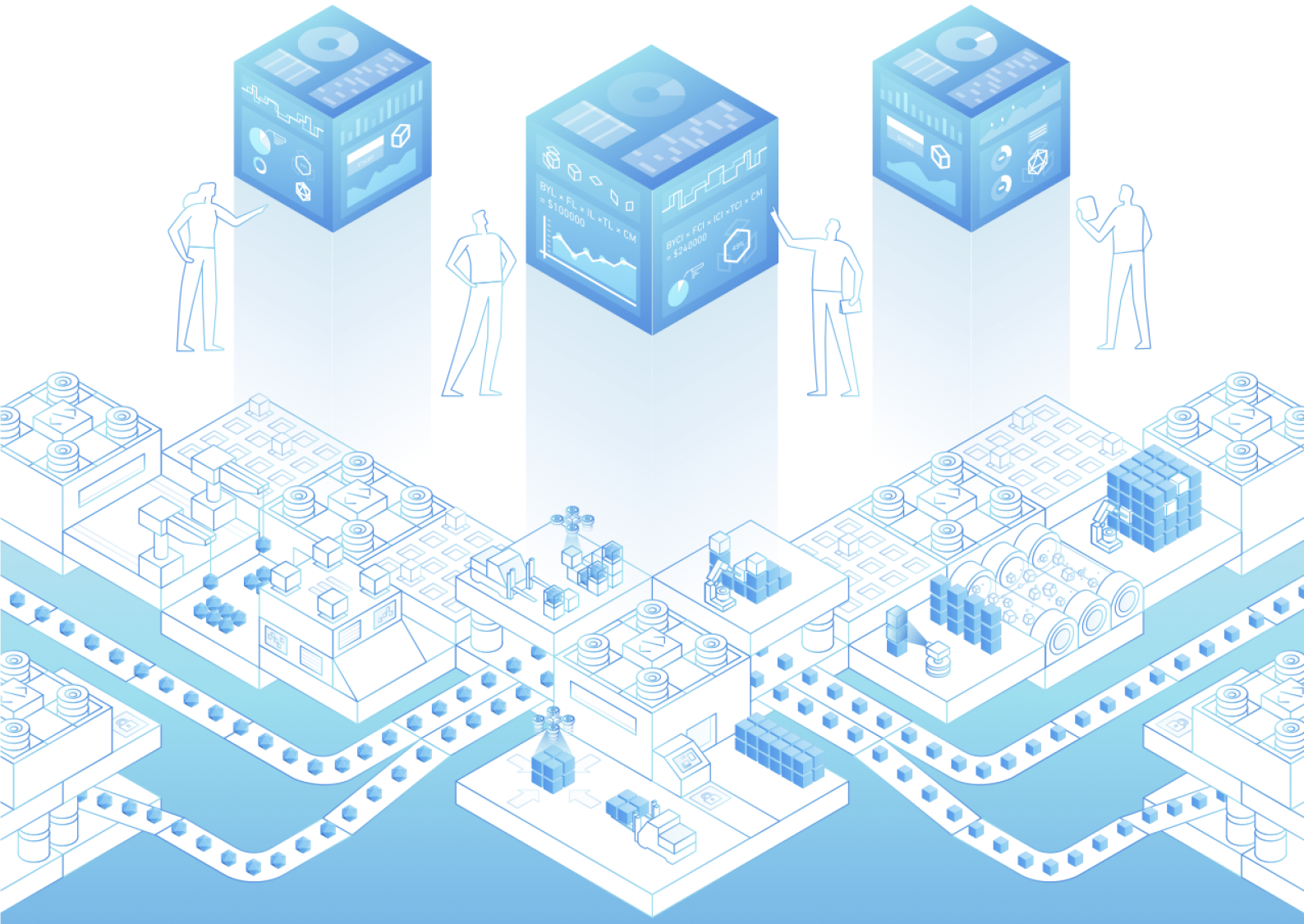
THE DEVELOPER PRODUCTIVITY ENGINEERING HANDBOOK

A Complete Guide to Developer Productivity Engineering for Practitioners



Hans Dockter et al.

Gradle

THE DEVELOPER PRODUCTIVITY ENGINEERING HANDBOOK

A Complete Guide to Developer Productivity Engineering for Practitioners

Hans Dockter et al.

Version 2.0-local, 2022-03-11

Table of Contents

THE DEVELOPER PRODUCTIVITY ENGINEERING HANDBOOK: A Complete Guide to Developer Productivity Engineering for Practitioners

Published by Gradle Inc.

Copy Editor: Wayne Caccamo, with Justin Reock and Raju Gandhi
Cover 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.

What is clear to us from working with you is that Developer Productivity Engineering is quickly becoming a critical practice in any software engineering organization that takes industry leadership and the developer experience seriously. We welcome your continued contributions to the ideas in this book.

We would also like to thank all the talented and dedicated engineers at Gradle Inc.

Hans Dockter and the team at Gradle Inc.

Preface

Developer Productivity Engineering (DPE) is a software development practice used by leading software development organizations to maximize developer productivity and happiness. This book represents the proverbial “living document” that will be updated with new Developer Productivity Engineering concepts, best practices and enabling technologies as we invent and discover them over time. This is the second major update to the initial document which was published in 2019. We will notify you about future updates if you have registered to receive this book, and we continue to encourage you to give us your feedback at devprod-eng-book@gradle.com.

About this book

In this book we share developer productivity engineering techniques and stories about 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 test distribution to speed up feedback cycles
- Use data to proactively improve the reliability of the development toolchain
- Find and fix frequent errors and noisy signals like flaky tests

It consists of four parts:

- Part 1 - DEVELOPER PRODUCTIVITY ENGINEERING DEFINITIONS, DRIVERS & CONCEPTS
  Part 1 defines Developer Productivity Engineering (DPE) and important related concepts, makes an intuitive case for investing in a DPE initiative now, and positions DPE in the broader developer productivity solution landscape.
- Part 2 - FEEDBACK CYCLES AND ACCELERATION
  Part 2 describes the essential role of fast feedback cycles in achieving DPE excellence and three key enabling technologies for achieving feedback cycle acceleration: Build caching, Test Distribution, and Performance Profiling & Analytics.
- Part 3 - TROUBLESHOOTING FAILURES AND BUILD RELIABILITY
  Part 3 describes failure types and their origins, principles of efficient troubleshooting, and the importance of toolchain reliability for improving productivity and the developer experience. It also covers best practices for achieving build reliability leveraging failure analytics and flaky test management tools.
- Part 4 - ECONOMICS
  Part 4 provides a recipe for quantifying the observable and hidden costs of slow builds and tests as a basis for determining the potential ROI of a DPE initiative investment in improving build and test speeds and making troubleshooting more efficient. It also introduces a straight-forward ROI calculator (online tool) that you can use to estimate your own DPE initiative ROI leveraging your own data.

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

About the cover art

Charles Aweida and Ari Zilnik designed the cover art in collaboration with 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 use the following convention for calling out short anecdotes that 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 led 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.

Wayne Caccamo is the VP of Marketing at Gradle Inc. and has served in executive roles at several successful technology startups and has held leadership posts at Oracle and Hewlett-Packard where he was the founder and director of the HP Open Source Solutions Operation (OSSO).

Eric Wendelin leads the Analytics team at Gradle Inc. His mission is to make software assembly faster and more reliable through data. Prior to joining Gradle, he led engineering teams at Twitter and Apple.

Marc Philipp leads the Test Distribution team at Gradle Inc. as a Senior Principal Software Engineer. He is a long-time core committer, maintainer, and team lead for the JUnit open source project and initiator of the JUnit Lambda crowdfunding campaign that started what has become JUnit 5.

About Gradle Inc.

Our purpose is to empower software development teams to reach their full potential for joy, creativity, and productivity. We are the provider of Gradle Enterprise, the premier enabling technology for the practice of Developer Productivity Engineering and the company behind the popular open-source Gradle Build Tool used by millions of developers.

To learn more, contact us at: https://gradle.com/enterprise/contact/.

Introduction

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 from every industry and every region.

This book represents not only our learnings from that exciting journey but also what we see by working with cutting-edge teams from companies like AirBnB, Netflix, Microsoft, LinkedIn, Spotify, Twitter and many others. These teams make Developer Productivity Engineering (DPE) a dedicated practice and have executive sponsorship for this effort. The company as a whole realizes that their developer's ability to innovate at their full potential is critical for leading their respective industry. They have dedicated teams of experts with a sole focus on DPE to improve the toolchain, accelerate feedback cycles, and provide a great developer experience. The lessons we present in this book apply to any technology stack, language, framework or build system.

DPE is not only for visionaries and innovators with formal dedicated teams in place. It should appeal to any company whose success depends increasingly on the productivity of development teams and ensuring a positive developer experience to recruit and retain talent. Specifically, we wrote this book for:

- **Senior software developers** that are responsible for their teams' productivity and morale of their team(s) and are particularly focused on increasing trust and confidence in the toolchain to encourage the right behaviors
- **Build engineers** responsible for enabling fast and frequent software delivery, improving trust and confidence in toolchain, and making teaming and collaborations with the development team more efficient
- **DPE managers and champions** looking for a comprehensive guide to the DPE practice that they can reference, share, and use to socialize the key concepts and tools across the organization
- **Engineering management** (e.g. VP or Director level) open to new strategies to decrease time-to-market (TTM) and minimize cost while maintaining and improving quality
- **Members of the development team** looking to make an impact, build influence and gain respect by bringing solutions that address personal and team frustrations, or individuals that see themselves as future DPE champions but need help understanding and building a business case
- **CI teams** who want to learn how to contribute to the developer experience beyond providing CI as a service. This includes faster CI builds and more efficient ways for developer and CI engineers to troubleshoot failed CI builds, as well as contain the spiraling costs of their CI infrastructure and compute resources. Most importantly, this means changing from a reactive pattern to a proactive pattern when it comes to CI reliability.

No matter what your role or objective, this book is chock full of personal experiences, stories, and lessons learned that address how to:

- Understand the importance of fast feedback cycles and catching errors earlier
- Quantify the costs of a 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 to avoid unnecessary work such as caching and distribution that speed up feedback cycles and improve CI efficiency
- Efficiently discover and eliminate some of the most egregious and avoidable productivity bottlenecks such as frequent errors and false signals like flaky builds and tests

Hans Dockter, Founder and CEO of Gradle, Inc.

Part 1 - DEVELOPER PRODUCTIVITY ENGINEERING DEFINITIONS, DRIVERS & CONCEPTS

Developer Productivity Engineering Defined

The Intuitive Case for Developer Productivity Engineering

Forming a Developer Productivity Engineering Team—Why Now?

Developer Productivity Engineering's Place in the Broader Developer Productivity Solution Landscape

DPE Solutions Overview

1. Developer Productivity Engineering Defined

Developer Productivity Engineering (DPE) is a software development practice used by leading software development organizations to maximize developer productivity and happiness. As its name implies, DPE takes an engineering approach to improving developer productivity. As such, it relies on automation, actionable data, and acceleration technologies to deliver measurable outcomes like faster feedback cycles and reduced mean-time-to-resolution (MTTR) for software failures. As a result, DPE has quickly become a proven practice that delivers a hard ROI with little resistance to product acceptance and usage.

This contrasts with a Developer Productivity Management (DPM) approach that views developer productivity as a people challenge that can be addressed by applying management best practices and developer activity-based metrics.

Organizations apply the practice of DPE to achieve their strategic business objectives such as reducing time to market, increasing product and service quality, minimizing operational costs, and recruiting and retaining talent by investing in developer happiness and providing a highly satisfying developer experience. DPE accomplishes this with processes and tools that gracefully scale to accommodate ever-growing codebases.

Most mature industries have engineering disciplines dedicated to making production efficient and effective. This includes chemical engineering and industrial engineering sectors that have process experts widely understood to be essential to their firm’s economic success and long-term competitiveness.

Developer Productivity Engineering is of similar importance when it comes to the production of software with similar stakes in achieving a company’s economic potential and global competitive impact. It is quickly becoming a critical practice within software engineering organizations that strive for industry leadership across diverse and highly dynamic marketplaces.

2. The Intuitive Case for Developer Productivity Engineering

By Hans Dockter

When making a business case for investing in any kind of business or technical initiative it's important to provide management with a strong ROI story. Fortunately, the practice of DPE delivers a hard and compelling ROI. A financial model for generating an ROI estimate is described in Part 4 of this book.

However, process improvement initiatives will fail without bottom-up, grass-roots user acceptance. The **intuitive** case for DPE is not based on the impact to management and financial objectives like development costs, time-to-market, and quality. Instead, it is based on a visceral understanding of the pains experienced every day by developers from problems that DPE addresses — slow builds and inefficient troubleshooting, to name two — and the impact they have on the developer experience and productivity.

The intuitive case for DPE starts with and is built upon the intuitive premise that software productivity depends heavily on toolchain effectiveness. This is because toolchain effectiveness can be defined as the ability to facilitate developer creative flow, collaboration, and team productivity. It concludes that software productivity suffers without DPE.

2.1. The creative flow of software developers depends on toolchain effectiveness

Software development is a highly creative process. It is similar to the creativity of an experimental scientist. Scientists have a hypothesis and then enter into a dialogue with nature via experiments to determine if the hypothesis is correct. Our hypothesis is that our code runs and behaves as we expect it to and our dialogue is with the compiler, unit tests, integration, performance tests, and other feedback mechanisms to validate our hypothesis. The quality of the creative flow for your developers depends on the timeliness and quality of the response.

In an ideal world the answers are available instantaneously, they are always correct, and changes are immediately available at runtime. This was more or less our experience when we started developing software as kids. It was fun, productive, and addictive. But what if, when you wrote your first Hello World, it took a full minute to render that familiar and friendly greeting? Would you have continued with coding? I'm not sure I would have.

2.2. Effective collaboration depends on toolchain effectiveness

Now, many of us are enterprise software developers and instant feedback has given way to long wait times, complex collaborative processes and additional dialogue with stakeholders and tools. For example, first you must correctly interpret a business expert or customer requirement and translate it into code. Then, you need to run your code to get feedback on the correctness of your interpretation. Few developers operate in a vacuum in these environments. The quality of the collaboration depends on how quickly you can iterate.

2.3. Overall team productivity depends on toolchain effectiveness

Team productivity is determined by the quality of the individual developer's creative flow and the effectiveness of your collaboration. Both are strongly correlated to the toolchain effectiveness. Your toolchain can create a major bottleneck for your productivity and your success. The developer toolchain in the enterprise is an ever-changing and complex piece of machinery with ever-changing and complex inputs. Sub-optimal feedback cycle times and unreliable feedback is the norm. This is commonly the case since most enterprises do not performance manage or quality control their toolchain by leveraging modern acceleration or troubleshooting technologies. In a recent study, over 90% of survey respondents indicated that improving time spent waiting on build and test feedback was a major challenge (source: TechValidate). As a result, developers are stuck believing that slow, unreliable feedback is a given in the enterprise. It doesn't have to be that way.

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

2.4. Software productivity suffers without Developer Productivity Engineering

As a company grows, it needs a dedicated team of Developer Productivity Engineering experts. This practice has the sole focus of optimizing the effectiveness of the developer toolchain to achieve a high degree of automation, and fast and reliable feedback.

The job of DPE is to improve developer productivity across its entire lifecycle and to close the significant and growing gap between actual and potential software development team productivity. Too often CI/CD teams are not responsible for developer productivity and make decisions that optimize for other metrics, such as auditability or vulnerability checking without taking into account the adverse effects to developer productivity. The priorities and success criteria for a DPE team are primarily based on data that comes from a fully instrumented toolchain. For such a team to be successful it needs a culture that values and is committed to the continuous improvement of developer productivity.

A developer's happiness depends on their productivity. Most developers want to work in an environment that enables them to work at their full potential. Organizations that cannot provide such an environment are seeing a drain on their talent leading to project slowdowns or outright abandonment.

The economic benefits of not applying this practice are dramatic. Development productivity improvements provide significant return for every dollar invested in software development. The amount of features that can ship is heavily affected by it. The productivity gap that comes from not applying the practice of Developer Productivity Engineering is a major competitive disadvantage. Innovation is at the heart of any business to survive. Today most innovations are software-driven. In fact, a December 2020 report from IDC predicted that 65% of the global GDP will be digitally transformed by now (2022). That means that the productivity of software development teams is crucial for your business to survive, and developer productivity on the whole has a global impact.

Developer Productivity Engineering applies broadly across the software development process landscape. It's impact starts with the coding process and extends to CI/CD to the point where DevOps distributes and scales production applications.

