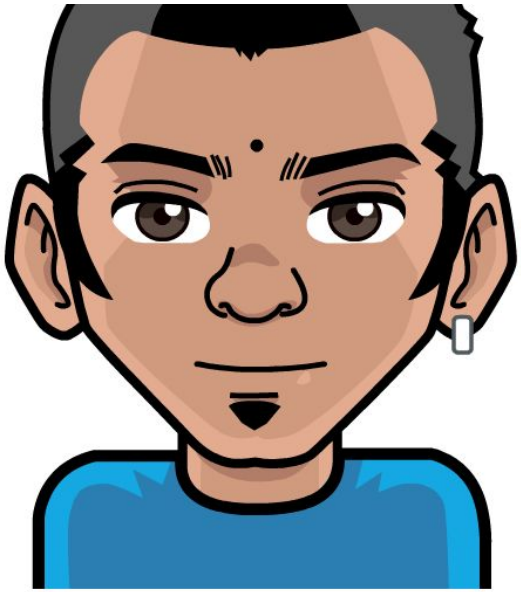
Introducing **Developer Productivity Engineering** Workshop

Gradle

Raju Gandhi



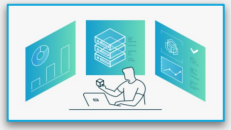
- Developer Advocate at Gradle
- DevOps enthusiast
- Based in Columbus, OH, USA.

@looselytyped

What if you could...

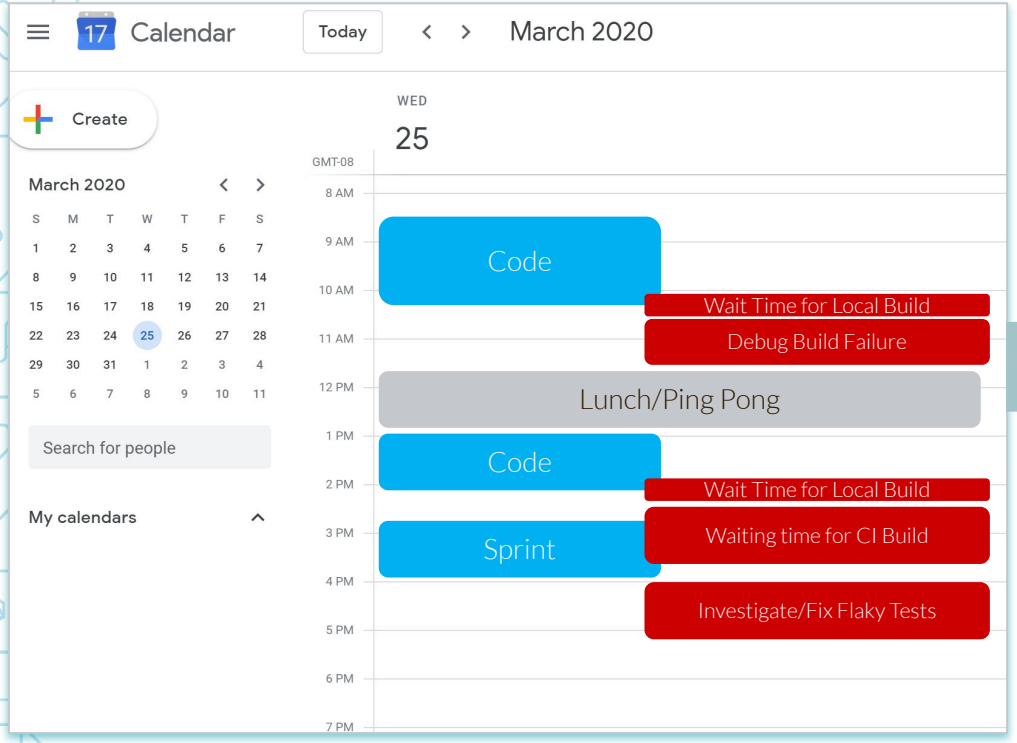
Cut average build & test time in half instantly?

Decrease build and test failure time to resolution by over two-thirds?



Get 100% visibility to data needed to reduce build, test and CI failures and avoid performance regressions?

A Day in the Life of Your Developers



Your Opportunity

Give developers back 1 hour per day in lost productivity

Lab Setup

https://gradl.es/mdpw

- Download the zip file and extract to any location
- Follow the instructions in the README.txt file

Introducing Developer Productivity Engineering

DPE is a new software development discipline that uses acceleration technologies to speed up the software build and test process and data analytics to make troubleshooting more efficient. The aim is to achieve:

- Faster feedback cycles
- More reliable and actionable data
- A highly satisfying developer experience

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.

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.

Agenda

+ The Cost of Inaction & Next Steps

DPE Addresses 5 Major Pain Points

WAITING FOR BUILDS & TESTS TO COMPLETE

INEFFICIENT TROUBLE-SHOOTING

FLAKY TESTS & OTHER AVOIDABLE FAILURES

NO OBSERVABILITY

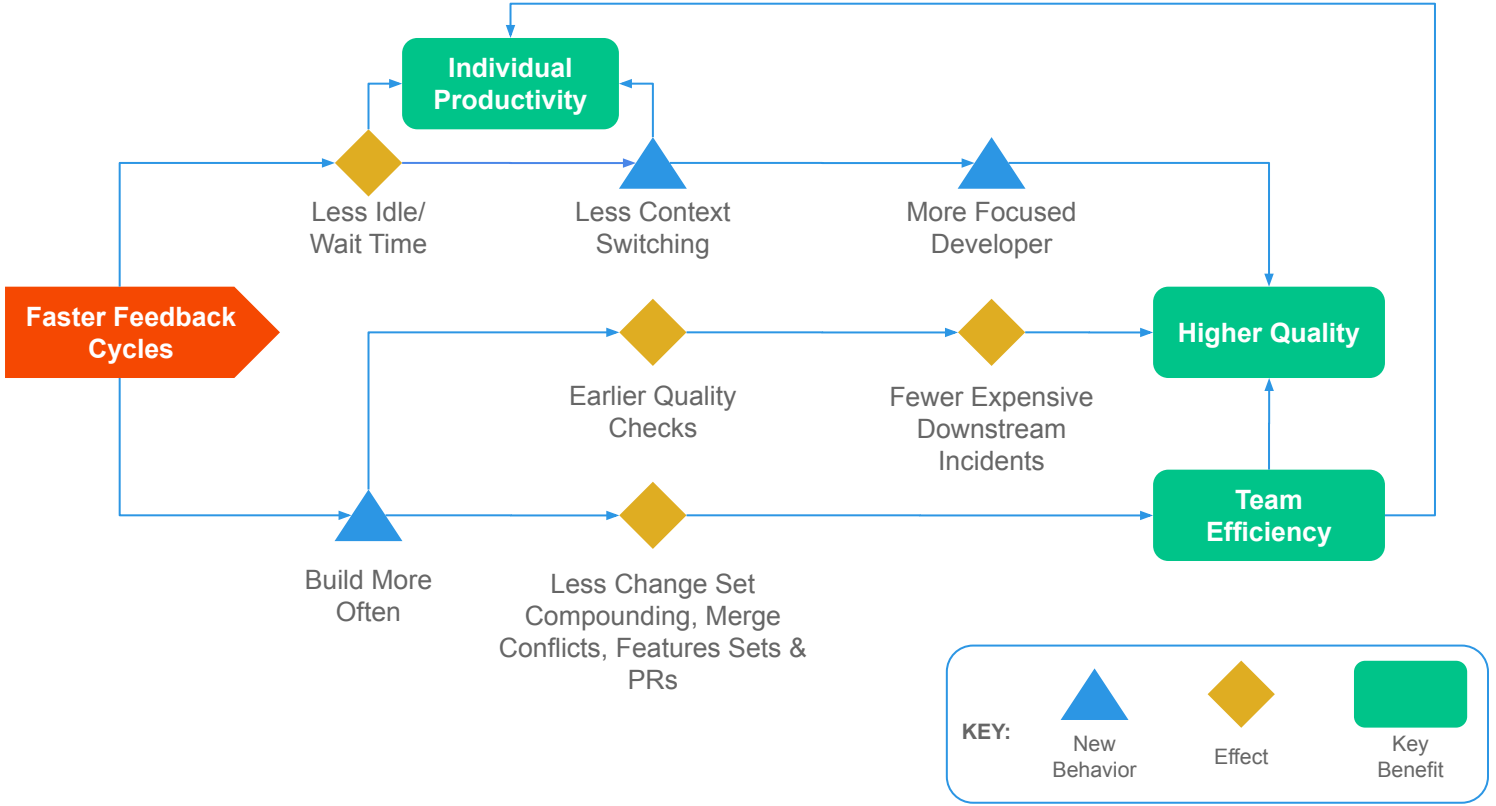
INEFFICIENT USE OF CI RESOURCES

Pain Point #1

Waiting for Builds & Tests to Complete

Pain Point #1	Benefit	Solution
Waiting for Builds & Tests to Complete	Faster Feedback Cycles	Acceleration Technologies

Why are fast feedback cycles so important?
Let's break it down



Flow

> being completely involved in an activity for its own sake. The ego falls away. Time flies. Every action, movement, and thought follows inevitably from the previous one, like playing jazz. Your whole being is involved, and you're using your skills to the utmost.
>
> - Mihály Csíkszentmihályi

