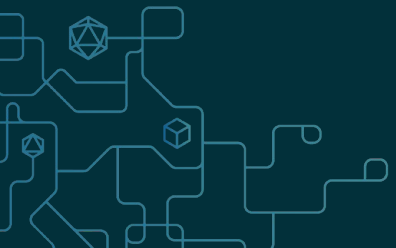
DevProd Engineering with Gradle Enterprise

Jenn Strater, Developer Advocate at Gradle

Gary Hale, Senior Software Engineer at Gradle

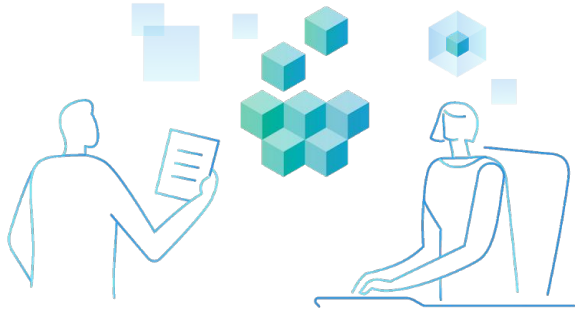
Table of Contents



Lab Setup

https://gradl.es/mdpw

1. Download the zip file & extract
2. Follow instructions in the README.txt



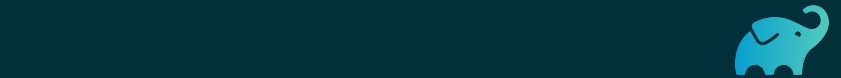
The case for developer productivity engineering

Software development is a creative process

- The creativity of a developer is similar to the creativity of a scientist
- Creativity that requires a dialogue with the toolchain
- The quality of the creative flow depends on the quality of the dialogue with the toolchain.
- Ideally you get answers instantaneously and the answers are always reliable

When you wrote your first code and it would have taken 1 minute waiting time to change the color of the circle your were drawing, would you have continued with coding?

Enterprise Software Developers face a new set of challenges

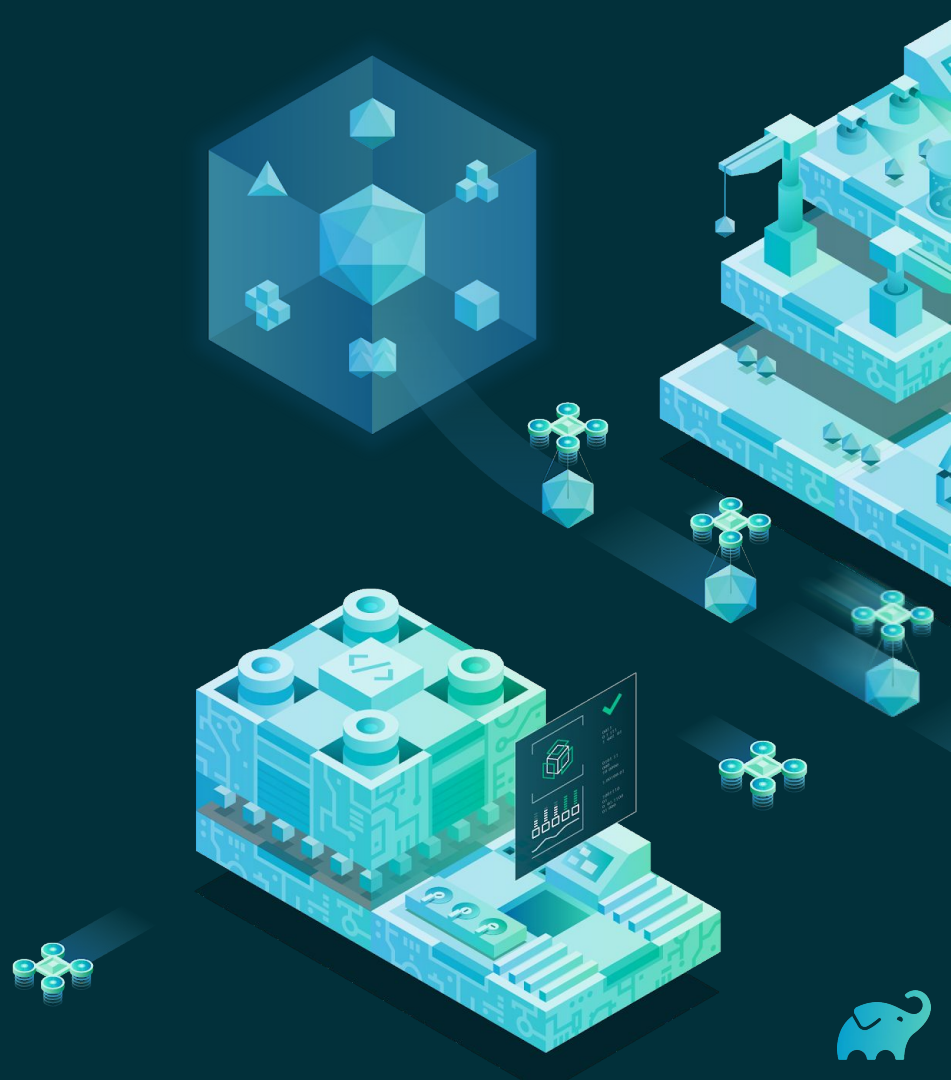


Enterprise Software development requires collaboration

- Collaboration is with the business experts/customer
- Code is an interpretation of a business idea
- Collaboration requires iterations around the correctness of the interpretation
- The effectiveness of the collaboration depends on how quickly you can iterate
- Agile software development requires such an effectiveness

Quality of Creative Flow
+ Collaborative Effectiveness

Team Productivity



Enterprise Software development requires complex machinery

- The developer toolchain is complex machinery with complex inputs
- It affects
  - Speed of iterations
  - Feedback cycles
  - Reliability of feedback
- Toolchain efficiency is a key enabler of Creative Flow and Collaborative Effectiveness

Project success negatively impacts toolchain efficiency

In successful projects all those metrics are growing, often exponentially:

- Lines of Code
- No. of developers
- No. of repositories
- No. of dependencies
- Diversity of tech stack

Result:

- Toolchain efficiency severely degrades if unmanaged
- The return of growing the team becomes more and more marginal

We need Developer Productivity Engineering

- A culture where the whole organization commits to an effort to maximize developer productivity.
- A team of experts whose sole focus is on optimizing the effectiveness of the developer toolchain with the objectives to have:
  - High degree of automation
  - Fast feedback cycles
  - Correctness of the feedback
- Priorities and success criteria is primarily based on data that comes from a fully instrumented toolchain.

Teams work far from their true potential

- There is a significant gap between the actual productivity of software development teams and their full potential.
- The gap is growing over the lifetime of a project for most organizations.
- The gap can be significantly reduced with the practice of developer productivity engineering.

Developer Productivity affects Developer Happiness

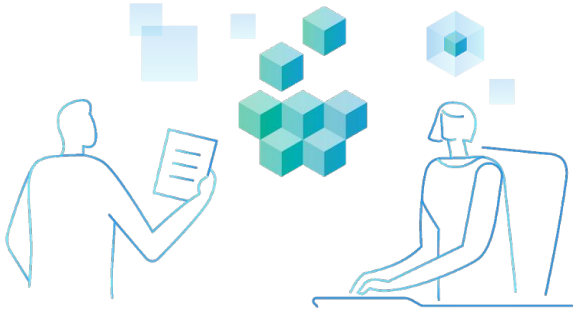
- Most developers want to work in an environment that enables them to work at their full potential.
- Organizations that can not provide such an environment will lose talent.

Low Developer Productivity is blocking business innovations

- Development productivity improvements provide a significant leverage for every dollar invested into software development.
- The productivity gap that comes from not applying the practice of developer productivity engineering is a significant competitive disadvantage.
- Business innovations need to be funneled through software.