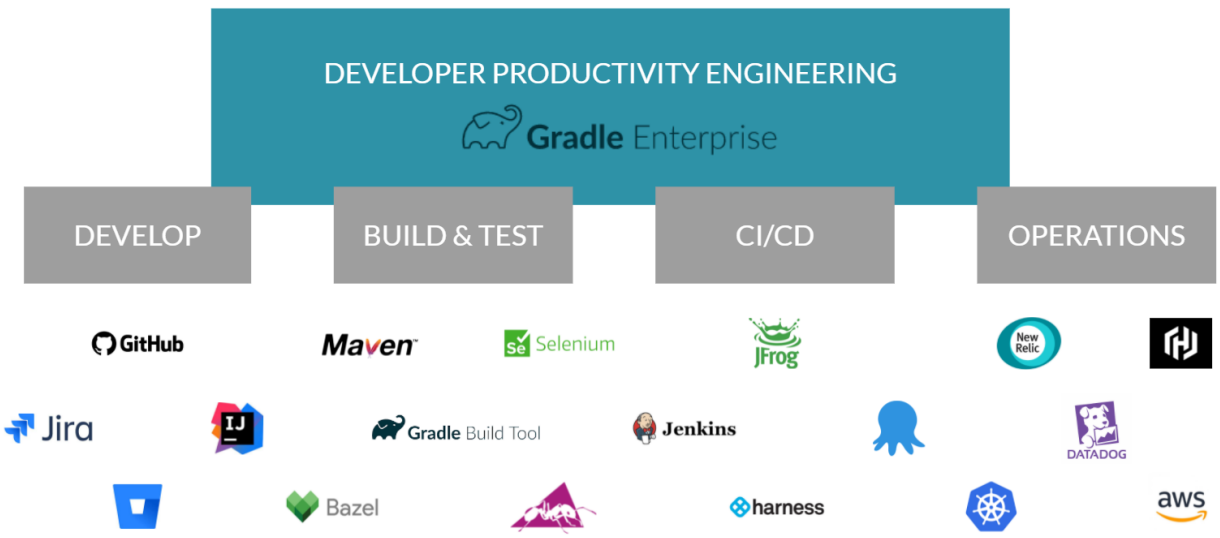


Figure 1. DPE is not a new phase of the SDLC, but a practice that supports multiple phases with a current focus on Build, Test and CI.

To be clear, Developer Productivity Engineering is not about treating developers like factory workers but instead like creative craftspeople. It is about the developer experience, unlocking creative flow, restoring the joy of development, and facilitating better collaboration with the business.

It's easy to understand at an intuitive level why DPE is important to developer productivity and the developer experience. Fortunately, one can also make a straightforward economic argument for DPE based on a hard ROI that has been proven in over a hundred of the most important technology and global business brands. This is covered in Part 4 - ECONOMICS.

3. Forming a Developer Productivity Engineering Team—Why Now?

By Wayne Caccamo

DPE is becoming the most important movement in software development and delivery since the introduction of Agile, lean software development, and DevOps methodologies and tools. For many companies, initiatives and experiments to boost developer productivity have been spearheaded primarily by lead developers and build engineers, who strive to mature their processes and find enlightened ways to create a cutting-edge development experience for their teams.

Rather than this largely ad hoc approach, the most successful DPE-enabled businesses formalize those investments by establishing a dedicated team. Here are three reasons why it makes sense for your company to launch a DPE initiative now.

3.1. DPE requires focus

To achieve excellence in any discipline requires focus, and focus in business requires dedicated resources. Dedicated teams have a better chance to succeed for several reasons, particularly when it comes to DPE. First, the mere existence of a DPE team signals that this is a management priority with executive buy-in, and that developer productivity engineers (like release engineers and build engineers) are first-class members of the engineering team.

Second, by standing up a dedicated team, management acknowledges that a separate set of goals and metrics are needed to assess productivity and, ideally, will make meeting those goals the focus of at least one person's full-time job.

Dedicated teams also enjoy the autonomy to manage and develop their own talent and unique skill sets. For DPE to succeed, it is necessary to find software development professionals that have a passion and affinity for supporting development teams in matters of productivity and efficiency.

Finally, the mere presence of dedicated teams reflects a milestone in the maturity of any new business or technical discipline. In this context, it is noteworthy that many high-profile companies that have achieved or aspire to a high level of maturity in this area have long ago established dedicated DPE teams.

3.2. DPE delivers a compelling ROI

While you can realize many DPE benefits without a dedicated full-time team, the magnitude and sustainability of those gains won't be as great. For example, it's easy to quantify the annual dollar savings from using DPE build and test acceleration technologies, like build caching and test distribution, to shave minutes off your average build time:

```
cost per engineering minute * average build time reduction (minutes) * average number of builds per year
```

A dedicated team not only provides the expertise to implement these DPE acceleration technologies and optimize their results, but can also ensure that build, test, and restore times don't regress as the

project or product evolves .

For many moderately sized development teams this quickly translates into double digit savings, measured in engineering years, or budget dollars measured in millions. This alone can justify your investment in a dedicated DPE team and tools many times over, without even considering the myriad other hard and soft benefits. Such benefits include dramatically reduced mean-time-to-resolve (MTTR) for software incidents, better management of flaky tests and other avoidable failures, and more efficient CI resource consumption.

3.3. DPE provides competitive advantage

Winning, or at least not losing the war for software development talent, is mission-critical for many companies. As a result, attracting and retaining top talent depends on the quality of the developer experience you can provide.

We know that the best way to improve the developer experience is to give them back the time they spend doing things they hate, like waiting for builds to complete and debugging software. That time can be better spent doing the one thing they love most—building great software features that delight end users.

These days, many companies use the existence of their dedicated DPE teams as a recruiting and retention tool because it demonstrates their commitment to providing a rewarding developer experience.

Businesses have a history of establishing dedicated teams at all levels to drive strategic initiatives aimed at gaining a competitive edge or remaining competitive, such as innovation and digital transformation teams. For example, it's not uncommon to see dedicated teams driving productivity and efficiency initiatives in areas like business process engineering (e.g. Lean, Six Sigma), manufacturing (e.g. Total Quality Management, JIT), and industrial process engineering. Given the global strategic importance of software development and delivery, shouldn't the experience of creating that software have the same priority?

3.4. Conclusion: Forward-thinking development organizations are all in with DPE

For most organizations, DPE should no longer be viewed as an informal, reactive, and opportunistic job to be done by developers in their spare time. Forward-thinking engineering teams have discovered that:

1. Systemic developer productivity gains require organizational focus
2. The business case for DPE is a no-brainer
3. The reward for taking action is sustainable competitive advantage

4. Developer Productivity Engineering's Place in the Broader Developer Productivity Solution Landscape

By Wayne Caccamo

As alluded to in chapter 1, there are two primary—and complementary—approaches to improving developer productivity. Both aim to use resources more efficiently and help ship code faster, while optimizing the developer experience.

Developer Productivity Management (DPM) focuses on the people, and answers questions like, "How can we get more output out of individual developers and teams by defining and tracking the right metrics?" Such metrics typically help to quantify output, evaluate performance and competencies, build and manage teams, and optimize collaboration and time management.

Developer Productivity Engineering (DPE) focuses on the process and technology, and answers questions like, "How can we make feedback cycles faster using acceleration technologies to speed up builds and tests?" and "How can we make troubleshooting easier, faster, and more efficient using data analytics?"

In analogous terms, suppose you're the owner of an auto racing team. To win, you need both the best drivers and the fastest cars. In the software engineering world, the drivers are your developers, while the cars are your highly performant processes and technology toolchain.

This chapter surveys the relative advantages and disadvantages of DPM and DPE, while ultimately recommending a DPE-first approach. In other words, no matter how good your drivers are, it's hard to win races without the fastest cars.

4.1. Advantages of Developer Productivity Management

DPM can provide management insights through cohort analysis

DPM aims to give engineering management in-depth insight into the performance of their individual developers and teams who are frequently geographically spread throughout the world. These insights may be used to increase leadership's understanding of the organization, give decision makers more confidence, and provide opportunities for continuous improvement. DPM does this through cohort analysis, which filters productivity metrics and trends by team locations (on-site or local), domain (front-end or back-end), seniority, geography, team sizes, and technical environment.

In contrast, while DPE also provides activity and outcome metrics by team member and technical environment, these metrics are mainly used to identify, prioritize, and fix toolchain specific problems like builds and tests.

DPM may be useful in building and managing teams and time

DPM can help you build high-performing teams through cohort analysis data about ideal team sizes, makeup, and structure. Further, DPM metrics can be used to improve project forecasts and estimates, prioritize work, evaluate project health and risk, and assign developers to teams based on demonstrated skills and needed competencies.

Managing time is a primary focus of many DPM tool vendors. Using data to simply strike the right balance between coding time and meeting time, for example, can make a tremendous difference in team output. Deeper DPM data that focuses on the days of the week and times of the day that optimize meeting effectiveness can further improve productivity.

DPM may be useful in optimizing individual performance

DPM metrics can be used both to measure individual output, and as inputs for evaluating individual performance. They can also help with identifying and closing skill gaps and allowing individuals to showcase their own domain expertise and competencies. Perhaps most importantly, DPM can be used to monitor changes in individual activity levels (e.g., velocity) to detect if a developer is stuck and requires some kind of intervention.

4.2. Limitations of Developer Productivity Management

Human productivity metrics are flawed

DPM assumes there is a reliable and meaningful way to measure developer productivity. After all, you can't measurably improve productivity if you can't define a baseline. In "The Pragmatic Engineer," Gergely Orosz argues that you can't measure the productivity of knowledge workers like doctors, lawyers, chemists, and software engineers. He concludes that for software engineering in particular, individual metrics can be gamed, don't tell the whole story or, in some cases, are completely arbitrary.

Many metrics violate "Do no harm"

The fact is there is no real consensus on what is a valuable human productivity metric that makes sense across the industry. Moreover, many human productivity metrics can lead to behavior changes—such as the number of commits per day—that can be counterproductive and antithetical to the business objectives they are designed to support.

In "The Myth of Developer Productivity" (Nortal Cloud Team, 2018), the authors review several common developer productivity management metrics, such as hours worked, source lines of code (SLOC) produced, bugs closed, function points, and defect rates. They conclude that most are more harmful than useful in managing developer productivity, in part because they lack context. Consider the developer whose commits and code review metrics are down, but only because s/he spent more time mentoring others and helping them increase their own productivity. Also consider the "highly productive" developer who cranks out many lines of inefficient code and who will fly under the radar when a business isn't scrutinizing both people and outcomes.

Gut instinct is still king

According to developer productivity thought leader Bill Harding ("Measuring developer productivity in 2020 for data-driven decision makers"), due to the challenges of measuring developer output and using that data to improve team productivity, managers often hone their own intuition to serve as their 'north star'. Thus, a key goal of DPM is to find and retain technical managers whose past intuitions led to success. But how do you define success, and how is that measured?

Big brother can be a tough sell

Another potential DPM challenge is gaining buy-in from the developer teams that are being measured. Defendants of DPM dismiss the inevitable developer pushback as largely a matter of messaging and claim that it's a natural evolution of a maturing software development management process. They point to the great companies that initially resisted external measures of performance—like Consumer Reports, Glassdoor, and Yelp—and later embraced them, failing to acknowledge that they had no other option.

DPM is a choice. To be effective, it should be implemented with great care, considering your company culture, objectives, and ability to execute (i.e,. the ability to translate insight into action). Developers understand that their activities may be monitored (e.g., whether they are active on Slack or not) and understand the need for measurement. The sensitivity stems from the purpose of the measurement and the underlying problems management is trying to solve. When management focuses on understanding the root cause of delivery speed variations, for example, DPM may be welcome, but expect the developer antibodies to surface when metrics are used as input for comparative performance evaluation.

DPM solutions may collide with other tools

The benefits that DPM targets often overlap with existing and established technologies, including IDEs, collaboration and workflow tools, project management systems, and learning management platforms. As a result, it's up to you to decide where one system/process starts and another ends, and deal with potentially multiple integration points to rationalize workflows and facilitate reporting.

In contrast, DPE is a mostly accretive solution in terms of the kind of data it generates and the ways in which it solves problems and augments workflows. Integrations with systems of record for management reporting (e.g., Tableau) or CI integration (like Jenkins and TeamCity) are simple and straightforward.

4.3. Advantages of Developer Productivity Engineering

DPE provides measures of outcomes

Perhaps the clearest distinction between DPM and DPE is that DPM provides mostly measures of activity and DPE focuses more on measuring outcomes. Compare DPM metrics like hours worked, SLOC, and bugs closed to DPE metrics such as build and test times for local and CI builds, failure rates, and build comparison data that can be used to pinpoint the root cause of performance and

stability regressions.

DPE metrics cannot be gamed, are not arbitrary, and can be more easily marshalled to make logical connections to higher-level performance and efficiency metrics. These metrics can, in turn, be aligned unequivocally with C-level business objectives, like time-to-market, quality of services, talent retention, and cost savings.

DPE delivers measurable ROI

If you Google "developer productivity measurement benefits," you will be hard-pressed to find a straightforward explanation from industry commentators or DPM tool vendors on the benefits of tools used to measure individual developer and team productivity. It may be that a direct answer to the query is hard to find not because it does not exist, but because it's obvious and assumed. That is, more productive developers write higher quality code that ships faster and at a lower cost. It would not be unreasonable to take this on faith, but to know if the return on your investment in DPM is a lot or a little is difficult to say.

With DPE, measuring ROI for several key benefits is straightforward. For example, to calculate the hard savings from achieving faster build and test feedback cycles using DPE acceleration technologies like build caching and test distribution, the formula is simple: Multiply the average time saved waiting for builds to complete by the number of builds per year to get total time savings in engineering years, and then multiply that by the cost of a fully-loaded engineering year. For many moderate-size development teams, this quickly translates into double-digit savings, measured in engineering years, or millions of dollars if put into budgetary terms.

DPE improves the developer experience

While DPM tool vendors may claim that many features support a better developer experience (e.g., team assignments that play to their skill and competency strengths), the limitations of DPM described above suggest DPM may do more to negatively impact the developer experience. As a result, developers are not the driver for DPM adoption—in fact, they are more likely to be the primary skeptics, while software engineering management is the primary user/practitioner.

Compare this scenario to DPE, in which the senior and lead developers are not only the drivers of DPE initiatives, but are also the power users of the tools. Furthermore, DPE focuses primarily on directly improving the developer experience by improving the tooling. The benefits they enjoy are not soft, indirect, or unquantifiable. They often directly experience the benefits of faster build times and more efficient troubleshooting a dozen times or more a day. As a result, it is not uncommon for companies with established and dedicated DPE teams to use their DPE practice as a tool for recruiting and retaining top engineering talent.

DPE has more robust success stories

It is difficult to find evidence of DPM successes beyond testimonial quotes. In contrast, you can find hour-long webcasts presented by DPE leaders within some of the world's leading business and technology brands. They cover business drivers and objectives, solutions deployed, verifiable benefits achieved, and next steps in their DPE journeys.

4.4. Limitations of Developer Productivity Engineering

DPE is relatively new and less well understood

Compared to DPM, DPE is a relatively new and emerging discipline that is socializing new concepts and solutions—including local and remote build caching, test distribution, predictive test selection, build scans, and build and test failure analytics. Some concepts get confused, which may create some initial friction in their adoption, such as the functional difference between build caches and dependency repositories. This is a normal part of the growing pains associated with establishing a new software solution category.

DPE solutions do not yet support all developer environments

DPE solutions, like Gradle Inc.'s Gradle Enterprise, currently focus on the JVM and Android ecosystems and may provide more limited support for other environments. It is expected that DPE solutions will expand their footprint over time to fully support all popular developer environments. While some DPM tool metrics are toolchain/technology specific, some apply more broadly across ecosystems.

4.5. Conclusion

Key Characteristics	**DPM**	**DPE**
First level focus is on measuring…	People	Process & Technology
Second level focus is on measuring…	Activity	Outcomes
SDLC lifecycle focus is on…	All phases	Build & Test & CI
ROI is primarily	Soft	Hard

DPM and DPE are complementary, not competing approaches to improving developer productivity. However, they can compete for the lion’s share of both your mind and your wallet. If you’re living in a JVM ecosystem, start with investing in the fastest car: Developer Productivity Engineering.

5. DPE Solutions Overview

The diagram below outlines the key pain points and benefits addressed by DPE and aligns them with the key available solution technologies and tools designed to deliver these benefits. Part 2 of this book describes in some level of detail the key concepts needed to understand why these solution capabilities are important, how they work, and what you can expect in terms of business impact.

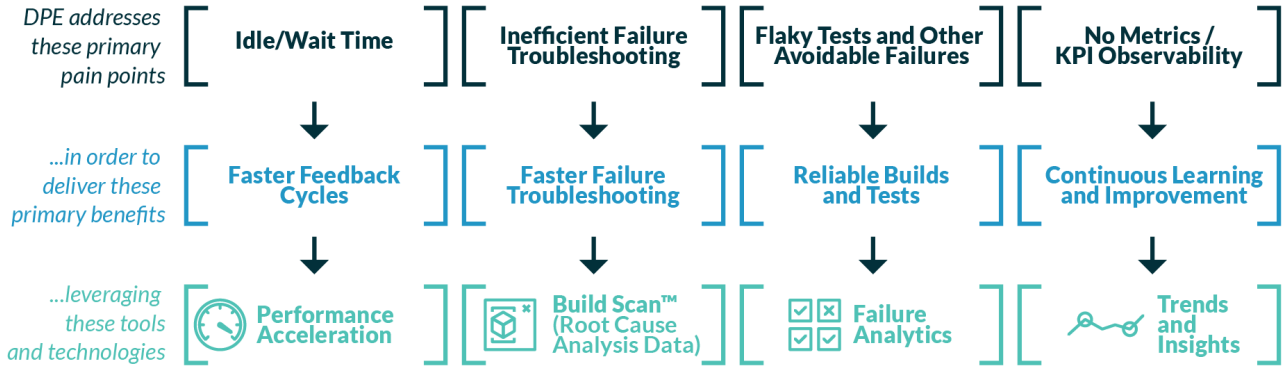


Figure 2. A Developer Productivity Engineering solution framework that aligns pain points, benefits and key solution technologies

Part 2 - FEEDBACK CYCLES AND ACCELERATION

The Essential Role of Faster Feedback Cycles

Faster Builds by Doing Less with Build Caching

Test Distribution: Faster Builds by Distributing the Work

Performance Profiling and Analytics

6. The Essential Role of Faster Feedback Cycles

By Hans Dockter

A central tenet of Developer Productivity Engineering is that feedback cycles across the software development and delivery lifecycle should be minimized. It's not possible to truly buy into the practice of DPE without having a full appreciation of the role feedback cycles play in optimizing developer productivity. One reason is the positive effects it has on engineering behaviors and productivity at the individual and team level in ways that might not be immediately apparent.