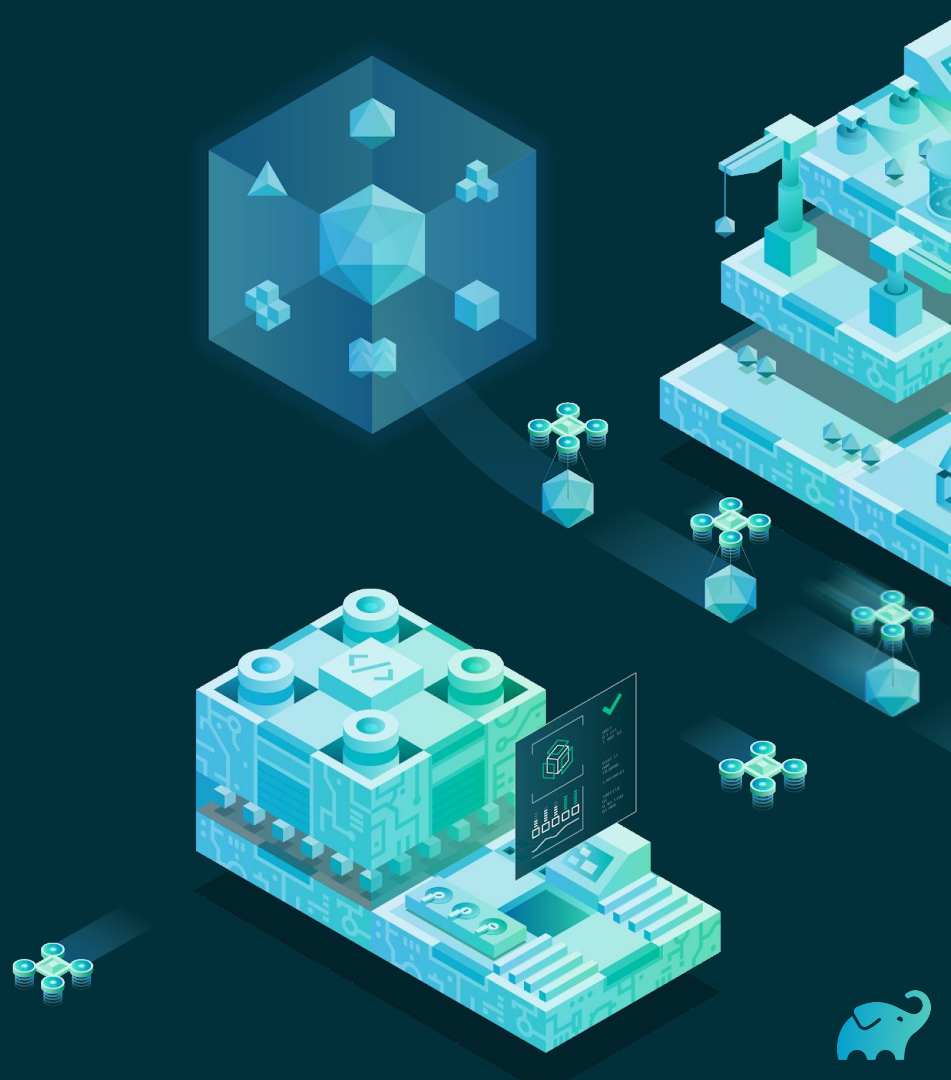
Deep Work

> Professional activity performed in a state of distraction-free concentration that push your cognitive capabilities to their limit. These efforts create new value, improve your skill, and are hard to replicate.

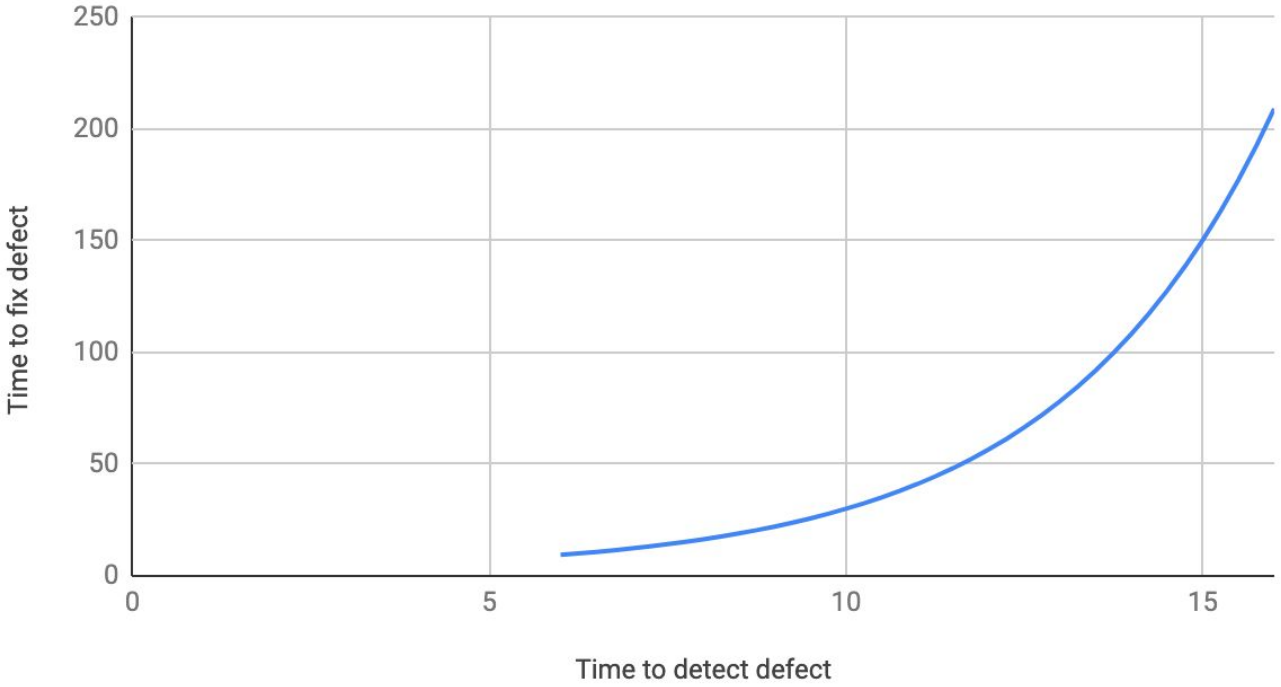
- Cal Newport

Quality of Creative Flow
+ Collaborative Effectiveness

Team Productivity

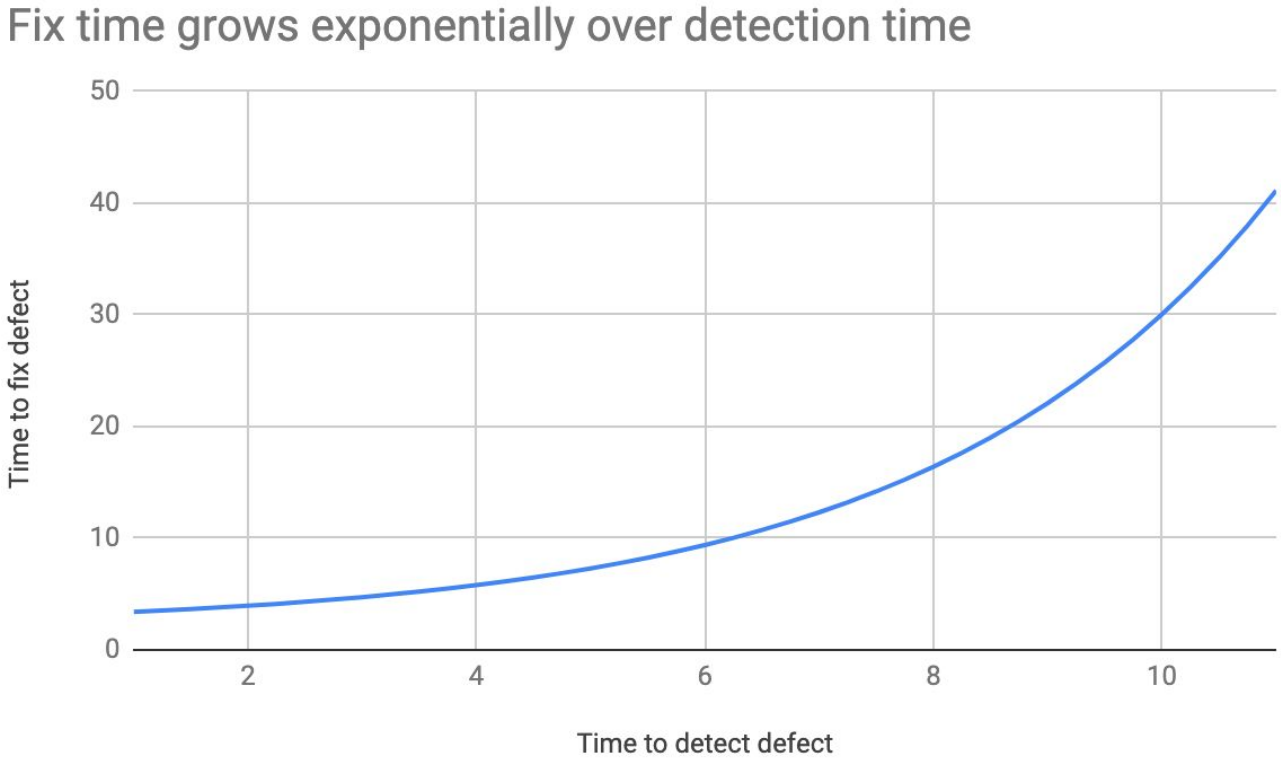


Fix time grows exponentially over detection time



- 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.


Feedback should be *instant*!



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

The faster the build, the more productive you are

- 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 faster the build, the more you stay focused

- 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

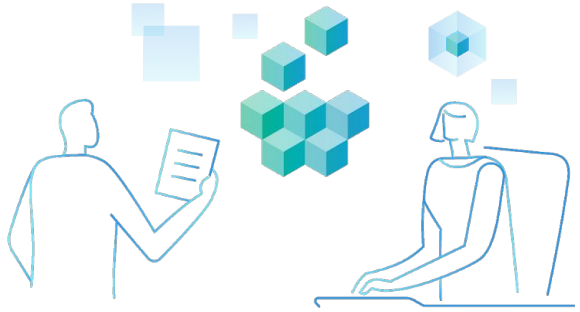
The faster the build, the more you build

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

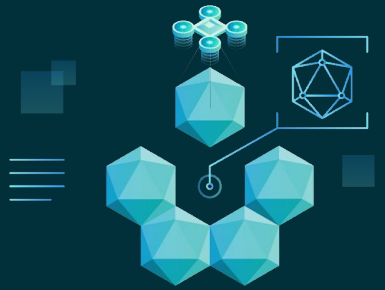
- The faster the feedback is, the more often devs ask for feedback
- The more often they ask for feedback, the more fine grained they can work.

The faster the build, the easier it is to debug

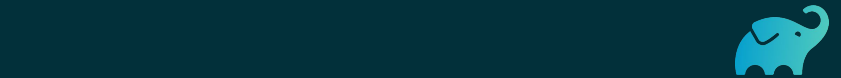
- When local builds take longer the change set of a single push increased
  - The bigger the change set is the longer it will take in average to debug a failure. This affects local builds as well as the resulting CI builds.
- When CI builds take longer:
  - The number of contributors with changes per 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.