Areas of focus for this workshop

- Efficient support for developers
- Fast Feedback cycles
- Proactively improve toolchain reliability



Unblock developers with fewer incidents and better support

Failure Types

- Verification Failures
  - Syntax error detected by compilation
  - Code style violation detected by checkstyle
  - Misbehavior in the code detected by a JUnit test
- Non-Verification Failures
  - Flakey Test
  - Binary repository down
  - Out of memory exception while running the build
- Slow Builds

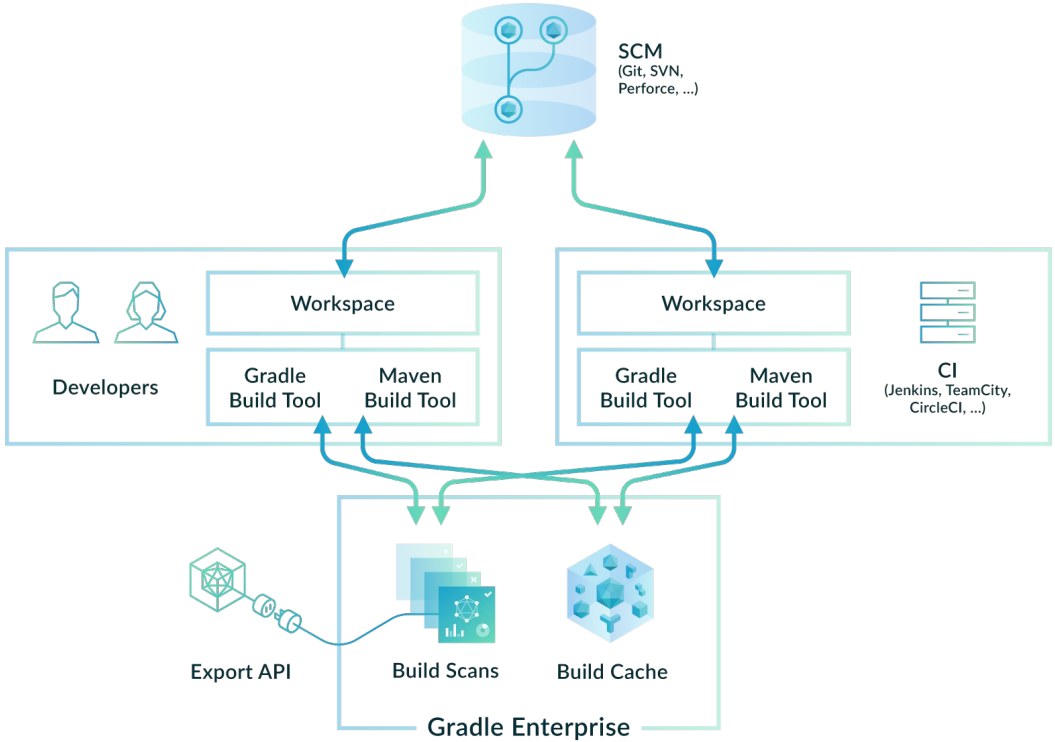
Triaging and prioritization is often difficult

- Non-verification failure masks as verification failure (flakey test)
- Verification failure masks as non-verification failure (snapshot dependency issue)
- Non-verification failure might be caused by bug in a plugin or user mis-configuration
- Many issues are flakey and hard to reproduce
- Not enough information available to help efficiently
  - No data for local build is collected and only limited data for CI builds
  - Most troubleshooting sessions begin with a game of 20 questions
  - Person asking for help often doesn't know what context is important
  - Helpers can burn out on helping
  - Root cause analysis often impossible without the helper reproducing the problem
  - Impact analysis is not data-driven

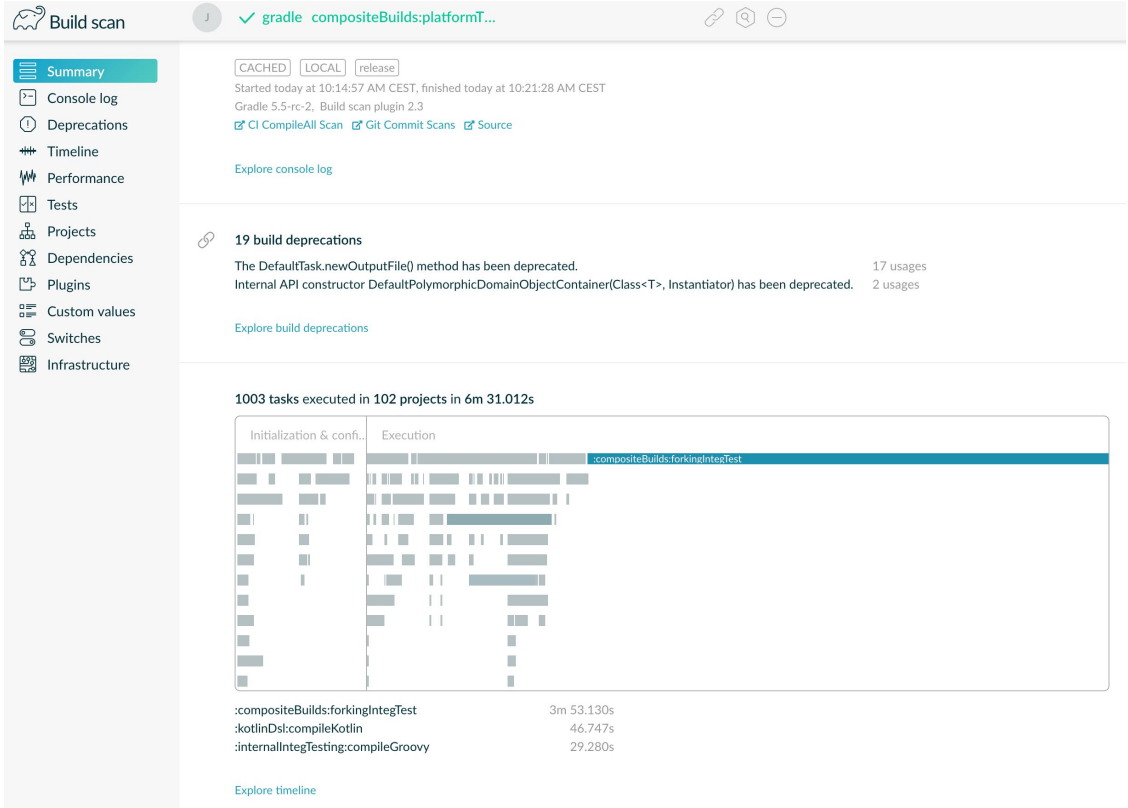
Capture data from every build run (local & CI)

- The only way to effectively diagnose flakey issues
- The data has to be comprehensive to allow for root cause analysis without reproducing
- Having all the data allows for impact analysis

What is Gradle Enterprise?



Gaining insights with build scans



https://enterprise-training.gradle.com/s/rzht6qise5ujg

Training Environment

https://enterprise-training.gradle.com/scans

- Username: attendee
- Password: gradle

Lab 01-troubleshooting-with-build-scans

For **Gradle** see:

labs/01-troubleshooting-with-build-scans/gradle/README.txt

For **Maven** see:

labs/01-troubleshooting-with-build-scans/maven/README.txt

Download: https://gradl.es/mdpw

Username: attendee

Password: gradle

Extending Build Scans

Builds interface with other tools, such as CI, VCS, IDE.

Extend build scans with cross-references to other systems:

- Tags (CI, local, dirty, branch, …)
- Values (Commit ID, Build number, …)
- Links (CI build URL, GitHub URL, …)

Can be used as search criteria

Lab 02-extending-build-scans

For **Gradle** see:

labs/02-extending-build-scans/gradle/README.txt

For **Maven** see:

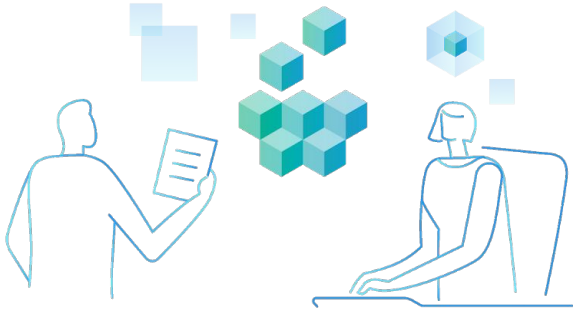
labs/02-extending-build-scans/maven/README.txt

Reducing number of incidents

- Often developers don’t know whether it is user, non-verification or verification failure
  - Self service (Demo comparison)

Reducing number of incidents (II)

- More ways to reduce number of incidents. We will discuss in the next sections
  - Proactive performance optimizations
  - Proactive reliability improvements



Fast feedback cycles are important

Faster builds improve the creative flow

	Team 1	Team 2
No. of Devs	11	6
Build Time	4 mins	1 mins
No. of local builds	850	1010

- The faster the feedback is, the more often devs ask for feedback
- The more often they ask for feedback, the more fine grained they can work.

