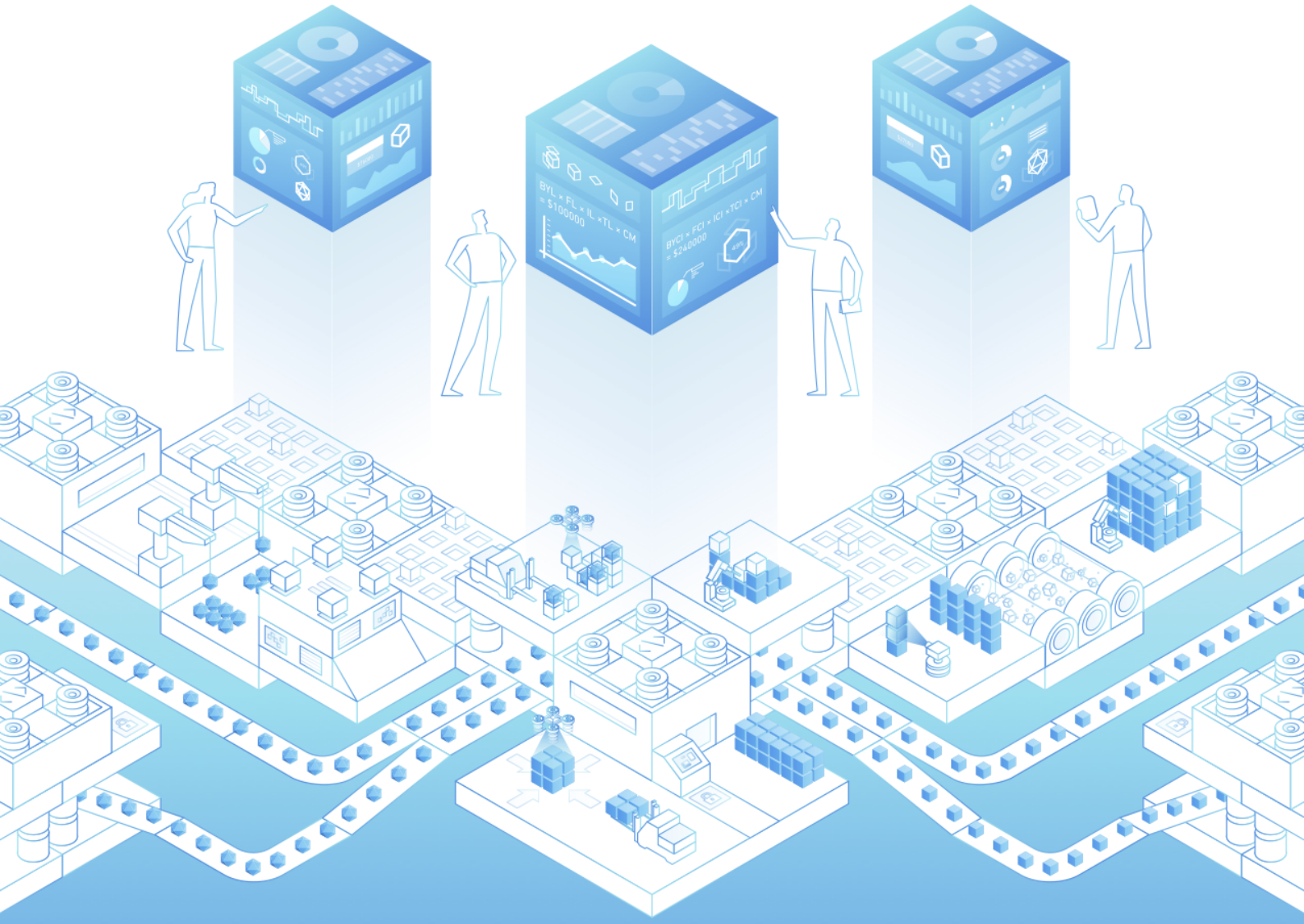
Developer Productivity Engineering

Using data and acceleration techniques to improve the developer experience and essential software development processes



Hans Dockter
Sam Snyder

Gradle

Developer Productivity Engineering

Hans Dockter, Sam Snyder

Version 1.1-build, 2020-02-11

Table of Contents

Developer Productivity Engineering

Published by Gradle Inc.

Production Editor: Scott Regan
Copy Editor: Jennifer Strater
Cover and Interior Design: Charles Aweida

Acknowledgements

Over the last decade, we have been fortunate to work with literally hundreds of the world's top engineering teams and thousands of developers who have so generously shared their experience, stories, challenges, and successes in the software development process.

We would also like to thank all of the talented and dedicated engineers at Gradle, Inc, and contributors to the Gradle open source community.

Finally, we want to recognize the support of our families and friends that enable all of us to pursue our passions in the creative process and craft of software development.

Hans Dockter, Sam Snyder, and the team at Gradle, Inc.

Preface

About the Early Access Edition of this book

This version of the book is provided to you as an early access edition and is free for everyone.

If you registered with Gradle when you downloaded this early access edition of the book, you will receive periodic updates and a free copy of the final version. Otherwise, you can check the website for periodic updates to the early access edition and purchase a copy of the final version once it becomes available.

About this book

In this book we share developer productivity engineering techniques and stories on how to:

- Understand the importance of fast feedback cycles and early discovery of bugs
- Quantify the costs of low productivity development environments that waste time waiting for slow builds, tests, and CI/CD pipelines
- Organize the data required to understand, troubleshoot, and improve essential development processes like builds, tests, and CI/CD pipelines
- Use acceleration technologies like caching and distribution to speed up feedback cycles
- Use data to proactively improve the reliability of the development toolchain
- Find and eliminate frequent errors and noisy signals like flaky tests

It’s split into 3 parts:

- Time Is Joy (and Money)
- Using Data to Make Builds Faster and More Reliable
- Economics

The chapters are independent stories so feel free to skip around and read the sections you find most useful now.

Chapters 1-2 in Part 1 cover the reasons why you should care about build speed and developer productivity in general. We use real-world examples to show the consequences of following or not following the practice of developer productivity engineering.

Chapters 3-5 in Part 2 highlight the problems you’ll first encounter when starting your journey to developer productivity engineering happiness and how to fix them.

Chapters 6-7 in Part 3 discuss the benefits of developer productivity engineering and how to calculate the ROI of these efforts for management and reporting.

The Appendix includes a list of helpful resources for taking the next step on your journey to developer productivity engineering.

About the cover art

The cover art was designed by Charles Aweida and Ari Zilnik of the Gradle Design team in collaboration with our illustrator Roger Strunk.

The illustration represents a developer productivity engineering team looking at metrics and collaborating with the developers to increase the productivity of their software engineering factory.

Conventions

> As this book is about real-world examples and stories, we don't have a lot of code to show. We do, however, use the following convention for calling out short anecdotes that are meant to complement the text.

Reader feedback

Feedback is always welcome and we hope to continue to add to this body of experience and thought leadership. You can reach out to us at devprod-eng-book@gradle.com.

About the Authors

Hans Dockter is the founder and project lead of the Gradle build system and the CEO of Gradle Inc. Hans is a thought leader in the field of project automation and has successfully been in charge of numerous large-scale enterprise builds. He is also an advocate of Domain Driven Design, having taught classes and delivered presentations on this topic together with Eric Evans. In the earlier days, Hans was also a committer for the JBoss project and founded the JBoss-IDE.

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. Previously he was an engineer at Microsoft and graduated with a BS in Computer Engineering from Ohio State.

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.

Introduction

When I started the Gradle project, it was out of deep frustration working in low productivity environments.

The amount of time it took to make progress and get feedback on changes in the enterprise environment was incompatible with how I work. It was taking the passion and satisfaction away from the craft of software development.

Across the entire software industry, organizations are struggling to ship software. Teams staffed by highly-skilled, professional engineers realize only a fraction of their theoretical peak efficiency, held back by innumerable problems.

Over the last decade, we have been fortunate to work with literally hundreds of the world's top engineering teams and thousands of developers who have so generously shared their experiences, stories, challenges, and successes in the software development process.

This book represents not just our learnings from that exciting journey but also what we see emerging on the horizon from working with the most respected software development teams in the world, such as LinkedIn, Salesforce, and Tableau, and more.

These leading-edge teams made developer productivity engineering an executive-level initiative. They have realized that to be successful at scale, they must provide an environment that allows their developers to innovate at their full potential. To achieve this, they staffed developer productivity engineering teams with teams of experts whose sole purpose was improving the developer experience.

We wrote this book to share what we've learned with everyone interested in making their software organization the highest performing it can possibly be.

Hans Dockter, Founder and CEO of Gradle, Inc.

What is Developer Productivity Engineering?

Developer productivity engineering is the discipline of using data to optimize software development practices, thereby improving the happiness and enhancing the productivity of software developers. Organizations successfully applying this discipline enjoy a high degree of automation, fast and reliable developer feedback cycles, and gracefully handle ever-growing codebases.

Most mature industries have engineering disciplines dedicated to making production efficient and effective. Fields such as chemical engineering and industrial engineering have process experts widely understood to be essential to the successful & economical operation of their firms.

Developer productivity engineering is of similar importance when it comes to the production of software. It is quickly becoming a critical discipline in software engineering organizations striving for industry leadership across diverse, competitive markets.

The Case for Developer Productivity Engineering

By Hans Dockter

Software productivity depends on toolchain effectiveness

The creative flow of software developers depends on the effectiveness of the toolchain

Software development is a very creative process. It is similar to the method experimental scientists use when formulating hypotheses, entering into a dialogue with nature to confirm or deny them. Our hypothesis is our code and the dialogue is with the compiler, with the unit/integration/performance tests and other validations, and with the running software. So our creativity also requires a dialogue. The quality of the creative flow for your developers depends on how efficient that dialogue is. How quickly do you get responses and what is the quality of the response?

In an ideal world, you get the answers instantaneously, the answers are always correct and your changes are immediately available at runtime.

This was more or less the experience when we all started with software development. This was so much fun; it was so productive, and we were hooked. But what if, when you wrote your first code, it took a full minute to change the color of the circle you were drawing? Would you have continued with coding? I'm not sure I would have.

The collaborative effectiveness depends on the toolchain effectiveness

Now, many of us are enterprise software developers and instant gratification is replaced with complex collaborative processes and all the additional dialogue we need to have. A business expert or customer has an idea that you interpret in a certain way via code. Then, you need to run your code to get feedback from the other person on how correct your interpretation is. The quality of the collaboration depends on how quickly you can iterate, which to a large degree depends on the effectiveness of your toolchain.

The overall team productivity depends on the toolchain effectiveness

The team productivity is determined by the quality of the creative flow and the collaborative effectiveness, which again are strongly correlated to the toolchain effectiveness. Your toolchain efficiency becomes a major bottleneck for your productivity and your success. The reason for that is that the developer toolchain in the enterprise is complex machinery with complex inputs that usually provide much lower than possible feedback cycle time and quality of feedback. The reason for that is that enterprises do not performance manage or quality control their toolchain and do not make use of acceleration technology that is available.

Successful projects, in particular, suffer from inefficient toolchains. An unmanaged and unaccelerated toolchain will collapse under the pressure when the number of lines of code, the number of developers, locations, dependencies, repositories, and tech stack diversity all grow exponentially.

Software productivity will be low without developer productivity engineering

As your company grows, you will need a dedicated team of experts in the field of developer productivity engineering. This discipline has the sole focus of optimizing the effectiveness of the developer toolchain to achieve a high degree of automation, fast feedback cycles and correct, reliable feedback.

Developer productivity engineering is the discipline of using data to improve the developer experience and essential software development processes to ensure a high degree of automation, fast feedback cycles, and reliable feedback.

The job of developer productivity engineering is to improve productivity through the entire lifecycle from the IDE to production. Too often CI/CD teams are not responsible for developer productivity and make decisions that optimize for other metrics, e.g. auditability without taking adverse effects on developer productivity into account. The priorities and success criteria for this team are primarily based on data that comes from a fully instrumented toolchain. For such a team to be successful it needs a culture where the whole organization commits to improving developer productivity.

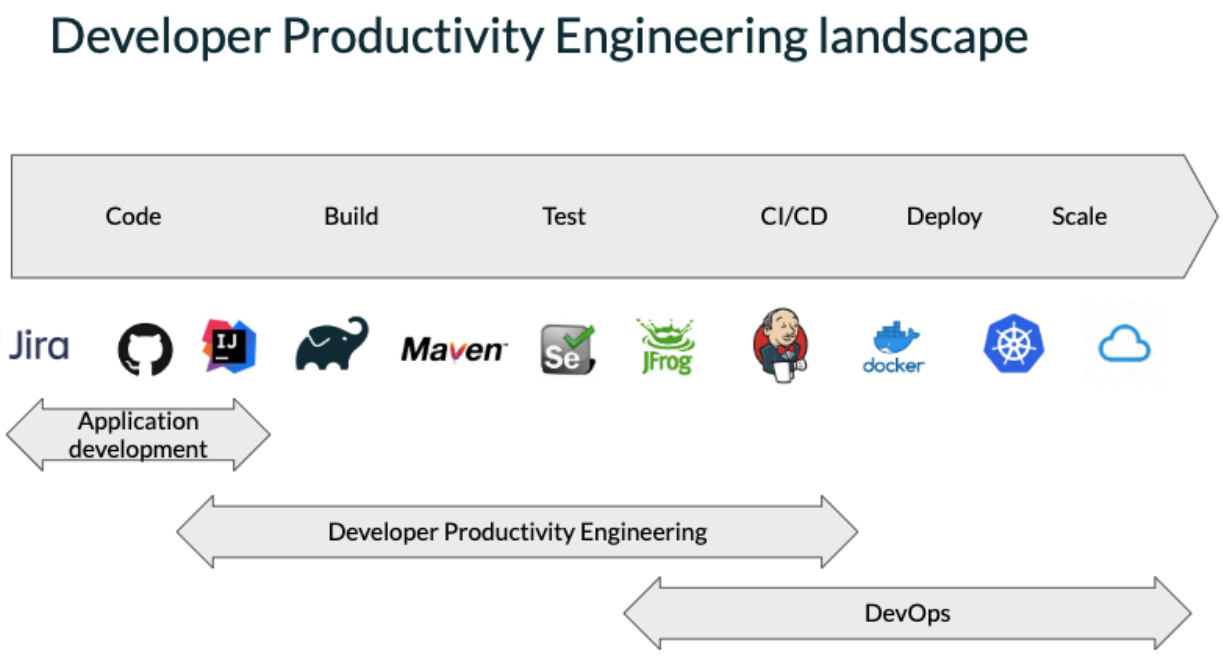
There is a significant gap between the actual productivity of software development teams and their full potential. The gap is growing over the lifetime of a project for most organizations but it can be significantly reduced with the practice of developer productivity engineering.

The happiness of your developers depend on their productivity. 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 are seeing a drain of talent leading to even lower productivity.

The economic benefits of not applying this practice are dramatic. Development productivity improvements provide significant leverage for every dollar invested in software development. The amount of features you can ship is heavily affected by it. The productivity gap that comes from not applying the practice of developer productivity engineering is a significant competitive disadvantage. Innovation is at the heart of any business to survive. Most innovations nowadays, for any enterprise, must be funneled through software. That means the productivity of your software development teams is crucial for your business to survive.

Developer productivity engineering applies to a wide part of the software development process landscape. It begins with the coding process and applies through to CI/CD and the point where DevOps takes built applications for distribution and scaling.



It is important to say that developer productivity engineering is not about treating developers as factory workers but instead as creative craftspeople. It is about the developer experience and unlocking creative flow and restoring the joy of development as well as the collaboration with the business.

An approach for developer productivity engineering

I started the Gradle project out of deep frustration in working in low productivity environments. The amount of time it took to make progress and get feedback on changes in the enterprise environment was incompatible with how I work and was taking the passion and satisfaction away from the craft of software development. Later I was joined by our Gradle co-founder, Adam Murdoch, and many other talented developers. Over the last decade we have worked with hundreds of the top development teams in the world from every industry.

This book represents not just our learnings from that exciting journey but also what we see emerging on the horizon from working with the most respected software development teams in the

world, such as LinkedIn and Tableau.

These leading-edge teams make developer productivity engineering an executive-level initiative, realizing that it is critical to lead their respective industry and their developer's ability to innovate at their full potential. They prioritize developer productivity engineering with a team of experts whose sole purpose is to improve the toolchain, unblock feedback cycles, and fix the developer experience.

In this book we share techniques and stories in how to:

- understand the importance of fast feedback cycles and catching errors earlier, including incorrect signals like flaky tests
- quantify the costs of a low productivity environment with wasted time waiting for builds, tests, and CI/CD pipelines
- organize the data required to understand, troubleshoot, and improve essential development processes like builds, tests, and CI/CD
- use acceleration technologies such as caching and distribution to avoid unnecessary work, speeding up feedback cycles
- find and eliminate frequent errors and false signals like flaky tests to unblock developers and better support developers
- make the practice of developer productivity engineering a respected discipline as in every other major industry

In many industries, there are engineering disciplines dedicated to the practice of making production efficient and effective, such as chemical engineering and industrial or process engineering. They have their own degrees and are much respected within their broader field.

Developer productivity engineering is of similar importance when it comes to the production of software. The software industry is at the beginning of this realization. It is a very exciting time to work in this area!

Part 1 - Time Is Joy (and Money)

Every Second Counts

Faster Builds by Doing Less with Build Caching

1. Every Second Counts

By Hans Dockter

Shortening feedback cycles can save teams lots of time because there are fewer interruptions to the creative process, developers spend less time waiting for each individual build, and they spend less time debugging problems that occur from compounding changesets. Shorter feedback cycles are also safer because developers make smaller changesets (fewer bugs and conflicts), changesets don't get stale and cause bug-inducing merge conflicts, and companies aren't tempted to cut corners like turning off tests and other safety measures in order to ship faster.