Fast feedback cycles are important



Build Cache



Build Caching

- Introduced to the Java world by Gradle in 2017.
- Available for Maven and Gradle
- Complementary to dependency cache
  - A dependency cache, caches binaries that represent different source repositories
  - A build cache accelerates building a single source repository
  - A build cache caches build actions (e.g. Gradle tasks or Maven goals)

Build Caching

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

Cache Key/Value Calculation

The *cacheKey* for Gradle Tasks/Maven Goals is based on the Inputs:

cacheKey(javaCompile) = hash(sourceFiles, jdk version, classpath, compiler args)

The *cacheEntry* contains the output:

cacheEntry[cacheKey(javaCompile)] = fileTree(classFiles)

For more information, see:

https://docs.gradle.org/current/userguide/build_cache.html

Other Cacheable Tasks/Goals Executions

Gradle Test/Maven Surefire inputs

- Test Source Files
- Runtime Classpath
- Java version
- System properties
- etc...

Checkstyle inputs

- Source Files
- Checkstyle version
- Checkstyle Config
- etc...

Caching is generic. It can apply to any task or goal meeting its requirements.
For IO-bound tasks/goals caching has no benefits (e.g. clean, copy).

Caching is **particularly** effective for multi-module builds

pom.xml

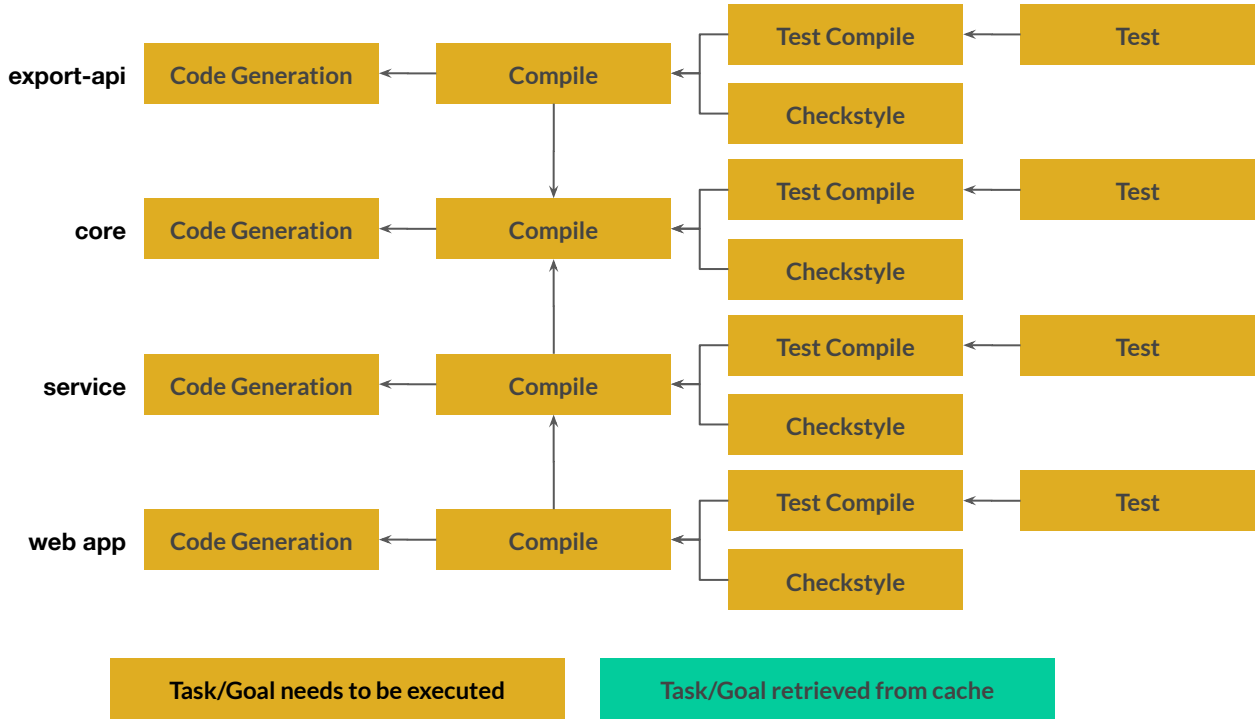
```
<modules>
  <module>core</module>
  <module>service</module>
  <module>webapp</module>
  <module>export-api</module>
</modules>
```

settings.gradle

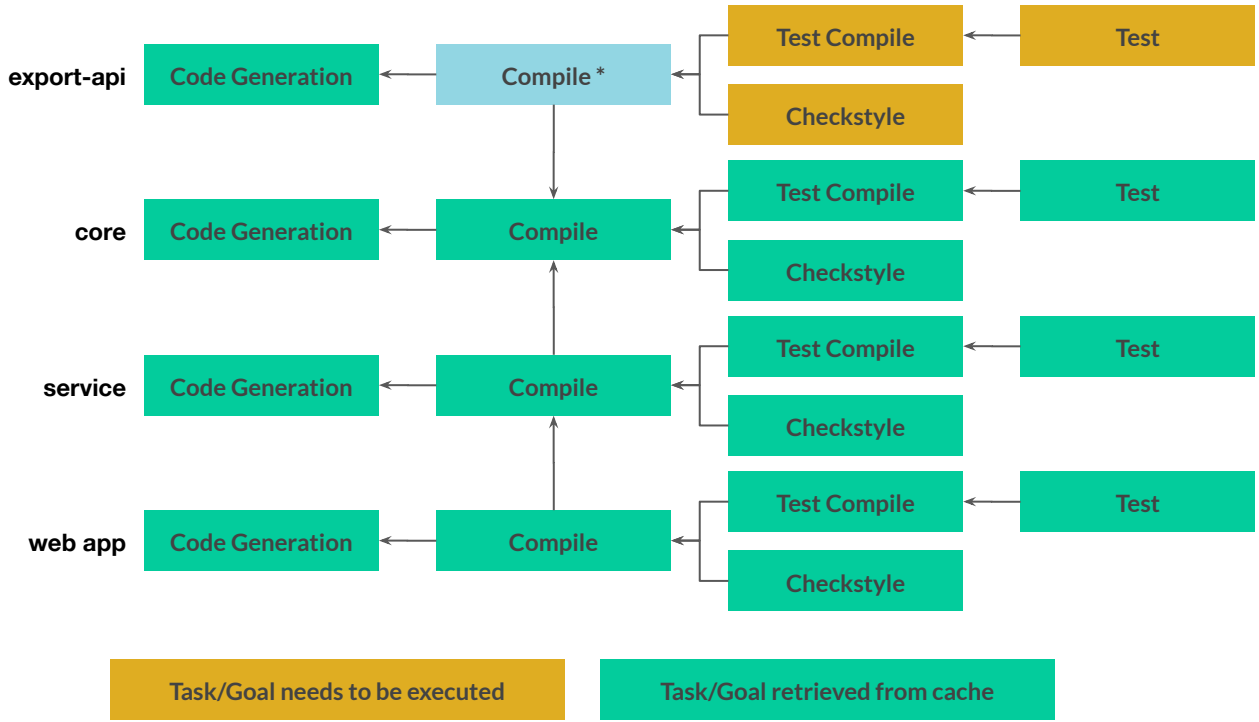
```
include ‘core’
include ‘service’
Include ‘webapp’
Include ‘export-api’
```

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

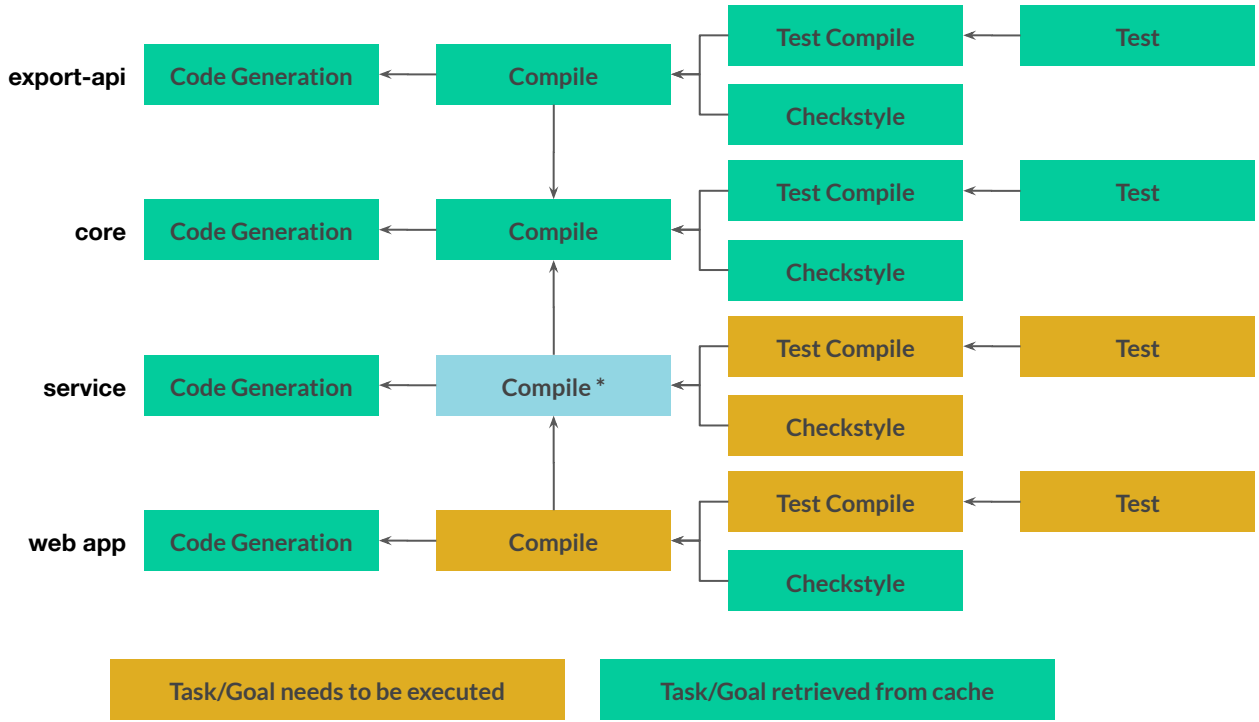
When not using the build cache, with Maven any change will require a **full build**. For Gradle this is the case when doing clean builds and switching between branches.