This diagram shows the complex relationship between the effects of faster build and test feedback cycles on software developer behavior and productivity and the quality benefits of those effects.

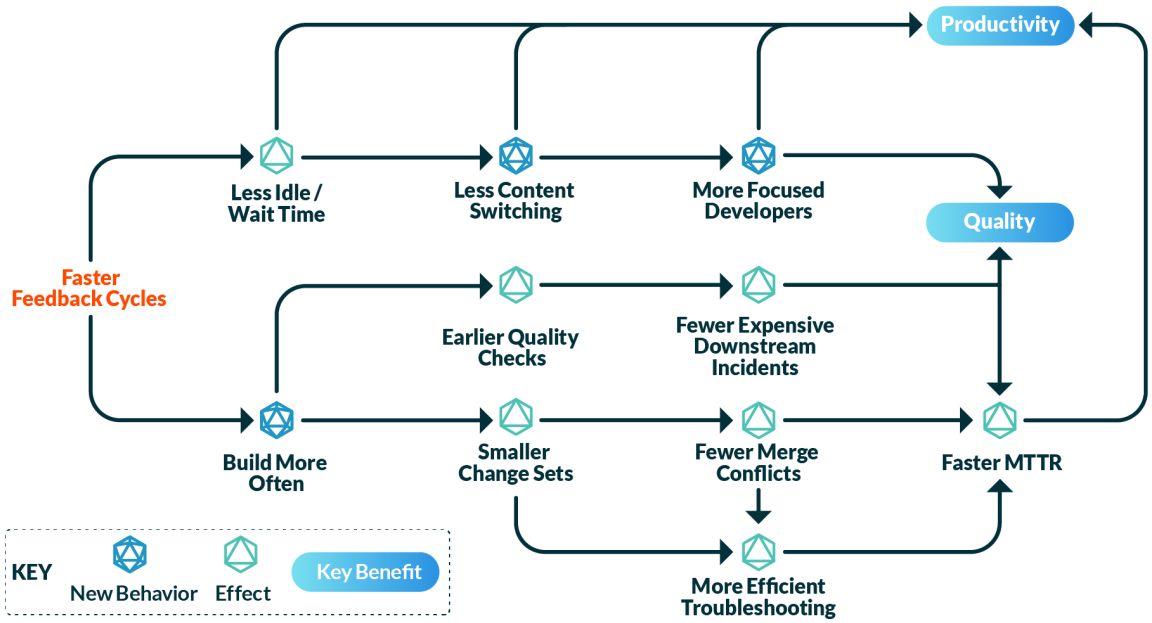


Figure 3. The anatomy of the effects and benefits of faster feedback cycles

6.1. Faster builds improve the creative flow

Previously the impact from faster feedback cycles on creative flow was discussed. Consider a customer example comparing feedback cycles from two teams.

	Team 1	Team 2
Number of developers	11	6
Build time (with tests)	4 minutes	1 minute
Number of local builds	850	1010
Number of builds per developer	77	180

The developers on Team 2 ask for feedback twice as often as the developers on Team 1. This correlation can be observed between build times and the number of builds consistently across organizations, even when the build time is only a few minutes. The most likely reason for this is that their builds and tests are faster. Such a correlation is not surprising when build times are painfully long. Interestingly, even in environments where builds and tests are not perceived to be slow, the data shows that the number of builds is inversely-proportional to the duration.

The underlying cause of this behavior is that our brains function much better when disruptions to the creative flow are minimized. Endorphins help us enjoy our work during flow states. But, when developers have to ask the toolchain for feedback, which ideally is frequently, it has to be fast or they will not stay in the flow. With faster builds and tests developers can find a more productive balance between the conflicting forces represented by the need to interrupt oneself to get feedback and the need to preserve one's creative flow.

6.2. Developer time is wasted waiting for builds and tests to finish

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 and a lot of the build and test time is pure downtime.

The aggregated cost of wait time is surprisingly high even for very fast builds and even moderate improvements are often worthwhile. The table below shows that, although the team with a one-minute build time is doing very well by industry-standards, reducing the build time further by 40% (down to 36 seconds) saves 44 developer days per year. For a team of six engineers, this is a significant productivity improvement.

Team	Number of developers	Local builds per week	Build Time	Build Time with DPE	Savings per year
Team 2	6	1010	1 minute	36 seconds	44 days

On a larger team, this scales exponentially as a team of 100 developers nearly cuts their build time in half for a savings of 5,200 developer days saved. Assuming 220 working days a year per developer, 5,200 developer days is roughly 25% of all engineering time spent. Significant performance improvements like this are game-changers for productivity.

Team	Number of developers	Local builds per week	Build Time	Build Time with DPE	Savings per year
Team 3	100	12000	9 minutes	5 minutes	5200 days

6.3. Longer builds mean more context switching

As build time increases, more and more developers 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 usually 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, developers pay the cost of context switching twice—once when going back to the previous task and once when continuing with the new one. This often costs developers 10-20 minutes per switch and the reason why most prefer to wait for builds that take less than10 minutes to run.

6.4. 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 effective average build time 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 both dramatically increase the average wait time as well as the context switching frequency.

6.5. Longer builds are harder to troubleshoot

The longer a build takes, the bigger the changesets will become. This is because for most organizations the build time is roughly the same regardless of the size of the changeset. This build time is a fixed tax paid to get any change through the software development lifecycle. If this fixed tax per build run is high, the logical way to minimize this tax is to not run builds as often. The amount of coding done between the build runs will increase.

Unfortunately, the tax is not fixed if the build fails. The more changes you have, the longer on average triaging takes. This increased debugging cost caused by larger individual changesets can be significant because it has to be paid for failing local builds, failing pull request builds, and failing integration branch builds.

Depending on how a CI process is organized, additional effects may come into play for the integration branch CI builds. The fewer integration branch CI builds per day that can be run due to their duration, the fewer merge points there will be. 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 this will potentially involve a lot of unrelated changes.

The most extreme cases we have seen are large repositories that take an entire day to build. If this build fails, it is possible that any one of the hundred commits could be the culprit and one should expect triaging to be extremely complicated and time-consuming. But even with much shorter build times, this effect may be experienced. In general, there is a non-linear relationship between the number of changes and debugging time.

With a build cache and modularization your build times will be on average much faster for smaller changes compared to larger ones, thus incentivizing a development workflow where small changes are pushed frequently and quickly to production.

6.6. Time to troubleshoot grows exponentially with problem detection time

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

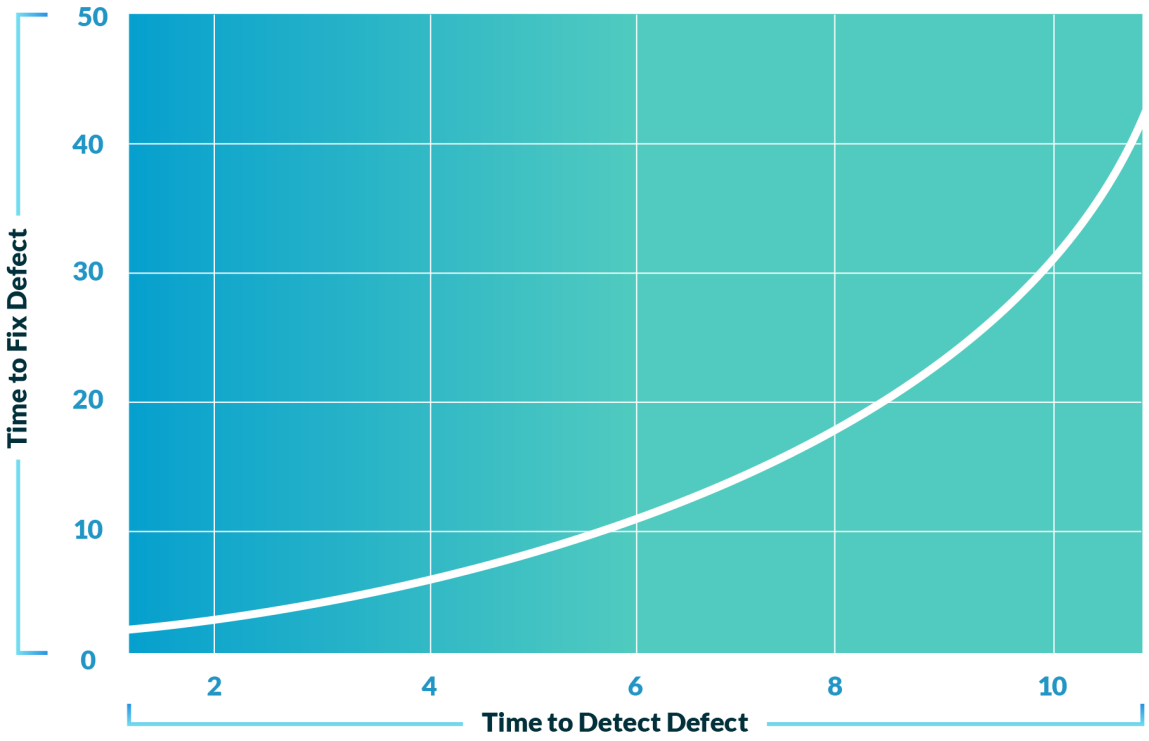


Figure 4. Fix time grows exponentially over detection time

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

There is a similar effect in a multi-repository environment where the consumer team is responsible for detecting if a consumer contract has been broken and organizing fixes as needed with the producer team. The more dependencies that have changed during a build run, the exponentially harder it becomes to figure out which dependencies and teams are responsible.

6.7. Longer builds lead to more merge conflicts

Both bigger changesets and more merged pull requests per integration branch CI build increase the surface area of the changes and thus the likelihood of merge conflicts. Another effect is at play here. Bigger changesets also increase the debugging time and thus the average time of a successful CI integration branch build. This again reduces the frequency of merges and for the reasons discussed, increases the likelihood of merge conflicts. This is a vicious circle. Merge conflicts are one of the most frustrating and time-consuming issues developers must confront.

6.8. The effect of build speed on large versus small projects

Smaller projects will benefit from faster builds because of less wait time and improved creative flow. They will also be affected negatively by larger change sets as their build time increases. Smaller projects usually don't suffer from overlapping merges as much as larger projects, as those are less likely to occur with a smaller 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 impacted and has to deal with the problem when this occurs. The triaging of that is often difficult and very time-consuming.

6.9. Many companies are moving in the wrong direction

As build times grow there is increasing friction between the process of getting feedback early and reducing the wait time before changes can be pushed.

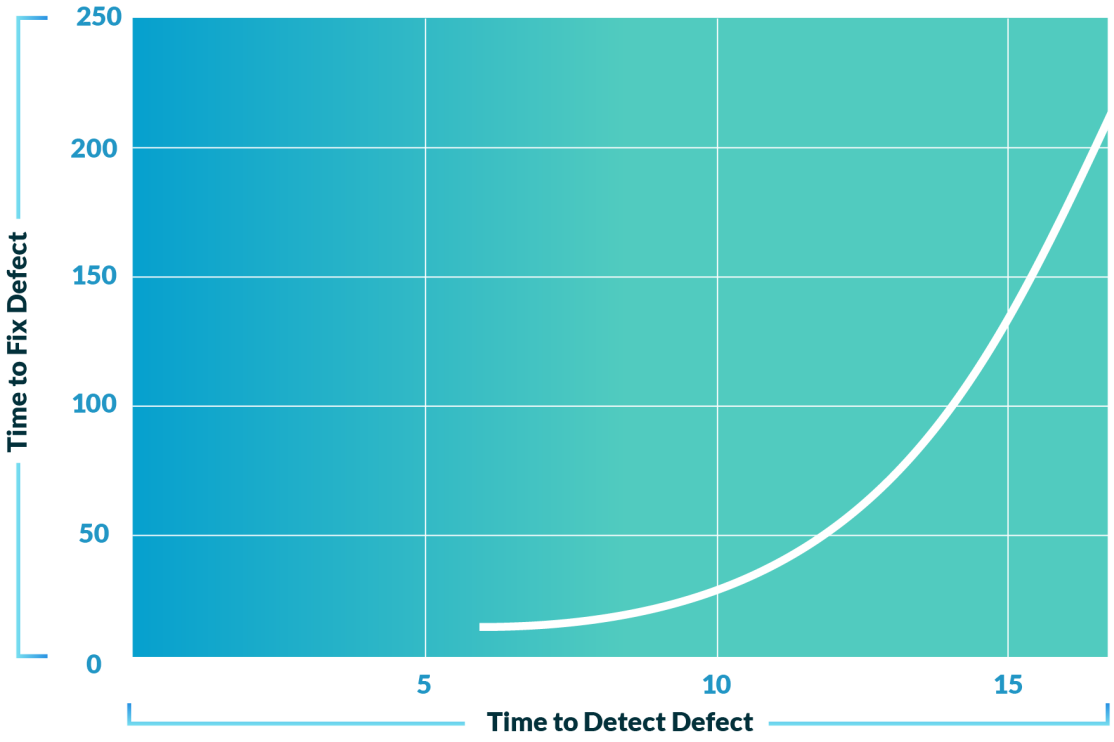


Figure 5. Delaying feedback exponentially increases fix time over detection time

This is a regrettable situation and most organizations do not yet have or are not even aware of the tools and practices that fundamentally solve this problem. Instead they go down the path of shifting the feedback later in the development cycle, or 'to the right'. This results in weakening the local quality gate by running fewer tests and the pull request build may be the first to run the unit tests.

This is the opposite of what a healthy CI/CD pipeline process looks like where feedback is provided as early and as conveniently as possible. Shifting feedback to the right will only prolong the

inevitable and all the compounding negative effects discussed previously will land even more forcefully because they will occur closer to production.

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

6.10. A final word of caution: The problem grows with your success

The problems described here will grow as developer teams expand, code bases bloom and repositories increase in number. Exacerbating this 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 will look for other opportunities.

The practice of Developer Productivity Engineering will put you in a position to improve or at least maintain your current productivity and truly reap the benefits of CI/CD.

STORIES FROM THE TRENCHES: IT'S A DOG EAT DOG WORLD

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 lamented: "If I only had the quality of your software engineers." The Bay Area company was very developer-productivity-focused and their engineering leader replied: "Guess where we got our developers from? From organizations like yours."

6.11. Conclusion

Faster feedback cycles encourage and reinforce good habits and developer best practices. Hopefully, this chapter has untangled the cause-and-effect relationship with other indirect benefits you may have not considered.

There is one other "soft" benefit that may yet be the most important: the impact on developer happiness. Fast feedback cycles mean developers waste less time waiting for builds and tests to complete and performing tasks they enjoy less like troubleshooting problems. The more time reserved for pursuing their passion—being creative and writing great code—the happier they will be.

And, a good way to discourage developers from engaging in negative behaviors is to minimize avoidable frustrations. Conversely, the best way to encourage positive behaviors is to make it easier to always do the right thing by investing in processes and tools that support a great developer experience. A good place to start is those that speed up feedback cycles.