1.1. Faster builds improve the creative flow

Earlier we talked about how the creative flow depends on the speed of the feedback cycle. Consider a customer example comparing feedback cycles from two teams.

	Team 1	Team 2
No. of Devs	11	6
Build time (with tests)	4 mins	1 min
No. of local builds	850	1010

The developers on Team 2 get feedback twice as often as the developers on Team 1. The most likely reason is that their build and tests are faster. Over time, the developers on Team 2 will be much more productive because there are fewer interruptions and they spend less time waiting for each individual build to run.

We see this correlation between build time and the number of builds consistently across organizations, even when the build time is less than a few minutes. The most likely reason for that is that their build and tests are faster. This is obvious when the build times have become painfully long and people explicitly say that they avoid as much as they can to run build and tests. What is interesting though, is that even in environments where building and testing are not perceived or described as slow, the data shows that the number of builds is inversely-proportional to the duration.

First of all, our brains work much better when the disruption to the flow is minimized. Endorphins help us enjoy our work during flow states, motivating us, but in modern offices, we have numerous distractions from meetings and loud coworkers to urgent email messages taking away our focus. With fewer interruptions from builds, we can use our remaining time more effectively.

There is another obvious reason why developers on Team 2 are more productive. A lot of the time waiting for feedback is actually just that, waiting time. When builds and tests run in less than 10 minutes, context switching is often not worth the effort, as a lot of the build and test time is downtime.

1.2. Developer time is wasted waiting for builds and test to finish

The aggregated cost of waiting time is surprisingly high even for very fast builds and even moderate improvements are often worthwhile. If you look at the table below, even though the team with a 1-minute build is doing quite well, reducing the build time by 40% to .6 minutes, saves 44 developer days per year.

Team	No of devs	Local builds per week	Build Time	Build Time w. GE	Savings per year
Team 2	6	1010	1 mins	0.6 mins	44 days

On a larger team, this scales exponentially as we see the team of 100 devs nearly halves their build for a savings of 5200 developer days saved. Assuming 220 working days a year per developers, 5200 developer days is roughly 25% of all engineering time. Significant performance improvements like this is a game-changer for productivity.

Team	No of devs	Local builds per week	Build Time	Build Time w. GE	Savings per year
Team 3	100	12000	9 mins	5 mins	5200 days

1.3. The longer the build the more context switching

As build time increases, more and more people switch to do different tasks while the build is running. If the build is the final build to get a feature merged and the build is successful, this is not much of a problem (although longer builds are more likely to lead to merge conflicts). But if the build fails or was necessary to provide intermediate feedback you have to switch back to the previous task and pay the cost of context switching. This often costs 10-20 minutes per switch and is the reason why for builds that are faster than 10 minutes people would rather wait. The cost has to be paid twice, when going back to the previous task and then again when continuing with the new one.

1.4. Longer builds are harder to debug

The longer the build takes, the bigger the changesets per build will become. In the case your changes pass the build, the overhead of building and testing is roughly like a flat tax. Whatever the amount of changes is, the tax is the same. So if the tax is high and has to be paid per build run, you are minimizing this tax by not running the build as often. So the amount of coding you do in between the build runs will increase.

Unfortunately, the tax is not flat if the build fails. The more changes you have, the longer on average the triaging will take. This increased debugging cost caused by larger individual changesets has to be paid for failing local builds, failing pull request builds and failing master builds.

Depending on how your CI process is organized, for the master CI builds you might have an additional effect. The fewer master CI builds per day you can run because of their duration, the less

merge points you have. Thus the average number of changes included in that CI build is higher. This may significantly increase debugging time for any failed CI build as we are talking possibly about a lot of unrelated changes.

The most extreme cases we have seen are builds of large repositories that take a whole day. If this builds fails possibly 100's of commits might be responsible for that failure and the triaging is often extremely complicated and time-consuming. But even with much shorter build times, you might have that effect. In general, there is a non-linear relationship between the numbers of changes and debugging time.

1.5. Longer builds lead to more merge conflicts

Bigger changesets and more pull requests per master CI build both increase the surface area of the changes and thus the likelihood of merge conflicts. There is another effect at play here. Bigger changesets also increase the debugging time and thus the average time of a successful CI master build. This again reduces the frequency of merges and for the reasons discussed this will increase the likelihood of merge conflicts. This is a vicious circle. Merge conflicts are one of the most frustrating and time-consuming things developers have to deal with.

1.6. The effect of failed builds on build time and context switching

On average 20% of all build and test runs fail. This is a healthy number as it is the job of the build to detect problems with the code. But this number significantly affects the average time until a build is successful and the context switching frequency.

```
average time for a successful build = Average build time + (failure frequency * average build time) + average debugging time
```

```
context switching frequency = failure frequency * number of builds * 2
```

An unreliable toolchain, for example, one with a lot of flaky tests, can thus both drastically increase the average waiting time as well as the context switching frequency.

1.7. Does this apply to every project?

Smaller projects will benefit from faster builds because of less waiting time and improved creative flow. They will also be affected by larger change sets as their build time increases. Smaller projects usually don't suffer from overlapping merges as much as larger projects will suffer from the same issues, as those are less likely to occur with a lower number of developers.

Many larger teams break up their codebase into many different source repositories for that reason. But this introduces other challenges.

The producer of a library, for example, is no longer aware of whether they have broken a consumer after running their build and tests. The consumer will be affected and has to deal with the problem when this happens. The triaging of that is often difficult and very time-consuming.

1.8. Debugging time grows exponentially with the detection time of a problem

Fixing problems later take disproportionately more time than fixing them earlier.

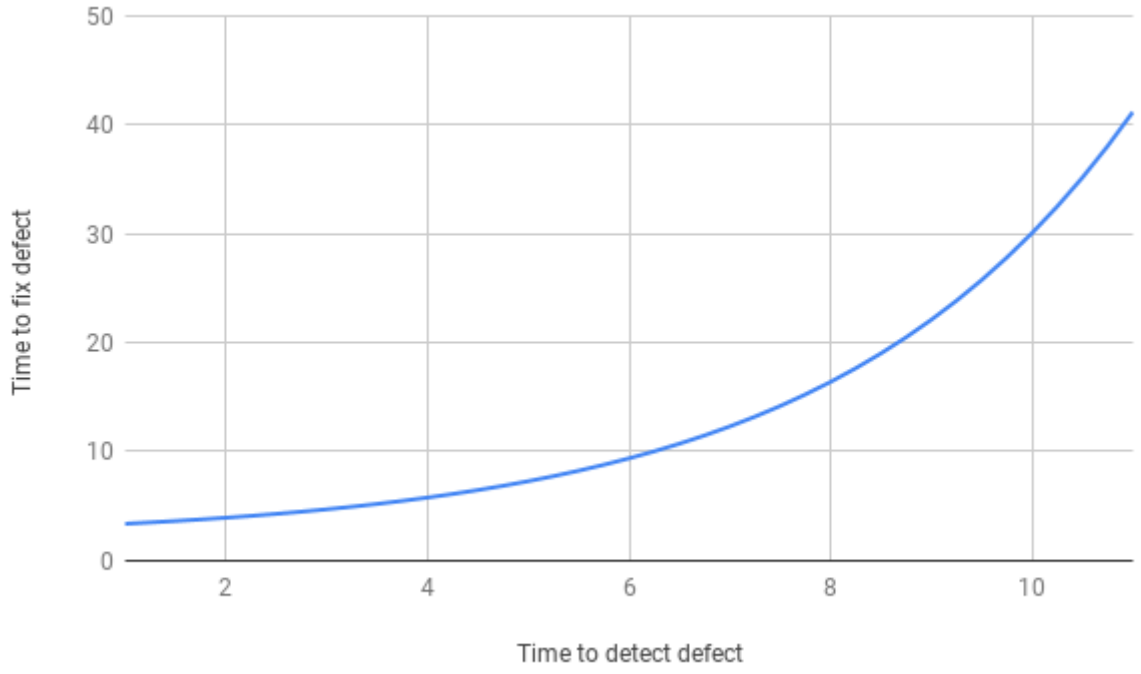


Figure 1. Fix time grows exponentially over detection time

There are multiple reasons for this. We already mentioned the non-linear relationship between the size of the changeset and the time required to fix the problem. Context switching also becomes more expensive.

You have a similar effect in a multi-repository environment where the consumer is responsible for figuring out that the consumer contract has been broken and organizing a fix. In this case, the more dependencies that have changed during a build run make it exponentially harder to figure out which dependency and team are responsible.

1.9. Many companies are moving in the wrong direction

As build times grow there is increasing friction between getting feedback early and reducing the waiting time before you can push your changes.

This is a terrible situation and there is no good choice. Many organizations go down the route of shifting the feedback later in the development cycle, or ‘to the right’. For example, they weaken the local quality gate by no longer running any tests, and the pull request build will be the first build to run the unit tests.

This is the opposite of a healthy CI/CD pipeline where you want to get feedback as early and as conveniently as possible. All the exponential effects we discussed above will hit you even stronger.

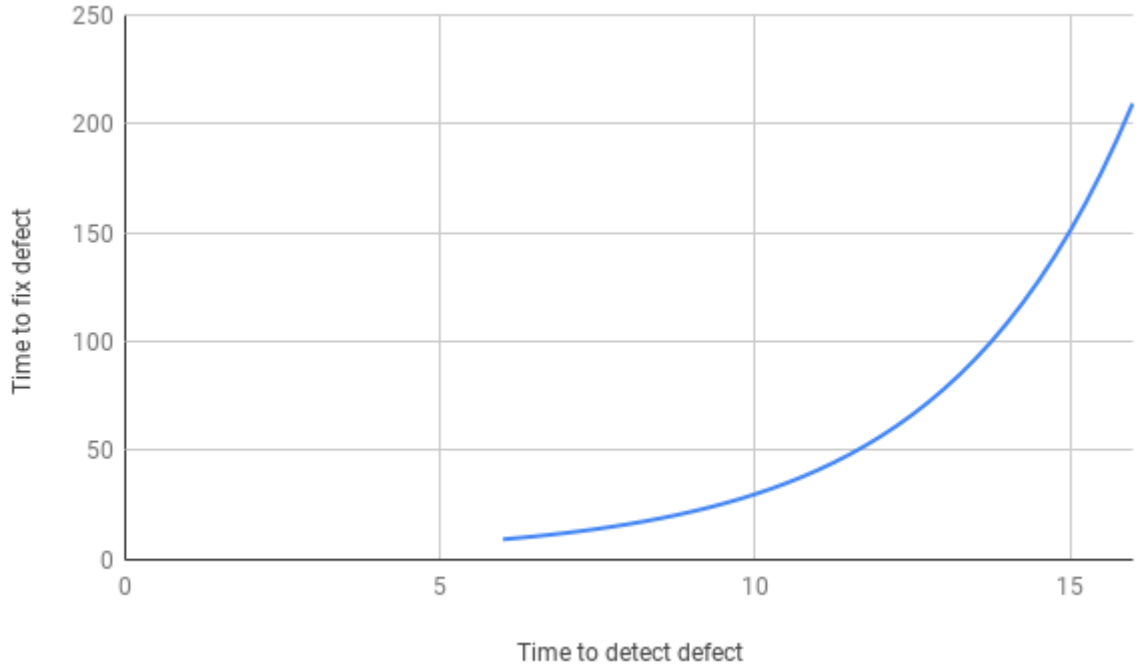


Figure 2. Delaying feedback exponentially increases fix time over detection time

The only solution to this problem is to make your build faster.

1.10. The problem grows with your success

The problems described here will grow as does your number of developers, code bases and repositories.

Only the practice of developer productivity engineering will put you in a position to improve or at least maintain your current productivity and truly practice CI/CD.

Even worse is the problem of unhappy developers. Most developers want to be productive and work at their full potential. If they don't see an effort to turn an unproductive environment into a more productive one, the best ones look for other opportunities.

A discussion between executives

There was a fireside chat a couple of years ago with an executive of a wall street bank and an engineering leader from a bay area software company.

The executive said something along the line of "If I had only the quality of your software engineers".

The bay area company was very developer-productivity-focused and her reply was: "Guess where we got our developers from? From organizations like yours".

2. Faster Builds by Doing Less with Build Caching

By Hans Dockter

The concept of build caching is pretty new to the Java world. It was introduced by Gradle in 2017. Google and Facebook have been using it internally for many years. A build cache is very different and complementary to the concept of dependency caching and binary repositories. Whereas a dependency cache is for caching binaries that represent dependencies from one source repository to another, a build cache caches build actions, like Gradle tasks or Maven goals. A build cache makes building a single source repository faster.

A build cache is most effective when you have a multi-module build. Here are examples for multi-module build declarations for Maven and Gradle.

Section in parent pom.xml

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

Section in settings.gradle

```
include "core"
include "service"
include "webapp"
include "export-api"
```

There are many reasons why you want to have multi-module builds even for smaller projects. They introduce separation of concerns and a more decoupled codebase with better maintainability as it prevents cyclic dependencies. Most Maven or Gradle builds are multi-module builds. Once you start using a build cache, more modularization will also enable much faster build and test runs.

When you run a build for such a multi-module project, actions like compile, test, javadoc, checkstyle, etc are executed for each module. So in our example above, we have four src/main/java and src/main/test directories that need to be compiled. For all four modules the unit tests are run. The same goes for javadoc, checkstyle, etc …

In the case of Maven, the only way to build reliably is to always clean the output from the previous run. This means that even for the smallest change you have to rebuild and re-test every module from scratch:

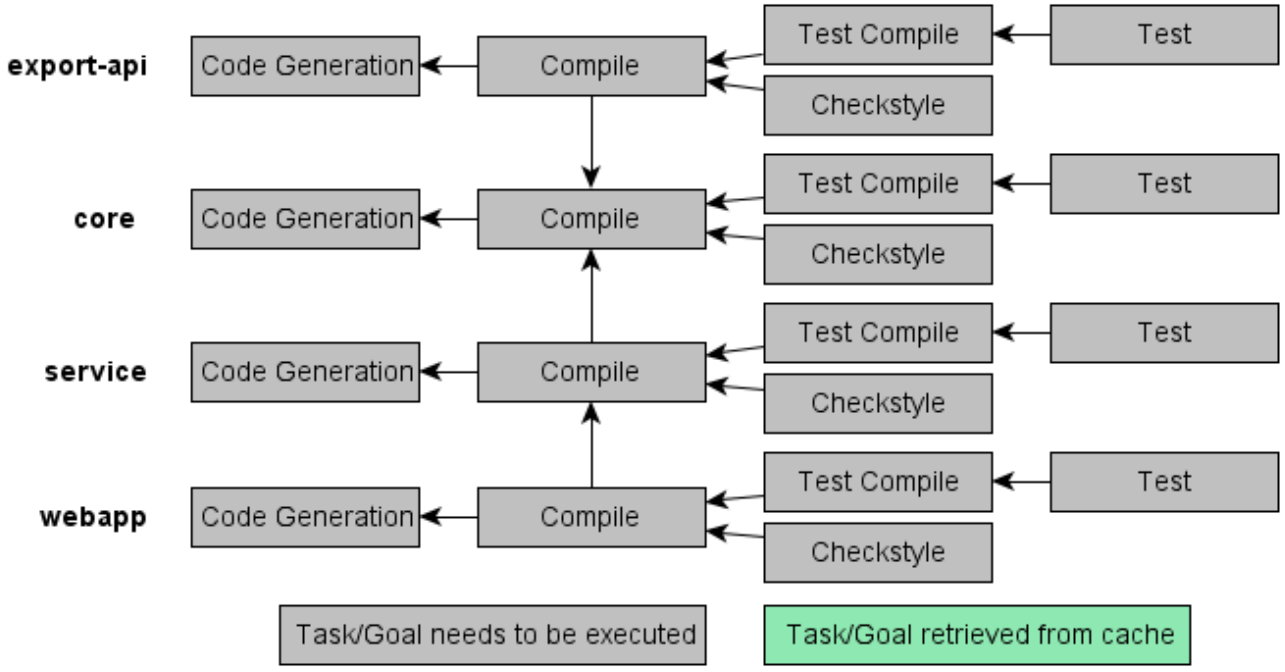


Figure 3. Rebuilding everything from scratch. Arrows indicate dependencies between tasks/goals

Gradle is more advanced than Maven. It does not require you to clean the output of previous runs and can incrementally rebuild the output depending on what has changed. For more details see: Gradle vs Maven: Performance Comparison

But in the case when you switch between branches or when doing clean CI builds you will need to build everything from scratch.

2.1. How does a build cache work?

Let's say you have built the project and after that you are changing a line of code in the export-api module from the example above. You add an argument to a public method and we assume no other module has a dependency on export-api. With such a change and the build-cache in place you need to run only 20% of the goals or tasks.