Builds that take less than 10 mins cause significant waiting time

- The slower the builds are the more developers are idle while waiting for the build to finish.
- The aggregated cost of waiting is surprisingly high even for very fast builds
- Even moderate improvements are worthwhile just from the perspective of reducing idle time

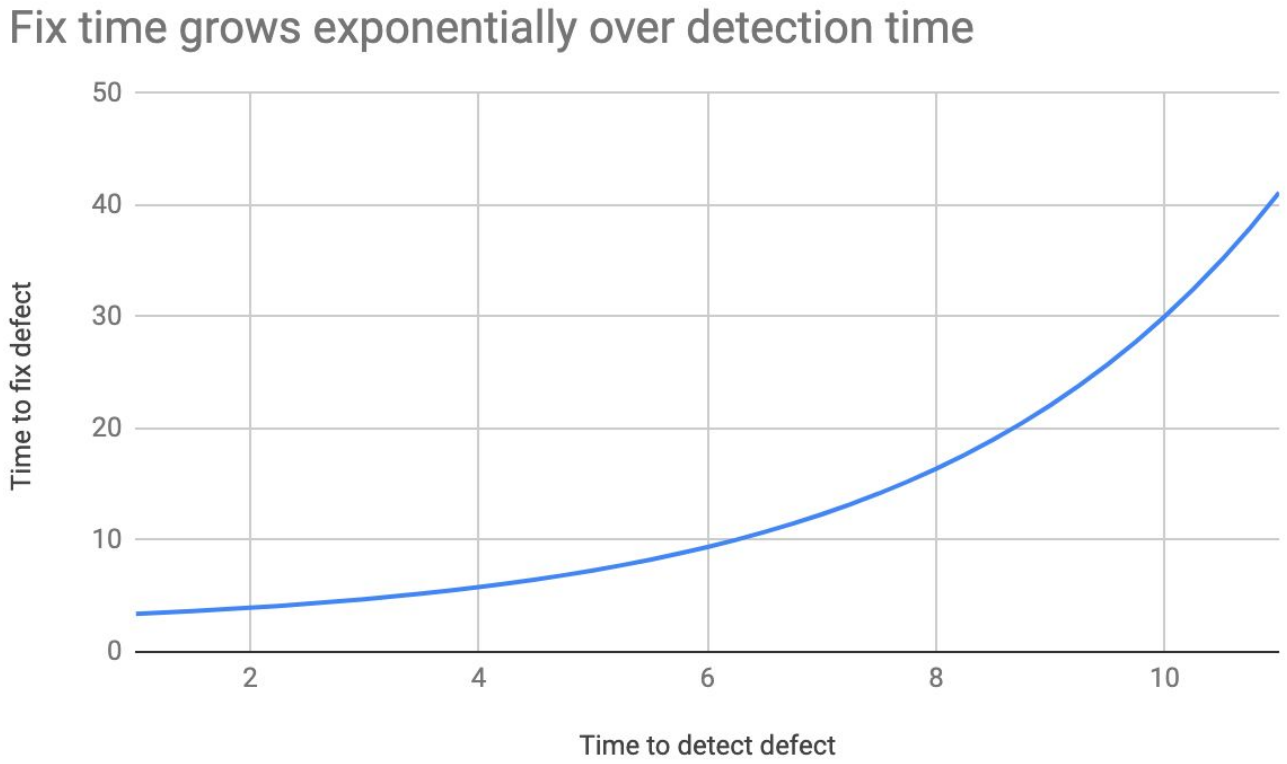
No of devs	Local builds per week	Build Time	Build Time w. GE	Savings per year
6	1010	1 mins	0.6 mins	44 days
100	12000	9 mins	5 mins	5200 days

The longer the build the more context switching

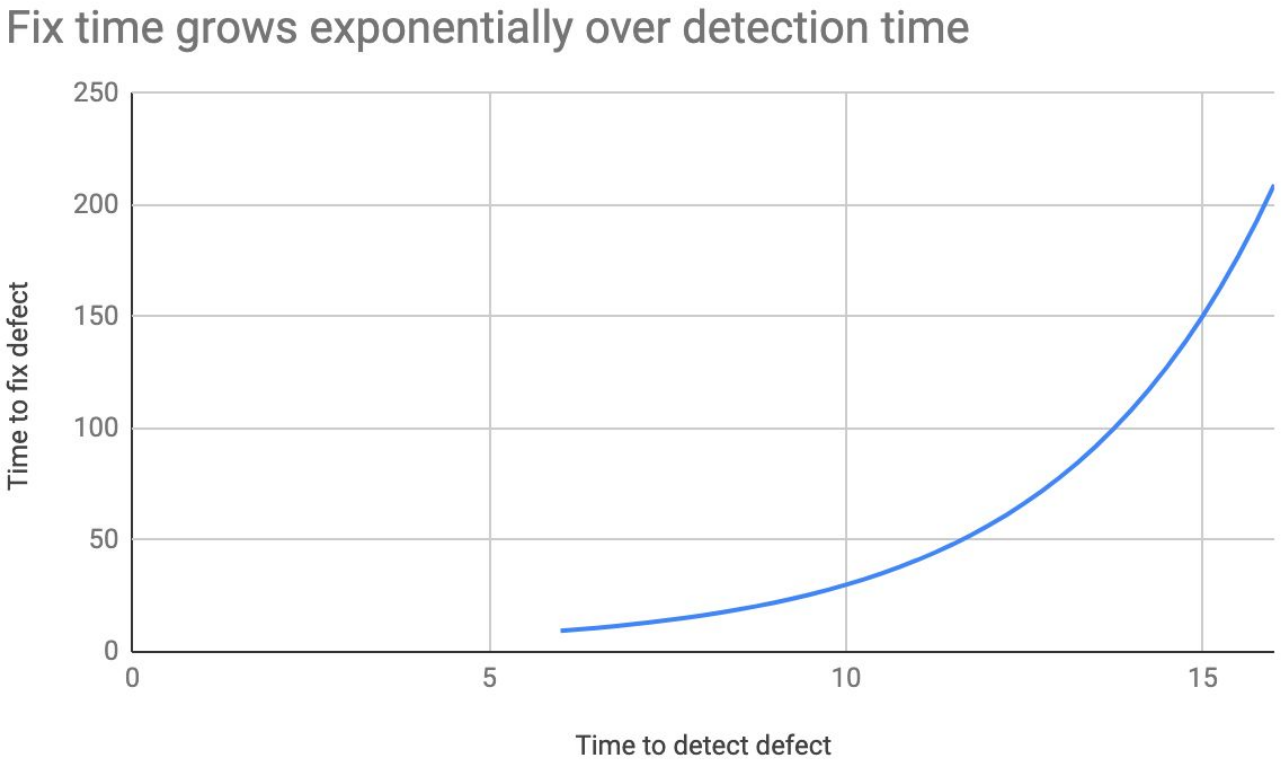
- As build time increases, people switch more and more to do different tasks while the build is running.
- Now the cost of context switching has to be paid, whenever work on the previous task has to be continued after the build finishes:
  - Whenever the build fails
  - Whenever the build was necessary to provide intermediate feedback
- Every context switch cost approximately 10-20 minutes.
- Has to be paid twice:
  - New Task - Fix build of Previous Task and start new build - Go back to new task
- A unreliable toolchain substantially increases this cost

The longer the build the harder to debug

- When local builds take longer the change set of a single push increased
  - The bigger the chage set is the longer it will take in average to debug a failure. This affects local builds as well as the resulting CI builds.
- When CI builds take longer:
  - The number of contributors with changes per CI build increases (e.g. for the master build) and debugging a failure is often significantly harder.
  - The pull request builds takes longer, which increases the likelihood of merge conflicts.



- In the case of a failure, the time fixing the failure is growing exponentially with the time it takes to detect it.



- Because of growing build times, test and builds are pushed to a later point in the life cycle.
- The exponential costs for debugging is increased by that.
- It also increases the change set size as it becomes inconvenient to get feedback.

Vicious circle

- Example merge conflicts
  - Long builds increase change set size
  - The increased change set size increase the repair time of a failed PR build
  - The average time for a *successful* PR build increases significantly
  - The likelihood of merge conflicts increases

Does all this apply to every project?