7. Faster Builds by Doing Less with Build Caching

By Hans Dockter

The concept of build caching is relatively 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. Maven and Gradle multi-module builds are configured a bit differently. Here are some example build declarations for each:

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"
```

For many reasons, multi-module builds are good practice, even for smaller projects. They introduce separation of concerns and a more decoupled codebase with better maintainability as it prevents cyclic dependencies. Additionally, modular applications will have an easier time adapting to future development paradigms. For example, highly modular API-driven applications are easier to deconstruct into microservices than ones without that level of modularity. Most Maven or Gradle builds are multi-module builds. Once you start using a build cache, increased modularization will enable even faster build and test runs by increasing the overall cacheability of the project.

When you run a build for such a multi-module project, actions like compile, test, javadoc, and checkstyle are executed for each module. In the example above, there are four `src/main/java` and `src/main/test` directories that need to be compiled. The associated unit tests will be run for all four modules. The same is true for javadoc, checkstyle, and so on.

With 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 it is necessary to rebuild and re-test every module from scratch:

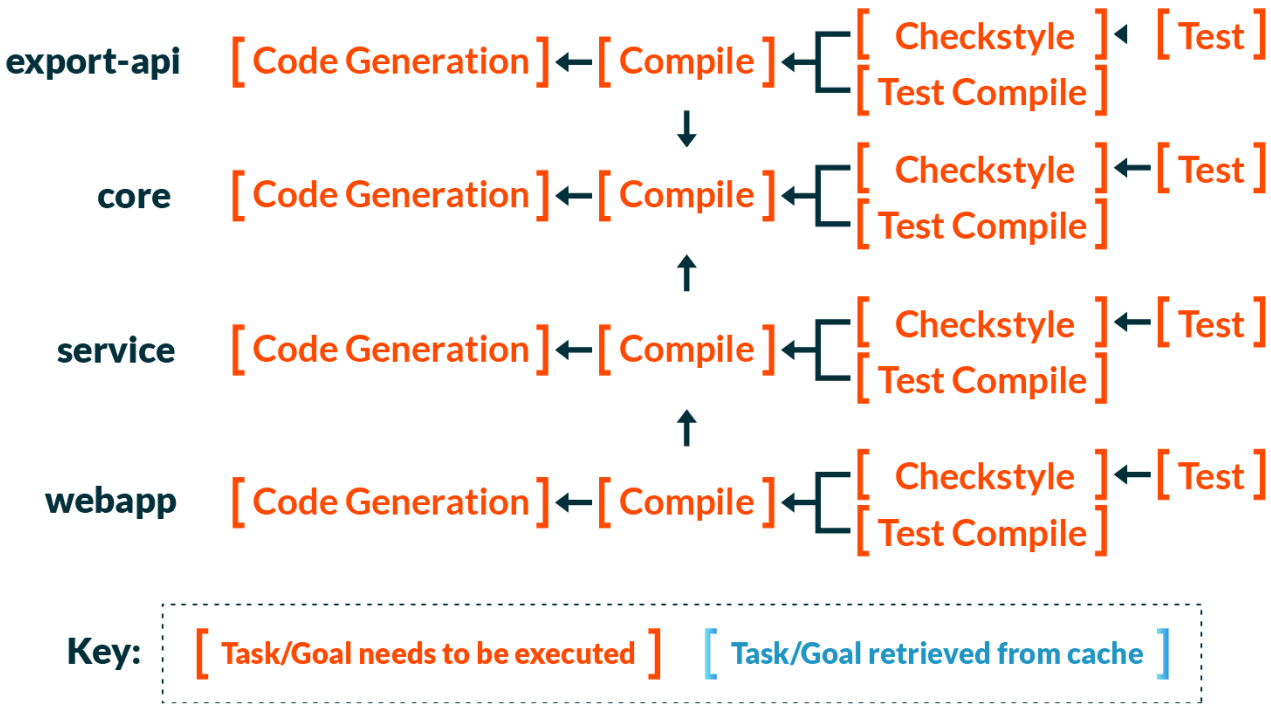


Figure 6. Rebuilding everything from scratch (arrows are dependencies between actions)

Gradle is an incremental build system. It does not require cleaning 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 cases where one switches between code branches, pulls new changes or performs clean CI builds it will still be necessary to build everything from scratch even when building with Gradle.

A build cache can improve this scenario. Let's say a project has been built once. After that, a line of code is changed in the `export-api` module from the example above. Next, an argument is added to a public method and it is assumed that no other module has a dependency on `export-api`. With such a change and the build-cache in place, only 20% of the actions need to be run.

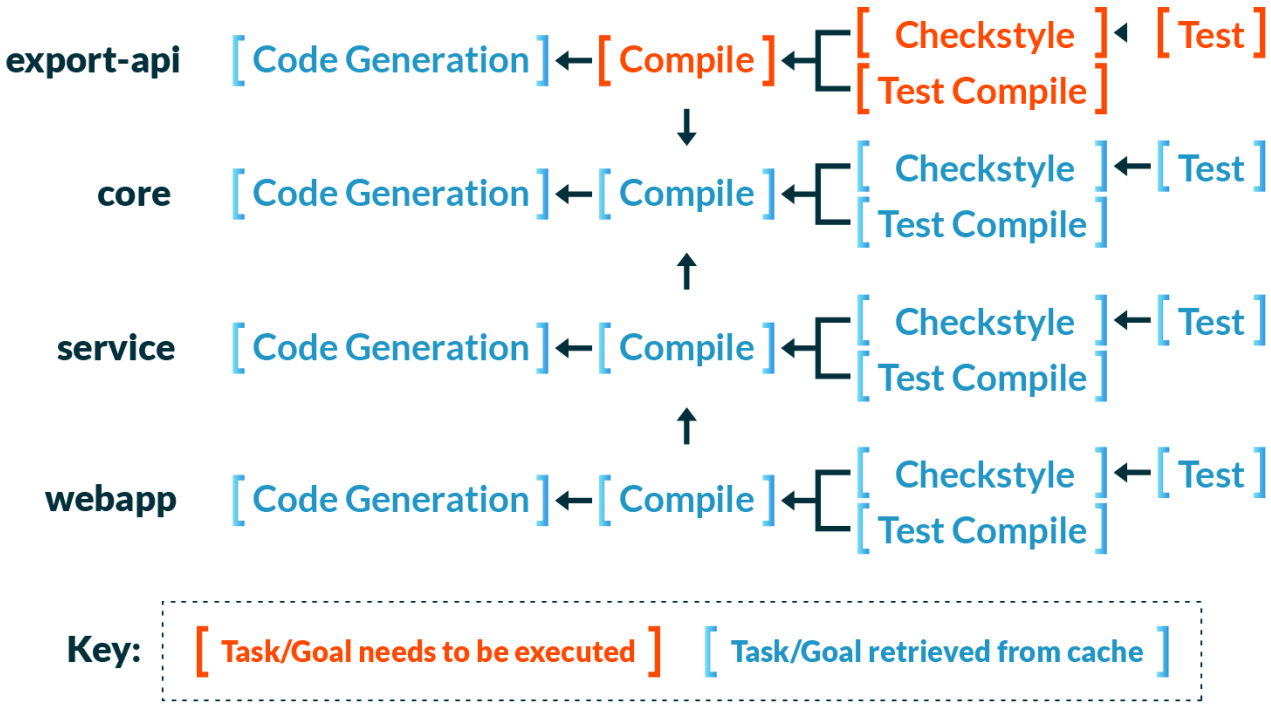


Figure 7. Changing a public method in the `export-api` module

How does this work? All 20 build actions have inputs and outputs. For example, the compile action 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 action has the test sources and the test runtime classpath as an input, and possibly other files. The outputs are the `test-results.xml` files.

The build cache logic hashes all inputs for a particular action and then calculates a key that uniquely represents those inputs. It then looks in the cache to see if there is an entry for this key. If one is found, the entry is copied into the Maven or Gradle build output directory of this module and the action is not executed. The state of the build output directory will be the same as if the action had been executed. Copying the output from the cache is much faster than executing the action. If an entry for a key is not found in the cache, the action will be executed and its output will be copied into the cache associated with the key.

In our example, four actions belonging to the `export-api` module had a new key and needed to be executed. That's because one of its inputs, the compile action of the production code (the source directory) has changed. The same is true for checkstyle. The compile action 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 action has a new key because its runtime classpath has changed for the same reasons.

As another example, let's say a method parameter needs to be added to a public method of the `service` module. `webapp` has a dependency on `service`.

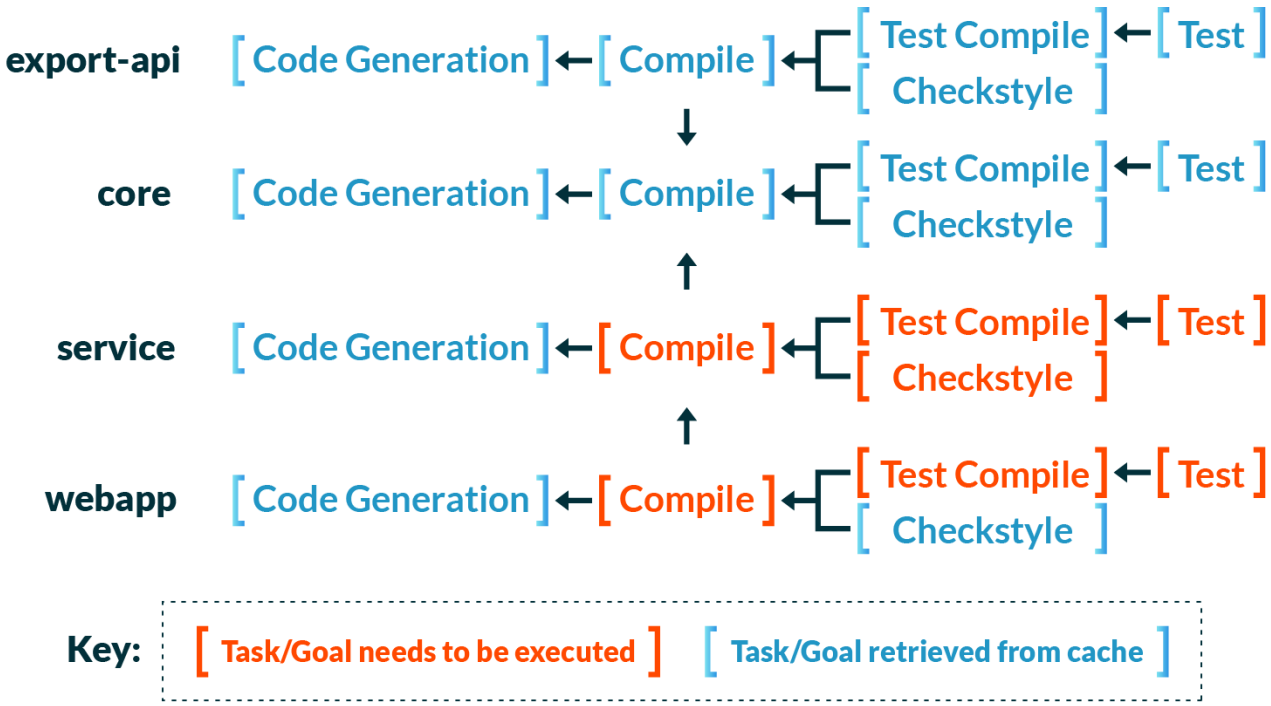


Figure 8. Changing a public method in the `service` module

Now, not only do the actions for the module that have changed need to be re-executed, but the actions for the dependent module will need to re-execute as well. This can be detected via the action inputs. Because the `service` module code changed, the classpath of the `webapp` compile and test compile action has changed, as well as the runtime classpath for its test action. As a result, all these actions need to be executed. Yet, compared to rebuilding everything, only 40% of the actions need to be executed.

Now let's use the same example as before, but instead of adding an argument to a public method in the `service` module, only a change to the method body will be made. A smart build cache can now do further optimizations:

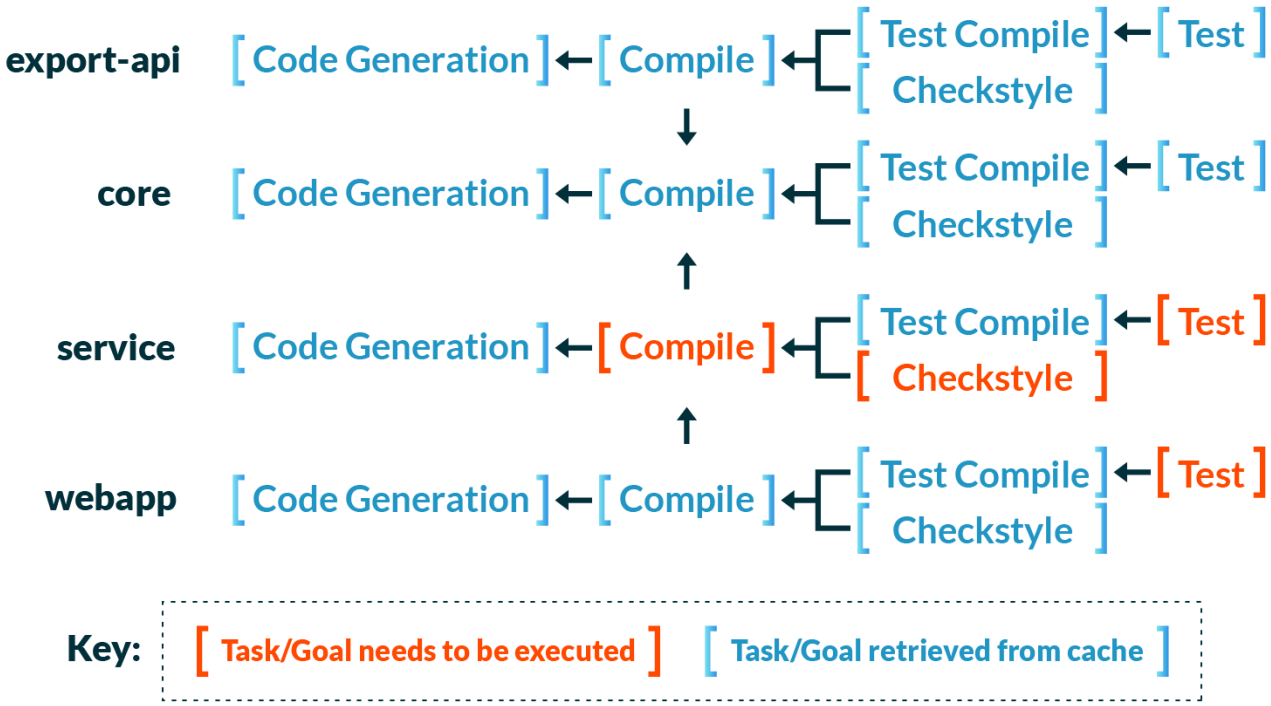


Figure 9. Changing an implementation detail of a method in the `service` module

The cache key calculated from the compile classpath only takes the public API of the classpath items into account. An implementation change does not affect that key, reflecting the fact that any such change has no relevance to the Java compiler. As a result, the execution of three compile actions can be avoided. For the runtime classpath of the test actions, implementation changes in your dependencies are obviously relevant and lead to a new key, which in turn, results in executing the test actions for `service` and `webapp`. With this optimization, just 20% of the build actions need to be executed.

Let's look at another example where every other module depends on the `core` module and a change is made to the implementation of a method in the `core` module.

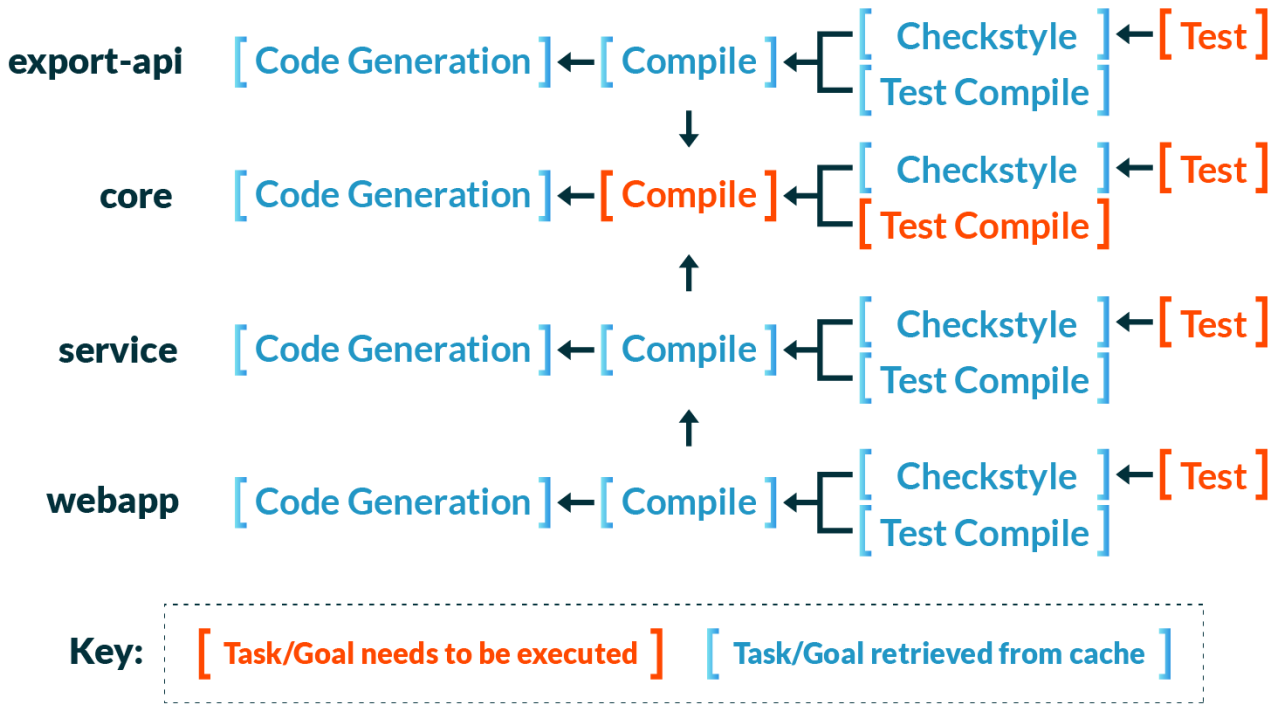


Figure 10. Changing an implementation detail of a method in the `core` module

This is a very invasive change but even in this instance only 30% of the actions need to be executed. Even given that executing the test actions will probably consume more time than the other actions, it still saves a lot in time and compute resources. The worst-case change would be adding an argument to a public method in the `core` module.

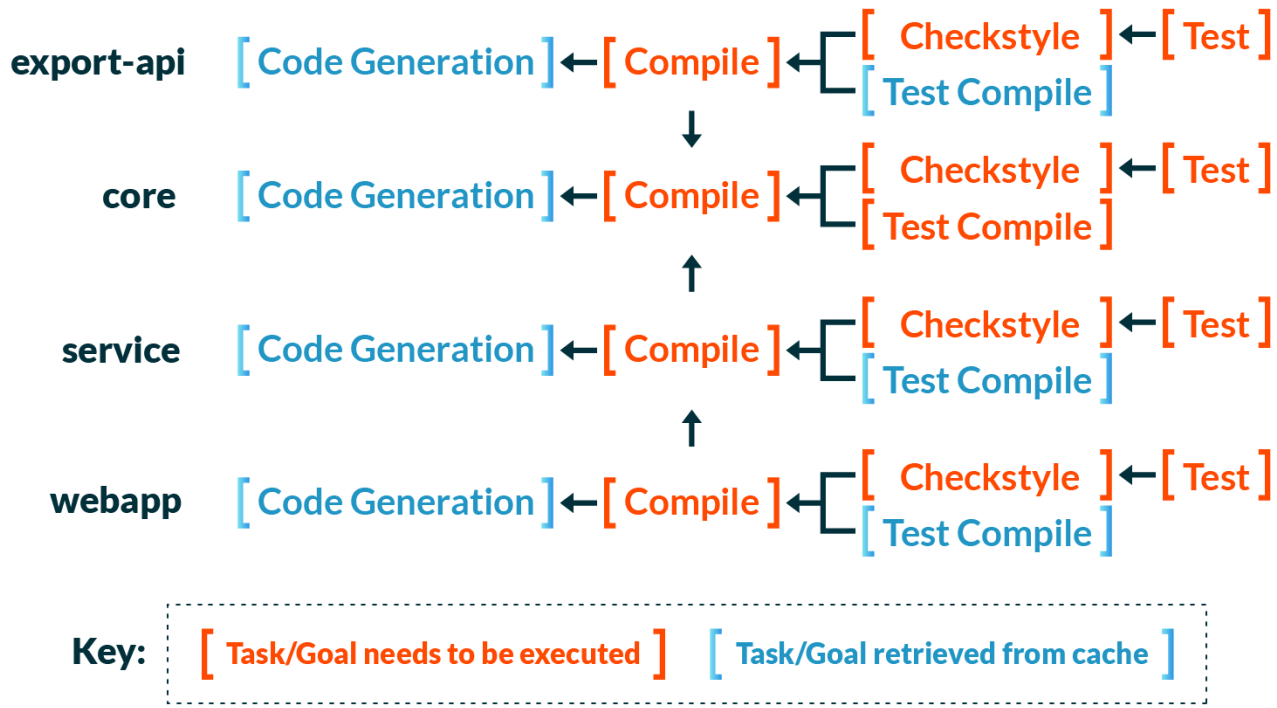


Figure 11. Changing a public method in the `core` module

Even here one still gets significant savings from the cache as only 65% of all the actions need to be executed.

7.1. Local vs remote build cache

There are two types of build caches. One is a local cache, which 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 code is committed and pushed. Many organizations weaken the local quality gate because of long build times. A local cache is key to strengthening the local quality gate and getting more feedback in less time before code is pushed. The other type is a remote cache, which shares build output across all builds in the organization.

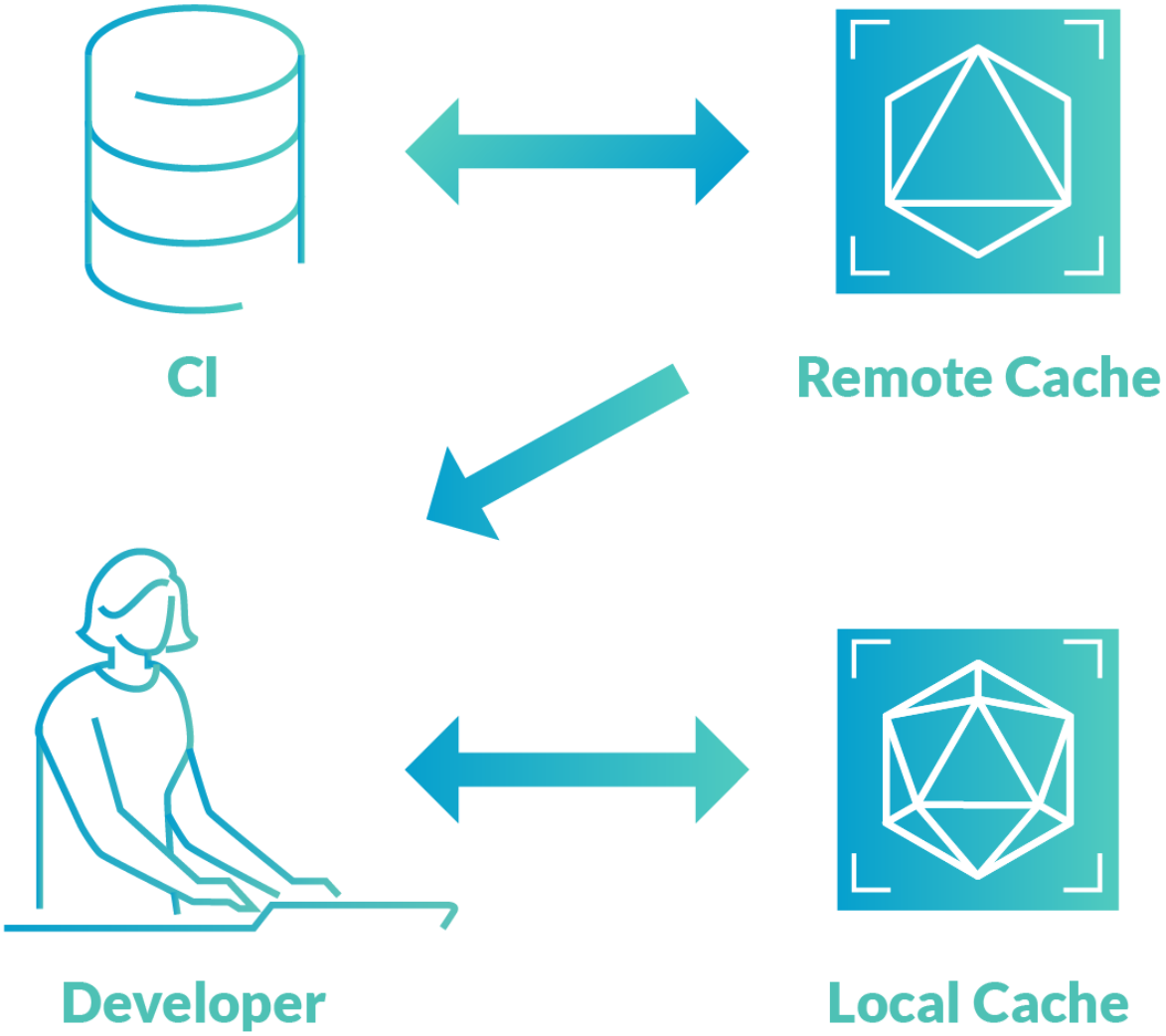


Figure 12. Cache Node Architecture

Local builds write into the local cache. Usually, only CI builds write to the remote cache, while both local and CI builds read from the remote cache. This speeds up the average CI and local build time significantly. You often have a pipeline of multiple CI jobs that run different builds against the same set of changes (e.g., a quick check job and a performance test job). Often those jobs run some of the same actions and a build cache makes this much more efficient.

A build cache not only improves build and test execution times, it also dramatically reduces the amount of compute resources needed for CI. This can be a significant economic saving. Furthermore, most organizations have a problem with CI queues. Specifically, new changes pile up

and there are no agents available to start the build. A build cache can improve agent availability considerably. The diagram below was provided by a company we work with when they started to introduce the cache and optimized its effectiveness.

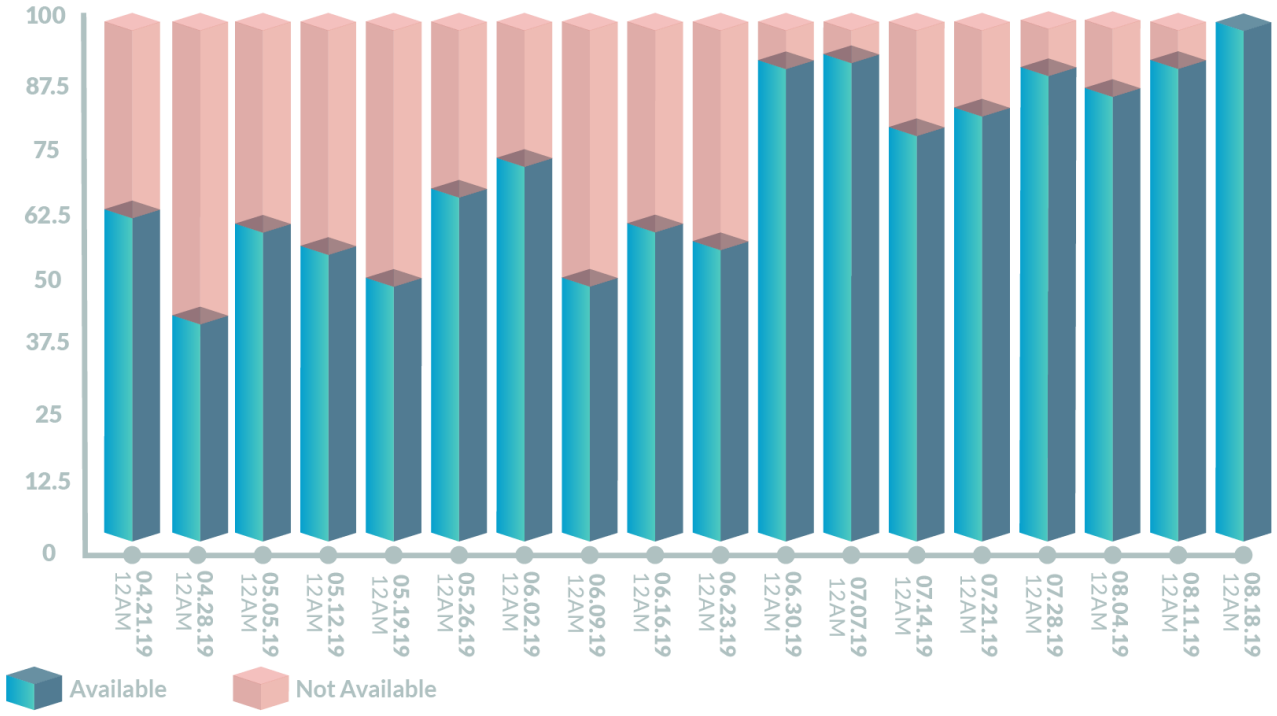


Figure 13. Improving CI agent availability by leveraging and optimizing the build cache

7.2. Build cache effectiveness

If all code and tests live in a single module there will be little benefit to caching, except on CI where there are often multiple CI jobs running against the same set of changes. Even with a few modules, the benefits can be substantial, and benefits grow exponentially with the number and size of modules. For larger multi-module builds, often half of all 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 that have some level of modularization. Investing in further modularization improvements not only reduces the technical debt related to evolving your codebase, it also provides immediate build performance benefits.

7.3. Sustaining cache benefits over time

To maintain build cache performance gains, build environments need to be continuously monitored. This includes the caching infrastructure. The infrastructure should support sufficient download speeds from the remote cache nodes, as well as storage capacity to preserve cache entries that are still in use.

Another important aspect is the reproducibility of builds. Volatile inputs are the enemy of caching. Let's say there is a build action that always adds 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. It's important to effectively detect such issues as they can have a major

effect on cache effectiveness.

7.4. Summary

Nothing is faster than avoiding unnecessary work. For my own company's projects, we can see a spectacular 90% average time saving by using previous build output via caching. Across industries a minimum of 50% savings can be observed from organizations that have started to adopt build caching. Build caching is a foundational technology to address the pain points of slow feedback cycles and achieve the obvious and hidden benefits elucidated in Chapter 6.

8. Test Distribution: Faster Builds by Distributing the Work

By Marc Philipp and Hans Dockter

Often when developers say their builds are slow, they really mean their tests are slow or perhaps they believe their builds are fine, but their tests are slow. Either way, test time is often the key driver of build times. Many of our large customers report that test execution time makes up 60%-90% of their build time. Many factors contribute to test time, including the growing number of integration and functional tests to cover a wide range of inputs, running tests sequentially, and dependencies on expensive external services.

When tests are slow, developers often don't run them locally but rely on their CI server to eventually notify them of their status. The negative effects of getting feedback less frequently and later is discussed in detail in Chapter 6. This is where test parallelism comes in and saves the day (or at least the hour).

This chapter introduces fine-grained, transparent Test Distribution which is a modern approach to test parallelism that accelerates test execution by taking existing test suites and distributing them across remote agents to execute them faster—locally and on CI. It contrasts with traditional test parallelism options described next.

8.1. Traditional test parallelism options and their limitations

No parallelism

Companies struggle frequently with any form of test execution parallelism. There are various reasons for this. For example, parallelism can be problematic when there are implicit dependencies between tests (i.e., one test can only succeed if another test runs first) or when test resources such as a test database cannot be used in parallel. Ideally refactoring tests will fix these problems. There are also less intrusive ways to mitigate these problems. For example, tests can be run with dependencies separately from tests with no dependencies so that the latter can be run in parallel. Not being able to parallelize your tests is a serious constraint which will become more and more costly as your test base grows. With available state-of-the art test distribution solutions, the delta between your theoretical achievable test execution time and your actual test execution time will be measured in orders of magnitude.

Single Machine Parallelism

Many organizations run some or all of their tests in parallel which often results in a significant acceleration. Obviously the parallelism is limited by the machine's CPU and memory, which has several implications:

- **Developer machines become unavailable for other tasks.** First, tests that consume a lot of resources on a local developer machine can make that machine slow and even unavailable to

perform other tasks.

- **Organizations overspend on local developer machine compute capacity.** Some organizations provide powerful workstations to all of their developers to mitigate the problem caused by tests consuming too much machine resources. We know organizations that spend $8 million per 1,000 developers to arm them with $8,000 workstations in addition to their laptops. While this improves the test execution time, even these machines have parallelization limits. When tests are not running, which is the majority of the time, the significant compute and memory resources of these high-end machines are usually heavily underutilized. Test Distribution is a much less expensive solution than investing in top-of-the-line workstations.
- **Organizations invest in expensive underutilized servers to run CI.** For builds that run on CI, parallelization and test execution time are similarly limited by the CPU and memory resources of the CI agent. Some organizations we work with use $150K+ machines as CI agents to push those boundaries. In general, providing very powerful machines as CI agents to improve test execution time is inefficient because it inevitably results in massively underutilized machine capacity. As soon as a CI build starts, machine resources are fully dedicated to this particular CI build. The CI build is doing other work and not just running tests that will not benefit from those machine resources. For example, I/O heavy tasks like downloading dependencies don't need powerful compute resources. The rest of the build actions are usually not as parallelizable as the test execution, which is another reason why this machine might be underutilized. Even tests cannot run simultaneously since different tests have different dependencies on other build actions. As a result, there may only be a small time window when the machine is fully utilized. Compare this with remote test agents that can be reallocated from one build to another while those builds are running. This underutilization will significantly increase your infrastructure costs if you run 1000's or 100,000's of CI builds per day. Further, utilizing the most powerful machines comes at a premium. The price per compute resource is more expensive than for less powerful machines. Most importantly, those machines will still limit your ability to run tests as fast as you want.
- **Fragility due to resource sharing.** Single-machine parallelization requires tests in multiple local JVMs to not interfere with each other. They share the same file system and they often share the same "process space" so servers started from tests need to use different ports. A fine-grained test distribution solution overcomes those constraints by running tests only sequentially per agent.

CI Fanout

To achieve parallel execution of tests, many organizations create multiple CI jobs per build, where each CI job runs on a subset of the tests on a different agent. This approach has many major disadvantages. However, because for many teams this has been the only option, it has proven extremely valuable compared to not leveraging any form of test distribution. What are the disadvantages?