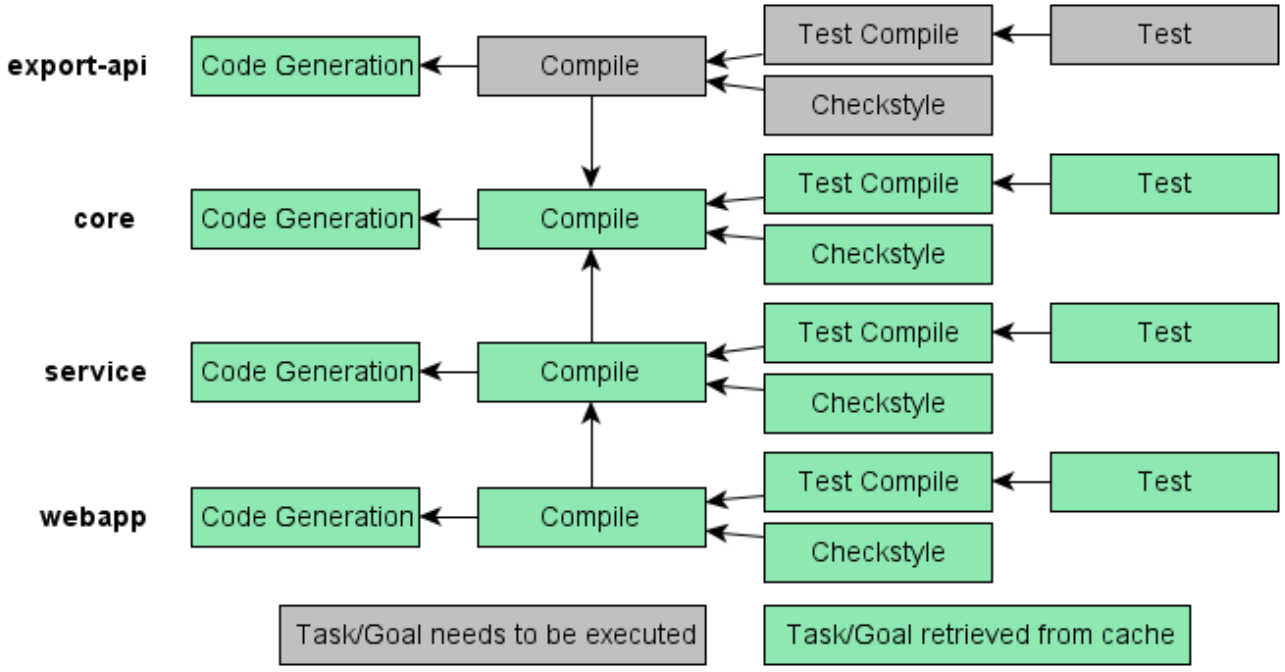


Figure 4. Changing a public method in the export-api module

How does this work? All of those 20 goals have inputs and outputs. For example the compile goals have the sources and the compile classpath as an input as well as the compiler version. The output is a directory with compiled .class files. The test goal has the test sources and the test runtime classpath as an input, possible also other files. The outputs are the test-results.xml files.

What a build cache does is to hash all those inputs for a particular goal and then calculate a key that uniquely represents those inputs. It then looks in the cache if there is an entry for this key. If so, the entry is copied into the Maven build output directory of this module and the goal is not executed. The state of the Maven output directory will be exactly the same as if the goal had been executed. Copying the output from the cache though is usually much faster than actually executing the goal or task. If an entry for a key is not found in the cache, the goal will be executed and its output will be copied into the cache, associated with the key. In our example, four goals belonging to the export-api module had a new key and needed to be executed. The compile goal of the production code, because one of its inputs, the source directory, has changed. The same is true for checkstyle. The compile goal for the tests has a new key because its compile classpath changed. The compile classpath changed because the production code of export-api was changed and is part of the compile classpath for the tests. The test goal has a new key because its runtime classpath has changed for the same reasons.

Let’s look at another example. Let’s say you have built the project and are then adding a parameter to a public method of the service module. Webapp has a dependency on service.

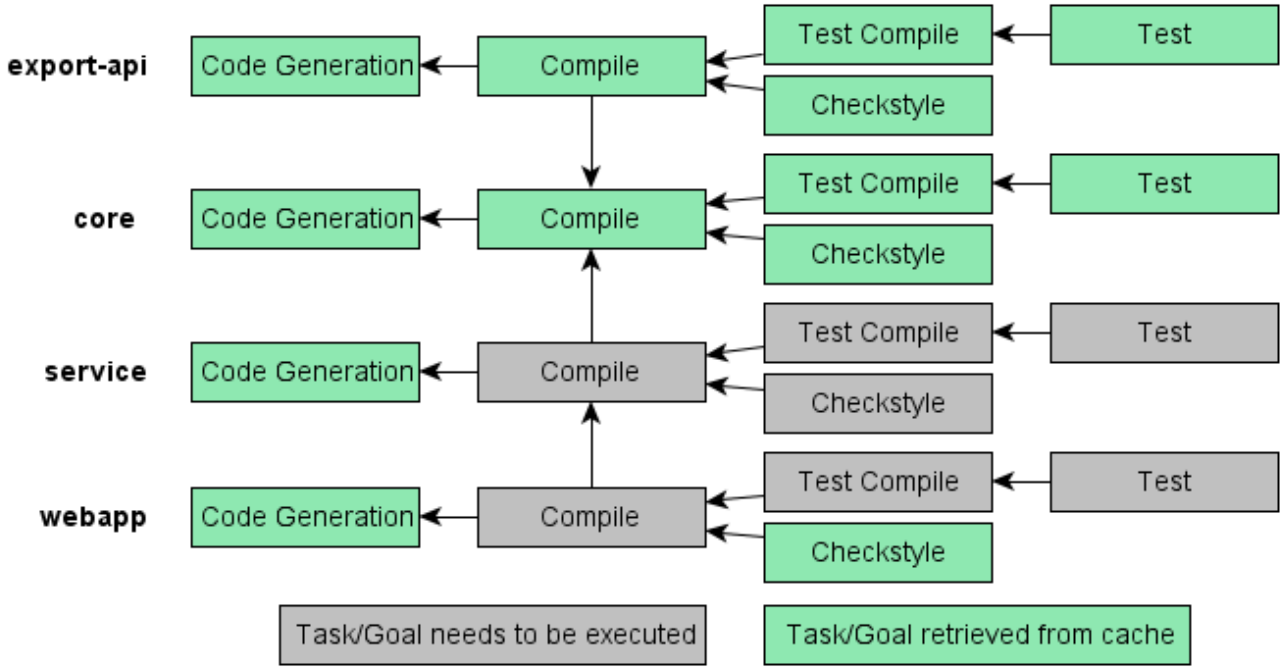


Figure 5. Changing a public method in the service module

Now not just the goals for the module that has changed needs to be re-executed, but also for the module that depends on this module. This can be detected via the inputs of the goals. Because of the change in the code of the service module, the classpath of the webapp compile and test compile goal has changed as well as the runtime classpath for its test goal and thus all those goals need to be executed. Compared to rebuilding everything, you are still only executing 40% of the goals.

Let's use the same example as before but instead of adding an argument to a public method in the service module, we are changing just something in the method body. A smart build cache can now do further optimizations:

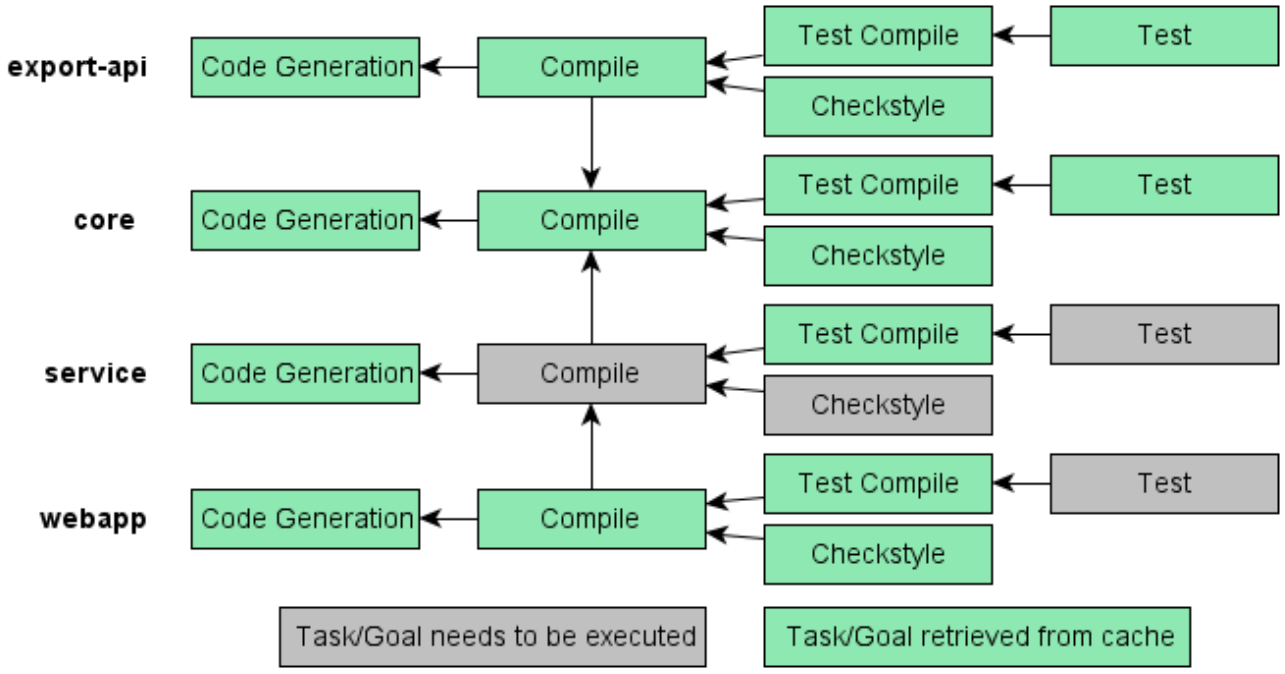


Figure 6. Changing an implementation detail of a method in the service module

The cache key calculated out of the compile classpath only takes the public API of the classpath items into account. An implementation change will thus not affect that key, reflecting the fact that any such change has no relevance for the Java compiler. Because of that, the execution of three compile goals can be avoided. For the runtime classpath of the test goals, implementation changes in your dependencies are obviously relevant and lead to a new key resulting in executing the test goals for service and webapp. With this optimization only 20% of the goals need to be executed.

Let's look at another example. Let's say every other module depends on the core module and you change an implementation of a method in core:

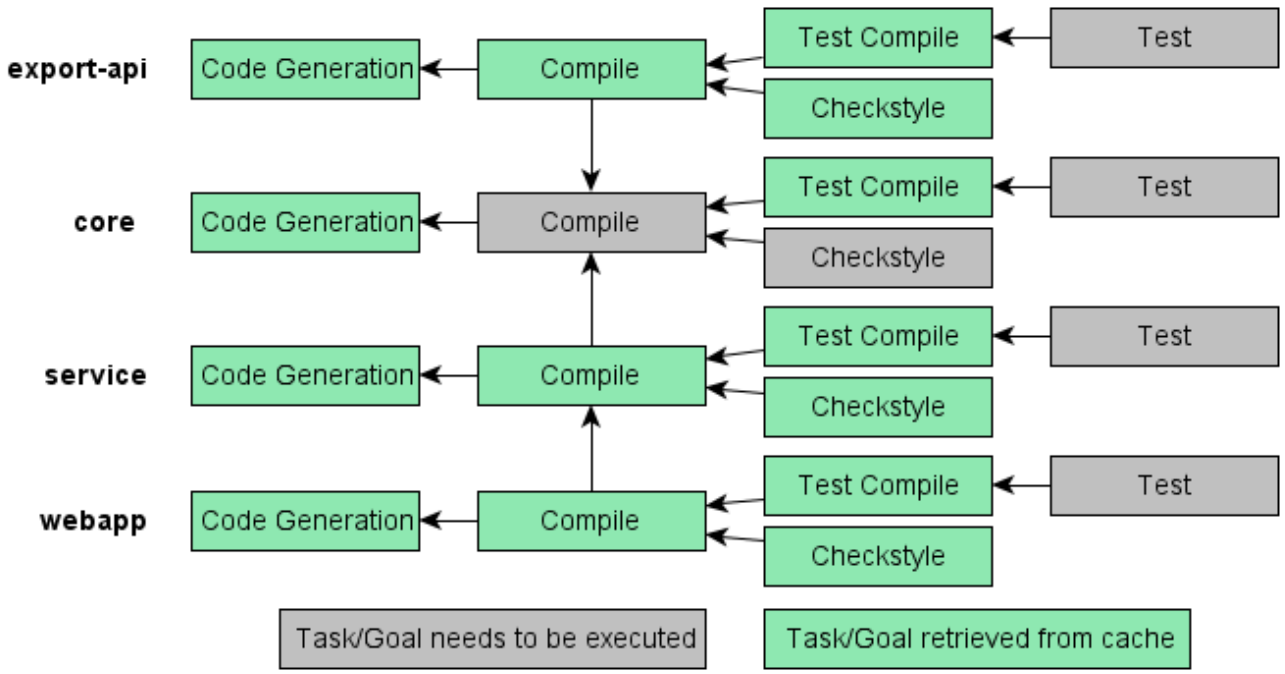


Figure 7. Changing an implementation detail of a method in the core module

This is a pretty invasive change but even here only 30% of the goals need to be executed. Granted, executing the test goals will probably consume more time than the other goals, it is still a big saving in time and compute resources.

The worst-case change would be adding an argument to a public method in the core module:

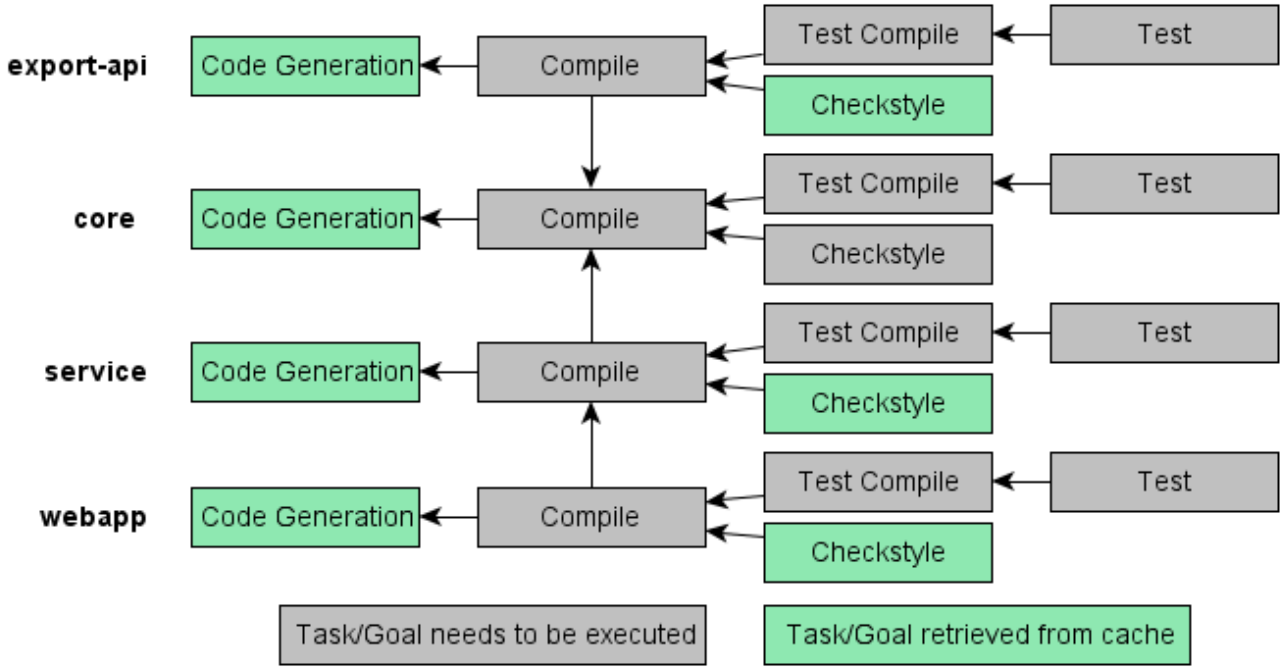


Figure 8. Changing an public method in the core module

Even here to get relevant savings from the cache as only 65% of all the goals need to be executed.

2.2. Local vs remote build cache

There are two types of build caches. One is a local cache. It is just a directory on the machine that is running a build. Only builds that run on that machine add entries to that cache. A local cache is great for accelerating the local development flow. It makes switching between branches faster and in general, accelerates the builds before you commit and push your code. Many organizations weaken the local quality gate because of the long build times. A local cache is key to strengthen the local quality gate and get more feedback in less time before you push your code.

The other type is a remote cache. A remote cache node shares build output across all builds in the organization.