- Projects with relatively fast builds pay a high waiting time cost.
- Projects with long builds pay both, high waiting time and context switching costs.
- Projects who have a low number of committers are less affected by merge conflict problems.
- What about microservices with many repositories?
  - Building and testing a single repository is relatively fast
  - Producer build no longer detects that consumer is broken
  - Consumer has to figure out why they are broken
  - Triaging is often very time consuming and complex
  - Integration problems are often discovered at a late stage

Build Caching

When the inputs have not changed, the outputs can be reused from a previous run.

Cacheable Task/Goal Executions

Gradle Compile/Maven Compile

- Source Files
- Compile Classpath
- Java version
- Compiler configuration
- etc...

Gradle Test/Maven Surefire

- Source Files
- Runtime Classpath
- Java version
- Compiler configuration
- etc...

Caching is a generic feature and applies to all tasks/goals.
For IO-bound tasks/goals caching has no benefits (e.g. clean, copy).

Caching is effective for multi-module builds

pom.xml

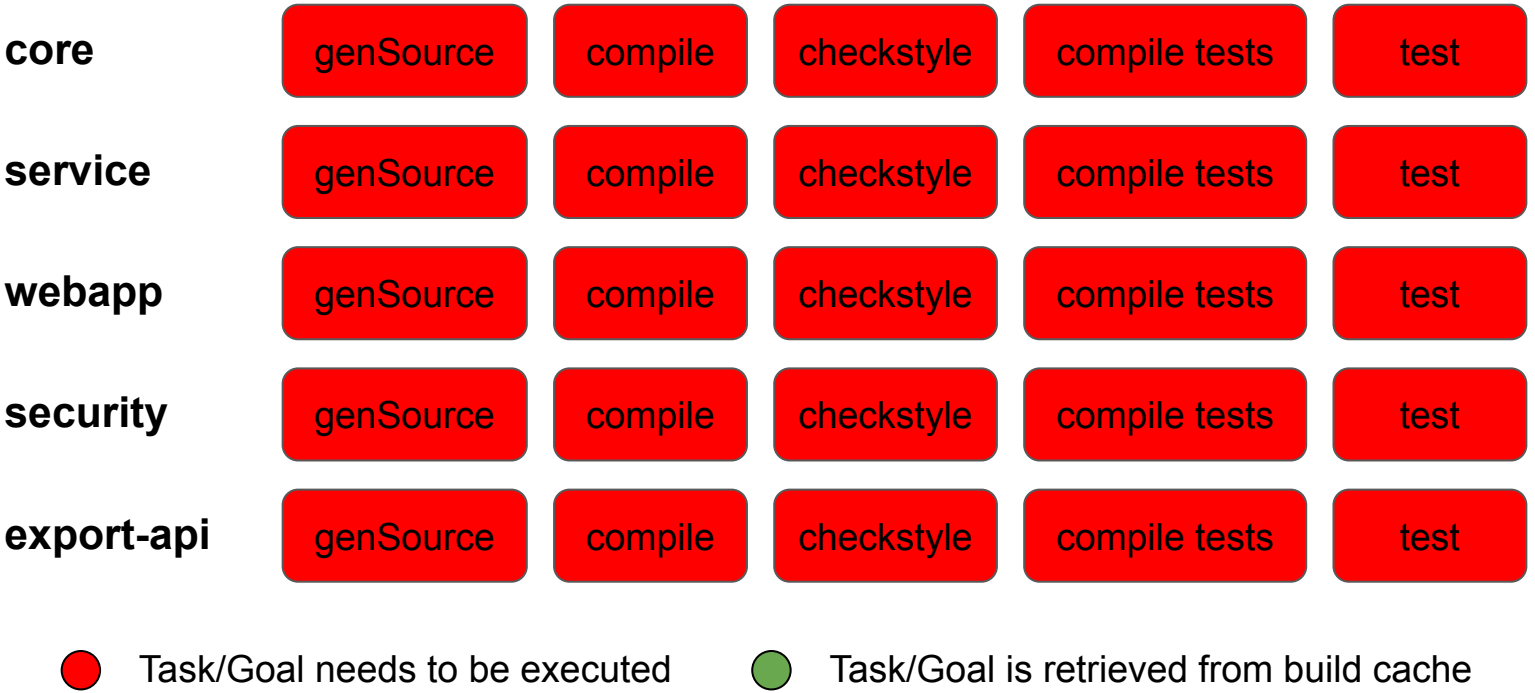
```
<modules>
  <module>core</module>
  <module>service</module>
  <module>webapp</module>
  <module>export-api</module>
  <module>security</module>
</modules>
```

settings.gradle

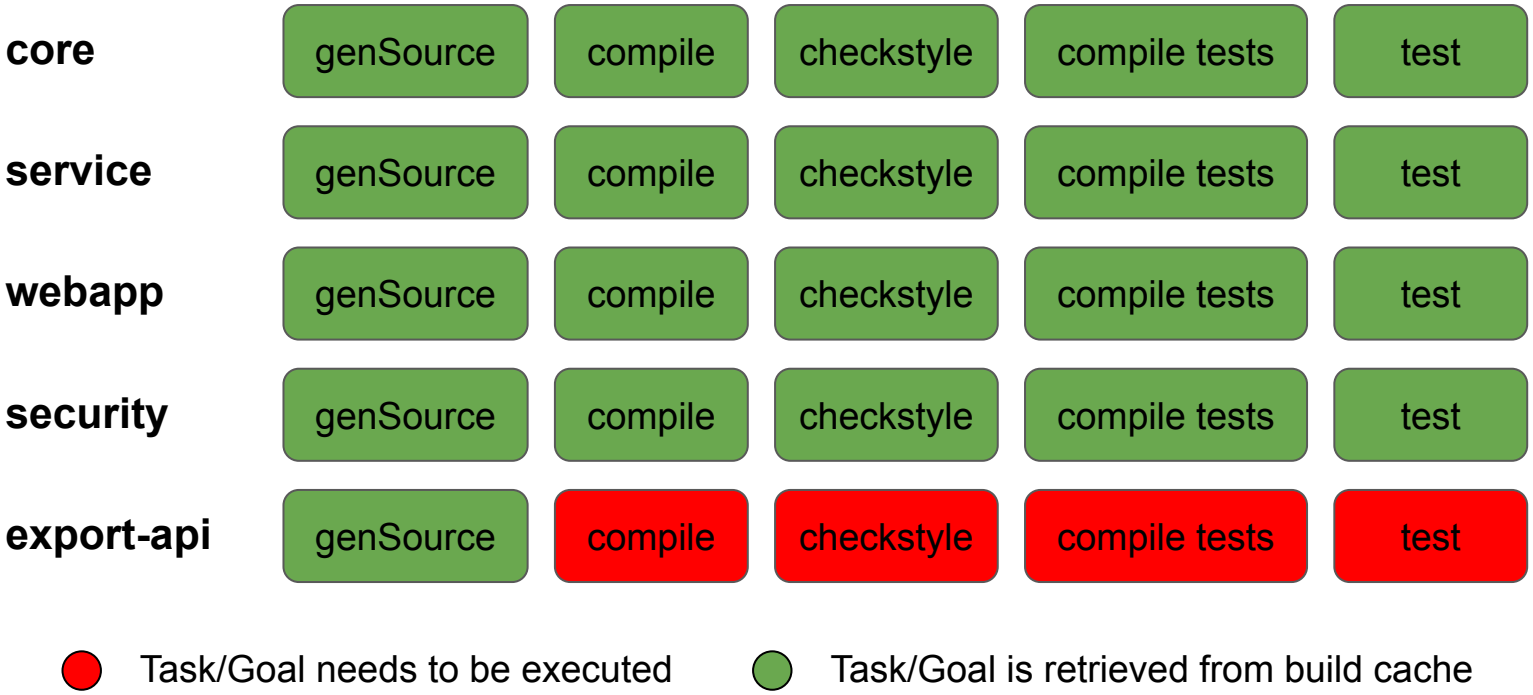
```
include ‘core’
include ‘service’
Include ‘webapp’
Include ‘export-api’
Include ‘security’
```