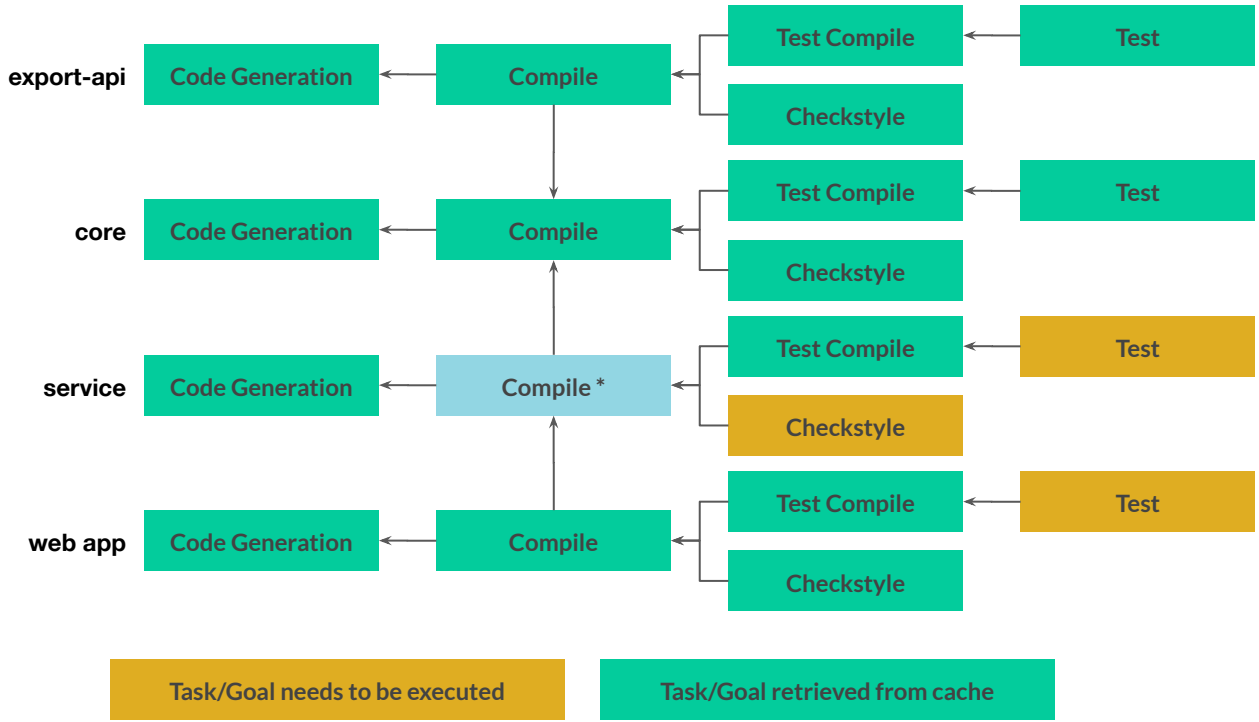
Changing a **public method** in the **export-api** module



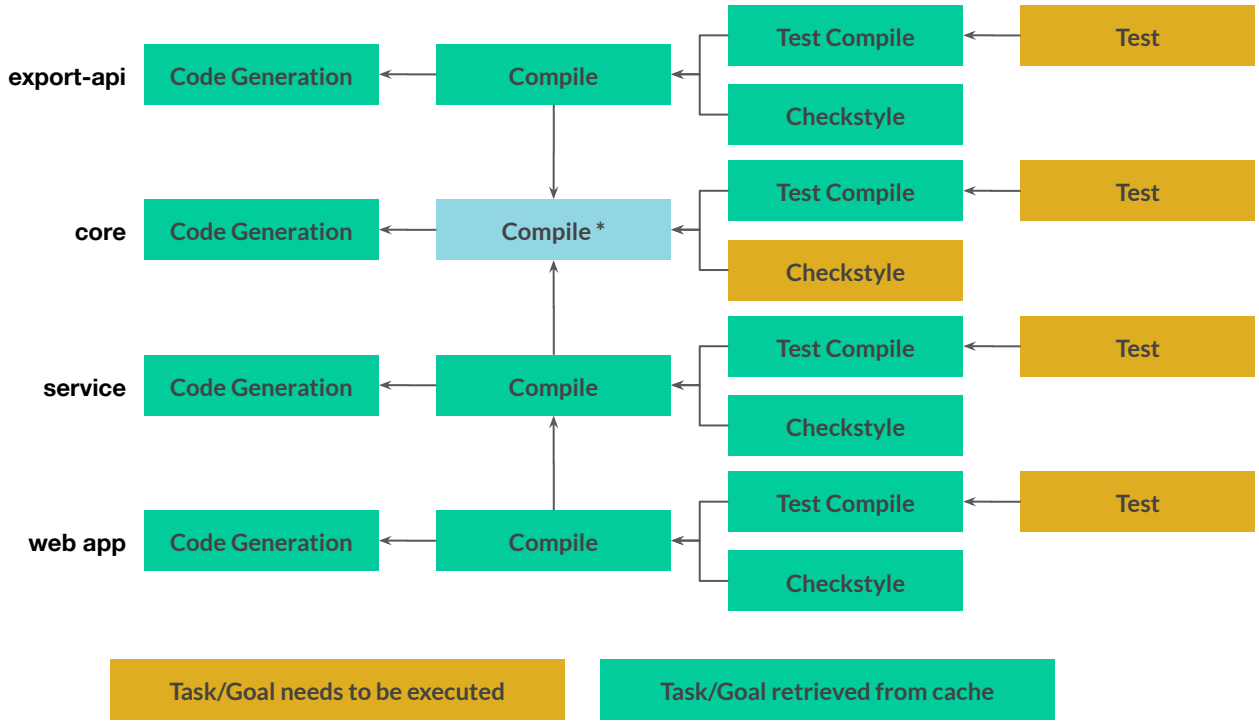
Changing a **public method** in the **service** module



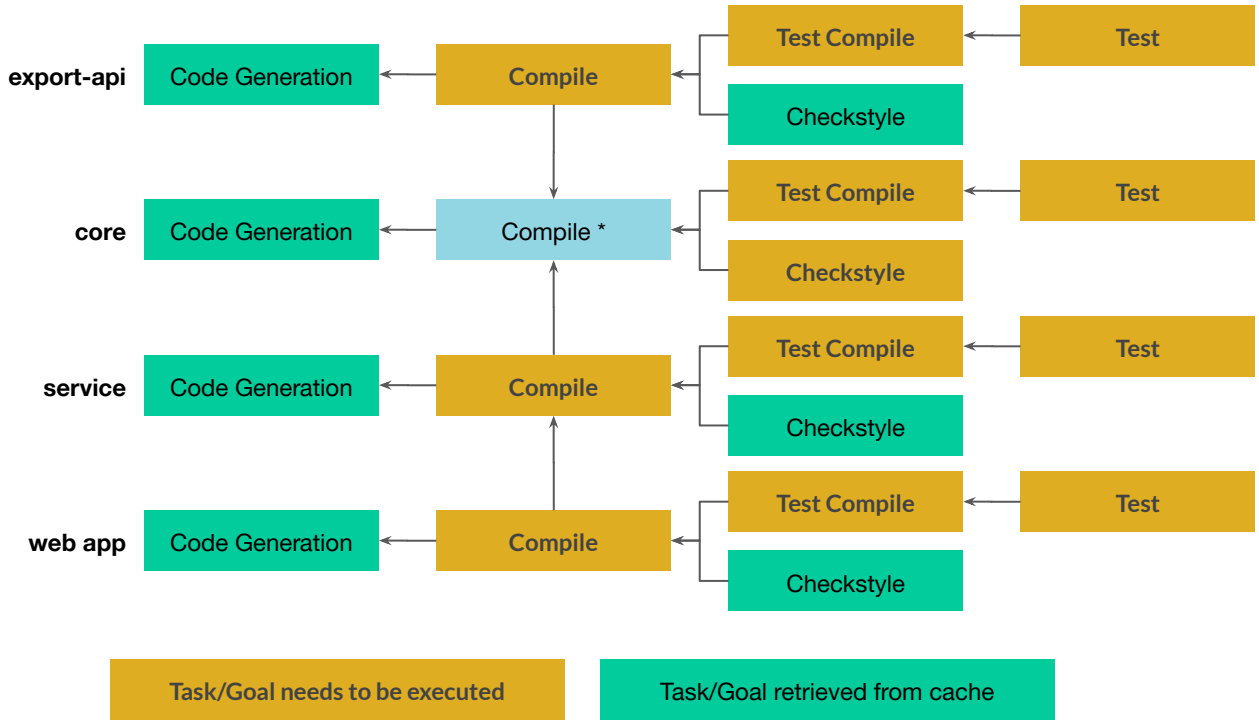
Changing an **implementation detail** of a method in the **service** module



Changing an **implementation detail** of a method in the **core** module



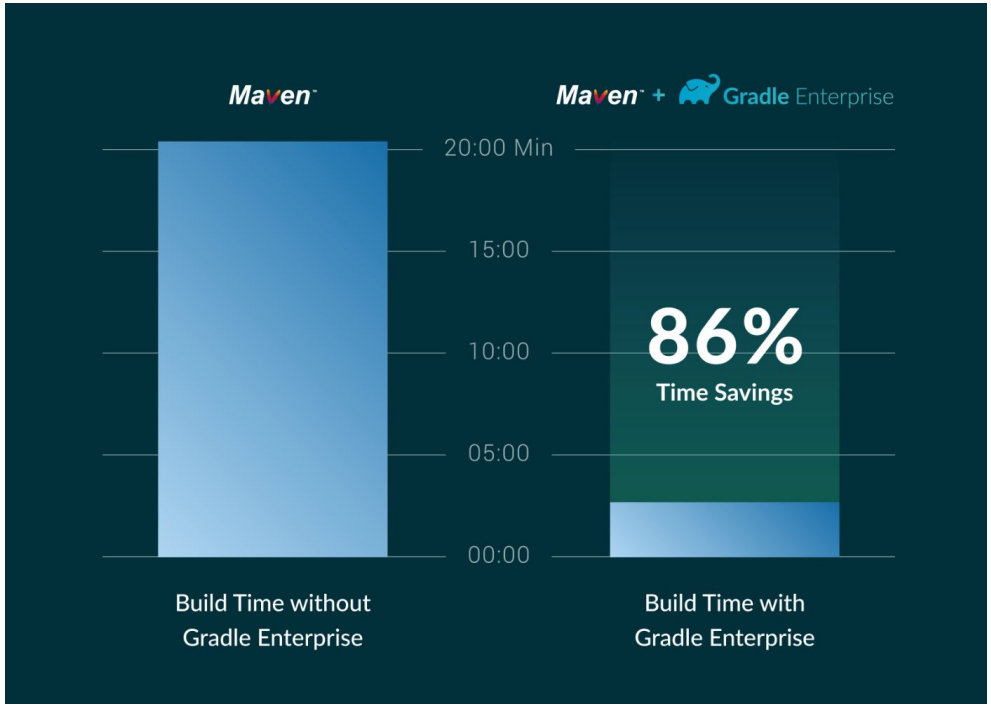
Changing an **public method** in the **core** module



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 leaf modules
  - Leaf build times is reduced by ~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

Build Cache Results - Maven Example



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 01: Using the Local Build Cache

For **Gradle**

labs/01-local-cache/gradle/README.txt

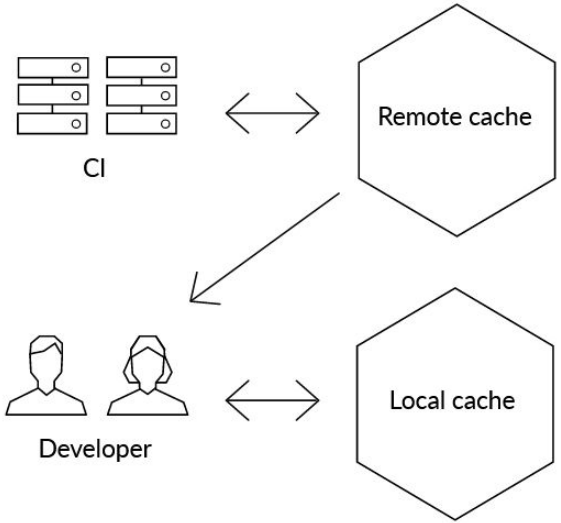
For **Maven**

labs/01-local-cache/maven/README.txt

Download:
https://gradl.es/mdpw

GE Credentials:
Username: **attendee**
Password: **gradle**

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 02 - Help Luke Accelerate his build

Luke has heard that everyone else is enjoying faster builds due to the remote cache. Luke is frustrated that his builds aren't any faster.