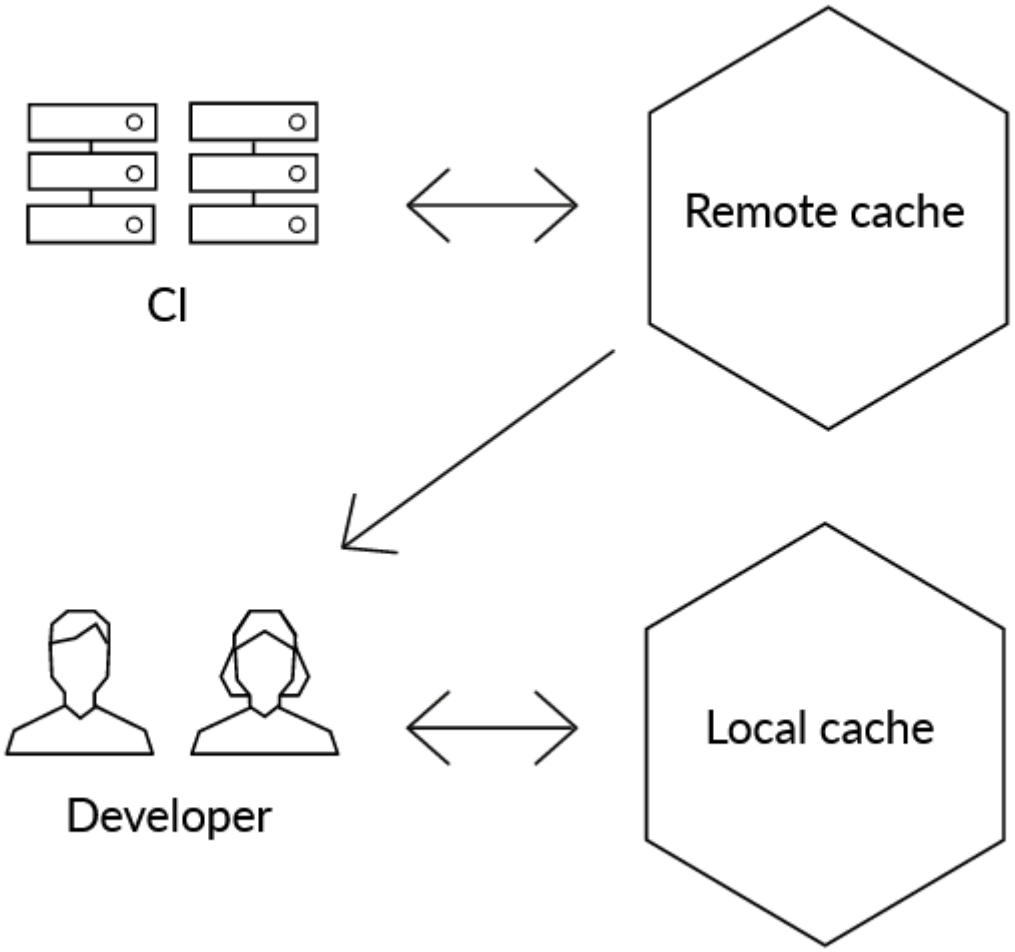
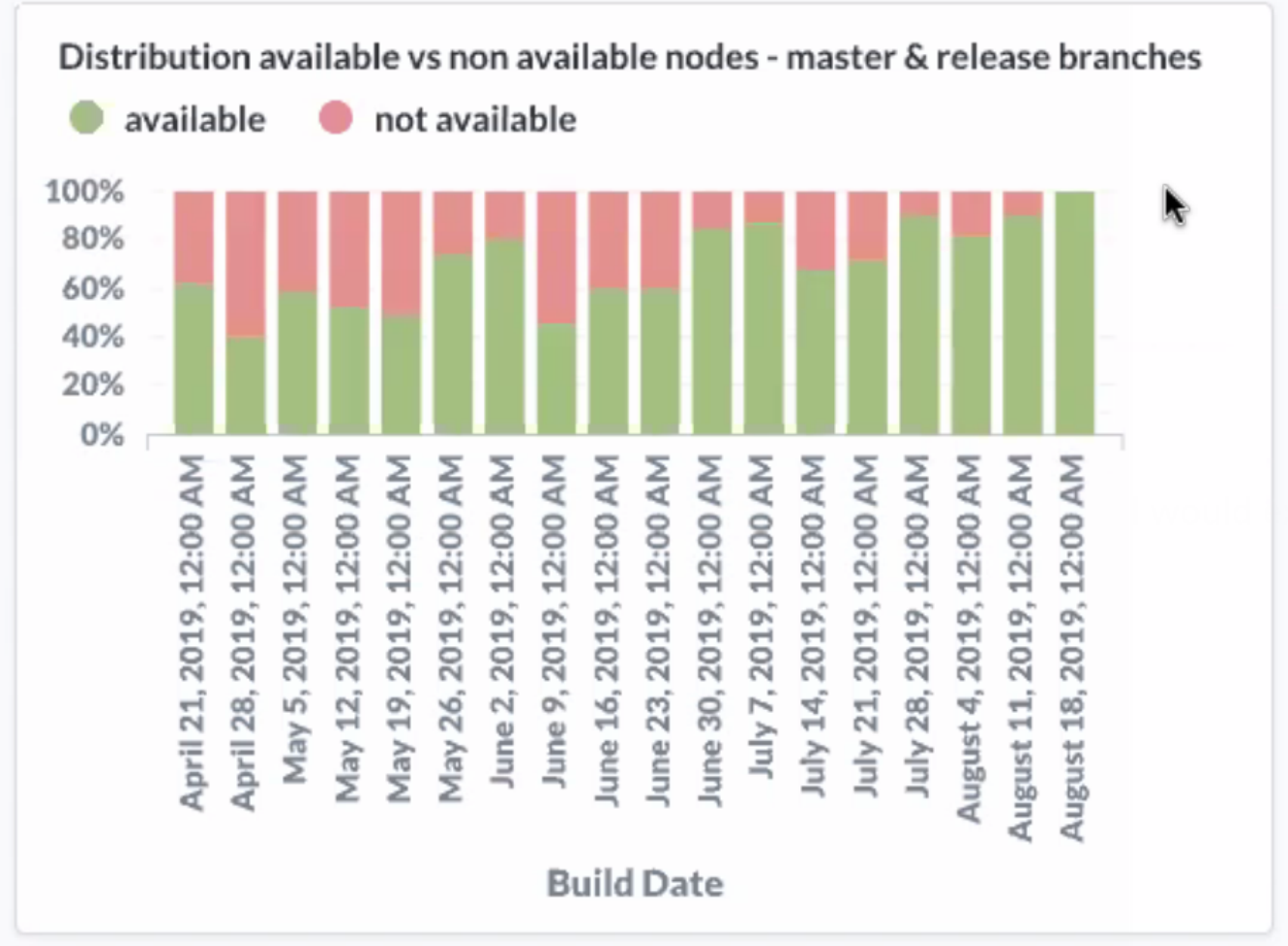


Figure 9. Cache Node Architecture

Local builds write into the local cache. Usually, only CI builds are writing to the remote cache while both, local as well as CI builds are reading from the remote cache. It dramatically speeds up the average CI build time. You often have a pipeline of multiple CI jobs that run different builds against the same set of changes. For example a quick check job and a performance test job. Often those jobs run some of the same goals and a build cache makes this much more efficient.

A build cache does not just improve build and test execution times, it also drastically reduces the amount of compute resources needed for CI. This can be a very significant economic saving. But it also helps with CI build times. Most of the organizations we talk with have a problem with CI queues, meaning new changes are piling up and there are no agents available to start the build. A build cache can significantly improve agent availability. The below diagram is from one of our customers as they started to introduce the cache and optimized its effectiveness.



2.3. How effective is a build cache?

As already mentioned, if all your code and tests live in a single module you will not get many benefits from a cache. You will get some though. Especially on CI where you often have multiple CI jobs running against the same set of changes. But the majority of the benefits come once you have multiple modules.

Even with a few modules, the benefits will be significant. With many modules, things can become very efficient. For larger multi-module builds often 50% of all the modules are leaf modules, meaning no other module depends on them. If n is the number of modules, rebuilding such a leaf module will only take roughly 1/n of the time compared to building the whole project. A build cache allows you to work much more effectively with larger source repositories if you have some modularization. Investing in further improving the modularization will not reduce the technical debt when it comes to evolving your codebase but will give you immediate benefits when it comes to build performance.

A build cache needs to be continuously monitored when it comes to its effectiveness. One part of this is the caching infrastructure, including things like download speed from the remote cache nodes as well as enough storage space to not lose cache entries that are still in use.

Another important aspect is the reproducibility of your build. Volatile inputs are the enemy of caching. Let’s say you have a goal or task that always add a build.properties file to your jar that contains a timestamp. In such a case the input key for the test action will never get a cache hit as the runtime classpath always changes. You have to be able to effectively detect such issues as they

can drastically reduce your cache effectiveness.

2.4. Summary

Nothing is faster than avoiding unnecessary work. For our own projects, we can see a spectacular on average 90% time saving because of reusing previous build output via caching. We see savings at least over 50% all across the industry once organizations start adopting the build cache. It is must-have technology to address the problems outlined in "Why are long feedback cycle times poisonous?"

3. Dependency Management

By TBD

Managing dependencies. Behind this simple expression lies one of the biggest lies of our industry: composing software is easy. The reality is far from this dream and we only have to look at the number of "package managers" or dependency engines to realize how difficult it is: `aptitude`, `rpm`, `npm`, `yarn`, `maven`, `gradle`, `ivy`, `pip`, ... name your own! There are *many* tools available, often specific to an ecosystem, and some ecosystems battle to find the "right" one.

Dependency management is such a hard problem that some build tools willingly ignore it and let the developer deal with this problem alone: *Ant*, for example, is a build tool which has no dependency management whatsoever. Developers are required to put their dependencies *file by file* into a *lib* directory, without any convention. While this certainly has some advantages, building applications at scale with this technique doesn't work. That's why better Java dependency management engines were born in Maven 2 and Gradle.

3.1. The Stone Age of Dependency Management

What do all dependency management engines have in common? They are trying to save the user from the pain of having to declare what should be installed for an application to *work*. "Work" is already difficult to define: what you need to build an application can be different from what you need to run it. For example, a developer will need C++ header files to build an application (often found in `-dev` packages), but the end-user does not need them to execute the application.

More importantly, the actual artifacts that the engine needs to bring (`.so` files, `.jar` files, ...) are dependent on the context:

- what *target platform* is going to be used? (`i386`, `arm64`, Java 8, Java 13, ...)
- what are the dependencies *used for* (compilation, runtime, tests, ...)

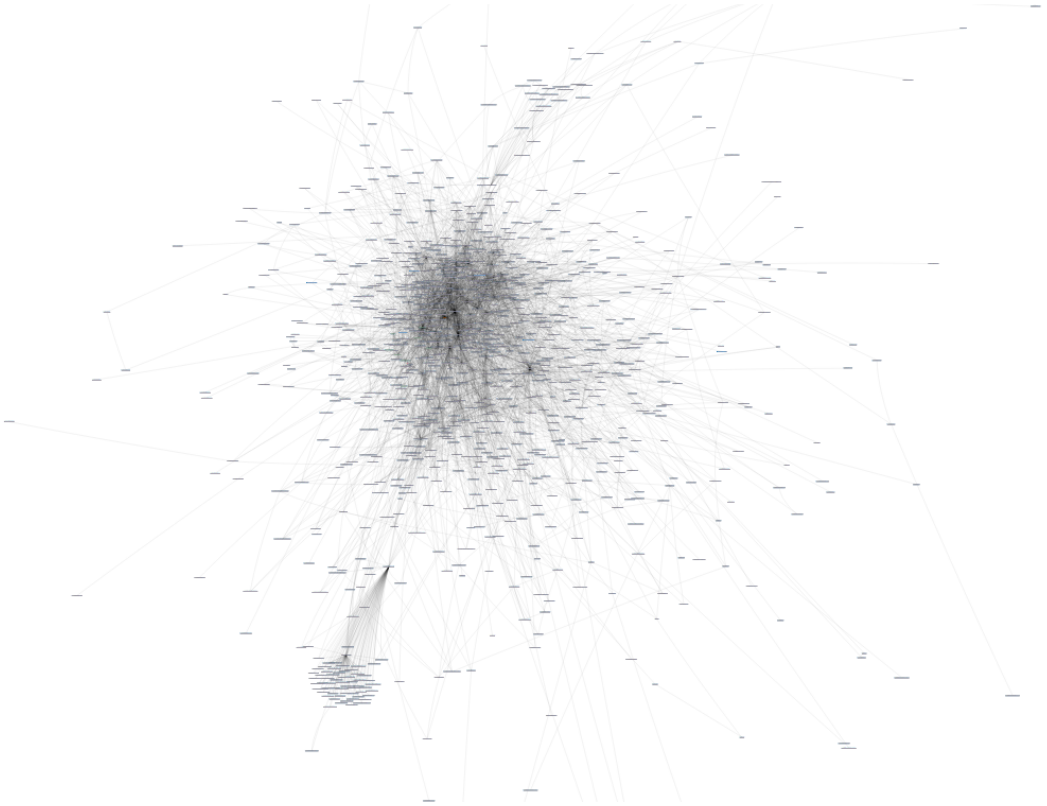
Where does the complexity come from? Software composition. Because, of course, nobody wants to rewrite everything from scratch. Modern development relies heavily on frameworks and libraries which are shared between a number of applications. The explosion of Open Source Software, the rise of platforms like GitHub and public repositories like Maven Central makes it extremely easy for developers to share code which can be reused either in source or binary form.

While these seem like a huge step forward and make it look easy to reuse software components, the devil is in the details and with the growth of the project comes more and more challenges to address:

- Should you build your dependencies from sources or use binaries?
- How do you deal with conflicts, for example when two libraries depends on different versions of the same dependency?
- How do you make sure that everything *works together*?
- How much do you *trust* the sources?

- Can you *trust* the binaries you can find in Maven Central or other repositories?
- How do you deal with bugs and vulnerabilities, by definition discovered *after* a release?

The picture below gives a sense of what we call dependency hell. It's a real world dependency graph of an application from a famous Silicon Valley company:



Having such a large dependency graph is common and one must realize that what we describe here can exist in different forms:

- a static, compilation, dependency graph
- or a runtime, micro-services based, dependency graph

Done incorrectly, dependency management can have a dramatic impact on developer productivity, by requiring the developer to fix issues they shouldn't even have to care about: - a build passes on a laptop but breaks on CI because of the use of a local repository or a mirror - a build suddenly breaking without any code change because a dependency was upgraded automatically (new releases) - multiple implementations of the same API end up in the same dependency graph (e.g, logger bindings) - use of source dependencies with the consequence of slow feedback time because everything has to be rebuilt again and again - having to arbitrarily split modules into pieces for the sake of build performance - implementation details are not identified as such and are used as if they were public API, making it impossible to change them later

All those issues are well documented but we are still killing the productivity of millions of developers by having them solve the same problems again and again.

3.2. Let's Move to the Bronze Age

The good news is that it doesn't have to be that way. Dependency management is, surprisingly,

about *managing dependencies*. This means, making sure that a developer expresses requirements, and that the a tool, a dependency management engine, fullfils those requirements. Today, most of the problems in dependency management are on one side related to the poor modeling of those requirements (e.g. why does a client need this dependency?) and, at the other end, poor modeling of what *variants* a library offers.

Let's take a simple example: most dependency management tools today let users describe a dependency in terms of versions. For example, a user would say:

> I need library `awesome-lib` version 1.2

but what does it *mean* to say 1.2? Is it:

- the latest release the moment the dependency was added to the build file?
- a version the user was familiar with and didn't care checking if there were newer versions?
- not the latest but the user knew 1.2.1 wouldn't work because it has a bug?
- copy/pasted from StackOverflow?
- etc...

In fact, there are *many* things to say about such a dependency declaration. First, it forces users into saying "I need `awesome-lib`", when in theory, they should really care about *features*, for example, "I need something which can compute as many decimals of π as I want to". Second, users shouldn't really have to declare a version either, but they do, it should convey more information.

With this in mind, we designed the Gradle Build Tool to offer a much richer way to express requirements and therefore allow the developer to express requirements in terms of constraints. For example, users would be able to express:

> Any version between 1.0 and 2.0 excluded should work. However, if nobody cares, choosing 1.2 is fine. Oh, by the way, don't use 1.2.1 it's buggy!

By expressing this information in terms of constraints, a modern dependency management engine can select the *best solution*, if there is one. If there's not, then it should *fail early*, that is to say as soon as possible during the development process, with the assumption that the earlier a build fails, the lower the cost of fixing it. More importantly, if there is a solution for all the constraints, then users can be relatively confident that the application is going to work, which can remove some of the burden from testing.

Something which is also critical to recognize is that not all software components are the same. For example, a traditional monolithic Java application on Apache Tomcat isn't the same thing as a Java utility library. They are also different from a Quarkus micro-service, which is itself different from a Spring Boot micro-service and likewise a Micronaut one. All of these components have different requirements, deployment strategies, build strategies, dependencies, ... So why would the build tool force everyone to use the same model for all of them? As an example, tools like Apache Maven, despite providing dependency management, do not model the differences between such components. This means that they cannot express properly what are the specifics of each and put a burden on both the producer of the component and their consumers.

For example, why would the build tool force the developer to declare dependencies on all the pieces of their framework, when it could just ask them to tell what *features* they use and add the dependencies for them? Why would the tool force the user to choose a *specific variant* of a library when the build tool (more specifically the dependency management engine) can make the decision for them? A typical example of this is the variants of a native library: depending on the target architecture, the consumer has to declare if they're going to use `i386` or `amd64` dependencies, but they also have to say if they want the *dynamically linked* or *statically linked* dependencies. All this information is *known by the engine* and shouldn't leak into the model adding a maintainance burden that nobody wants to pay.

Gradle, being a *variant aware dependency management* tool, is built to recognize that all software components are different and come with their own requirements. Long story short: a Java library is a software component. A C++ library is a software component. A platform is a software component. A Boot application is a software component. Everything is a software component, and everything needs a declarative model, with, potentially, its own publication model.

By making the producer and the consumer models richer, the dependency management engine can make smarter decision and reduce the overhead of maintaining large dependency graphs.

Part 2 - Using Data to Make Builds Faster and More Reliable

Toolchains Need to Be Monitored and Managed to Be Fast and Reliable

Improving Developer Experience with Proactive Investment in Reliable Builds

The Black Pit of Infinite Sorrow: Flaky Tests

4. Toolchains Need to Be Monitored and Managed to Be Fast and Reliable

By Hans Dockter

With very few exceptions, organizations have no relevant and actionable insight into the performance and reliability of the toolchain their developers are using. The tech industry that has enabled other industries to become so efficient has so far not applied those practices to itself.

The unsung heroes

In my workshops, I frequently run the following exercise: Assume the code base in your organization grows by over 30% and despite the growth, you make building and testing faster by 10%.

- Who would notice?
- Who would praise you?
- Who would know?

The common answer I get is usually 'no one' to all those questions. Even the build engineers themselves don't have the data to measure the impact of their work. To put this into perspective, for a 100 person developer team, we are probably talking about a multi-million dollar annual savings. And yet this doesn't show up on anyone's radar. Companies don't even have the most basic metrics to measure and improve developer productivity.

4.1. What is toolchain reliability?

What exactly do we mean with reliability? A toolchain is unreliable whenever:

- It is asked to do something and fails without producing any result. For example, compilation or test runs fail when there is an out of memory exception or a dependency can not be downloaded because the repository is down. We call these non-verification failures.
- The toolchain produces the result it is asked to, but the result is not correct. That could be a flaky test but also an archive that is not assembled correctly. If the result is a failure, like a flaky test, we call this also a non-verification failure.