- **Reporting is broken.** Test reports are separated per agent and as a result there is no aggregate test reporting.
- **Debugging is inefficient and painful.** Running the same build many times often makes debugging build issues a nightmare. It may be necessary to deal with 10's or 100's of build logs and any build flakiness will add to the confusion, This leads to a poor user experience . Finally,

CI and build analytics is heavily skewed by this approach.

- **No efficient load balancing.** The test partitioning has to be done manually and is almost impossible to optimize. As a result, test execution time will not be well balanced across the agents.
- **CI administrators may become overburdened.** The administrative CI overhead of running tens or hundreds CI jobs for the same logical build step is significant.
- **The necessity to run full builds per agent impacts feedback cycles and resource consumption.** For each CI agent a full build must be run before a subset of the tests is run. This wastes resources and slows feedback cycle times. A build cache will help tremendously but even then there will be significant overhead per CI job by running full builds. This overhead not only creates waste but it also limits how finely-grained tests can be distributed. Due to coarse-granularity, it may still be necessary to use single-machine parallelism to achieve acceptable test execution times and this may introduce fragility as discussed in [sec-fragility].
- **CI fanout only works on CI.** Perhaps the biggest disadvantage with this approach is that it only works on CI. Chapter 6 discusses how important it is for the developers to get quick feedback to stay in the creative flow. With CI Fanout a pull-request has to be created just to get fast feedback on tests. For developers to stay in the flow and be able to ask for feedback frequently, they must not be responsible for running tests fast.

8.2. Capabilities of a fine-grained and transparent test distribution solution

Here are some key capabilities of modern test distribution solutions:

- **Available for all builds regardless of where invoked.** Test Distribution should be available to all builds, not just those triggered by CI. This means Test Distribution should work from the local command line, from CI and from the IDE (if the IDE delegates test execution to the build system).
- **No change to developer workflow.** With modern test distribution solutions, the workflow for developers stays exactly the same, but instead of running tests locally, tests are run transparently and automatically in a distributed fashion. This also means there is only a single build invocation and a single CI job for achieving this. As a result, the CI user experience will be much better compared to CI fan out. If properly implemented, test report aggregation happens as if tests are all run locally.
- **Significantly reduced overhead.** The build steps that are prerequisites for running the tests need to run only once. This results in a dramatic reduction in resource consumption.
- **More reliability and fewer dependencies.** Efficient fine-grained distribution allows for sequential execution of tests per agent which increases the reliability. See also [sec-fragility].
- **Balanced test execution load between test agents.** Test Distribution test history and dynamic scheduling services can be used to properly balance the test execution load between the different test agents to optimize test execution and compute resources. Considering that many local and CI builds run at the same time, a Test Distribution service is necessary that manages the allocation of the test agents efficiently and elastically and according to the priority of the different build types without adjusting the CI pipeline. For example, the service should be able

to spin up additional test agents to accommodate increased demand and stop them again when they are no longer needed to save costs.

- **Local build execution fallback.** Test Distribution solutions should always support the ability to fall back to local build execution if no remote test agents are available or provide an optional hybrid local/remote execution mode to maximize available resources.

8.3. Test Distribution complements build caching

Build caching and Test Distribution are complementary and should always be used together for optimal results. Build caching is the fastest possible test accelerator as it avoids executing a subset of the tests all together if the inputs for a given set of tests have not changed.

Depending on the type and size of the code change, sometimes the acceleration impact of either solution can vary widely from build to build. When little needs to be rebuilt because most of the build output is retrieved from the build cache, the impact of Test Distribution will be small. If a change leads to a vast number of cache misses, the impact of build caching will be insignificant and the acceleration will come almost exclusively from Test Distribution. For many builds, results will fall somewhere in the middle of these two extremes.

If projects are hardly modularized or not modularized at all, then build caching will not add much value and the impact of Test Distribution will be massive. But this is not a good state to be in as Test Distribution can be very resource-intensive. Build caching is a huge mitigating factor for resource consumption. For this reason as well as many others, it is best to invest in better modularization.

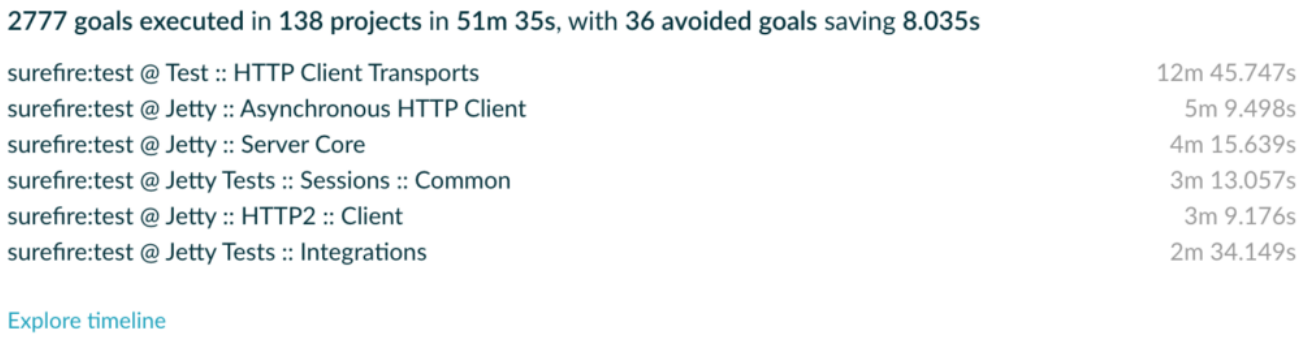
Ideally build caching and Test Distribution combine to form a virtuous cycle that provides very fast and efficient feedback cycles. This leads to more frequent builds with smaller changesets. This, in turn, results in even more effective build caching. The net effect is that fast and efficient feedback cycles can be achieved continuously as the codebase grows.

8.4. Test Distribution case study - Eclipse Jetty project

This case study demonstrates how Test Distribution and build caching was used with very little effort to reduce the total build time from over 50 minutes to about 15 minutes for a typical code change. Options for reducing build time even further by optimizing the build for Test Distribution will also be discussed. In this scenario, the Eclipse Jetty open-source project used the Test Distribution extension of Gradle Enterprise to achieve these results.

Establishing a Performance Baseline

The Jetty project contains 145 modules that comprise about 450,000 non-comment lines of Java code. After cloning the project the first build is ready to be run. The screenshot below shows that the build takes 51m 35s. This will be used as the baseline and referred to as (0). The summary screenshot below shows the total execution time along with the slowest goals – all of them test goals.



The timeline shows that 92% of the total build time (47m 35s) is spent executing tests. Maven provides a means to parallelize test execution by running multiple forks of the test JVM. However, as alluded to above the number of JVMs – and hence tests – that can be run in parallel is limited by the build machine's CPU and memory resources.

Assessing the Effectiveness of Test Distribution

To assess the effectiveness of Test Distribution, it was enabled for all Jetty test goals and a build with 10 remote agents was run. The build (DT1) takes 22m 35s (18m 37s for test goals) instead of the initial 51m 35s (0). When digging a little deeper into a tool called the Build Scan™ (described in detail in the following chapter)), it is evident that for most test goals there are a few test classes dominating the overall execution time.

Test Distribution uses test classes as the unit of execution. Hence, a test goal can never be faster than any of its individual test classes. To demonstrate that splitting slow test classes reduces the overall execution time, 6 test classes were split. When running the build again, the overall build time (DT2) drops to 20m 12s (16m 14s for test goals).

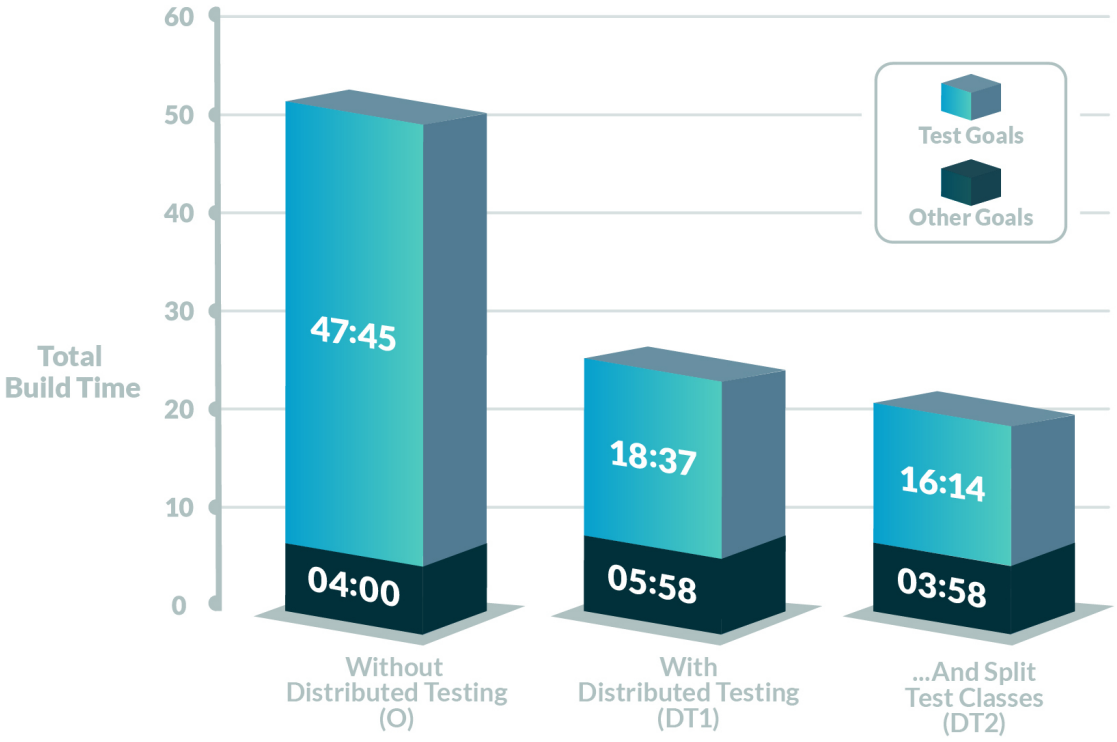


Figure 14. Impact of Test Distribution on execution time

As the above results show, Test Distribution dramatically reduces test execution time, the degree to which depends on the project. Projects with slow test goals tend to benefit more than those with a larger number of faster test goals. While Jetty falls into the latter category, Test Distribution still resulted in reducing test execution time from 47m 35s to 16m 14s. This represents a ~3x speedup. Allocating more agents could reduce this time even further.

Combining Test Distribution with the build cache

To simulate a typical developer use case, a small change to the implementation of a private method in the jetty-client module is made. Running another build without Test Distribution yields a total execution time of 39m 37s (C1) of which 36m 33s is spent executing tests. Some modules don't depend on jetty-client so they are not affected by the change and their goals' outputs can be loaded from cache. Thanks to compile avoidance, even compile goals of dependent modules can be loaded from cache since the application binary interface (ABI) (i.e. the visible API, they are compiled against did not change). However, all tests have to be run again since the code on the test runtime classpath differs.

After making another small non-ABI change in the same place as before, the build with build cache and Test Distribution enabled is ready to run. This time, the build (C2) takes 14m 50s (11m 50s for tests) which is a 2.7x speedup compared to the build without Test Distribution (C1) and a 3.5x speedup compared to the initial build (0).

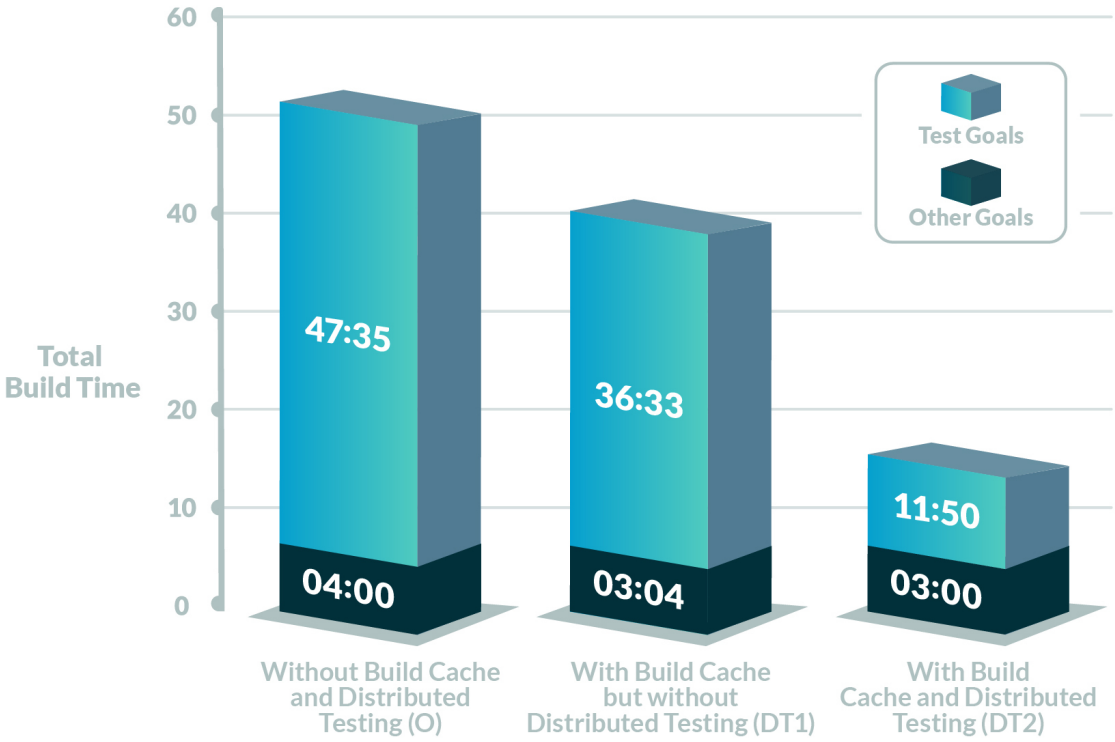


Figure 15. Impact of Test Distribution and build cache on execution time

Enabling parallel goal execution

Jetty's build is a large multi-project build (145 modules), which begs the question of how Maven's multi-threaded mode affects build times. In the results observed so far, Maven was executed with a single thread (the default) and hence all test goals were executed sequentially. As a consequence, test classes from different goals were not executed in parallel – even when Test Distribution was enabled.

Two additional builds were run while enabling Maven's multi-threaded mode (via the -T 1C command line argument, (i.e., one thread per CPU core). Without build cache and Test Distribution, the build takes 35m 27s (OP); with both enabled, it finishes after 10m 49s (C2P). That's a 3.3x speedup and is similar to comparing single-threaded builds. It is fair to conclude that Test Distribution and the build cache work well in both cases.

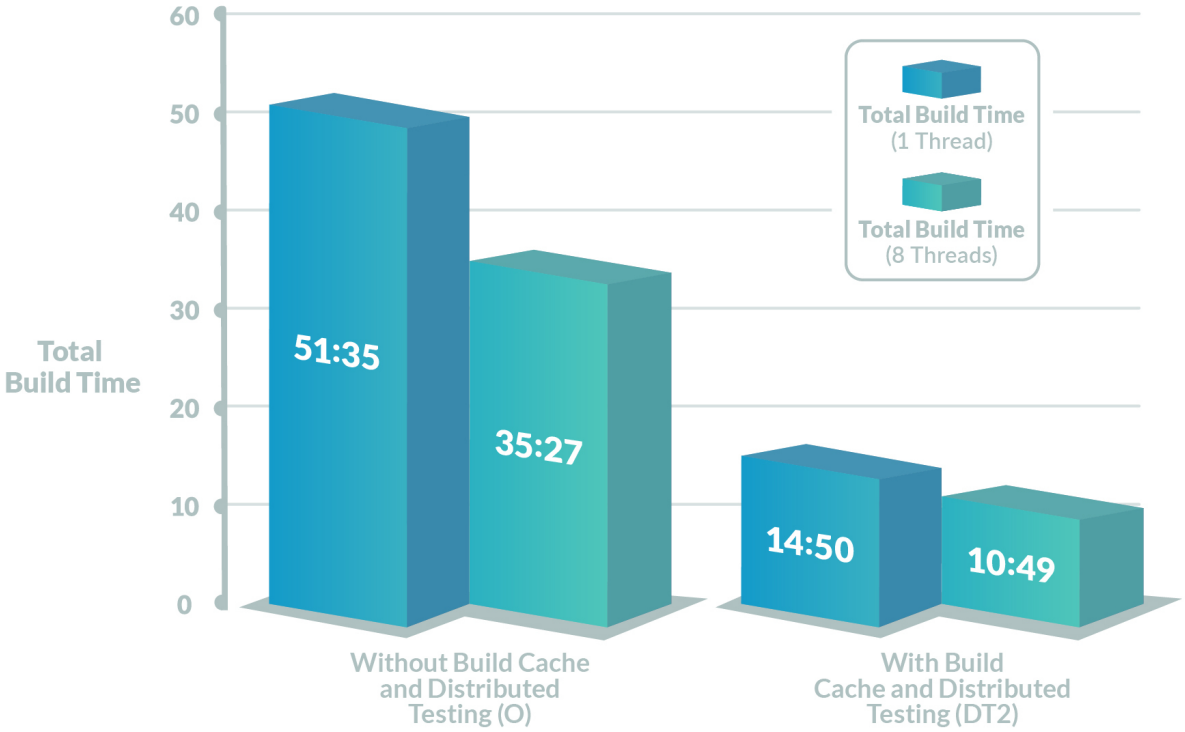


Figure 16. Test Distribution and build cache execution time while running single- versus parallel-threaded builds

8.5. Conclusion

Code changes in software projects often require large parts of the test suite to be re-executed – even when using the build cache. Thus, test execution usually dominates overall build times. Test Distribution accelerates test execution by extending test parallelism from a single machine to many, and increases developer productivity by shortening feedback cycles. Unlike other solutions, such as manually partitioning the set of tests into groups for CI builds, it also works for local builds, requires little setup, and produces a single report and Build Scan for all tests.

9. Performance Profiling and Analytics

By Hans Dockter, Eric Wendelin

With 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 by providing data visibility, analytics, and insights has so far not applied productivity practices to itself. In this chapter we will introduce the concepts of Maximum Achievable Build Performance (MABP) and Actual Build Performance (ABP) and how working with data is the key to keeping the delta between them as small as possible. A grasp of these concepts provides a foundation for understanding performance profiling concepts and tools introduced later in this chapter.

9.1. Maximum Achievable Build Performance (MABP) vs Actual Build Performance (ABP)

For any given software project there is a maximum achievable build performance which is a function of the technology stack. For example, the maximum achievable build performance for a Scala project will be lower than that for a Java project because the Scala compiler is significantly slower than the Java compiler. There is nothing that can be done about this, unless one is capable and has the resources to write a custom Scala compiler that is faster than the default compiler. There might be good reasons to choose Scala over Java, but the consequence will be a lower MABP. The same is true for the choice between Gradle and Maven. There may be good reasons to choose Maven, but the MABP will be higher with Gradle.

A high MABP is not worth much if the ABP is low. For example, Java has a very fast compiler, but if poorly written custom annotation processors are used, the actual compile time might be very long. Reducing the delta between MABP and ABP is where the majority of DPE performance-related benefits accrue. At the same time, if the MABP is low, the return on investment from reducing the delta of the ABP is also low.

That is why DPE focuses not only on closing the MABP-ABP gap by improving actual performance, but also on increasing the MABP.

The higher the MABP, the more it makes sense to invest in closing the gap with the ABP. For example, for single-module Maven builds (involving a large single-repository Java code base), modularizing this code base to increase ABP may close the gap moderately by increasing the ABP derived by better leveraging parallel Maven builds. However, if build cache is enabled, the MABP will increase dramatically and the payoff for the modularization effort will be enormous as will the investment in using DPE data and analytic tools to optimize the build cache. Since the ABP can never exceed the MABP, raising the MABP ceiling is the only way to ensure that continuous performance improvement is practiced over time.

It is important to note that the acceleration technologies discussed in previous chapters like build cache and Test Distribution fundamentally change the DPE investment calculation in multiple ways. First, those technologies easily increase the MABP by an order of magnitude for any build system they support. Second, they can mitigate some negative effects on the MABP from various

technology choices. For example, by using a build cache, compilation is performed substantially less often. This means the difference between the Java MABP and the Scala MABP is reduced. Also, in the case of choosing between Gradle and Maven, Maven has no incremental build capabilities (a major reason why Gradle is much faster). With build caching Maven has a coarse-grained equivalent to incremental builds and the performance difference between the two is still significant but not as enormous as it was before.

9.2. Recognize the importance of inputs

Here are some scenarios where the ABP is significantly lower than the MABP:

- **Parallel builds not leveraged for Maven or Gradle.** Sometimes it is as easy as switching it on to get significant performance benefits.
- **Parallel builds with low utilization of compute resources.** This can be caused by a single large module being on the critical build path. Breaking it up would significantly accelerate parallel builds. Alternatively, removing obsolete dependencies from this module can achieve a similar effect.
