows us that flakiness was substantially reduced - but not completely eliminated - by your fixes.

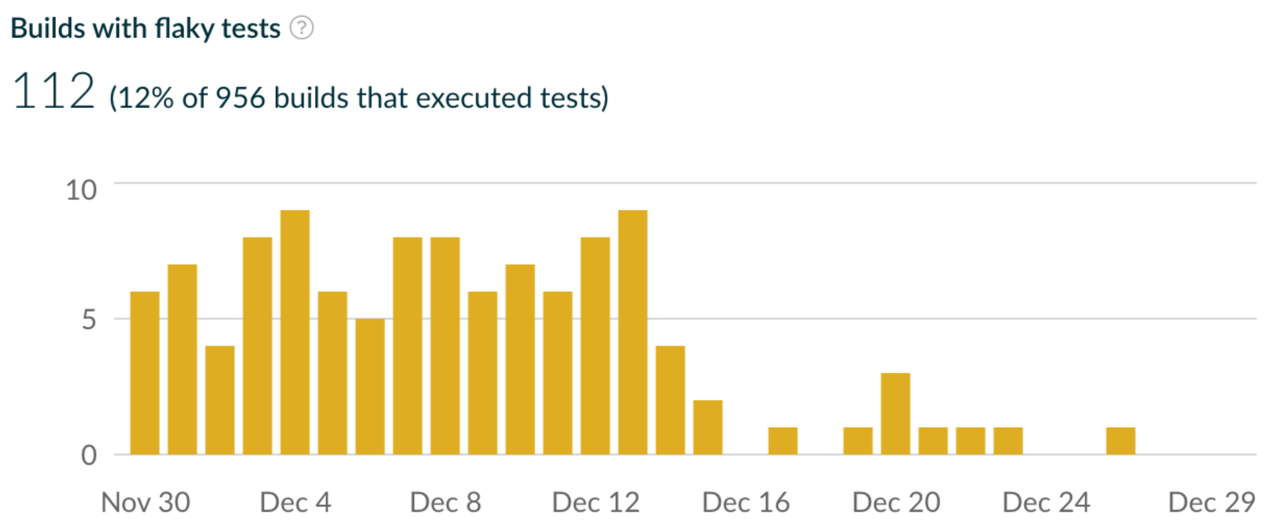


Figure 31. The builds with flaky tests chart showing the entire time range, including both local and CI builds.

Even if you had fixed every single flaky test, more will eventually appear as new development brings new complexity. But when that happens these tools & techniques can be applied again to regain control and keep your tests healthy.

Configuring Your Build to Detect Flaky Tests

Flaky test detection in Gradle Enterprise is based on the same test emitting "passed" and "failed" results in a single build.

Users of the Maven build tool can get Gradle Enterprise to recognize flaky tests by configuring the Surefire plugin with a `rerunFailingTestsCount` value of at least 1. See: https://maven.apache.org/surefire/maven-surefire-plugin/examples/rerun-failing-tests.html

Users of the Gradle build tool can get Gradle Enterprise to recognize flaky tests by adding our test retry plugin and setting a `maxRetries` value of at least 1. See: https://github.com/gradle/test-retry-gradle-plugin

Automatically retrying failed tests is a powerful tool for detecting & managing flaky tests, but it can also be used to bury your head in the sand. Set 1000 retries on every test failure and watch all the flaky failures vanish - along with hours and hours of your time. Maintain vigilance and your tests will provide the most signal for the least frustration for years to come.

To learn more about how to tackle flaky tests, see this chapter from our book Developer Productivity Engineering: https://go.gradle.com/dpe-flaky-tests

Next steps: Where to go from here

The following free resources are available from Gradle, Inc. to help you learn about technology to implement your developer productivity engineering effort.

- Build Cache Deep Dive - Get hands-on training on how to tackle build performance issues with the Gradle build cache.
- Getting started with free build scans for Gradle and Maven - A build scan is a shareable record of a build that provides insights into what happened and why. You can create a build scan at scans.gradle.com for the Gradle and Maven build tools for free.
- 30-day Gradle Enterprise Trial - larger enterprise software development teams can work closely with our technical team to dramatically improve build/test/CI over the course of this 30-day trial. Request a trial at https://gradle.com/enterprise/trial/

Chapter 9. About the Author

Sam Snyder is a Senior Software Engineer at Gradle Inc. Previously, he was a Senior Software Engineer at Tableau where among other things he led a focus on improving developer productivity and build infrastructure. Before that he was an engineer at Microsoft and graduated with a BS in Computer Engineering from Ohio State.

Chapter 10. About Gradle, Inc.

At Gradle Inc., our mission is to make enterprise software development productive and joyful so that both companies and developers can innovate at their full potential.

We are the company behind both the popular open-source Gradle Build Tool used by millions of developers and Gradle Enterprise, a developer productivity application that immediately accelerates and fixes the essential software processes - build, test, and CI.

We've spent the last decade working with hundreds of the world's leading software development teams to accelerate and improve their software development processes and create a great developer experience.

To learn more contact us at gradle.com.