A build that failed because a test detected a code defect is a reliable build! We call this a verification failure.

Reliability issues kill the morale

Frequent reliability issues have a quantifiable impact on feedback cycle time and developer productivity. Their negative effects go way beyond that though. Imagine you are repairing something under a sink. You had to squeeze yourself into a very tight cupboard and are lying there in an uncomfortable position. It took you minutes, requiring a lot of patience, to get the screwdriver attached to a wobbly screw. Finally, you start to loosen the screw and then the screwdriver breaks. You have to start all over again including getting out and in after having picked up a new screwdriver. Now imagine that would be your job and that happens to you every day, multiple times a day while you have pressure to repair a certain number of sinks. That might give you an idea of how many developers are feeling today. It is extremely frustrating to be held back by flawed tools while doing challenging work.

Reproducible builds

Let's first define what we mean with reproducible builds:

A build that is reproducible always produces the same output when the *declared* input has not changed.

So whenever the toolchain relies on something with an undefined state which affects the output, the build is not reproducible. For example, as part of the build, a source code generator might be called. The build runs whatever version is installed on the machine. So if you change the installed version you might get a different output without changing the configuration of the build.

Individual Classpaths

We had a customer who had a build that needed a Tomcat library on the classpath of their application. For whatever legacy reason in the past, the build was configured to pick up the Tomcat library from the Tomcat server that was locally installed on the machines. People were no longer aware that this was the behavior. It didn't help that the Tomcat library shipped with Tomcat does not have the version number as part of its name. This had not been causing any known issues. But it is a timebomb waiting to explode! When it does, it will cause very hard to debug issues during development or production. The organization discovered the issue when they started to use build caching and were wondering why they were not getting as many cache hits as they expected. As they also started to gather comprehensive data about every build execution, they could now easily detect the problem. As a side note, build caching enforces build reliability as it is less effective otherwise.

Builds that do not provide reproducible results are not necessarily unreliable but they are a reliability risk in the (possibly very near) future. Changes to the environment might have indirect side effects on the behavior of your build, tests or production code. When this happens, this often will be extremely difficult to triage. Version control will tell you that nothing has changed as the build configuration was not touched. Compare this to a state where the input is declared. In the case of an environment change, the build will at least fail with a clear failure message, for example,

that a particular version of the source code generator is required but not installed on the machine. It's even better if the build automatically installs the required version.

4.2. Performance and reliability

Performance and reliability are closely related. As discussed in Every Second Counts, we have the following relationship:

```
average time for a successful build = Average build time + (failure frequency * average build time) + average debugging time

context switching frequency = failure frequency * number of builds * 2
```

Obviously non-verification failures add to the failure frequency and thus waiting time and context switching frequency. Additionally, non-verification failures have a much higher average debugging time than verification failures and thus substantially affect the average time until you have a successful build.

4.3. No data, no good performance, no good reliability

The software toolchain is complex machinery that includes the IDE, builds, tests, and CI and many other components. It has complex inputs that are always changing. Every other industry with such complex production environments has invested in instrumenting and actively managing this environment based on the data they are collecting. The data serves two purposes. One is to see quickly trends and pathologies. The second is to effectively figure out the root cause. Let's take a chemical plant that produces some liquid as an example. One important trend that will be observed is how much of that liquid is streaming out of the vessel in which the liquid is produced. If they see a drop in that number they react by looking at the parameters of the vessel environment, for example, the temperature, pressure, and concentration of certain chemicals. That will usually allow them to figure out or significantly narrow down the root cause for the drop in production. We need similar instruments for optimizing the software production environment.

Small things matter

One of our customers has a large Maven build. For years they were suffering from slow builds. Once they started instrumenting their toolchain they learned the following. Just building the jar files when running their Maven build took 40 minutes on CI. When they compared this with the data they had from the local build, they saw that locally this takes only between 1 and 2 minutes. The reason was that locally they were using SSD drives. So they could achieve a massive CI speed up just by replacing the hard disks.

Toolchain performance and reliability is sensitive to its inputs. That starts with infrastructure like the hard disks in the example above or networking. This also includes the build cache nodes, the CI agents and binary repository manager. For example, dependency download time is often a bottleneck for CI builds. Yet hardly any organization has an idea of how much time they spend on average on downloading dependencies and what the network speed is when this is happening.

The code itself is also a complex input for the toolchain. Developers, for example, add annotation

processors, new languages, and new source code generators that might have a significant impact on the toolchain performance (and reliability). And yet I haven't met a single company who would be able to tell me how much of the compile-time is spent on a particular annotation processor or how much faster on average a compiler for one language is compared to the compiler for another language for *their* particular codebase. Just changing the version of an already in use compiler or annotation processor might introduce significant regressions.

Other factors are memory settings for the build and test runtimes, code refactorings that move a lot of code between modules, and new office locations. The list goes on and on.

Surprises are everywhere

We worked with an insurance company and looked at their build and test times. Traditionally they only looked at CI build times. Once we started collecting data also from developer build and test runs we found something very interesting. We saw local builds that took more than 20 minutes, for the same project that was built and tested on CI in less than a minute. Digging in deeper we found out that those builds come from developers that work from home and use a network share for the build output directory.

4.4. Supporting developers

In most organizations, most of the performance regressions go unnoticed. But even if they are noticed they often go unreported. Why? Because reporting them in a way that they become actionable requires a lot of work on the side of the reporter. They need to manually collect a lot of data to let the support engineers reason about the problem. If the issue is flaky they need to try to reproduce it to collect more data. Finally, often reported performance issues do not get addressed which lowers the motivation to report them. Why do they not get addressed? Often the data that is provided as part of the report is not comprehensive enough to detect the root cause, particularly with flaky issues. Furthermore, the overall impact of this problem on the whole organization is hard to determine without comprehensive data. That makes it hard to prioritize any fix.

When a performance regression is really bad, it will eventually get escalated. That usually happens after the regression has already created a lot of downtime and frustration. Because there is no good data available it will also usually take longer than necessary to fix the regression.

All in all, the average build and test time is much higher than necessary and continuously increasing because the toolchain is not instrumented and is thus impossible to be kept in an efficient state.

Even the experts struggle

We were working with a potential customer to measure the impact of our build acceleration technology. It was a large team so the business opportunity for us was significant. They were super excited to start the trial as they were suffering a lot from performance issues. When they tried out our technology, they told us that build and test time has *increased* by a factor of 5 when connected with our supposedly acceleration technology. We were very surprised by that and doubted that their plain Maven build and the Maven build that connects with our product were run the same way.

But they were very clear that those two builds were definitely run the same way. And we reached a point where they were getting annoyed by us questioning that fact. Those were the build experts at that organization. We were wondering whether we have to fly in some of our engineers in to dive into this. Before that though, they agreed to create build scans, which is our data collection technology, for the two builds and shared them with us. And when we compared the two build scans we could see that the plain Maven build was run with 128 threads while the one using our technology was only run with one thread. No wonder!

The reason for this was tricky. They created a separate branch for the Maven build that uses our product. This branch shared the same parent directory as the plain Maven build. For connecting with our product you have to apply a Maven extension. To apply a Maven extension you have to create a `.mvn` directory in the project root directory and add an `extensions.xml` file to it that defines what extensions you want to apply. Maven only looks for the first `.mvn` directory in the directory hierarchy. They had a `.mvn` directory in the parent directory with a `maven.config` file that sets the number of processes to be used to 128. The plain Maven project had no `.mvn` directory and thus picking up the configuration from the parent directory.

The key for effective troubleshooting is not to rely on human assumption of what is happening but to have an actual snapshot of what has happened. When it comes to toolchain support we rely way too often on what people hypothesize. They think they have assigned a certain amount of memory. They think they have assigned a certain number of processes. But even experts are often wrong with that. For us, not finding this problem could have cost us a large amount of money in lost revenue. Not detecting performance problems in your organization is even more expensive, as the cost goes way beyond any license costs. Stop paying this cost!

For reliability issues, the story is a bit different. Failures are always noticed by at least one individual, the developer who is running the build and tests locally or the developers whose changes are causing the CI job to fail. The very first analysis they do is usually to decide whether this is a verification or non-verification failure. This is often not easy. We talk about this in detail in Developer Experience. Once the developer thinks it is a non-verification failure, they should report it to the people responsible for maintaining the toolchain. But reporting such issues and asking for help is painful for the reporter as well as for the helper for the same reasons it is painful to report and help with performance regressions as described above. But as they often block people they can not be simply ignored. So to avoid reporting them, developers run the build and test again and hope that the issue is flaky. If it still fails they might run it again with all the caches deleted. This is

extremely time consuming and often very frustrating. Only when they can not get to a successful build that way, they ask for help informally or by filing an issue. When you talk with the engineers who support developers with those issues as the build and CI experts, they most of the time have no idea about the root cause either. So guess what they often do as the first thing? Run the whole thing multiple times to see if the issues go away.

Many managers do care deeply about the job satisfaction of their developers. But even those who feel less strongly about that aspect should be very concerned about reliability issues with the toolchain. Features are not moving forward and the time from code committed to code successfully running in production is heavily affected by such issues while at the same time human and machine resources are wasted.

Data is the obvious solution

The biggest problem in supporting developers with performance or reliability-related toolchain incidents is that no comprehensive set of data is available that lets you effectively determine the root cause. That is why data of each and every build and test execution need to be collected. You never know upfront whether you will need it or not. If you have flaky issues that are hard to reproduce it is the only way you can find the root cause. This data has to be easily accessible and explorable so that everyone can look at it. Having the data of each and every build execution also allows people to compare what has changed. Let's say you can not easily determine the root cause for a failing CI build and are not even sure whether it is a verification or non-verification failure. But you know that a similar build was successful 15 minutes ago. You now have the data to compare the two and the comparison might tell you that different versions of dependencies have been used, or a different version of the compiler, etc …

4.5. Avoiding incidents is even better than efficiently supporting them

While you will always have issues that require direct support, one of the most important and game-changing practices all this data will allow you to do, is to actively observe and manage the toolchain to reduce the number of non-verification failures and continuously and pro-actively optimize the performance. Once you have comprehensive data of each and every build and test execution from IDE to local builds to CI, you can see trends and can compare and establish a baseline. Trends and comprehensive data together will enable you to:

- Immediately detect regressions and see who is affected
- Detect the root cause without the need to reproduce the problem
- Fix the problem before it causes a lot of harm and gets escalated to you
- Prioritize your work based on data and quantifiable impact
- Continuously make your toolchain faster and more reliable

We will talk in detail in Developer Experience how this can be done for non-verification failures.

Lunch discussions

I have encountered many lunch discussions at organizations I was visiting about how much the virus scanner is or is not slowing down the developers or how much faster local builds would or would not be with Linux machines compared to windows because of the faster file system. Those discussions go on for years without a resolution because no one can measure the impact and thus there is no resolution.

Many heavily regulated industries have a virus scanner enabled on developer machines. If developers reach out to management to deactivate the virus scanner for the build output directory with the argument that things would be faster, this often falls on deaf ears because there is no number attached. The reaction is, ah, the developers are complaining again. If they can provide data like that they have 1,300,128 local builds per year and the virus scanner introduces an additional 39 seconds of waiting time per execution they can make a strong and quantifiable case saying that 8 engineering years are wasted because of the virus scanner. That discussion will most likely have a completely different dynamic.

When I talk with teams that are responsible for the toolchain, I'm often surprised by their approach to performance. They are completely incident driven. If someone complaints they try to improve the problem that was reported. If no one complains, they consider everything to be fine and nothing needs to be done. But they are the experts. They should not wait until people are so miserable that they are complaining about a situation. They should turn into developer productivity engineers who understand how important toolchain performance is for the productivity of the developers. Their focus should be to get the performance to levels beyond what most developers would expect is possible.

Imagine you are in charge of an online shop. Would you wait to improve the page load time until some of your visitors start complaining about it? And after you have improved it, would you do nothing until they complain again? Probably not, as you would lose tons of business along the way. Less will be bought and customers will move to different online-shops. This often happens silently. Similarly, developers will produce fewer features and will look for different job opportunities if you keep them working from their true potential because of a slow toolchain.

5. Improving Developer Experience with Proactive Investment in Reliable Builds

By Sam Snyder

Developing software is a bit like carpentry. You have a variety of specialized tools that you use to measure, shape, connect, and polish your creations. But no self-respecting carpenter would use a hammer that, even when perfectly aimed, occasionally pirouettes around the head of the nail to smash their thumb. Yet we software developers deal every day with toolchains that unexpectedly, unfairly smash our metaphorical thumbs.

Our build automation exists primarily to save us the drudgery of verifying whether or not our code works as intended. In a more ideal world, the build would always do this with perfect accuracy and only fail because a real problem was detected with our code. We call such legitimate build failures "verification" failures.

In reality, not all build failures are verification failures. Your build is software - and often quite complex software - with all the opportunities for problems that come with software. There might be bugs in the build logic. Or an artifact repository might be unavailable. Or you have flaky tests that sprinkle the verification process with uncertainty and frustration. Or a developer may have misconfigured their build environment. We call these and other undesired build failures non-verification failures - aka cursed-hammer-thumb-smashes.

In most organizations when a build fails it is the developer who initiated it who deals with the consequences. There are good and bad reasons for this. There are many verification failures where the reason is obvious and easy to fix for the person who encountered it. But there are also many failures where it is unclear to the developer not what has caused it but also what type of failure this is and who is responsible for fixing it.

There are non-failures that look like verification failures, like flaky tests or runtime errors from different versions of the same dependency on the classpath. There are verification failures where the developer thinks it is a non-verification failure. Such as when the build is using snapshot dependencies and the CI build picked up a later version than the local build but the developer has no straightforward way to figure this out.

The more unreliable the build and test pipeline is, the less a developer trusts whether a verification failure is actually a verification failure.

When we're unsure how to get our toolchain out of a bad state, developers often just try to run the build again. Often not because there is any indication that this will help, but because they don't know what else to do and they are blocked. If that doesn't work, we try clearing caches and eventually burn down the whole workspace and clone it anew. This wastes a lot of our valuable time and costs a great deal of frustration. Asking for help often requires knowing who to ask for help, which isn't always trivial in large organizations and large codebases.

And when we're in the position of being the helper, it's often very difficult to efficiently provide aid. The problem may only be occurring on a particular dev's environment - which might be down the

hall or on the other side of the planet. So triaging often begins with a game of 20 questions where the person asking doesn't necessarily know what context is important or how to reproduce the bad state. This can be exhausting to both the helper and the helped. I've seen helpers burn out and transition to different roles and the organization didn't even know how much it depended on that person to keep functioning until they were gone.

Complaint-driven development

At one software development firm I worked at, I got frustrated with poor build reliability and set out to do something about it. To start, we didn't have much in the way of telemetry, especially for local developer environments so the only signal available for prioritization was the intensity of complaints. One of my coworkers, let's call him Frank, would very loudly demand help when he was blocked.

Everyone knew when Frank was blocked by something. He'd go from chat room to chat room trying to find the relevant expert until he did. So Frank's problems would get fixed and Frank would get unblocked and get back to doing his work. And good for him. But once we set up Gradle Enterprise and had a detailed telemetry stream coming from every developer's build, I found out that while I had been helping Frank with something, 30 other developers had been suffering in silence with an unrelated problem.

I went to one of those developers and asked her why she hadn't let me know about the problem. She shrugged and said, "Builds are slow and unreliable, that's just how they are". The toolchain wasting time and productivity had become so normal it wasn't worth making any noise about. The best developers want to be productive, want to do their best work on your organization's most challenging problems. They may put up with sub-par tooling for a little while, but eventually, they will leave for organizations that provide a more productive environment.

Several developers did leave, citing poor developer experience as one of their motivations. Once we successfully implemented the data-based methods described in this chapter there was a marked decrease in complaints about reliability. I still wonder if we had started sooner how many of those valued colleagues might have stayed?

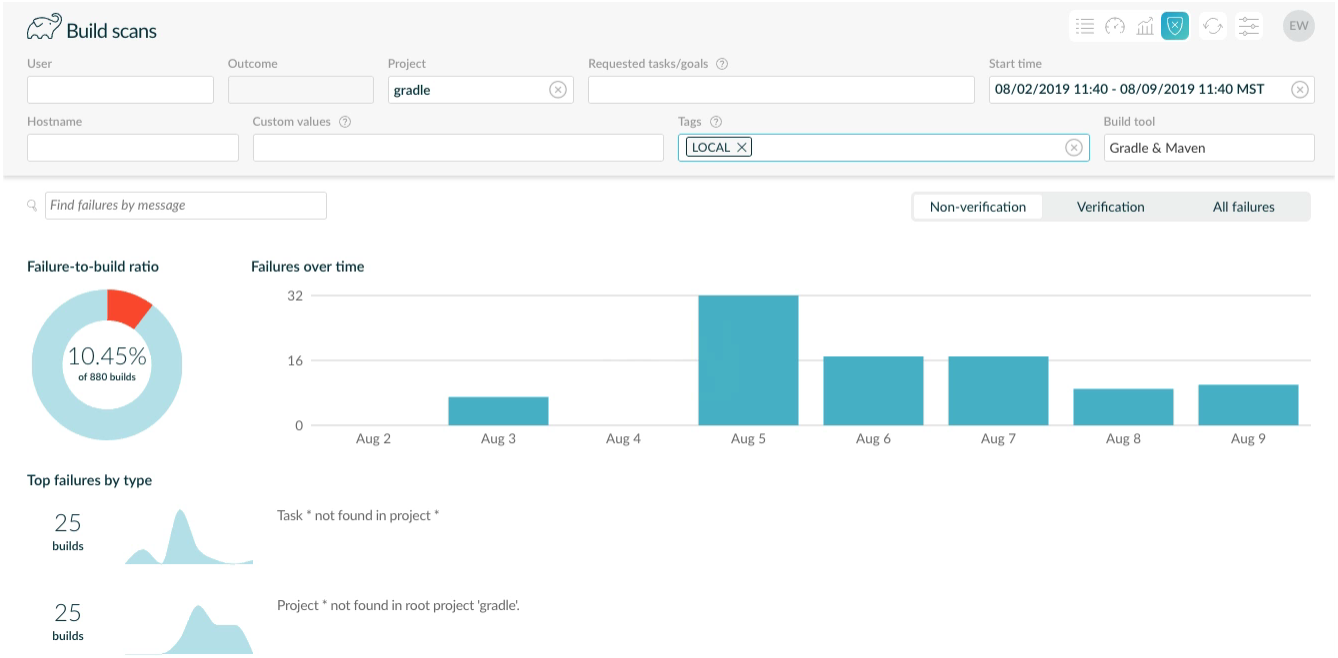
5.1. So how can we improve the situation?