- **Inefficient caching due to modules that change frequently.** Modules that change frequently may have a lot of other modules that depend on them. Because of the changing cache key for downstream build actions, the build actions always need to be rebuilt. Splitting those modules might significantly increase the number of cache hits.
- **Inefficient caching caused by build actions with unreproducible output.** An example of this could be a source code generator that adds a timestamp to the generated source files. This may drive the ABP to be lower than MABP since every build action that depends on the output of that build action will have a very low cache effectiveness.
- **Build actions that use a lot of memory** This can be caused by under or overallocation of memory.
- **Slow execution time for I/O heavy build actions** A virus scanner that is scanning the build output directory or slow hard drives could cause slow executions.
- **Slow Compile Time** This can be caused by the addition of low-performing annotation processors that may provide only minimal benefit.
- **Large test classes that limit the effectiveness of Test Distribution** Test classes with a large number of tests can limit the time savings that can be achieved by using Test Distribution as they may prevent a fine-grained and well balanced distribution across the test agents.
- **High latency between developer machines and caches** This can be caused by new office locations that do not have a build cache or binary cache nearby.

This list goes on and on and on. What's important to realize is that the software toolchain is complex machinery that includes the IDE, builds, tests, CI and many other components. It has complex inputs that are always changing and the toolchain performance and reliability is sensitive to these inputs. This includes infrastructure components like the hard disks, but also includes the build cache nodes, the CI agents and binary repository manager. These components and inputs individually and collectively interact to impact toolchain performance. For example, dependency download time is often a bottleneck for CI builds. Yet most organizations have no idea how much time is spent downloading dependencies and what the network speed is when this occurs.

The code itself is also a complex input for the toolchain. For example, developers add annotation processors, new languages, and new source code generators that might have a significant impact on the toolchain performance. And yet I haven't interacted with a single company that knows 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 examples are memory settings for the build and test runtimes, code refactorings that move a lot of code between modules, and new office locations. As said, the list goes on and on.

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 quickly see trends and pathologies. The second is to effectively determine the root cause of problems.

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 a drop is observed in that number they react by looking at the parameters of the vessel environment such as the temperature, pressure, and concentration of certain chemicals. This usually allows them to determine or significantly narrow down the root cause for the drop in production. Comparable instruments are needed for optimizing the software production environment. The tech industry needs to practice what it preaches.

LESSONS FROM THE TRENCHES: SMALL THINGS MATTER

For years one of our customers with a large Maven build suffered from slow builds. Once they started instrumenting their toolchain they learned that 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 one and two minutes. The reason was that they were using SSD drives locally. They achieved a massive CI speed up just by replacing the hard disks, an action that they would not have known to take without looking at the data.

9.3. Data is the obvious solution

It is impossible to have a great developer experience if your toolchain is not instrumented and data from every toolchain execution is not collected. This data should be collected with four objectives in mind.

1. To continuously monitor high-level ABP metrics and trends
2. To be able to drill into any past toolchain execution and view a very comprehensive performance profile to find ABP bottlenecks that can be removed
3. To be able to compare performance profiles with different inputs to understand what the root cause is of any ABP regression or improvement
4. To continuously monitor low-level aspects of the toolchain for particular pathologies, (e.g., the cache-hit rate for a particular build action across all builds to effectively find common root

causes for ABP regressions)

Without the data these capabilities provide, it is almost impossible to find performance related issues that occur irregularly (so-called flaky performance issues) because they are extremely difficult to reproduce. But with the data of an instrumented toolchain at your fingertips, this is now practical to pursue.

STORIES FROM THE TRENCHES: 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.

9.4. The collaboration between developers and the development-infrastructure teams

In most organizations, most performance regressions go unnoticed since they happen in smaller increments. But even if they are noticed they often go unreported. Why? Because reporting them in a way that becomes 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 contextual data available, often it will also take longer than necessary to address the regression.

In sum, the average build and test time is much higher than necessary and continuously increasing in most organizations because the toolchain is not instrumented.

STORIES FROM THE TRENCHES: EVEN THE EXPERTS STRUGGLE

We were working with a potential customer to measure the impact of our acceleration technology. It was a large team, so the business opportunity was significant. The company was suffering from performance issues and was excited to try out our DPE acceleration technology (build cache). But, after enabling the build cache their engineers reported that the experiment resulted in a build and test cycle time increase of 5X and they assured us they were running the before and after builds the exact same way. We were not convinced and had to convince their skeptical "experts" to run and compare the two builds using Build Scans (which is our build and test data collection and reporting service). An examination of the data revealed that the "before" Maven build was run with 128 threads while the "after" build using our technology was run with only one thread due to a misconfiguration of the directory hierarchy. The reason this happened was tricky and would have never been discovered without collecting comprehensive build data. 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. To connect to our product, a Maven extension must be applied which requires creating a `.mvn` directory in the project root directory and adding an `extensions.xml` file to it that defines what extensions should be applied. 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 128. The plain Maven project had no `.mvn` directory and thus picked up the configuration from the parent directory.

The lesson we learned together is that when it comes to toolchain support we rely way too much on what we think or hypothesize. We think we have assigned a certain amount of memory. We think we have assigned a certain number of processes. But, even the experts are often mistaken. Fact-based, data-driven software development organizations run their builds and tests faster, more efficiently, and avoid costly mistakes based on false recollections and assumptions. Tools are available that collect and organize the relevant data for you. Take advantage of them. 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!

9.5. Be proactive and less incident-driven

It is surprising to learn that most teams responsible for the toolchain are completely incident-driven. If someone complains they try to address 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 a lot of business along the way. Similarly, developers will produce fewer features and will look for different job opportunities if

they are kept working at their true potential because of a slow toolchain.

There is a flip side to this story, however. 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 made building and testing 10% faster. Who would notice? Who would praise you? The common answer I get to those two questions is usually 'nobody'. Even the build engineers themselves do not have the data to measure the impact of their work. To put this into perspective, for a 100-person developer team, this is typically a multi-million dollar annual savings. And yet this does not show up on anyone's radar. Companies do not have even the most basic tools to measure and improve developer productivity. As a result, it is unrealistic to expect that the build or CI team will proactively make improvements if the opportunities are hard to discover and their impact is hard to measure. To attract the best engineers to these teams, they need to be given the tools to make a difference and show the impact they are having on the organization.

9.6. See the big picture with performance analytics

Once detailed performance metrics are collected over time, informed executive decisions can be made on where to invest and how. Such planning requires measurement up front and at a steady cadence thereafter to adapt to an evolving situation.

Let's consider one strategy for measuring ABP and MABP using build data and assess the possibilities for planning to close the gap.

Performance metrics may be visualized as a trend line. Build performance varies quite a lot depending on the teams, the nature of code changes, and the build execution environments. It is wise to focus on one specific aspect of build performance at a time, such as "reduce the percentage of very slow builds" or "increase the percentage of very fast builds."

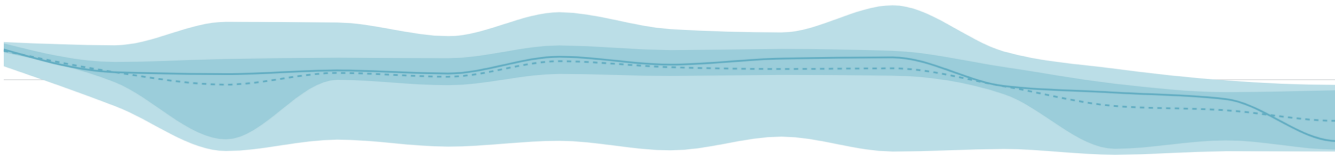
To close the gap between ABP and MABP, let's assume that the goal is to reduce the *median* build time to the 5th percentile build times. Said differently, the "typical" build time should be closer to the fastest build times. Note that the median is preferred over the average because the average is easily skewed by outliers.

With an understanding of the metrics we want to track to measure our success, the data is collected long enough to be able to visualize build performance over weeks, normalizing hourly and daily data which would skew our decision making.

There is now sufficient data to decide how much to invest. Using this same performance profiling data but broken up into smaller metrics, where and how to invest can be decided. For example, if slow build cache effectiveness is observed, the first step should be to identify causes of unnecessary cache misses. Or testing time is observed to be dominating overall build time, an investment may be directed at reducing it

These same performance trends can be used naturally to analyze the degree of success and plan high-level next steps.

Success in this scenario might look something like that of this developer team for a popular streaming service:



The darkest blue line represents median build time, the medium blue shaded area between the 25th and 75th percentiles, and the light blue area between the 5th and 95th percentiles. Clear improvement can be observed particularly in median build time and thus the gap between ABP and MABP.

9.7. Performance profiling with Build Scan™

Suppose the big picture has been seriously considered using performance analytics and trends data. That is, the metrics to optimize to reduce the ABP/MABP gap have been identified, and now the right build changes must be made to improve developer feedback time to meet your DPE goals. It is now time to use performance profiling to identify those changes.

A detailed Build Scan will show all inputs to a build, break down the performance impact of each unit of execution, and indicate where caches were hit or missed and why.

It is natural to start with the longest-running unit of execution and analyze the inputs and outputs to optimize cache effectiveness or raw performance. Such an effort will likely follow a graph-traversal-like process where one focuses on groups of related inputs and identifies improvements at some "vertices."

Here is an example of such a process. An integration test target has been identified which rarely benefits from build cache hits, causing the ABP of target builds to be much slower than MABP. Inputs to the integration tests are typical: build plugins, packaged production sources, test sources, library dependencies, installed toolchain binaries, and external infrastructure.

In this case we should compare detailed performance profiles from Build Scans for a cache hit scenario and a cache miss scenario from a homogeneous set of builds. Capturing a Build Scan for many builds to aid this process will prove to be very beneficial. Pay attention to the differences for each group of inputs. Should it be observed that the packaged production sources also suffer build cache misses simultaneously, the graph will be traversed and the upstream inputs will be inspected, and it can be seen that the toolchain binaries which package the production sources differ when build cache misses occur. We now understand the importance of standardizing the toolchain, and are able to track the effectiveness of our solution at a high-level.

You might see a difference in the installed toolchain this way:

Comparing **12 infrastructure items**, with **9 differences**

Operating system	Linux 5.14.12-zen1-1-zen	Mac OS X 11.5
CPU cores	12 cores	16 cores
Max Gradle workers	12 workers	4 workers
Java runtime	Eclipse Foundation OpenJDK Runtime Environment 11.0.12+7	AdoptOpenJDK OpenJDK Runtime Environment 11.0.11+9
Java VM	Eclipse Foundation OpenJDK 64-Bit Server VM 11.0.12+7 (mixed mode)	AdoptOpenJDK OpenJDK 64-Bit Server VM 11.0.11+9 (mixed mode)

Observe that the Java VMs have different vendors, which is a common cause of build cache misses.

9.8. ABP vs MABP revisited

Our industry should be very excited about the new acceleration technologies that are available. But no one should be under the illusion that this is “fire and forget” technology. If an organization provides a build cache, possibly even to a sizable and distributed team, and that organization thinks it will deliver benefits without being managed, they will be sorely disappointed. We have seen this at play multiple times.

We have frequently seen infrastructure teams that enabled build caching for their development teams, but made no investment in the tools to hone and tune the cache or actively manage its performance over time. After only a few months, developers were disappointed that the ABP was still low, the infrastructure team was frustrated that their effort was viewed by developers as a waste of time, and management started to wonder about the competency of their decision makers. Throwing unmanaged acceleration technologies at a performance problem without following any proper DPE practices and without having comprehensive data insights will not only fail to improve performance, but it could result in negative savings.