Use build scans on https://enterprise-samples.gradle.com to identify why Luke isn't benefiting from the remote cache

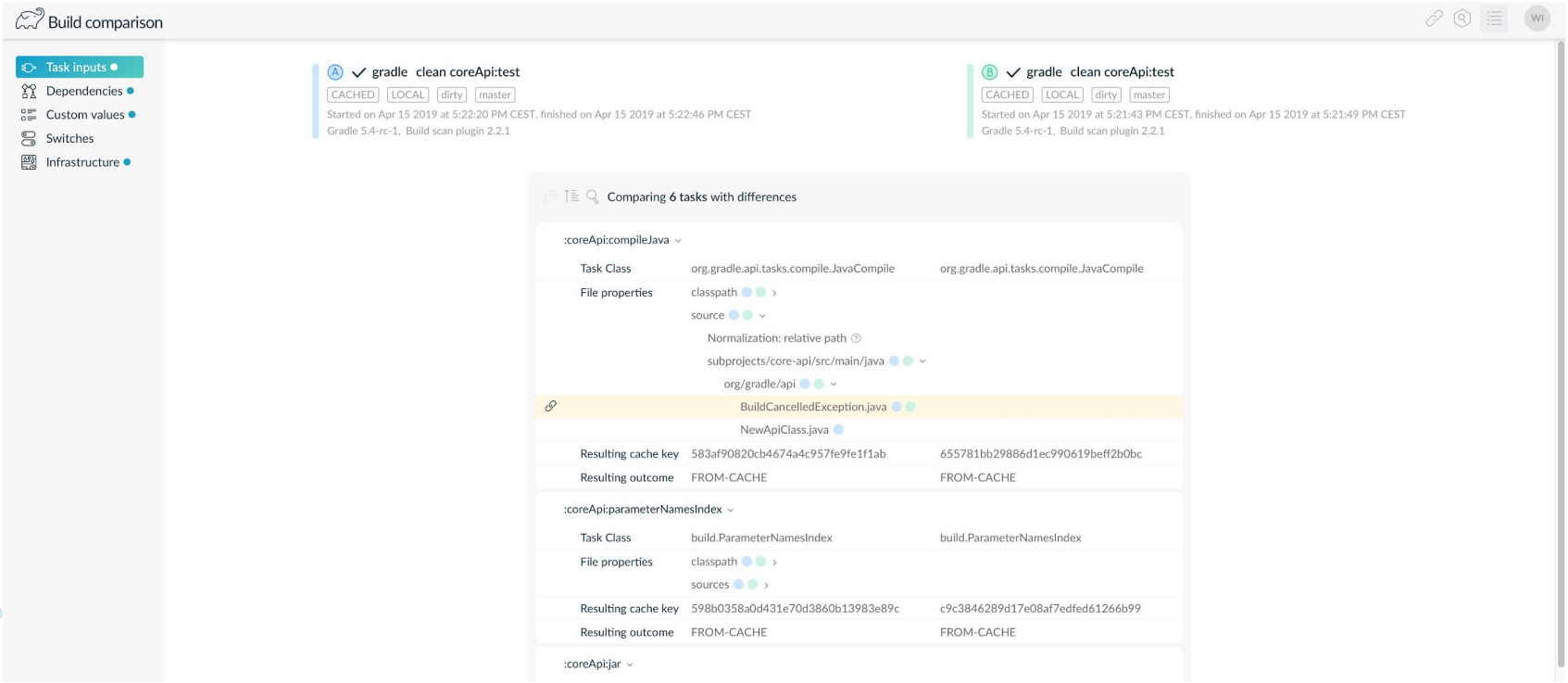
Hints:

- Filter on the custom value: **git branch name=sam/build-data-generation**
- You may have to expand the time range (**Set it to start in May 2020**)
- The performance dashboard shows avoidance savings, are Luke's builds benefiting?
- Go back to the scans list and use the "Compare Build Scan" button to compare one of Luke's builds to a CI build

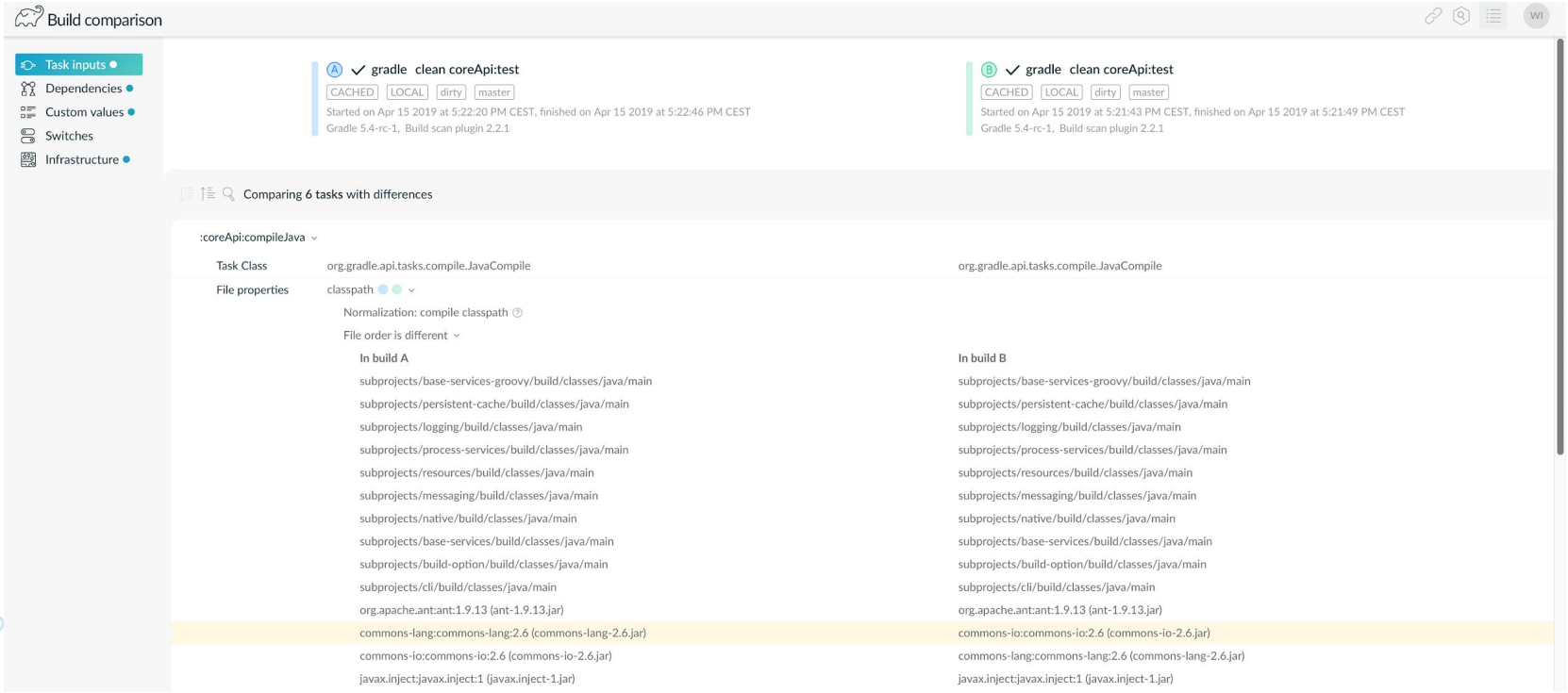
Input Volatility

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

Identifying differences in inputs



Debugging Volatile Inputs



Lab 03: Build Comparison

For **Gradle**

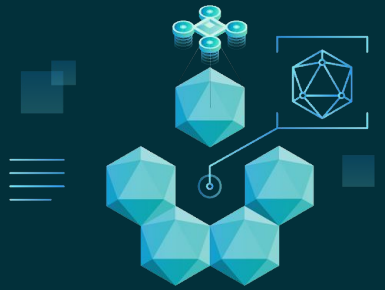
labs/03-build-comparison/gradle/README.txt

For **Maven**

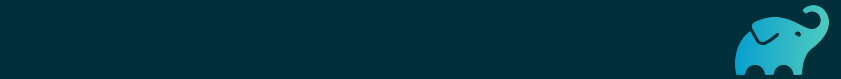
labs/03-build-comparison/maven/README.txt

Download:
https://gradl.es/mdpw

GE Credentials:
Username: **attendee**
Password: **gradle**



Distributed Testing



Existing solutions - single machine parallelism

Parallelism in Gradle is controlled by these flags:

`--parallel` / `org.gradle.parallel`
: Controls project parallelism, defaults to false

`--max-workers` / `org.gradle.workers.max`
: Controls the maximum number of workers, defaults to the number of processors/cores