The fastest code is the code that doesn't execute at all. The most efficient, painless support session is the one that doesn't need to take place at all because proactive investment has made the build highly reliable. To achieve this you need to collect data from all build and test run in the organization, both local and CI, and analyze the data to determine which issues have the greatest impact on developer productivity. To facilitate this type of analysis, we created the Failures Dashboard in Gradle Enterprise, first introduced in version 2019.3.

The mysterious missing project

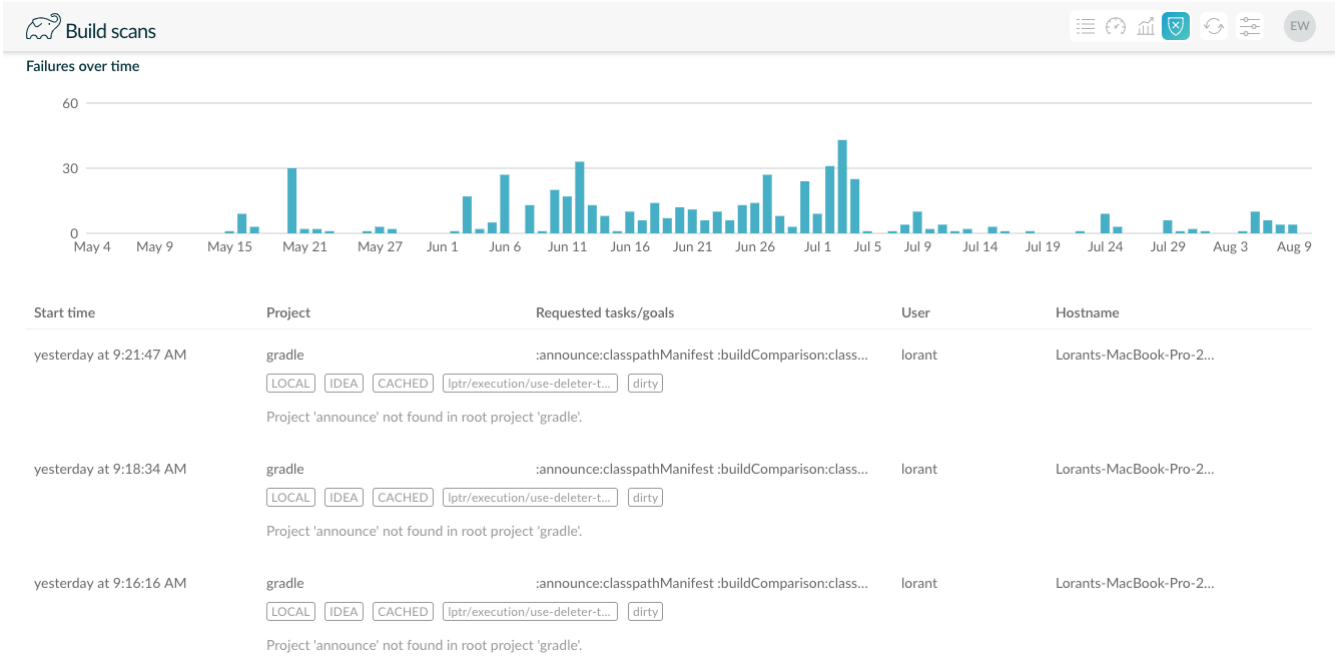
Here's how we used the Failures Dashboard to identify and fix a nasty problem in our own build. Here's the "Top Failures" view of the Failures Dashboard, filtered to focus on local builds and non-

verification failures:



The classification of “Verification” vs “Non-Verification” is based on semantic analysis of error messages. Then, the failures are clustered by similarity, and the exclusive phrases for each cluster are extracted to form a fuzzy matching pattern that uniquely matches all failures in the group. `Project * not found in root project 'gradle'` was a non-verification failure that we hadn’t known about - no one had complained! - despite 1 in 40 local builds a week failing with that error.

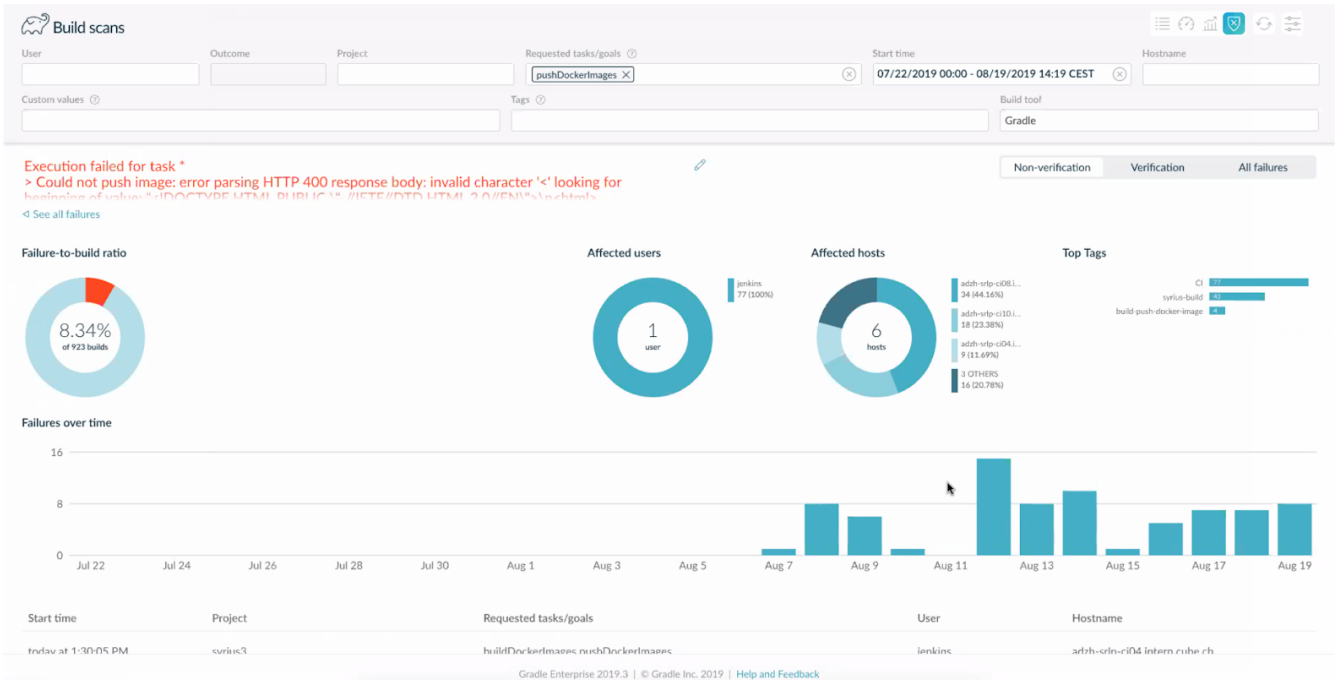
From this “Failure Analysis” view we can see that it started happening in early May. Interestingly, we see that the IDEA tag is one of the top tags. Most Gradle developers use IntelliJ IDEA for their Java development. Before digging into a single console log or build scan, the fact that this failure occurs in builds run by IntelliJ but not on CI or the command line already suggests that the root cause is related to IntelliJ configuration. The bottom of the page lists the 50 most recent occurrences of this failure and their details. We can click on a build to go to the corresponding build scan.



Glancing over the list shows that the “announce” project is the one being cited as missing. A few git blame invocations later and we found the commit that removed the defunct “announce” project from the build - but missed a few references in a way that didn’t break the command line but did impact IntelliJ’s ability to import the build. Once we had the data and could visualize it, effective prioritization & remediation of an issue we previously knew nothing about was natural and easy.

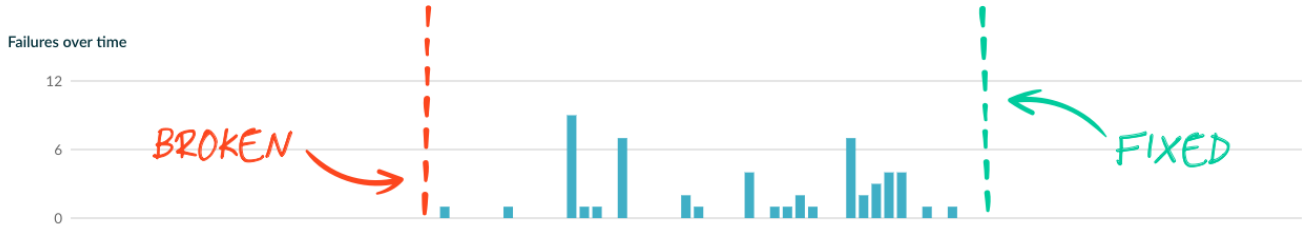
A Docker disruption

One of our customers shared this example with us. They had noticed that their test lab and deployments had been failing more often but didn’t know why. So they searched the Failures Dashboard for the error message and got this:



It shows that over 8% of their builds were failing with an issue related to the publishing of docker images. Armed with this evidence that the problem was severe and knowing exactly when it started happening, they were able to quickly track down the root cause as a bug introduced to their docker

configuration by an infrastructure team. Because they acted proactively when developers complained they were able to tell them it was a known issue actively being worked on, rather than being caught off balance.



When the issue was resolved they were able to send a report to all the stakeholders that showed when it started happening, when it was fixed, how many people were affected, and what procedures were being implemented to avoid similar problems in the future. Ultimately this incident led to increased trust between developers and their tooling, and of the developer productivity engineers supporting that tooling.

5.2. A proactive developer productivity process

On one developer productivity team we were very successful using a process like this one to stay on top of reliability in a rapidly-changing build environment:

1. Every day at standup look at the non-verification section of top failures view of the failure dashboard to see if any new issues have appeared
    a. Frequently this would lead to our team being the first to report on a new toolchain issue making the subsequent incident response feel proactive to the rest of the organization
2. Immediately triage any new failure types, evaluating if they were happening frequently enough to warrant interrupting our sprint for
3. If an issue was interrupt priority, send out an email & slack message to the affected communities of developers to let them know
4. The Engineer taking the issue would use metadata like when the failure started happening, who it happened to, where it happened, with what metadata, to begin their investigation
5. Depending on how long remediation took, send additional updates to stakeholders appraising them of our progress
6. Once a candidate fix was checked in, continue monitoring the analysis view for that failure type to verify that new incidents dropped to 0
    a. If the fix required any manual action on the part of developers, directly reach out to any still affected and let them know the availability of the fix
7. Conduct a short root cause analysis meeting where those who worked on the issue would discuss why it happened and what steps could be taken to prevent that category of failure in the future
8. Send final all-clear report with a description of the timeline of the incident, the result of the root cause analysis, the nature of the remediation, and the description of the guards against regression to be implemented

Ultimately the transparency, predictability, and effectiveness of this process built a great deal of trust between our developer productivity team and our peers working on the product. This credibility manifested as developers coming to us for advice on infrastructure and toolchain proposals, which allowed us to steer them away from anti-patterns and architectural mistakes. Being involved early on in these discussions let us avert many issues before they could ever be committed in the first place, producing a virtuous cycle.

5.3. You don't optimize what you don't measure

How much time do developers in your organization spend having their thumbs smashed by cursed hammers? How long does it take after an issue affecting developer productivity is detected before it's fixed? Ultimately only the continuous application of the principles and practices of developer productivity engineering will attain & maintain a great developer experience at scale. If you're not measuring it, don't act too surprised when some of the best leave for greener, better-tooled pastures.

Of course, build reliability is likely not the only pain point in your development toolchain. Flaky tests are another common source of intense developer pain which we will discuss in The Black Pit of Infinite Sorrow: Flaky Tests.

6. The Black Pit of Infinite Sorrow: Flaky Tests

By Sam Snyder

A test is “flaky” whenever it can produce both "passing" and "failing" results for the same code.

Test flakiness is a bit like having diabetes. It’s a chronic condition you can never be fully cured and done with. Left unmanaged, it can render a body (or body of tests) severely damaged. But your prognosis is good if you continually measure and act on the relevant health indicators.

Even if only 0.1% of your tests are flaky, with thousands of tests you can have problems with a considerable portion of your builds. This is common to even medium-sized development teams.

When confronted with flaky tests, many organizations (including Gradle) implement an automatic retry when tests fail. This trades increased build times for build reliability & retaining the test coverage provided by flaky tests. But this can potentially allow the problem to fester and new flakiness to be introduced until one retry isn’t enough anymore. And then, where does it end? Two retries? Ten?

Some organizations give up and surrender to the problem; their test reports end up telling them almost as much about the quality & shippability of their software as scrying tea leaves or consulting horoscopes.

At Gradle, we reached a point where flaky tests were the number one complaint of our developers. We had already implemented a single automatic retry but it wasn’t enough anymore. We didn’t want to sacrifice any more build time to additional retries and we were wary that retries could be hiding real product problems. A perfectly correct, stable test covering flaky product code looks exactly like a flaky test and so real issues can easily be hidden by retries.

So we configured our CI jobs such that any time a test fails once but passes on retry, an entry is made in a database that records the test and the time of the incident. After collecting data for a few weeks we analyzed it to see which tests were causing the pain. Several hundred tests had at least one flaky failure, but a mere 10 tests were contributing 52% of all our flaky failures! We immediately disabled those tests and filed issues for the owners of those areas to investigate.

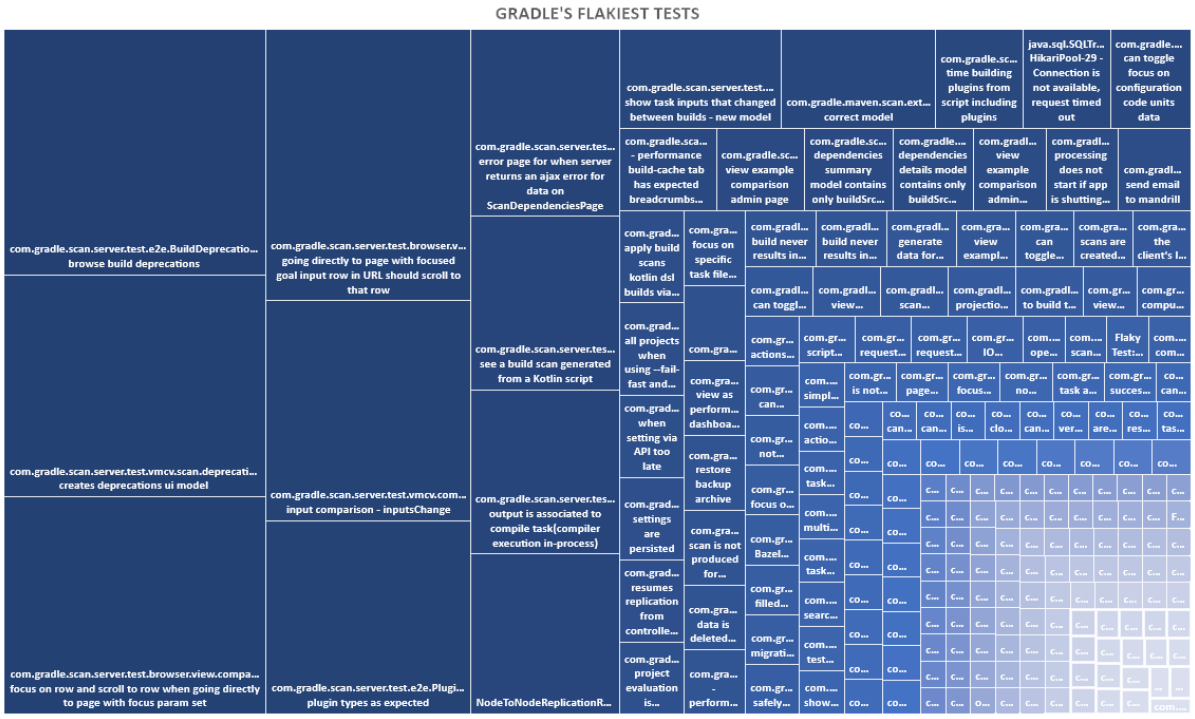


Figure 10. Each square is a flaky test. The bigger the square, the bigger the flake.

Reasonable engineers sometimes object to disabling flaky tests, worried that the disabled test will never be fixed and its coverage lost forever. This anxiety can indicate a lack of trust that quality will be prioritized as part of the planning process - an issue worthy of its own book! Setting aside the organizational pathologies it may signal, if the test is flaky enough that it can fail even when re-tried, you've *already lost that coverage.*

When flaky failures become a significant fraction of a test or test suite's failures, engineers reasonably begin to assume that any new failure from that suite is spurious. They may cease to prioritize investigations and so the suite becomes useless and bugs get shipped to customers that had been showing up red in test reports for weeks or months.