While a functioning build-cache is one of the cheapest and yet best-tasting lunches you will ever be able to buy for your team, it is not free. The same is true for Test Distribution and anything else related to the performance of your builds and tests. The good news though is that the potential savings you can get by combining acceleration technologies with a DPE practice is so enormous, that the economic argument to establish a DPE practice is overwhelmingly obvious. More on that in Part 4 - ECONOMICS.

Part 3 - TROUBLESHOOTING FAILURES AND BUILD RELIABILITY

Failure Types and Origins

Efficient Failure Troubleshooting

The Importance of Toolchain Reliability

Best Practices for Improving Build Reliability with Failure Analytics

Improving Test Reliability with Flaky Test Management

10. Failure Types and Origins

By Hans Dockter

When a build fails developers are usually stuck. Local build changes can't be pushed and pull requests can't be merged. Of course, work can be started on a different feature in a different branch. But the code that failed might be required by another team, by QA, or by your customers. And if the failure is in a team or integration branch, the whole team is blocked. Most failure incidents have a high time sensitivity to repair compared to performance problems since performance issues usually do not block developers from continuing their work.

10.1. Common build failure root causes

Build automation exists primarily to avoid the drudgery of verifying whether or not code works as intended. In a more ideal world, build automation works with perfect accuracy and indicates a failure only when a real problem is detected in the code. We call such legitimate build failures *verification failures*.

In reality, not all build failures are verification failures. Builds are software too and are often quite complex. As a result, all of the potential problems that arise when developing any software apply to build software. Failures associated with build software are called *non-verification failures* and include:

- Faulty build logic
- Unavailable artifact repositories
- Flawed build environments
- Faulty external services

Ambiguous Language

We say a "build is broken" or ask "who broke the build" or simply say that "the build failed." One can argue that if the build detects a problem with the code, it was successful and is definitely not broken. But if a bug in the build logic or in the CI configuration is preventing the toolchain from validating changes or assembling artifacts, then the build is certainly broken and it has failed to do its job. This language is ambiguous and uses the same terms for a completely different set of problems and responsibilities which adds to the confusion and often not well defined ownership of build failures.

10.2. Classifying failure types and determining ownership

Distinguishing failure types can be tricky. There are non-failures that look like verification failures, such as runtime errors from different versions of the same dependency on the classpath. There are verification failures that look like non-verification failures to a developer. For example, when the

build uses snapshot dependencies and the CI build later picks up a different version than the local build version, there is no straightforward way for the developer to discern this.

In most organizations, when a build fails, the developer who initiated it deals with the consequences. There are valid and invalid reasons for this. There are many verification failures for which the reason is obvious and easy to fix for the person who encountered it. But there are also many failures where a developer is not only unclear about the root-cause, but also the failure type, and who is the responsible party for making the fix.

10.3. The role of DPE in addressing build failures

For build failures, the objective of DPE is to:

1. Provide the tools and processes to improve the mean-time-to-repair for incidents and enable the appropriate resources to address them, and;
2. Minimize the occurrence of non-verification errors.

We will look at those two areas in detail in the next chapters.

11. Efficient Failure Troubleshooting

By Eric Wendelin

Troubleshooting broken builds is slow and frustrating for developers, build and CI teams, with hours spent on debugging, support, and reproducing problems. DPE practices empower teams by giving them comprehensive data about every build, so they can efficiently collaborate on build issues and get back to delighting users. In this chapter you will learn why data contextualization is the key to efficient failure troubleshooting. You will also learn about DPE tools that make troubleshooting more efficient and the role that historical data and build comparison capabilities play in making these tools effective at minimizing the productivity impact of unexpected build failures.

11.1. Data contextualization is the key to easier and more efficient troubleshooting

The truth is that most software assembly disruptions are easy to solve by those with the context to solve them. Therein lies the trouble. Engineers unfamiliar with the build and test infrastructure lack the tools to effectively solve or communicate problems.

As a result, there is a critical need for a comprehensive and organized record of what happened during a build. Ideally, this record should effectively convey every aspect of a software assembly process such that no one is left guessing the answers to questions such as which assembly tools participated in the build, which library dependencies were required, what repository was used, and how long did the build take.

It is also important to provide developers with insights to help them clarify, for example, if the root cause of a problem can be traced to an infrastructure problem that the build or CI team should deal with or a problem with the code that they are responsible for fixing. This is where streamlining communication with easily accessible and intuitively presented data is immensely useful. A troubled software developer no longer has to accurately recall and reproduce what happened when there is already a thorough audit record of what actually happened. As a result, the infrastructure team can help much more efficiently.

Key aspects of software assembly that are critical for troubleshooting:

Build environment	• Hardware • Operating system
Toolchain	• Infrastructure services that execute assembly binaries

Build inputs	• Software assembly/verification tools • Projects and libraries • Source code state • Build configuration
Build outputs	• Problems detected by verification tools • Debugging information • Component-level build time data • Best practices followed

To be effective, such a record must also be easily navigable and deeply-linkable. This gives those with knowledge about the problem area (the build infrastructure team or the team owning a certain dependency, for example) every chance at quickly solving disruptive issues so that everyone can get back to providing customer value.

STORIES FROM THE TRENCHES: ASTONISHING WASTE

In contrast to performance issues, failures are usually noticed by at least one individual, the developer who is running the build and tests locally or the developer whose changes are triggering a CI job that fails. Usually, the next step for the developer is to decide whether the failure is due to a verification error or non-verification error. This is often not easy and without the right tooling can be impossible to determine. Once a developer determines that the problem is caused by a non-verification error, they should report it to the team responsible for maintaining the toolchain.

But if a developer needs help making that determination, asking for help is often painful for both the reporter and the helper for the same reasons it is painful to collaborate on performance regressions as described in Section 9.4. Since reporting failures often blocks progress they cannot simply be ignored. To avoid reporting failures, a common practice is for developers to run the complete build again and hope that any problems are due to flakiness. If problems continue, complete builds may be run yet again with all caches deleted. This can be extremely time-consuming and frustrating. Only after these steps have been taken without success is help requested informally or an issue is filed. But the build engineers and CI experts responsible for responding to these requests for help often have difficulty determining the root cause of problems because they don't have the right process or tooling either.

Most managers care deeply about the job satisfaction of their developers. But even those who are less concerned about that 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 reliability issues while at the same time human and machine resources are wasted.

11.2. Implementation example: Leveraging Build Scan™

A good example of a *de facto* standard DPE tool that improves toolchain reliability by making troubleshooting more efficient is called Build Scan™. Build Scan does this by providing actionable insights for every local and CI build in one distilled and organized report. It is like an X-ray for your build that provides the task-level data and graphical views development and build engineering teams need to perform root cause analysis for build and test failures. It can be used by individual developers or in collaboration with build teams to identify the cause of many of the most common failures like dependency version conflicts. It can also be used in combination with build and test failure analytics tools (Chapter 13) to solve more complex problems and in combination with build cache (Chapter 7) as a data analytics tool to hone and optimize your build cache configuration or other performance related issues.

11.3. A spotlight on toolchain failures

Many teams consider it normal and unavoidable when developers are sporadically delayed by changes to the tools that assemble their software. After all, tools must be updated with reasonable frequency to apply security patches or give users access to new features.

Build failures caused by these changes to the underlying tools are common. In fact, data from large open-source projects shows that it's typical for 2-5% of builds to fail due to non-developer-related reasons. Many times these failures are surprising or confusing with each disruption spreading to close team members until the team is fully inoculated.

Developers first ask did I cause this or do I need to fix this now or is the broader team affected by this? A Build Scan should quickly answer these questions by indicating related failures, if any, and highlighting the parts of the toolchain which are most likely responsible. Once this is known, communication and investigation become more streamlined. If there are no other related failures, the developer can immediately investigate the relevant portion of their environment and solve the problem without escalating it.

11.4. The role of historical test data

Analysis of large open-source projects shows that it is common for 3-10% of test builds to fail due to test infrastructure problems. In practice this metric varies greatly from teams who only write unit tests to those who heavily rely on full integration test coverage.

Again, developers that encounter unexpected test failures must ask did I cause this, do I need to fix this now or is this a flaky test failure? Simply knowing whether this is the only recent failure of this test makes it easy for the developer to decide how to proceed. Most CI systems today only surface the test failure in isolation, forcing developers to work much harder to unblock themselves from these common problems.

Build Scan must fill another critical gap in this test failure analysis. And that is to capture and analyze local test failures. This allows developers a complete picture of test behavior which will be useful when debugging. This also allows a Build Scan comparison between a failing local build and

a passing CI build to narrow the problem space.

11.5. Comparing builds to facilitate debugging

When debugging a build failure, identifying what changed since the last success is a critical need. Build problems can be traced to either build inputs such as the source code and dependencies, or to build environment factors such as file system and network problems. Many factors must be considered, some of which are difficult or impossible to fully reconstruct after a build occurs. This is where an immutable record of all relevant builds can be invaluable.

A Build Scan of a failed build can be compared to another Build Scan of a successful build to identify the differences between them quickly. Eliminating potential root cause dimensions this way, especially common ones like dependency conflicts, is at the heart of efficient debugging and key to reducing the mean-time to resolve failures.

11.6. Summary

Capturing and presenting comprehensive build and test data—like toolchain, build input, and build output data—allows developers and DPE practitioners to streamline failure troubleshooting. This data is most powerful when collected and combined from multiple builds because it gives developers and build experts a complete historical picture which streamlines debugging and ultimately is helpful in reducing the severity of disruptions from unexpected build failures.

12. The Importance of Toolchain Reliability

By Hans Dockter

12.1. What is toolchain reliability?

A toolchain's reliability can be judged by the rate at which it causes non-verification failures. These failures come in two forms:

1. The toolchain is asked to do something and fails without producing any result
2. The toolchain produces the result it is asked to, but the result is not correct (e.g., a flaky test or an archive that is not assembled correctly)

The first type of non-verification failure is usually easier to identify than the second. Both cases cause downtime, waste compute resources and are a massive distraction depending on the frequency in which they occur. They also negatively affect the quality of the code that is shipped as there are less iterations to find problems in the code.

Builds become unreliable when problems are too expensive to find, too hard to reproduce for root cause analysis, and when fixes cannot be correctly prioritized because their relative impact is unknown.

To further sharpen your understanding of what could be perceived as a nebulous topic, this chapter highlights the importance, connections, and differences between several important reliability-related concepts. This includes:

- The connection between performance and reliability
- The connection between reliability issues and morale
- The importance of issue prevention; and
- The difference between reproducible and reliable builds

The more unreliable the build and test pipeline, the less a developer trusts whether a verification failure is actually a verification failure. When developers are unsure how to get their build out of a bad state, they 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, they may try clearing caches and eventually burn down the whole workspace and clone it anew. This wastes a lot of valuable time and generates a great deal of frustration. Asking for help often requires knowing who to ask and how to pin down the problem, which isn't always trivial in large organizations and with large codebases.

12.2. The connection between performance and reliability

Performance and reliability are closely related. As discussed in Chapter 6, 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 * number of context switches
```