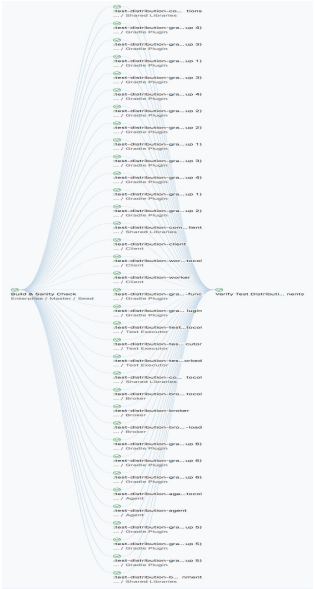
`test.maxParallelForks`
: Controls how many VMs are forked by an individual test task, defaults to 1

See https://guides.gradle.org/performance/#parallel_execution for more information

Existing solutions - CI fanout

Test execution is distributed by manually partitioning the test set and then running partitions in parallel on several CI nodes.

```
pipeline {
  stage('compile') { ... }
  parallelStage('test') {
    step {
      sh './gradlew :testGroup1'
    }
    step {
      sh './gradlew :testGroup2'
    }
    step {
      sh './gradlew :testGroup3'
    }
  }
}
```

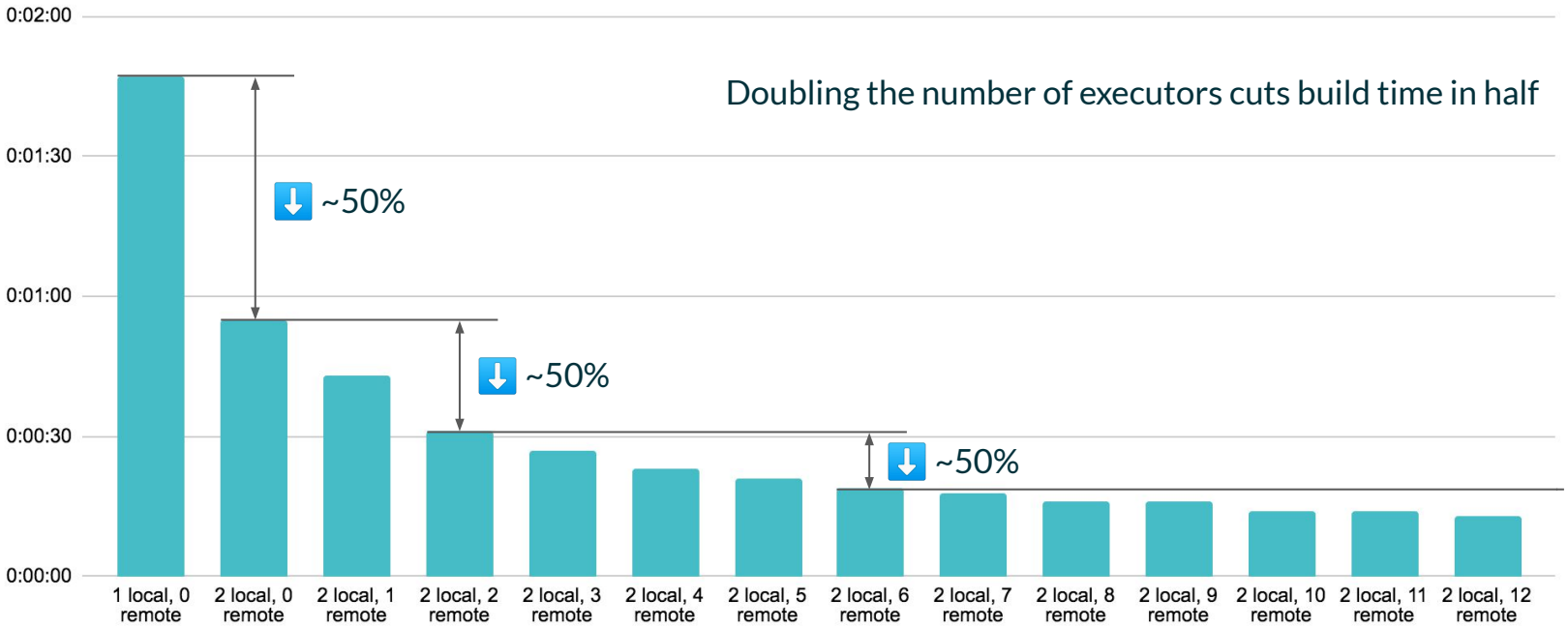


See https://builds.gradle.org/project/Gradle for an example of this strategy

Assessment of existing solutions

- **Build Caching** is great in many cases but doesn't help when test inputs have changed.
- **Single machine parallelism** is limited by that machine's resources.
- **CI fanout** does not help during local development, is inefficient (in particular on ephemeral CI agents or without build cache), requires manual setup and test partitioning, and result collection/aggregation

Distributed Testing Results



Measurements from the demo project

Prerequisites for using the Test Distribution plugin

- Gradle 5.4+
- Gradle Enterprise 2020.2 (to be released ~ April 2020)
- Tests running on JUnit Platform
  - ✅ JUnit Jupiter (introduced with JUnit 5)
  - ✅ JUnit 3.x/4.x (via JUnit Vintage Engine)
  - ✅ Spock Framework (via JUnit Vintage Engine)
  - ✅ Any JUnit Platform test engine
- Java 8+ test runtime

Support for

Maven™

Coming Soon!

Pain Point #2

Inefficient troubleshooting of broken builds

Pain Point #2	Benefit	Solution
Inefficient Trouble-shooting	Efficient Trouble-shooting	Root Cause Analysis Data

Not enough information available to troubleshoot efficiently

- 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

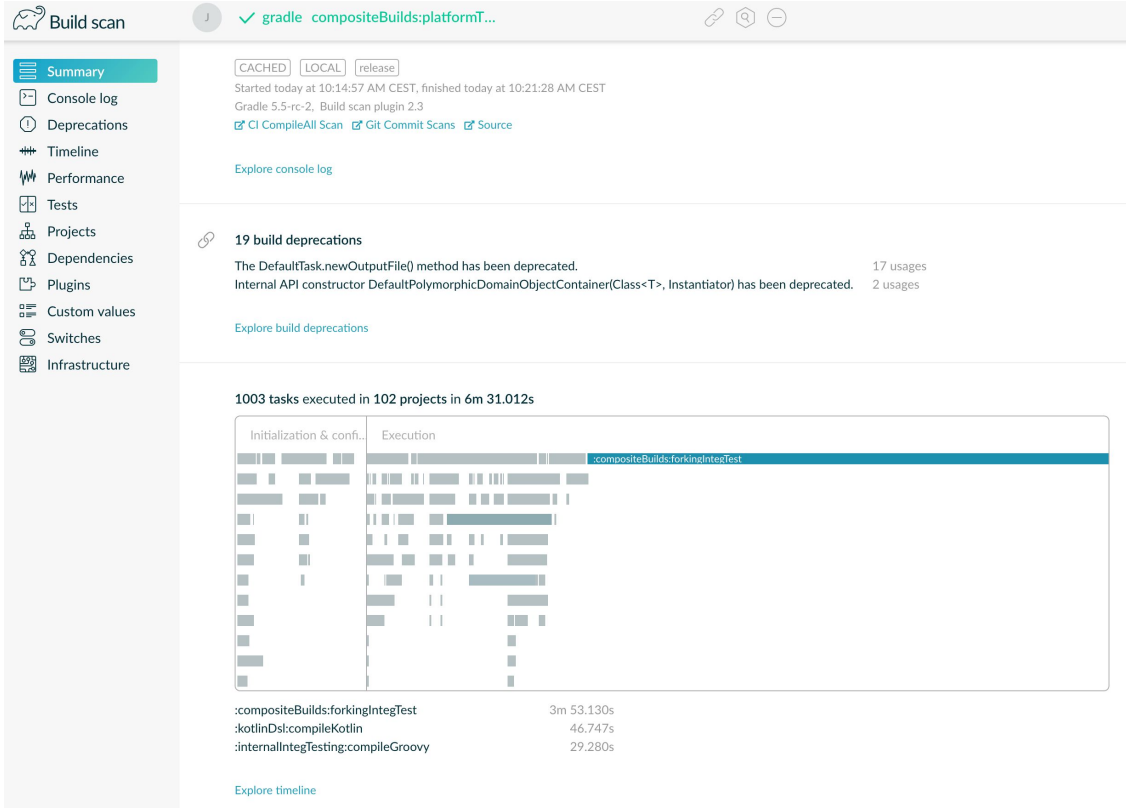
Capture data from every build run (local & CI)