A suite that died of flakiness

One company I worked at maintained a suite of roughly 10,000 browser tests for their server product and ran them nightly. These tests automated interactions with the web interface, which communicated with a fully running instance of the server. Since these were effectively end-to-end tests, problems anywhere in the product or infrastructure would substantially disrupt the results. On a good day the pass rate for this suite was 85%, on a bad day it could be as low as 55%.

There were roughly 50 engineers responsible for this suite. Investigating and fixing a test is sometimes quick, but very often takes hours of work. To get the whole suite passing before the next nightly test pass, each engineer would have had to fix dozens of tests. With so many unreliable tests, and more being added all the time, doing anything about it came to feel Sisyphean.

This suite passing was part of the process for merging work from the feature branch to the release branch we shipped from, a process enacted every two weeks. Some document somewhere probably said the suite was supposed to be passing before the integration happened. In practice, integration owners had to compare the results between the source and target branch to try and distinguish new failures from the thousands of existing failures.

This process of trying to scry real results out of the suite was ineffective and mind-numbingly tedious, hated by everyone who got stuck participating in it. Bugs were routinely integrated that tests had accurately reported as failing. Legitimate failure signals were thoroughly drowned in an ocean of red, flaky failures.

Digging ourselves out of that hole took a lot of time and effort that could have been spent on improving the product, had we not let the problem get out of hand in the first place. I still wonder about how many person-years were wasted.

6.1. Measuring your blood sugar: Quantifying flakiness

How much pain test flakiness causes an organization is exponential. A little flakiness is barely noticeable. A lot of flakiness and your testing is rendered completely worthless, any actionable signal drowned out in a blood-red sea of noise.

So it follows that the ROI on fixing flakiness is logarithmic: Fixing that most-flaky test reduces pain substantially, but once flakiness is down to acceptable levels it becomes difficult to justify the opportunity cost of investing in flakiness fixing instead of new features. Therefore an ongoing process is required to measure the severity of the problem in a way amenable to correctly prioritizing investments of time & resources.

So how much flakiness can you live with? The amount of flakiness you can tolerate is inversely proportional to the number of tests you have, the rate at which they flake, the number of builds you run, and how many times you retry on failure.

```
Flaky Failures/day = (Builds/day)(Count of Flaky Tests)(Chance a test flakes)^(1+retries)
```

Let's say your organization has 200 flaky tests that run in 500 builds per day and you want no more than 1% of your builds, 5 in this example, to be disrupted by a flaky test failure. With no retries, you need to keep the average rate at which they flake under 0.005%. With a single retry, you can maintain a 1% flaky failure rate with an average test flakiness rate of under 0.7%. This means your tolerance for flakiness goes down as the number of flaky tests you have, or the number of builds you run, increase.

6.2. Common sources of flakiness

A test can be flaky for an unbounded number of reasons. Here are several that occur across many different kinds of software projects.

- Accessing resources that are not strictly required
  - Why is that unit test downloading a file? It's a unit test, it should only interact with the unit under test.
  - This includes accessing the real system clock. Prefer a mocked source of time controlled by the test
- Insufficient isolation
  - When running tests in parallel, resource contention or race conditions are possible if each test doesn't get its own exclusive copy of the resource
  - Even with fully serial execution, tests that change system state may cause subsequent tests to fail if they don't clean up after themselves or crash before cleanup has occurred.
  - Tests interacting with a database should wrap all their interactions in a transaction that's reverted at the end of the test.
- Asynchronous invocations left un-synchronized, or poorly synchronized
  - Any fixed sleep period is a mistake. Too short it fails tests that would have passed. Too long and it pointlessly wastes your time. And "too short" and "too long" differ based on circumstances so no fixed value can ever be "just right". Always poll the result with a reasonable timeout.
- Accessing systems or services that are themselves not perfectly stable
  - Try writing most functional tests with mocked services. Use a smaller number of higher-level contract tests to ensure the mocks meaningfully match the real service.
- Performance tests that can be highly variable, even when you try to control the circumstances in which they run
  - Running the same scenario several times and discarding outliers can reduce noise at the cost of additional time & resources required to run the test
- Usage of random number generation
  - While obviously required for valuable techniques like fuzz testing, the random seed should always be logged & controllable otherwise reproducing any issues the tests uncover can be needlessly difficult.
  - When re-running a failed test that uses random number generation, the same seed should always be used

6.3. Taking your insulin: Keeping flakiness under control

Since flaky tests pose a procedural challenge as well as a technical one, procedural adaptations must be part of the solution. Here's an example process your organization could use you to keep the pain of flakiness tolerable and avoid having to write off whole test suites as a loss:

1. Setup your CI to re-run failed tests a single time (unless there are more than 10 failures)
2. Any test which fails then passes on re-run should be recorded, with a timestamp, as flaky in a database
3. Over the duration of a sprint, or whatever your planning cadence is, count how many builds you run total, how many fail due to flakiness, and how many flaky tests you have
4. Set a realistic tolerance threshold. If you have a million flaky tests you aren't going to get to 0 in one go
    a. Except for the very largest codebases, it's probably not worth the effort to de-flake a test that fails one in a million times
5. Disable any tests which flake at rate above your tolerance threshold
6. File bugs for each test disabled and send them to the teams that own those tests
    a. This is most effective if done in advance of sprint planning with notification of planners
7. A test should not be allowed to be re-enabled until the supposedly fixed version has been run enough times to demonstrate that it flakes at a rate that doesn't threaten your tolerance threshold
    a. A special CI configuration
8. Repeat this process on a regular cadence
9. Periodically check for tests that have been disabled for a very long time without being re-enabled and consider additional notifications or escalations

If performed completely manually, this can be a lot of work! Fortunately, many of these steps are highly susceptible to automation. After the CI jobs are in place, identifying flaky tests and filing bugs accordingly can be automated. Automatically disabling tests is harder, but still achievable with a script that knows how to perform & commit an abstract syntax tree transformation.

Part 3 - Economics

Quantifying the cost of builds

Investing in your build: The ROI calculator

7. Quantifying the Cost of Builds

By Hans Dockter

In this section, I want to provide a model on how to calculate the costs of your builds and the return you get on improving it.

Developers and build engineers are under constant pressure to ship faster and more frequently. This requires fast feedback cycles and thus fast builds. At the same time, cloud, microservices, and mobile have made our software stacks more complex. Together with growing codebases, and without taking any actions, this will slow you down as your builds become slower and build failures are harder to debug.

Improving this is a huge organizational challenge for build engineers and development teams. To trigger organizational change it is important to have quantitative arguments. Inefficient builds do not just affect your ability to ship fast they also waste a lot of your R&D bandwidth.

Some of our customers have more than 100,000 Gradle build executions a day. Most medium to large engineering teams will have at least thousands of builds a day. For many organizations, this is a multi-million dollar developer productivity problem that is right under your nose. And every effort to improve it should start with assessing its impact.

7.1. Meet our example team

Our example team consists of 200 engineers with the following parameters:

Parameter Name	Description	Value for the example team
C_M	Cost per minute of engineering time	$1
D_E	Working days of an engineer per year	230 days
$C_E = D_E * C_M * 8 * 60$	Cost per engineering year	$110,400
B_L	Local builds per day	2000
B_{CI}	CI builds per day	2000
D_W	Number of days the office is open	250 days
$BY_L = B_L * D_W$	Local builds per year	500000
$BY_{CI} = B_{CI} * D_W$	CI builds per year	500000

The example numbers above and later in the article reflect what we see typically in the wild. But the numbers out there also vary a lot. For some, we have better averages than for others. The example numbers are helpful to get a feeling for the potential magnitude of the hidden costs that come with builds. Your numbers might be similar or very different. You need your own data to get a good understanding of what your situation is and what your priorities should be. The primary purpose of this document is to provide a model with which you can quantify costs based on your

own numbers.

The number of builds depends a lot on how your code is structured. If your code is distributed over many source repositories you have more build executions compared to the code being in a single repository which then results in longer build times. But as a rule of thumb, one can say that successful software teams have many builds per day. It is a number that within your organization you want to see go up. As we evolve our model, we want to switch in the future to a lines of code build per day metric.

7.2. Waiting time for builds

Parameter Name	Description	Value for the example team
W_L	Average fraction of a local build that is unproductive waiting time.	80%
W_{CI}	Average fraction of a CI build that is unproductive waiting time	20%

7.3. Local builds

When developers execute local builds, waiting for the build to finish is pretty much idle time. Especially everything shorter than 10 minutes does not allow for meaningful context switching. That is why we assume WLas 80%. It could even be more than 100%. Let's say people check and engage on Twitter while the local build is running. That distraction might take longer than the actual build to finish.

Here is the cost for our example team of unproductive waiting time for each minute the local build takes:

$$BY_L * W_L * C_M = 500000 * 0.8 * \$1 = \$400{,}000 \text{ per year}$$

Vice versa, every saved minute is worth $400,000. Every saved second $6667. A 17 seconds faster local build gives you additional R&D resources worth one engineer. If you make the build 5 minutes faster, which is often possible for teams of that size, that gives you the resources back of 18 engineers which are 9% of your engineering team or $2,000,000 worth of R&D costs!

The reality for most organizations is that the builds are taking longer and longer as the codebases are growing, builds are not well maintained, outdated build systems are used, and the technology stack becomes more complex. If the local build time grows by a minute per year, our example team needs an additional 4.5 engineers just to maintain your output. Furthermore, talent is hard to come by and anything you can do to make your existing talent more productive is worth gold. If local builds are so expensive, why do them at all ;). Actually, some organizations have come to that conclusion. But without the quality gate of a local build (including pull request builds), the quality of the merged commits drastically deteriorates leading to a debugging and stability nightmare on CI and many other problems for teams that consume output from upstream teams. So you would be kicking the can down the road with even significantly higher costs.

It is our experience that the most successful developer teams build very often, both locally and on CI. The faster your builds are, the more often you can build and the faster you are getting feedback, thus the more often you can release. So you want your developers to build often, and you want to make your build as fast as possible.

7.4. CI builds

The correlation between CI builds and waiting time is more complicated. Depending on how you model your CI process and what type of CI build is running, sometimes you are waiting and sometimes not. We don't have good data for what the typical numbers are in the wild. But it is usually a significant aspect of the build cost, so it needs to be in the model. For this example we assume W_{CI} is 20%. The cost of waiting time for developers for 1 minute of CI build time is then:

$BY_{CI} * W_{CI} * C_M = 500000 * 0.2 * \$1 = \$100{,}000$ per year

Long CI feedback is very costly beyond the waiting cost:

- Context switching for fixing problems on CI will be more expensive
- The number of merge conflicts for pull request builds will be higher
- The average number of changes per CI build will be higher and the time finding the root cause of the problem will increase and it will often require all the people involved with the changes.

We are working on quantifying the costs associated with those activities and they will be part of a future version of our cost model. The CI build time is a very important metric to measure and minimize.

7.5. Potential investments to reduce waiting time

- Only rebuild files that have changed (Incremental Builds)
- Reuse build output across machines (Build Cache)
- Collect build metrics to optimize performance (Developer Productivity Engineering practice)

7.6. The cost of debugging build failures

One of the biggest time sinks for developers is to figure out why a build is broken (see the challenge of the build engineer for more detail). When we say the build failed, it can mean two things. Something might be wrong with the build itself, e.g., an out of memory exception when running the build. We will talk about those kinds of failures in the next section. In this section, we talk about build failures caused by the build detecting a problem with the code (e.g., a compile, test or code quality failure). We've seen roughly these statistics for a team of that size:

Parameter Name	Description	Value for the example team
F_L	Percentage of local builds that fail	20%
F_{CI}	Percentage of CI builds that fail	10%

I_L	How many percents of the failed local builds require an investigation	5%
I_{CI}	How many percents of the failed CI builds require an investigation	20%
T_L	Average investigation time for failed local builds	20 mins
T_{CI}	Average investigation time for failed CI builds	60 mins

Such failure rates for F_L and F_{CI} come with the territory of changing the codebase and creating new features. If the failure rate is much lower, I would be concerned about low test coverage or low development activity.

For many failed builds the root cause is obvious and does not require any investigation, but there are enough where you need to investigate which is expressed by I_L and I_{CI}. CI builds usually include changes from multiple sources. They are harder to debug, and multiple people might need to be involved. That is why T_{CI} is larger than T_L.

Costs

Debugging local build failures:

$$BY_L * F_L * I_L * T_L * C_M = 500000 * 0.2 * 0.05 * 20 * \$1 = \$100000 \text{ per year}$$

Debugging CI build failures:

$$BY_{CI} * F_{CI} * I_{CI} * T_{CI} * C_M = 500000 * 0.1 * 0.2 * 60 * \$1 = \$600000 \text{ per year}$$

Overall this is $700,000 per year.

People often underestimate their actual failure rate. At the same time, there is quite a bit of variation in those numbers out there in the wild. You may have teams with very long-running builds. Because the builds are slow, developers don't run them that often and there are also less CI builds. Fewer builds mean a lower absolute number of build failures. And long-running builds are saving money, right?

Not so fast: A small number of builds means a lot of changes accumulate until the next build is run. This increases the likelihood of a failure, so the failure rates go up. As many changes might be responsible for the failure, the investigation is more complex and the average investigation times go up. I have seen quite a few companies with average investigation times for CI failures of a day or more. This is expensive debugging but the costs of such long-living CI failures go beyond that. It kills your very capability to ship software regularly and fast.

The basic rule is that the later a failure shows up, the investigation time grows exponentially.

So following up on the section of local build time. If developers don't do a pre-commit build, it will push up the failure rate and investigation time on CI. Everything is connected. If you have very

poor test coverage your failure rate might be lower. But that pushes the problems with your code to manual QA or production.

Potential investments for reducing debugging costs

- Tools that make debugging build failures more efficient
- Everything that makes builds faster

7.7. Faulty build logic

If the build itself is faulty, those failures are particularly toxic. Those problems are often very hard to explore and often look to the developer like a problem with the code.

Parameter Name	Description	Value for the example team
F_L	Percentage of local builds that fail due to bugs in the build logic	0.2%
F_{CI}	Percentage of CI builds that fail due to bugs in the build logic	0.1%
I_L	How many percents of the failed local builds require an investigation	100%
I_{CI}	How many percents of the failed CI builds require an investigation	100%
T_L	Average investigation time for failed local builds	240 mins
T_{CI}	Average investigation time for failed CI builds	90 mins

F_L is usually larger than F_{CI} as the local environment is less controlled and more optimizations are used to reduce the build time like incremental builds. If not properly managed they often introduce some instability. Such problems usually require an investigation which is why the investigation rate is 100%. Such problems are hard to debug, for local builds even more so as most organizations don't have any durable records for local build execution. So a build engineer needs to work together with a developer trying to reproduce and debug the issue in her environment. For CI builds there is at least some primitive form of durable record that might give you an idea of what happened, like the console output. We have seen organizations which much higher rates for FL and FCI than 0.2% and 0.1%. But as this is currently very hard to measure we don't have good averages and therefore are conservative with the numbers we assume for the example team.

Cost

Debugging local build failures:

$BY_L * F_L * I_L * T_L * C_M = 500000 * 0.002 * 1 * 240 * \$1 = \$240{,}000$ per year

Debugging CI build failures:

$BY_{CI} * F_{CI} * I_{CI} * T_{CI} * C_M = 500000 * 0.001 * 1 * 120 * \$1 = \$60000$ per year

Overall this is $300,000 per year.

There is a side effect caused by those problems: if developers regularly run into faulty builds, they might stop using certain build optimizations like caching or incremental builds. This will reduce the number of faulty build failures but at the cost of longer build times. Also when it is expensive to debug reliability issues, it means they will often not get fixed. Investing in reliable builds is key.

Potential investments

- Collect build metrics that allow you to find the root causes effectively
- Reproducible Builds
- Disposable Builds

7.8. CI infrastructure cost

Often half of the operational DevOps costs are spent on R&D. The CI hardware is a big part of that. For our example team, a typical number would be $200K.

Potential investments to reduce CI infrastructure cost

- Reuse build output across machines (Build Cache)
- Collect build metrics to optimize performance (Build Performance Management)

7.9. Overall costs

We assume the following average build times for our example team:

Parameter Name	Description	Value for the example team
A_L	Average build time for local builds	3 mins
A_{CI}	Average build time for CI builds	8 mins

This results in the following overall cost:

Waiting time for local builds	$1,200,000
Waiting time for CI builds	$800,000
Debugging build failures	$700,000
Debugging faulty build logic	$300,000
CI hardware	$200,000

Total Cost	$3,200,000

While this cost will never be zero, for almost every organization it can be significantly improved.

Cutting it in half would give you the R&D worth of 15 engineers for our example team of 200. And keep in mind that if you don't do anything about it, it will increase year by year as your codebases and the complexity of your software stacks are growing.

There are a lot of other costs that are not quantified in the scenarios above. For example, the frequency of production failures due to ineffective build quality gates or very expensive manual testing for similar reasons. They add to the costs and the potential savings.

7.10. Why these opportunities stay hidden

I frequently encounter two primary obstacles that prevent organizations from realizing the benefits of investing in this area.

Immediate customer needs always come first

Especially when talking to teams who have many small repositories I hear regularly the statement that "build performance and efficiency is not a problem". What do they mean with not a problem? In their case, it is simply that developers do not complain about build performance. For those teams, unless the developers are really yelling, nothing is a priority. While developer input is very important for build engineering, anecdotal input from developers should not be the sole source of prioritization. You might leave millions of dollars of lost R&D on the table. Build engineering should operate more professionally and more data-driven.

Benefit is understated

For other teams, there is a lot of pain awareness, e.g., around long build times. Actually, so much of it, that the impact of incremental steps is underestimated, as they are not taking the pain completely away. With a cost and impact model, the value of incremental steps would be much more appreciated. Such a model is also helpful to demonstrate progress and is an important driver for prioritizing further improvements.