Builds with a single module will only moderately benefit from the cache

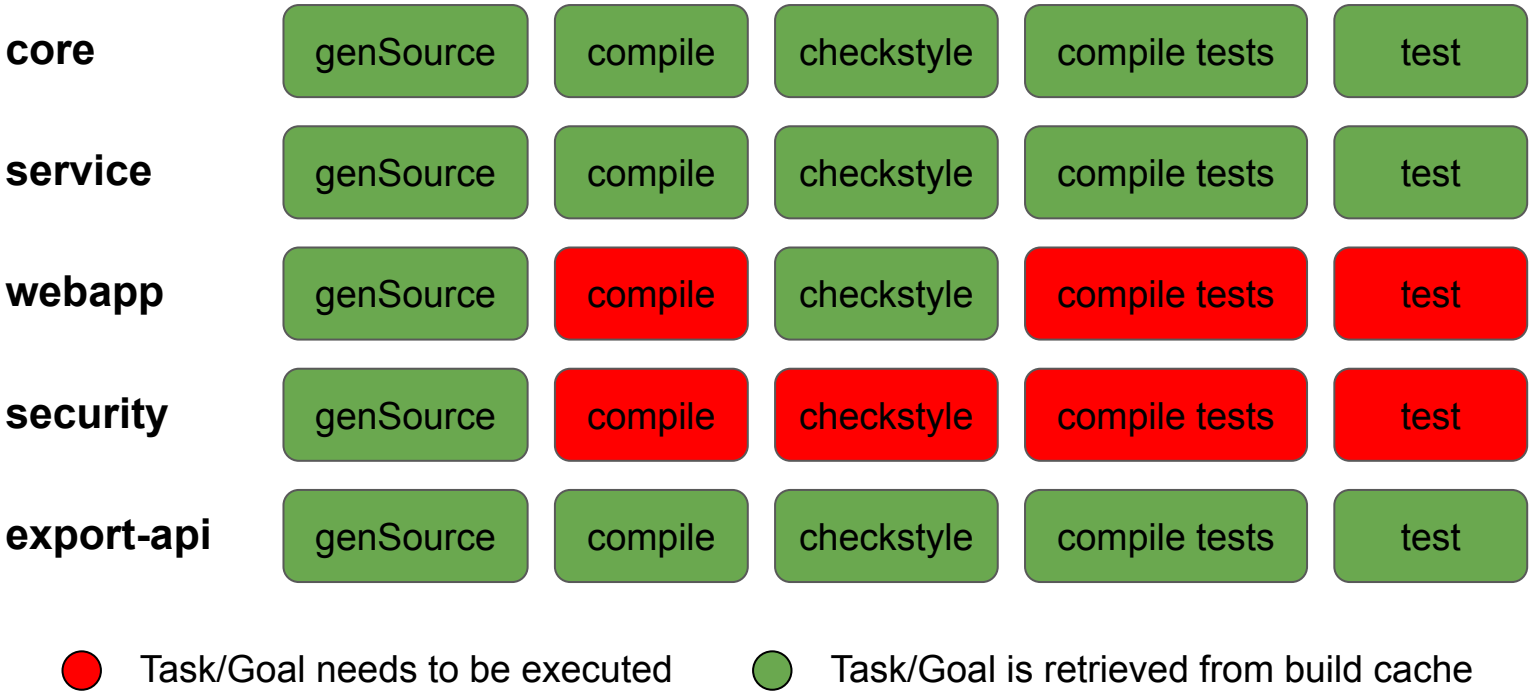
When not using the build cache, with Maven any change will require a full build. For Gradle this is the case when doing clean CI builds and switching between branches locally.



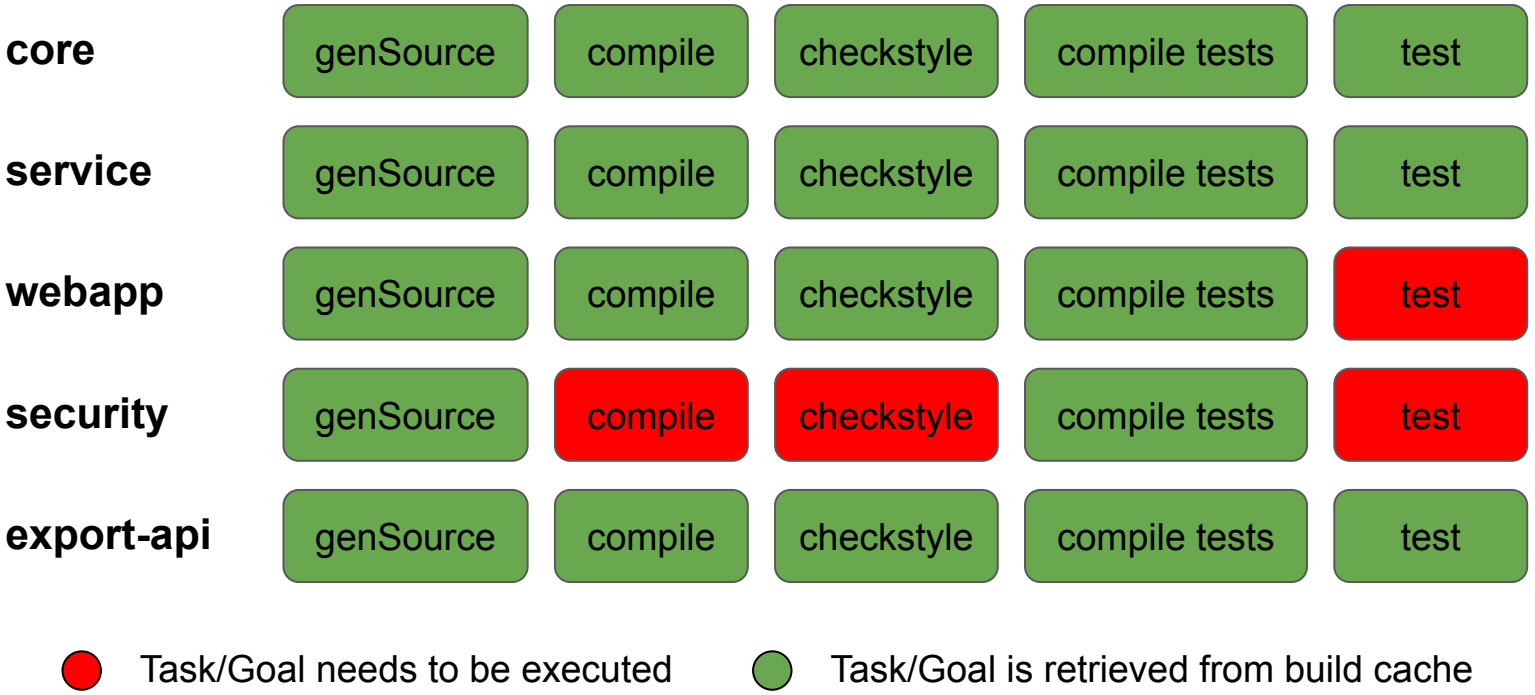
Changing a public method in the export-api module, no other module depends on export-api