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

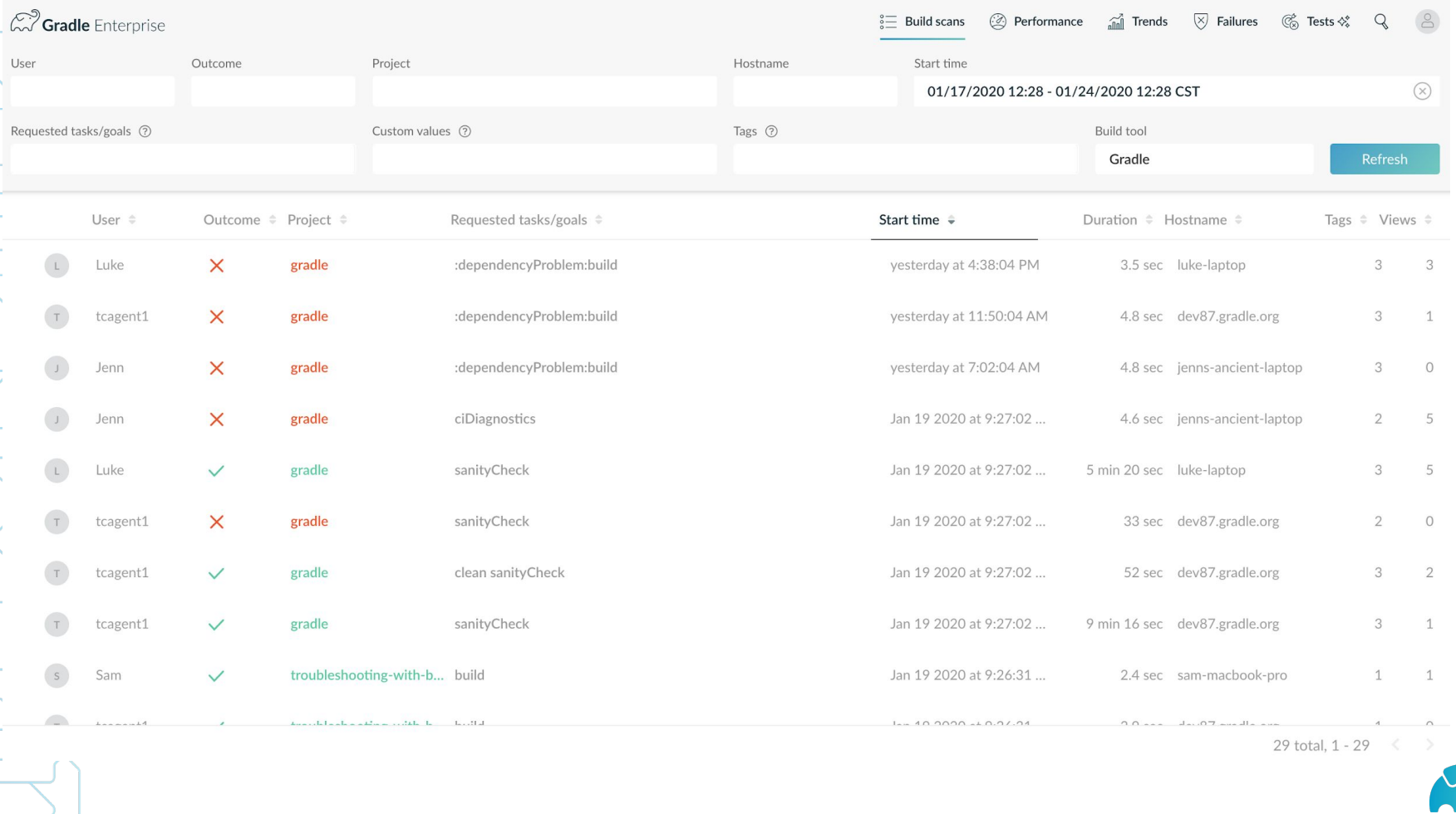
Benefits

- Self-service and/or collaborative build debugging
- Quick root cause analysis
- Decrease build failure MTTR

Gaining insights with build scans



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


Gradle Enterprise
Build scans
Performance
Trends
Failures
Tests
User
Outcome
Project
Hostname
Start time
01/17/2020 12:28 - 01/24/2020 12:28 CST
Requested tasks/goals
Custom values
Tags
Build tool
Gradle
Refresh
User
Outcome
Project
Requested tasks/goals
Start time
Duration
Hostname
Tags
Views
Luke gradle :dependencyProblem:build yesterday at 4:38:04 PM 3.5 sec luke-laptop 3 3
tcagent1 gradle :dependencyProblem:build yesterday at 11:50:04 AM 4.8 sec dev87.gradle.org 3 1
Jenn gradle :dependencyProblem:build yesterday at 7:02:04 AM 4.8 sec jenns-ancient-laptop 3 0
Jenn gradle ciDiagnostics Jan 19 2020 at 9:27:02 ... 4.6 sec jenns-ancient-laptop 2 5
Luke gradle sanityCheck Jan 19 2020 at 9:27:02 ... 5 min 20 sec luke-laptop 3 5
tcagent1 gradle sanityCheck Jan 19 2020 at 9:27:02 ... 33 sec dev87.gradle.org 2 0
tcagent1 gradle clean sanityCheck Jan 19 2020 at 9:27:02 ... 52 sec dev87.gradle.org 3 2
tcagent1 gradle sanityCheck Jan 19 2020 at 9:27:02 ... 9 min 16 sec dev87.gradle.org 3 1
Sam troubleshooting-with-b... build Jan 19 2020 at 9:26:31 ... 2.4 sec sam-macbook-pro 1 1
29 total, 1 - 29

Lab 04: Troubleshooting with build scans

For **Gradle**

labs/04-troubleshooting-with-build-scans/gradle/README.txt

For **Maven**

labs/04-troubleshooting-with-build-scans/maven/README.txt

Download:
https://gradl.es/mdpw

GE Credentials:
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 05 - Help Jenn fix her build

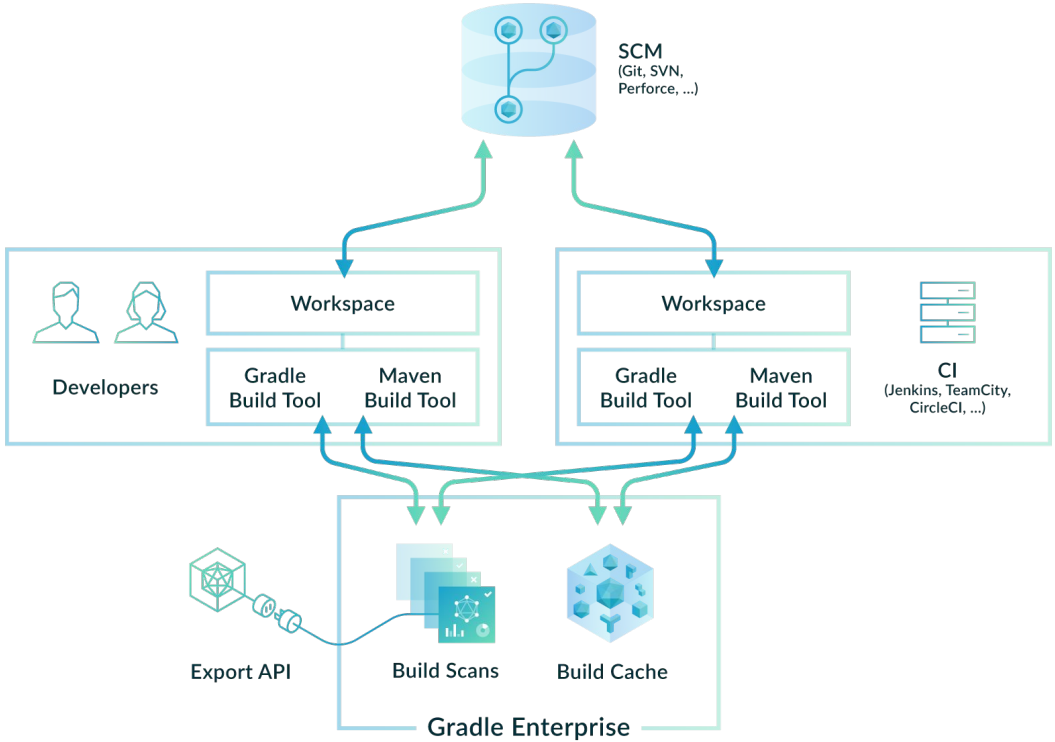
Gradle engineer Jenn has been trying to add new functionality to a task called "**ciDiagnostics**". It is passing on CI, but she hasn't been able to run it locally. Jenn is frustrated and has requested your help.

Use build scans on https://enterprise-samples.gradle.com to identify why the ciDiagnostics task isn't working for Jenn.

Hints:

- Filter on the custom value: **git branch name=sam/jenn-scenario**
- You may have to expand the time range (Go back to April 2020) and **find one of Jenn's builds**
- Look at the "Custom Values" for information about the state of Jenn's work

What is Gradle Enterprise?



Pain Point #3

Flaky Tests & Other Avoidable Failures

Pain Point #3	Benefit	Solution
Flaky Tests & Other Avoidable Failures	Reliable Builds & Tests	Failure 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

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
  - Flaky 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 (flaky 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 flaky and hard to reproduce

Why Reliable Builds & Tests are so Important

- Decrease lead times by making deployment pipelines more reliable, and therefore faster
- Improve confidence in the correctness of changes
- Dramatically increase team morale

Lab 06 - Flaky Builds: Bane of Developer Productivity

Go to this address: https://go.gradle.com/wb-ch6-g1 to access the Failures Dashboard, pre-filtered to the relevant time range.

Using the 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?
- If the most frequent failure were fixed, by how much would it reduce the total number of Non-Verification build failures?

Hints:

- The second item in each row, is a small preview of the "Failures over time" chart for that group of failures

Lab 07 - Better Development through Reliable Testing

Go to this address: https://go.gradle.com/wb-ch8-g1 to access the Test Dashboard, pre-filtered to the relevant time range.

Using the information presented on the dashboard, answer these questions:

- What percentage of all builds have failed tests?
- Which test class fails most often?
- Is this enough information to decide which test classes are problematic?

Pain Point #4

No Metric/KPI Observability

Pain Point #4	Benefit	Solution
No Observability	Continuous Learning & Improvement	Trends & Insights

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 flaky 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.

Benefits

- 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**

Lab 08 - Catching Performance Regressions Before They Catch You

Go to this address: https://go.gradle.com/wb-ch7-g1 to access the Performance Trends Dashboard, pre-filtered to the relevant time range:

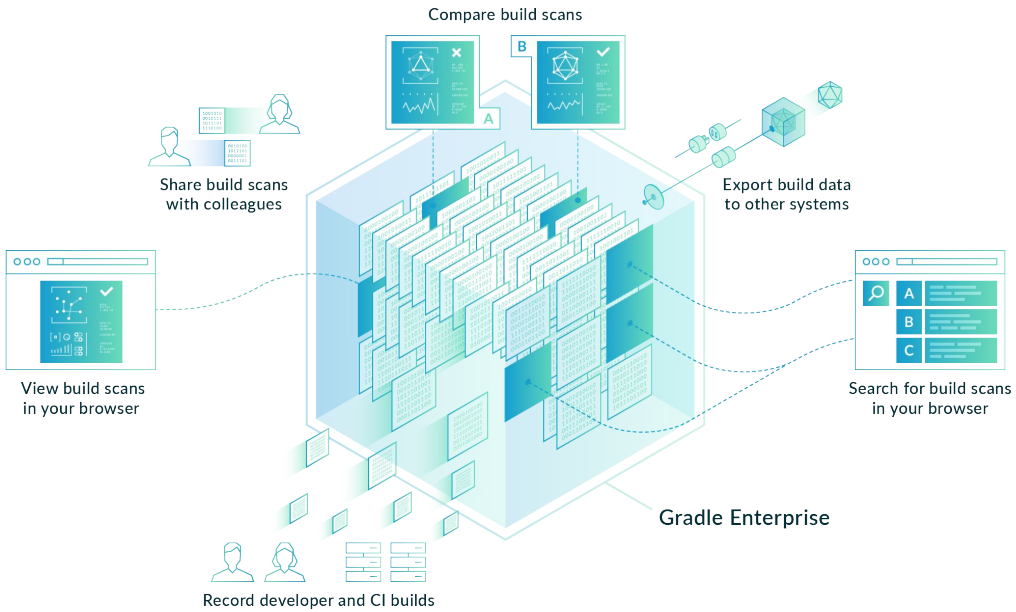
Using the information presented on the dashboard, answer these questions:

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

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.