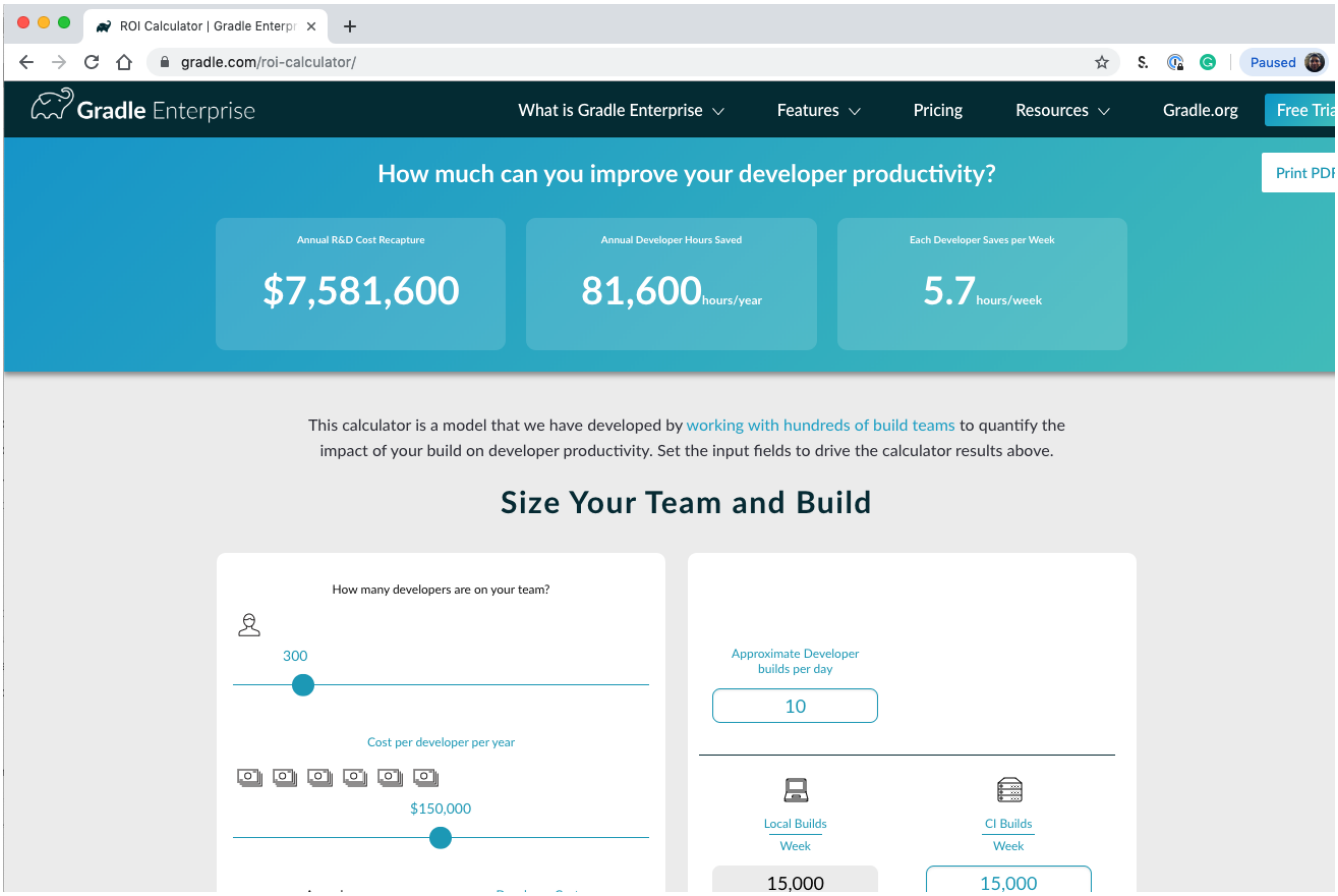
7.11. Conclusions

We haven't yet seen a company where investing in more efficient builds was not leading to a significant return on investment. The most successful software teams on the planet are the ones with an efficient build infrastructure.

8. Investing in Your Build: The ROI calculator

By Hans Dockter

In the previous chapter we discussed a model to calculate the cost of your build. We created an interactive ROI calculator to estimate the return from improving your build using this model. (Find it at https://gradle.com/roi-calculator.)



8.1. How to use the Build ROI calculator

The calculator estimates potential savings both in R&D dollars recaptured and developer hours saved at both a team and 'per developer' level. There are three sections which correspond to the savings drivers discussed in the previous chapter, including:

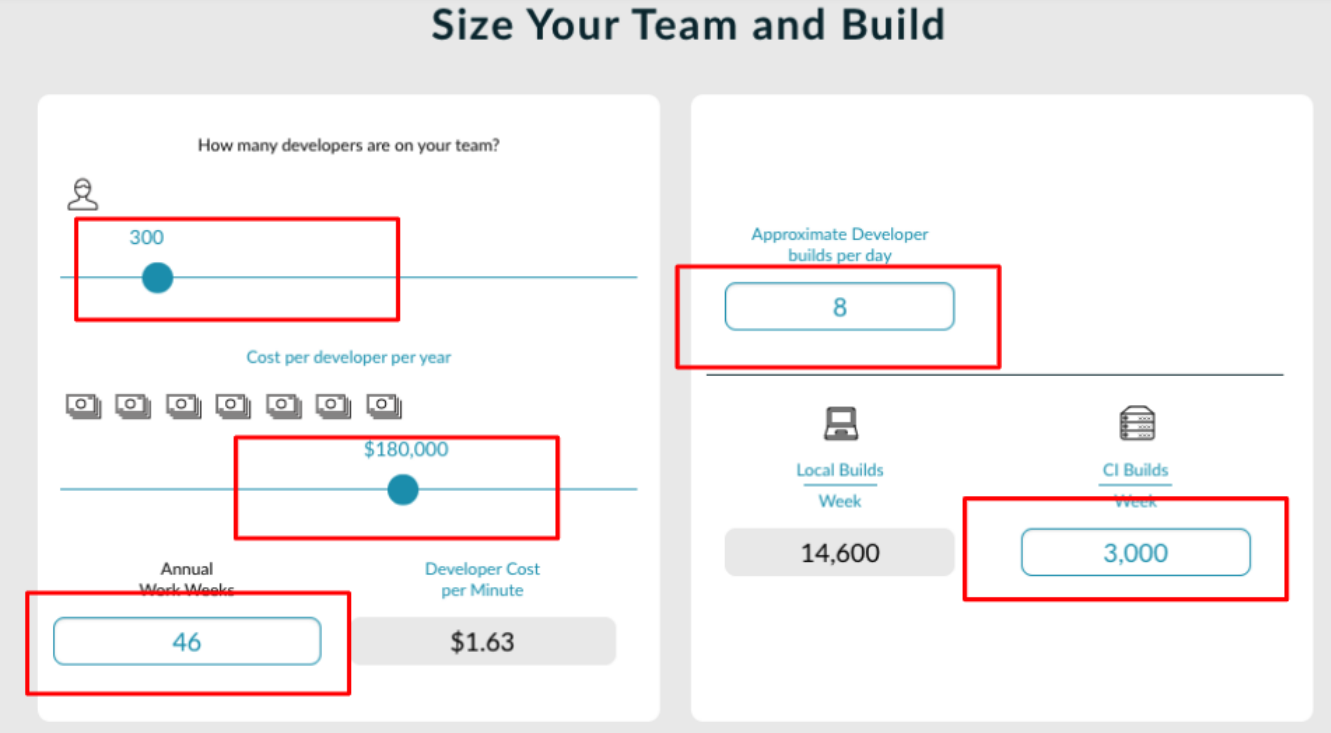
- Speeding up slow builds
- Preventing build regressions over time
- Faster build debugging

The savings figures for each of these drivers total into a summary depicted in the top banner. You set input parameters in each of the sections below to create this overall estimate or calculation for your team.

Step 1: Size your team and build

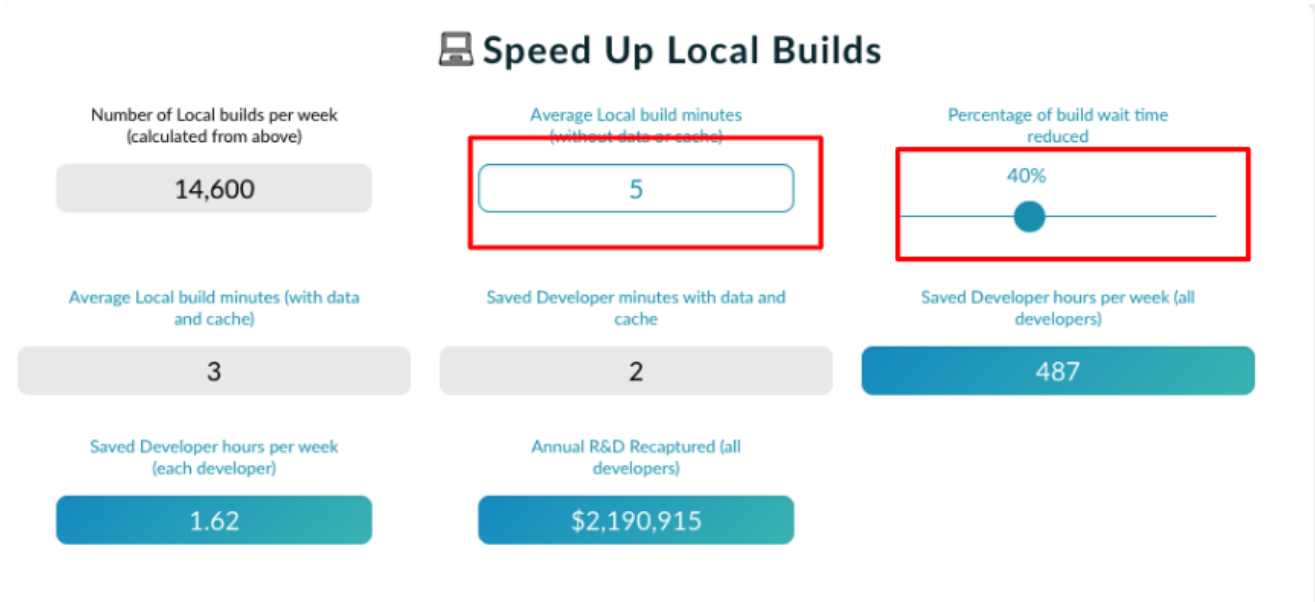
Start with the first, top section of the calculator entitled “Size your Team and Build”. On the left side, use the sliders to estimate the number of developers running local and CI builds on your team and the annual developer salary (including benefits). On the right side, update the blue input boxes for the weekly developers and CI builds.

The summary calculations at the top change to reflect your changes. In addition, tooltips explain how each field impacts your calculation.



Step 2: Speed up slow developer and CI builds

In the next section, set your average local build time in minutes and estimate the percentage of build wait time you expect to reduce using the cache.



Repeat the process in the next section Speed up CI Builds. This time there is an extra step - use the slider to estimate “Percentage of CI builds where developers are waiting”. In our experience, this is typically at least 20%.

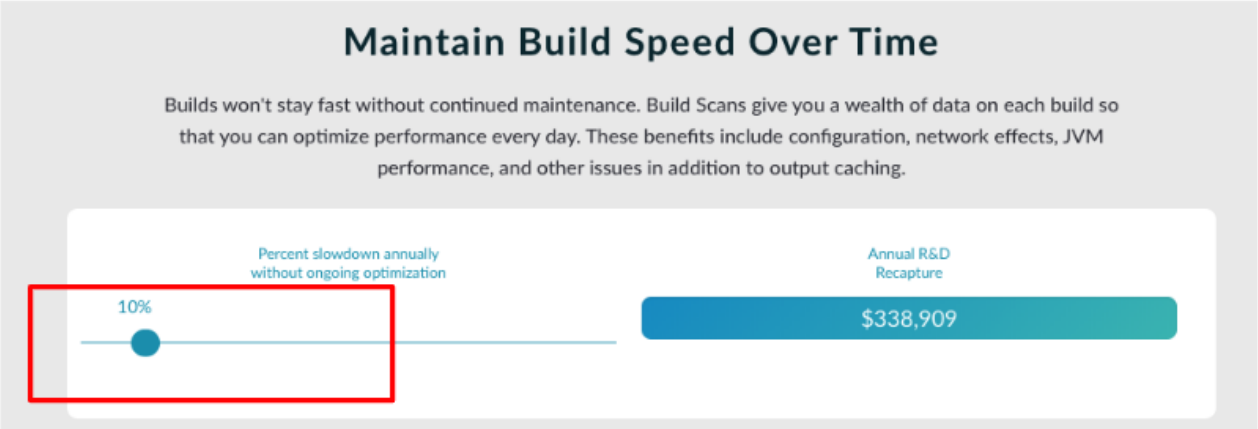
Notice how changing these settings updates both the total savings in these sections and at the top banner aggregate results.

Step 3: Maintain build speed over time

Builds grow larger, more complex, and slow down over time. Performance regressions go unnoticed unless data is used to continually observe trends, spot regressions and fix errors to keep builds fast.

In our experience working with hundreds of software development teams, an enterprise team can expect builds to slow down at 15% or more each year.

Progress to the section “Maintain Build Speed Over Time”. Use the slider to estimate how much you can save your team if you reduce or eliminate these regressions.



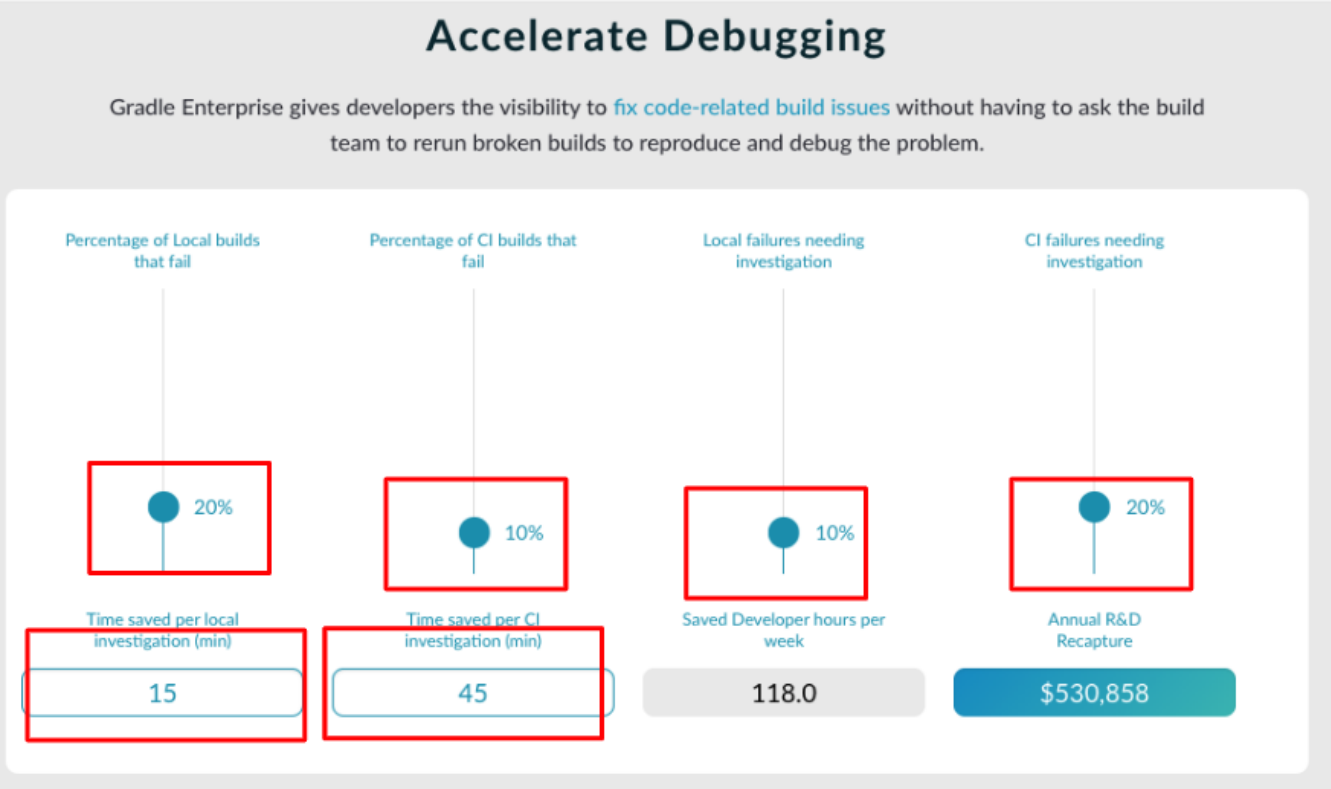
Step 4: Accelerate debugging

Debugging builds is a complex, time-consuming effort that consumes hours of developer time per week and often turns the build team into a reactive support organization. Netflix did an internal study that determined almost 25% of all engineering time was spent debugging builds!

Estimate how much time and money you can save your team by speeding up build debugging.

Progress downward to the final section entitled “Accelerate Debugging”. Set the sliders to the approximate percentage of time both developer and CI builds fail, the percentage of both local and CI build failures that require manual investigation.

Then use the text boxes underneath the sliders to estimate how much faster you could fix issues using build scans and analytics.



Create a PDF of the calculation

To save your estimate, use the “Print PDF” button in the banner to create a PDF of the current calculator state.

We look forward to your feedback on this calculator!

You can get a more precise calculation by using your own real data for many of these inputs. For example, in our experience, most teams underestimate both the number of developer and CI builds run each week. You can quickly get an accurate picture of your own build data and trends through a trial of Gradle Enterprise.

Request a trial of Gradle Enterprise to capture your own build data and understand how much productive time you can recapture for your organization.

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.

- Hands-on Developer Productivity Engineering Workshop - this free, 2-hour online workshop is offered monthly to train attendees both in the practice and the use of both free and commercial technology for developer productivity engineering.
- 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/