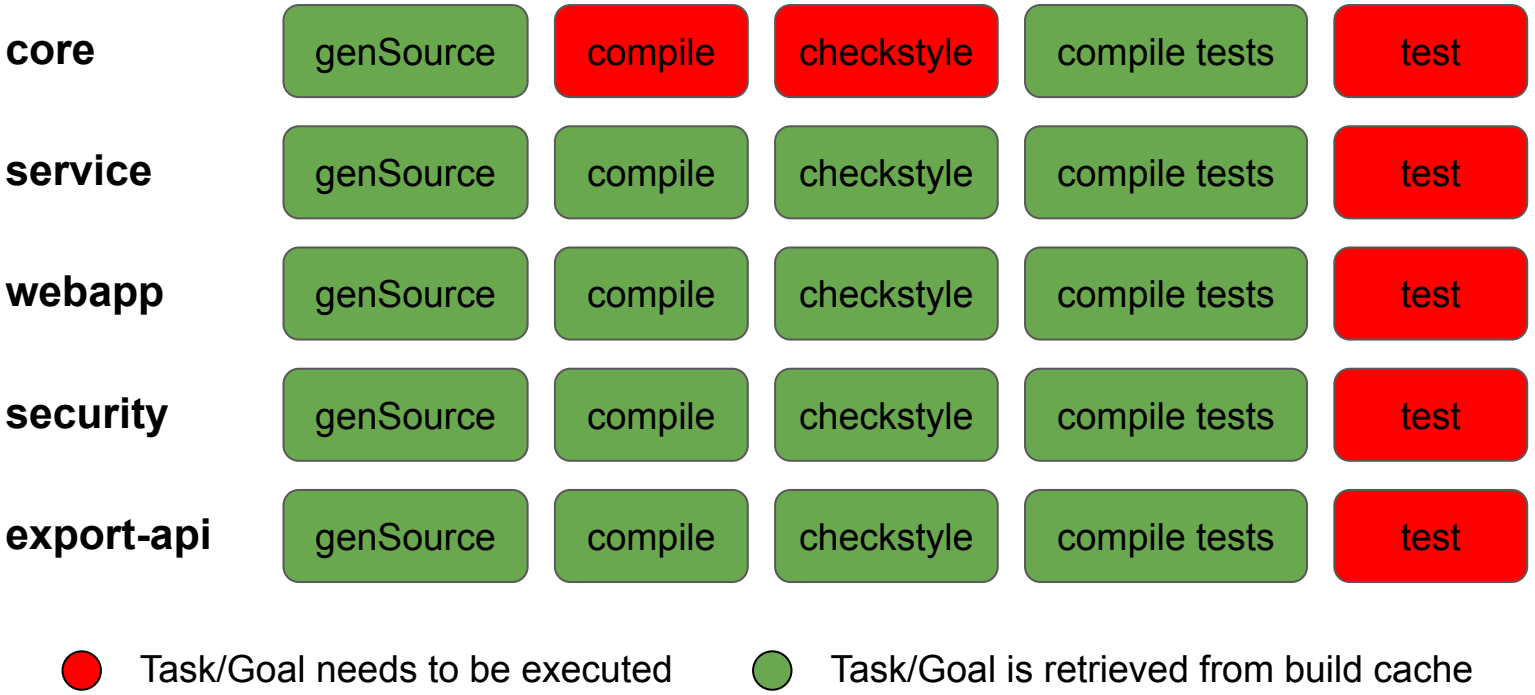
Changing a public method in the security module, webapp depends on security



Changing an implementation detail of a method in the security module



Changing an implementation detail of a method in the security module

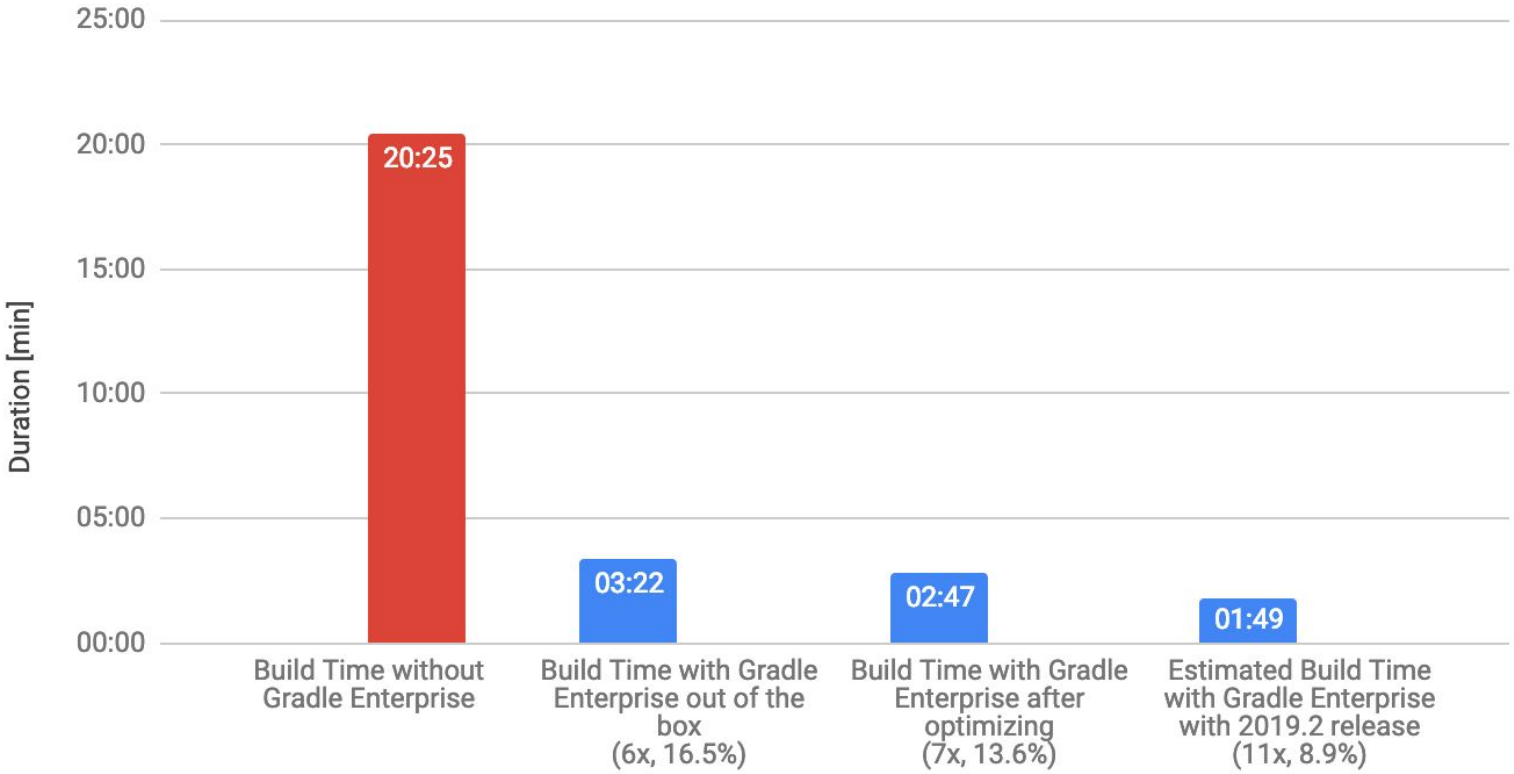


Cache effectiveness

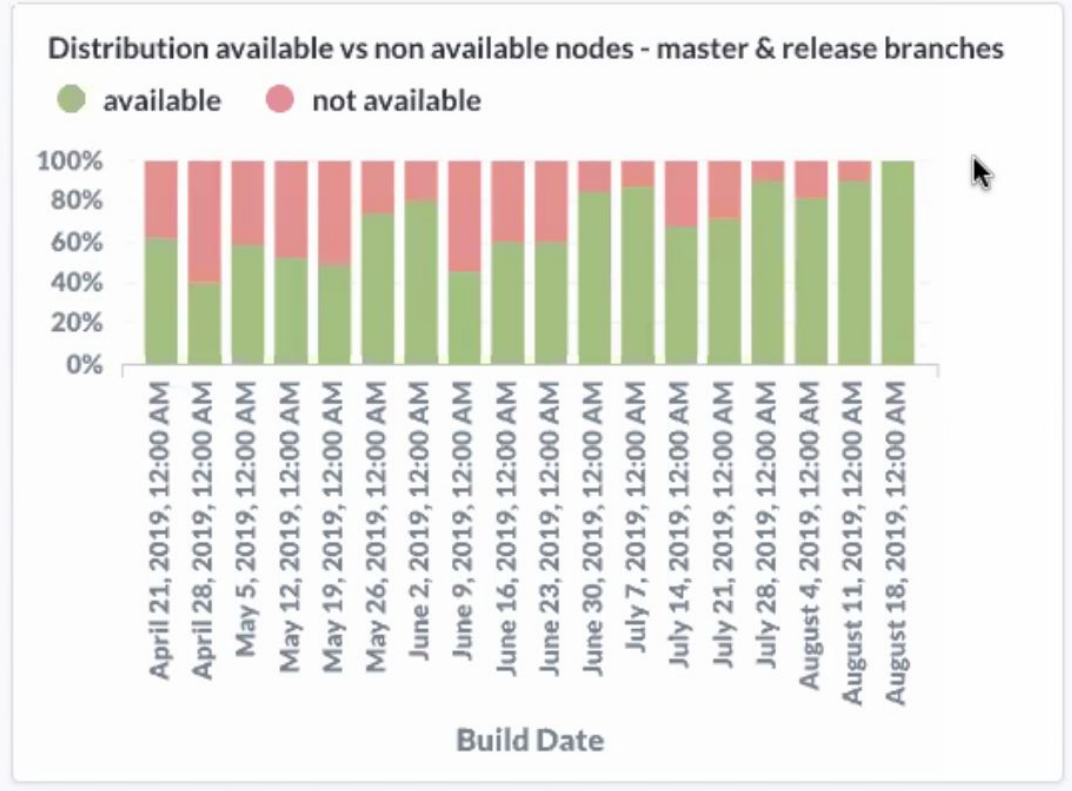
- Even with only a few modules a cache significantly reduces build and test times
- For larger multi-module builds often 50% of modules are leave modules
  - Build times is reduced by approximately 1/n with n being the number of modules
- Checking the inputs and downloading & unpacking items of the cache introduces overhead.
- Overhead is often very small compared to benefits
- Overhead needs to be monitored