Pain Point #5

Inefficient use of CI Resources

Pain Point #1	Benefit	Solution
Inefficient use of CI resources	CI cost & resource efficiency	Acceleration technologies

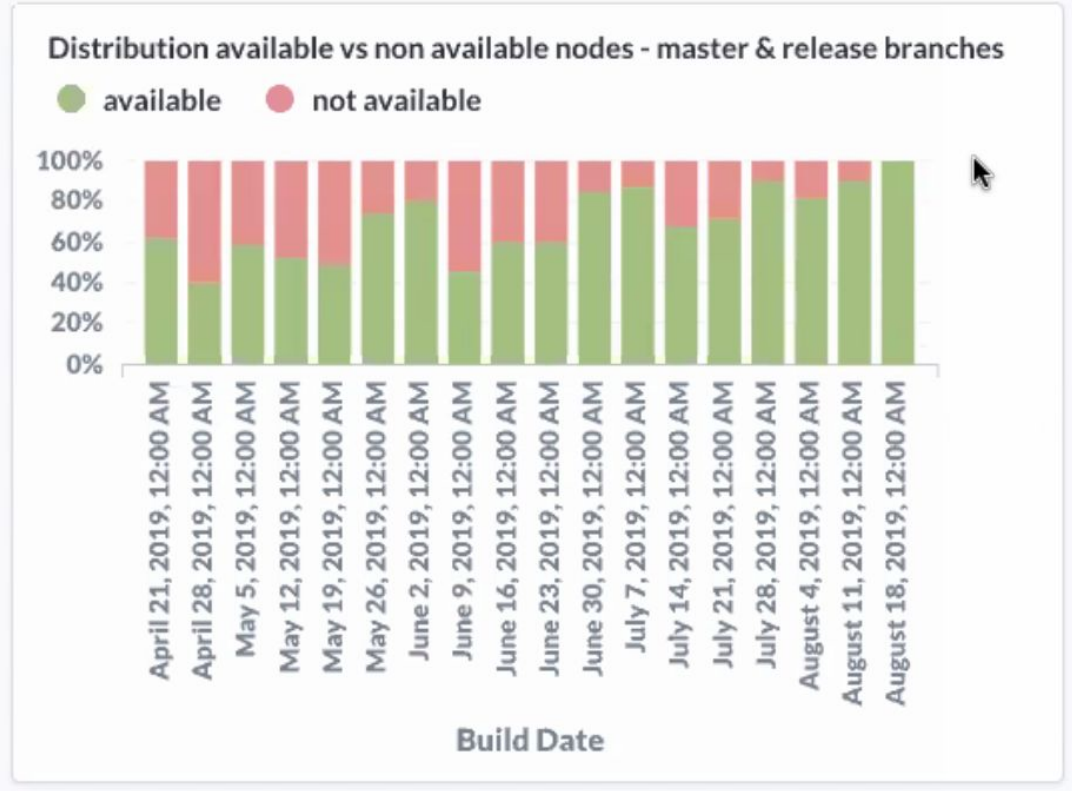
CI is often underutilized

- More idle/waiting times for developers
- Decreased agent availability and resource efficiency
- More engineering time required on infrastructure/provisioning/maintenance

Benefits

- Faster CI pipelines using build caching
- Faster CI pipelines using distributed testing
- Increased reliability with flaky test analysis
- Getting ahead of build regressions using trends and insights

Build Cache improves agent availability



The Cost of Inaction (COI)

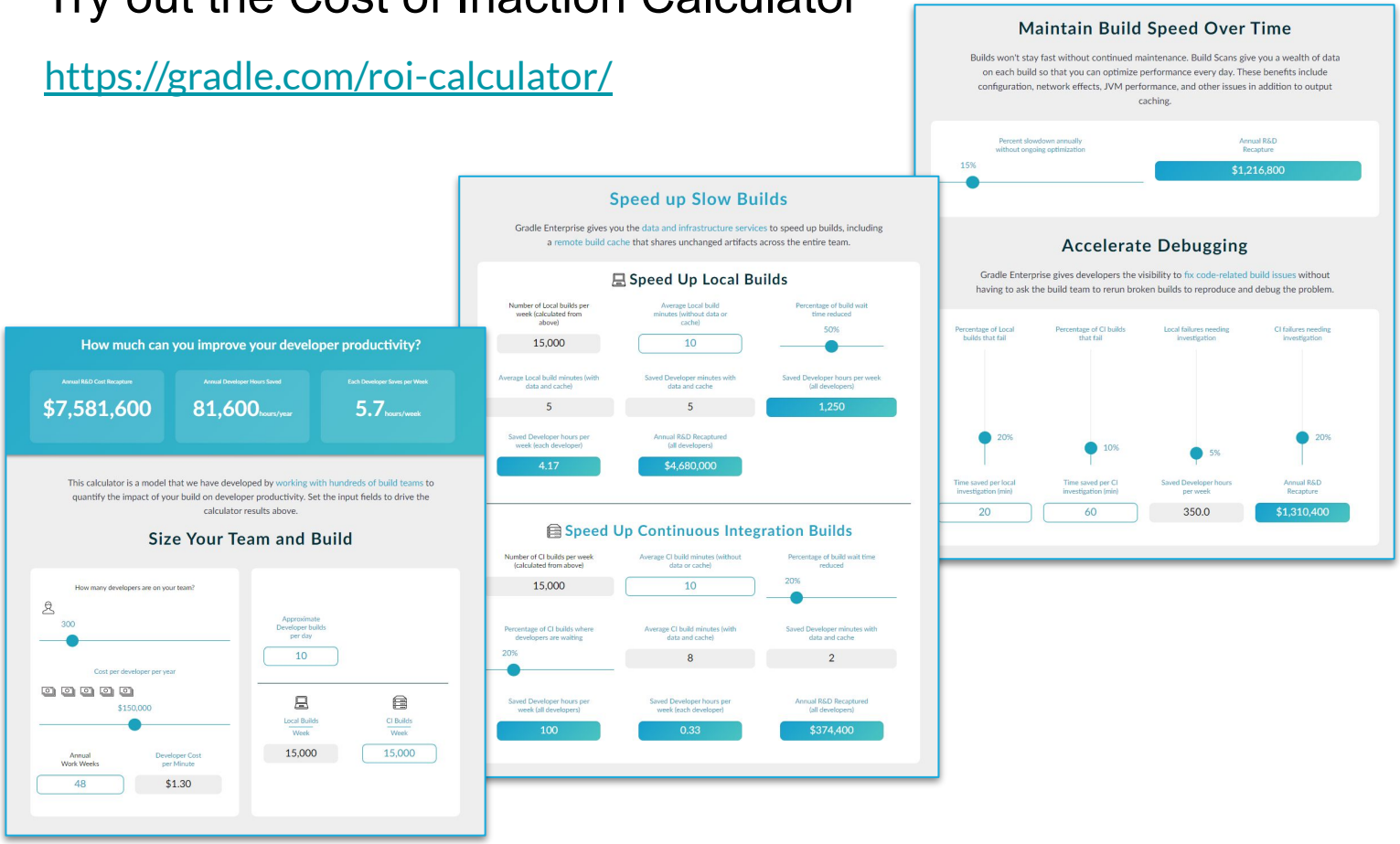
The Cost of Inaction ($$$)

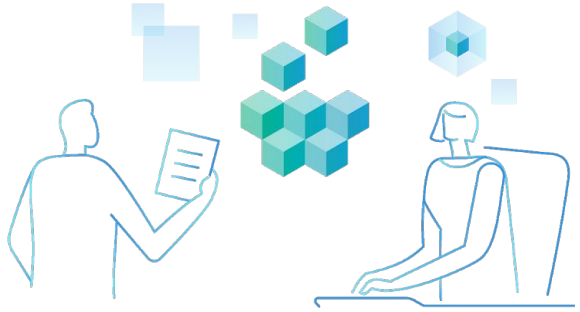
Example Team Assumptions	
# of Engineers	200
Cost per Engineer/Year	$100K
Local Builds/Year	500K
CI Builds/Year	500K
Average Build Time for Local Builds	3 min
Average Build Time for CI Builds	8 min

Cost of Inaction for…	Annual R&D cost	Eng. Years
Waiting for Local Builds	$1.2M	12
Waiting for CI Builds	$0.8M	8
Debugging Build Failures	$0.7M	7
Debugging Faulty Build Logic	$0.3M	3
CI Hardware	$0.2M	n/a
Total Cost	**$3.3M**	**30**

Try out the Cost of Inaction Calculator

https://gradle.com/roi-calculator/



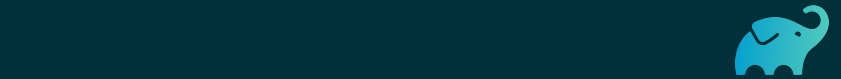


COI Calculator Demo

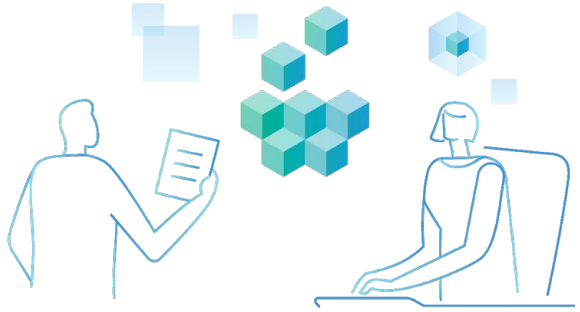
Summary of Developer Productivity Engineering Benefits & Solutions

PAINS	IDLE/WAIT TIME	INNEFFICIENT TROUBLE-SHOOTING	FLAKY TESTS	NO OBSERVABILITY	INEFFICIENT USE OF CI RESOURCES
BENEFITS	FASTER FEEDBACK CYCLES	FASTER TROUBLE-SHOOTING	RELIABLE BUILDS & TESTS	CONTINUOUS LEARNING & IMPROVEMENT	CI COST & RESOURCE EFFICIENCY
SOLUTIONS	BUILD CACHE & DISTRIBUTED TESTING	ROOT CAUSE ANALYSIS DATA (BUILD SCANS)	FAILURE ANALYTICS	TRENDS & INSIGHTS	BUILD CACHE & RESOURCE PROFILING

Next Steps



Next Steps



1. Run the Cost of Inaction Calculator using your own data
   https://gradle.com/roi-calculator/
2. Use the Developer Productivity Engineering eBook as a reference
   https://gradle.com/developer-productivity-engineering/
3. Checkout our Learning Center for a deeper dives into all DPE concepts
   https://gradle.com/learning-center/
4. Contact us to get a personalized demo of Gradle Enterprise
   https://gradle.com/enterprise/demo/

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 Pre-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

Please take 2 minutes and provide feedback

Thank you!