Spring Boot build time for compile and unit tests (fully cached)

https://spring.io/projects/spring-boot



Build Cache improves agent availability



Local Build Cache

- Uses a cache directory on your local machine
- Speeds up development for single developer or build agent
- Reuses build results when switching branches locally

Lab 3 - Using the Local Build Cache

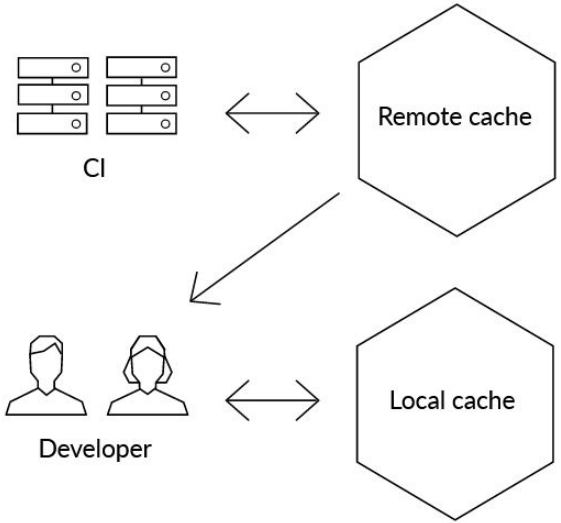
For **Gradle** see:

labs/03-using-the-local-build-cache/gradle/README.txt

For **Maven** see:

labs/03-using-the-local-build-cache/maven/README.txt

Remote Build Cache



- Shared among different machines
- Speeds up development for the whole team
- Reuses build results among CI agents/jobs and individual developers

Lab 04-remote-cache

For **Gradle** see:

labs/04-remote-cache/gradle/README.txt

For **Maven** see:

labs/04-remote-cache/maven/README.txt

Credentials:

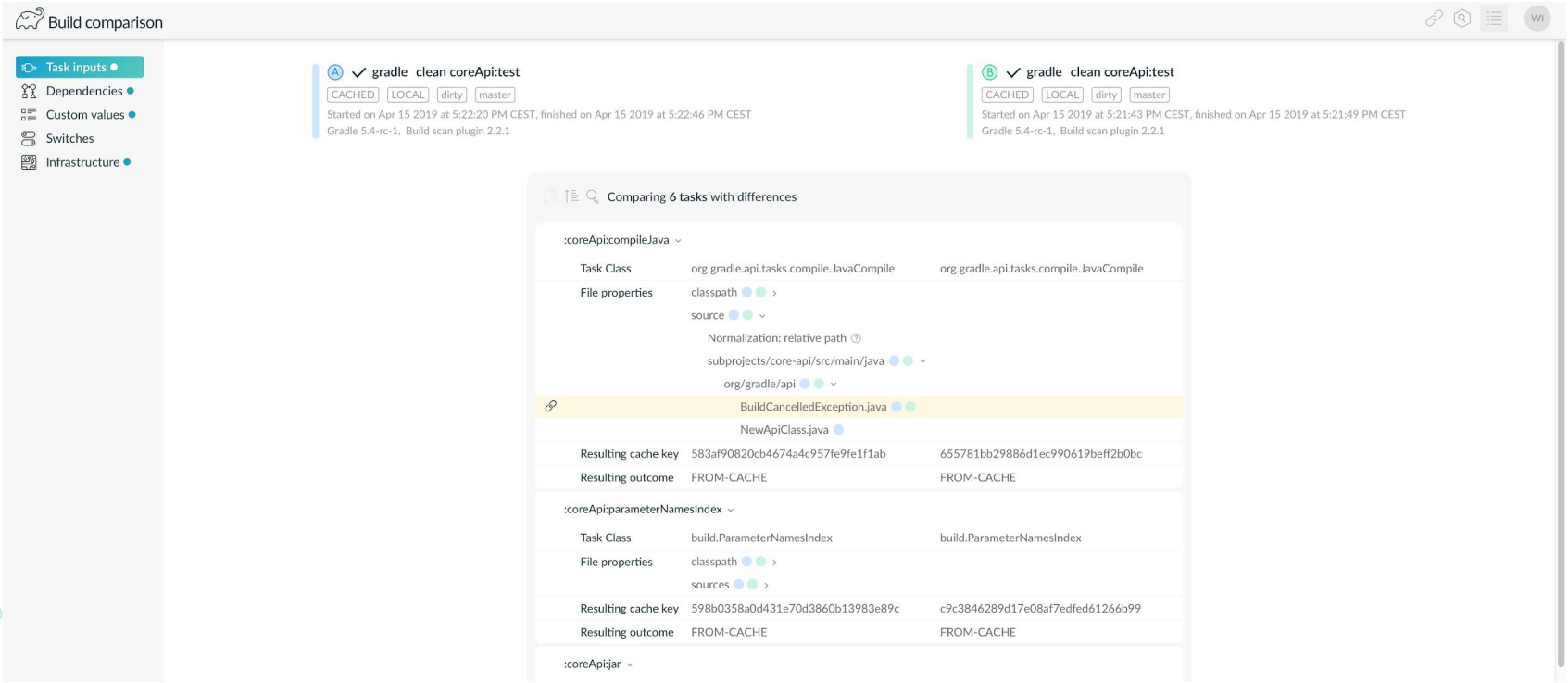
Username: attendee

Password: london99

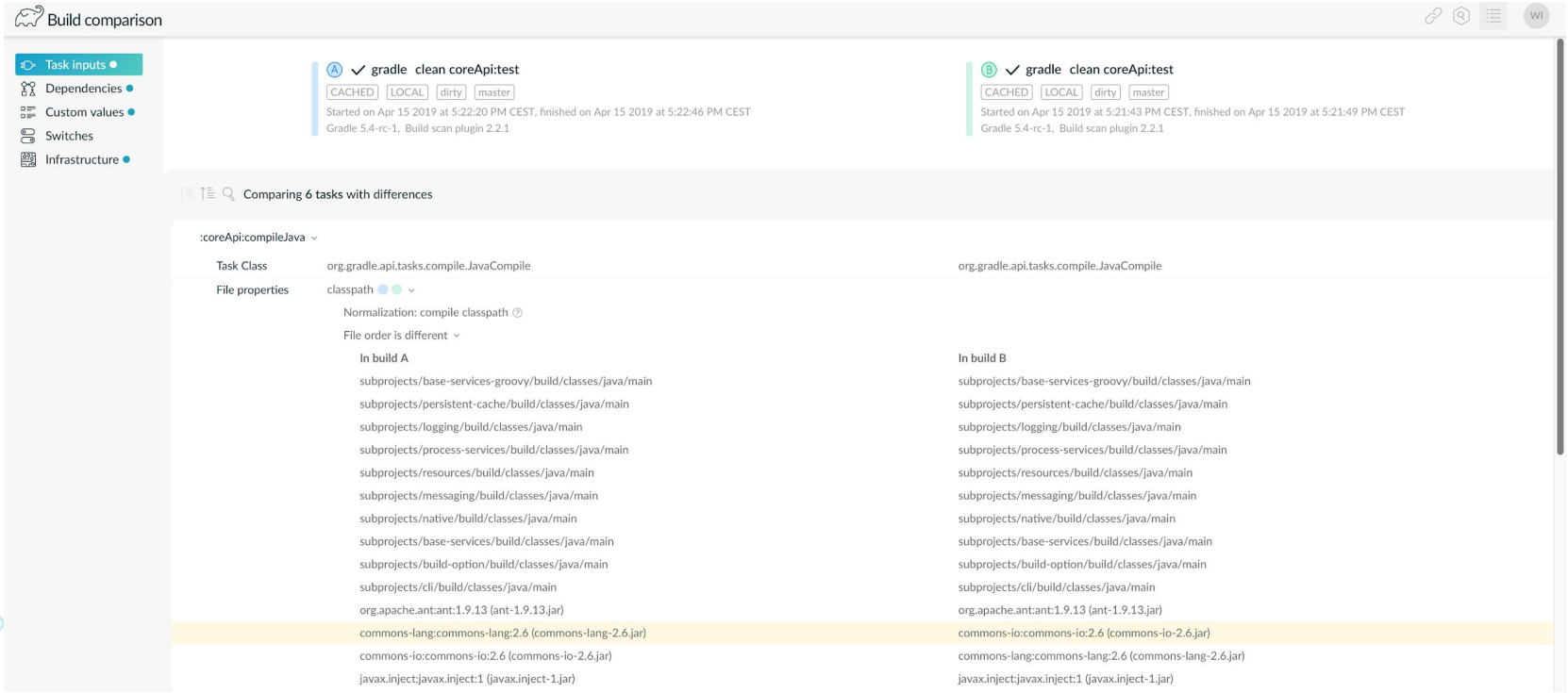
Input Volatility

- Inputs need to be stable and portable
- Common problems:
  - Timestamps
  - Absolute file paths
  - Non-deterministic ordering

Identifying differences in inputs



Debugging Volatile Inputs



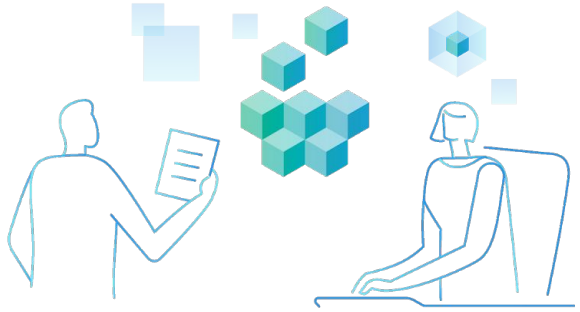
Lab 5 - Build Comparison

For **Gradle** see:

labs/05-build-comparison/gradle/README.txt

For **Maven** see:

labs/05-build-comparison/maven/README.txt



Why data is essential to keep builds fast

Performance regressions are easily introduced

- Infrastructure changes
  - Binary management
  - Caching
  - CI agents
- New annotation processors or versions of annotation processors
- Build logic configurations settings
  - Build tool version and plugins
  - Compiler settings
  - Memory settings
- Code refactoring
- New office locations

What happens today with most regressions

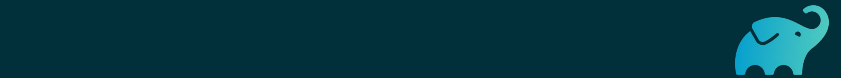
- Unnoticed
- Noticed but unreported
- Reported but not addressed
  - Root cause is hard to detect (especially with flakey issues)
  - Overall impact and priority can not be determined
- Escalated after they have caused a lot of pain
  - Problem gets fixed after it has wasted a lot of time and caused a lot of frustration amongst developers.
- Result: The average build time is much higher than necessary and continuously increasing.

Capture data for all builds

- Significant regressions can be detected immediately
  - With the available data, the root cause can often be easily detected
  - The problem can be fixed before it causes a lot of harm and escalation
- Having data from all builds allows for data-based prioritization of performance related improvements based on quantifiable impact.
- Performance related incidents can be supported much better
- Build scans are providing this capability
- Fewer incidents and builds that are getting continuously faster

Demo **Performance Analytics**

The importance of reliability



Flaky builds and tests are **maddening**

We've all been there

Things are going well with your change then you run verifications and something apparently unrelated breaks

- The break is cryptic, it's not your area
- You don't even know whose area it is
- Now your day is spent finding help instead of working on your thing

Complaint Driven Prioritization

Without data you don’t know which issues are important

- Not everyone speaks up
- Loud complaints may not be representative

Demo **Failures Dashboard**

Flaky Tests

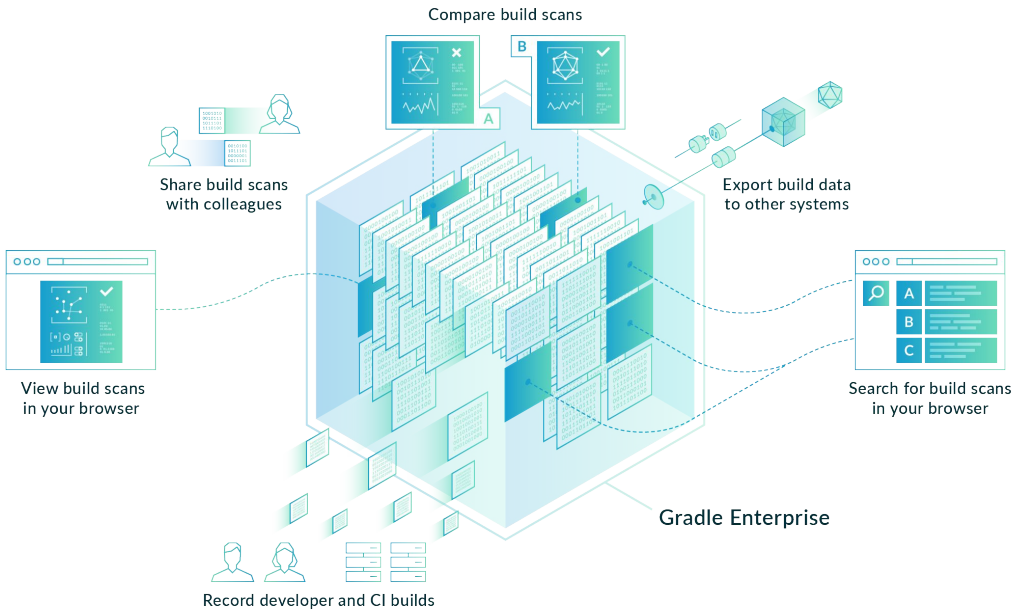
Flaky tests are a problem for everyone

We’re currently working on a solution

Plan to release this in Q4/2019

Interested in your ideas & thoughts

Gradle Enterprise is a data platform



- **Collect team data** - all the data about every build across the team creates unique dataset and insights.
- **Build Performance Management** - only with representative, actionable data will builds get and stay fast and reliable.
- **Debugging acceleration** - only with comprehensive, deep data is it possible to quickly discover the root cause for build failures.

Lab 06 - Live Dashboard

See:

labs/06-live-dashboard/README.txt

Username: attendee

Password: gradle

Next Steps

Trial Process Pre-Install

1. Installation
    a. Provisioning of license key
    b. Onboarding to our support system (ZenDesk and Slack) that give you direct access to our engineers.
    c. Installation usually will take 30-60 minutes.
2. Connect your local and CI builds with Gradle Enterprise
    a. Connecting your builds is easy. We will help you with any special configuration you need.
    b. Build will never fail because of Gradle Enterprise, even if Gradle Enterprise is down.
    c. The 30-day trial period will start once your builds are connected.

Trial Process Post-Install

1. Weekly meetings with Gradle engineering and Gradle account manager to analyze the data
    a. Screen sharing required as we don't have access to your data.
    b. Usually already very insightful after a couple of days of data to see average build times, failure rates, number of local and CI builds, performance bottlenecks, …
    c. We will identify cache inefficiencies and if necessary work with you on your build to resolve them.
2. Usually after 2-4 weeks we have enough data to make before and after case.
    a. We will work with you on a ROI report based on your data that calculates the quantifiable savings you will get from Gradle Enterprise.
3. After a purchase you can continue to use the trial instance including its data.
4. Your time investment
    a. Gradle Enterprise usually does not require much maintenance if any once installed.
    b. We expect you to dedicate a couple of hours of your time per week to analyze the data and, if required, work with us on your build to improve cache efficiency.

Resources

- Gradle Enterprise docs and tutorials: https://docs.gradle.com
- Build Scan Plugin User Manual: https://docs.gradle.com/build-scan-plugin
- Maven Extension User Manual: https://docs.gradle.com/enterprise/maven-extension
- Export API Manual: https://docs.gradle.com/enterprise/export-api
- Try out build scans for Maven and Gradle for free: https://scans.gradle.com

Thank you!