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 therefore substantially affect the average time to achieve a successful build.

An unreliable toolchain, like one with a lot of flaky tests, can dramatically increase the average feedback cycle time as well as the context switching frequency.

STORIES FROM THE TRENCHES: A FLAKY TEST NIGHTMARE

In 2016, just before Gradle 3.0 became available, flaky tests became an emergency in the Gradle Build Tool. It took up to two days to get a passing build on the main branch because flaky integration tests were not addressed. This caused serious delays in delivering new releases, as well as quite a bit of developer frustration.

An *ad hoc* team had to get the situation under control and we had no idea how bad the situation was. We started by collecting all test failure data in a system and provided reports to the engineering team on the most frequent causes of test failures and the most frequently failing tests. Through this process we learned that less than 3% of all changes ever got through the CI pipeline without flaky failures.

We instituted two "flaky fix-it" days where the entire Gradle team worked on the worst causes of flakiness. After just one week 30% of all changes got through the CI pipeline. After more work and another month 70% got through the CI pipeline. Today the number is above 99%. And, now a "flaky fix-it day" is held before every major release.

Visibility was *the* key to unlocking a change to our testing culture. As soon as we had data, not only was it easier to justify investing time, but fixing issues for which we had many data points became significantly easier.

12.3. The connection between reliability issues and morale

The negative effects of unreliable builds go well beyond the frustrations related in the story above. Imagine you are repairing something under a sink. You have to squeeze yourself into a very tight cupboard and are lying there in an uncomfortable position. It takes 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 you are a plumber and this happens to you multiple times a day, under the pressure of completing multiple jobs every working day. That might give you an idea of how many developers feel today. Working effectively on large codebases is a very challenging task. To do a context switch from one challenging problem domain into another to reason about a failure is like getting under this sink with unreliable tools. It is extremely frustrating to be forced to use flawed tools while doing challenging work.

12.4. The importance of issue prevention

You will always have issues that require in-depth root-cause-analysis by the developers or direct support from the infrastructure or build teams. Optimizing that workflow is a huge part of a great developer experience and was discussed in Chapter 11.

But arguably the most efficient thing you can do is to prevent non-verification incidents as much as possible. All the points we mentioned in Section 9.2 apply similarly to reliability. Achieving high build reliability needs to be a continuous practice embedded in the engineering culture paired with the right tooling. You need trends and comprehensive data 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 additional incidents
- Prioritize your work based on data and quantifiable impact
- Continuously make your toolchain and more reliable

We will talk in detail in Chapter 13 how this can be achieved.

12.5. The difference between reproducible and reliable builds

Often reliable builds and reproducible builds are used interchangeably. But they mean different things. 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.

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.

STORIES FROM THE TRENCHES: A TIME BOMB WAITING TO EXPLODE

We had a customer with a build that needed a Tomcat library on the classpath of their application. For legacy reasons, the build was configured to pick up the Tomcat library from the Tomcat server that was installed on local machines. Developers 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 was a time bomb waiting to explode! When it did, it would be very hard to debug issues during development or in 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 expected. Since they also started to gather comprehensive data about every build execution, they could now easily detect the problem. As a side note, less effective build caching is often the result of reproducibility issues. As a result, an added benefit of build caching is that it surfaces reproducibility issues and makes builds more reliable.

13. Best Practices for Improving Build Reliability with Failure Analytics

By Sam Snyder and Eric Wendelin

Build reliability can be improved with failure analytics by providing a way to view and revisit problems from a high level, which ensures that the toolchain can be adapted to more effectively meet evolving business needs. To achieve this benefit, it is necessary to maximize the frequency developers can iterate by reducing the amount of *unnecessary* build and test failures as much as possible.

This chapter covers three fundamental best practices that are critical to improving build reliability with Failure Analytics.

13.1. Avoid complaint-driven development

The best developers want to be productive and they 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 and make a concerted and proactive effort to provide a great developer experience. If you wait until the complaints start coming in, your best talent may already be gone.

STORIES FROM THE TRENCHES: SUFFERING IN SILENCE

I became frustrated with poor build reliability at my previous company and set out to do something about it. 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 very loudly demanded help when he was blocked. He would go from chat room to chat room trying to find the relevant expert until he found the answer. 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 the squeakiest wheels, 30 other developers were suffering in silence.

I went to one of those developers and asked her why she didn't raise her problems with me. She shrugged and said, "Builds are slow and unreliable, that's just how they are." The wasted time and lost productivity from toolchain unreliability had become so normal and accepted it wasn't worth making any noise about. 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.

13.2. Use data to systematically improve reliability

Time is limited and improvements must be prioritized by weighing the benefits to the organization.

The first step is to collect information from a representative sample of failures and group them by root cause to the extent possible.

Here are guidelines that can be used to prioritize build problems listed from highest priority to lowest

1. Frequent new non-verification failures
2. Long-standing and reasonably frequent non-verification failures which are actionable
3. Frequent verification failures
4. Other infrequent failure types that can be fixed quickly

Non-verification failures are considered severely disruptive because they are by definition less actionable for developers not fluent in the build logic and build infrastructure.

Frequent compilation, static analysis, or test failures should not be overlooked. It is possible that developers lack the tooling to avoid these types of failures. Introducing changes which assist developers in sidestepping avoidable failures is the best course of action. For example, if many "lint" failures are observed, introducing auto-formatting in IDE settings or via a git hook may substantially reduce these types of verification failures.

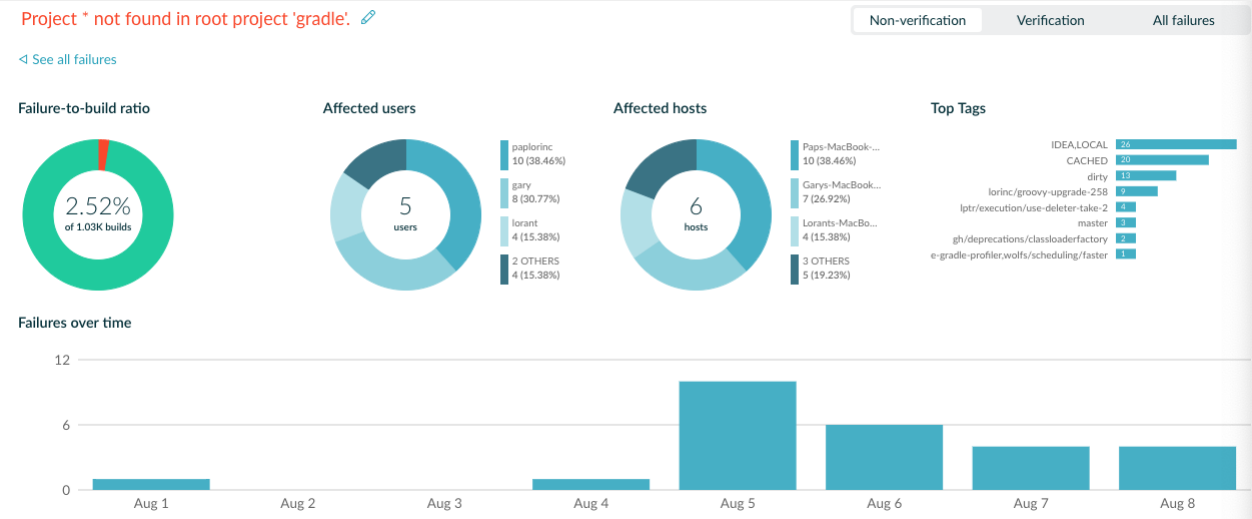
13.3. Continuously measure and optimize

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 organizations be able to attain and maintain a great developer experience at scale. If productivity metrics and outcomes are not measured, don’t be surprised when some of the best developers leave for greener, better-tooled pastures because they are not convinced about the impact of DPE.

Of course, build reliability is likely not the only pain point in your development toolchain. Flaky tests are another universal source of developer pain.

STORIES FROM THE TRENCHES: THE MYSTERIOUS MISSING PROJECT

Here's an example of how we used a failure analytics dashboard to identify and fix a nasty problem in our own build. The screenshot below shows the "Top Failures" dashboard view 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 the root project 'gradle' was a non-verification failure that we hadn't known about since no one complained despite the fact that 1 in 40 local builds a week failed with that error.

From this "Failure Analysis" view we can see that it started happening in early August. 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 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 and effective prioritization and remediation of an issue we previously knew nothing about was natural and easy.

14. Improving Test Reliability with Flaky Test Management

By Eric Wendelin

14.1. What is a flaky test?

A test is "flaky" when it produces both "passing" and "failing" results for the same code. An unreliable test suite with flaky tests wastes developers' time by triggering unnecessary test failure investigations that are not the result of their code changes and delaying the integration of their code.

The amount of waste varies by team and project, but it is often significant. Google reports a continual rate of about 1.5% of all test runs reporting a flaky result and that almost 16% of tests have some level of flakiness, while Facebook reports that "all real-world tests are flaky to some extent."

Developers must have confidence in the reliability of the test suite and the test infrastructure. Without that confidence they may become discouraged from running tests early and often, thereby creating quality and productivity issues later in the software development and delivery lifecycle.

There is a wide variety of causes for flaky tests, most commonly:

- Asynchronous code: deadlocks, contention, and timeouts
- Order-dependent tests: tests executed in a different ordering produces different outcomes
- Resource leaks: memory leaks, file handles, and database changes
- External infrastructure: unreliable services, container management, or even flawed hardware
- Time-based computation: system clocks jump ahead or backward (really)

Often tests which report flaky results are not themselves unreliable, but caused by flawed production code or test infrastructure. This makes flaky tests very time-consuming to find and fix because they are by-definition difficult to reproduce and often interact with outside elements, which stymies root cause identification.

This is the scenario where having data on flaky test executions is priceless since it allows developers to use historical test analysis to observe failure patterns which, in turn, leads to quicker root cause identification.

14.2. Flaky test identification strategies

Across the industry nearly every flaky test identification method involves retrying the test in a variety of ways:

- Re-run flaky tests in a different test order to identify order-dependent tests
- Re-run flaky tests in an isolated manner to identify resource leaks or asynchronous problems

- Re-run flaky tests in a different environment to identify infrastructure problems

Simply retrying a test immediately after failure in the same environment is the most common strategy which will identify a significant proportion of flaky tests. Most test runners or build tools have a mechanism for rerunning failed tests a pre-specified number of times.

Elite teams, such as those at Netflix and Mozilla, proactively identify flaky tests when they are created and changed using a combination of the above retry strategies aggressively for new and modified tests. This practice allows developers to eliminate flakiness at development time and helps to avoid copying flaky code, compounding the benefits.

14.3. Measuring the impact of flaky tests

A majority of flaky test executions are typically caused by a small fraction of the test suite. Fixing the flakiest test reduces pain substantially, but once flakiness is down to acceptable levels it becomes difficult to justify the opportunity cost of fixing flakiness over new features. An ongoing process is required to measure the severity of problems and use that information when prioritizing investments.

How much test flakiness is acceptable? The amount of flakiness an organization will tolerate depends on the culture, but all organizations should strive for fewer than 1% of developer-oriented test pipeline executions to fail due to non-verification failures. In other words, a CI pipeline that developers run before they merge their code should fail due to flaky infrastructure or flaky code in less than 1 in 100 runs.

A 1% failure frequency is sufficiently rare that build failures due to flakiness will never become the norm and the testing culture of a team can be preserved. Some teams may require higher standards with increased investment.

The likelihood of a failing test run due to flakiness can be calculated as the product of the number of tests, the rate at which they flake, and the number of build runs. The probability of a flake-less test run may be computed by multiplying the probability that each succeeds:

`P_pass = Π(1 - P_flaky)` for every flaky test in the suite. The probability of a flaky build failure is `1 - P_pass`; then the number of flaky failures per day can be estimated as:

```
Flaky Failures/day = (1 - P_pass)(Builds/day)
```

Let's say your organization has 200 flaky tests that run in 500 builds per day and the goal is to have no more than 1% of builds (5 in this example) disrupted by a flaky test failure. With no retries, the average rate at which each test flakes must be kept under 0.005%. With a single retry, a 1% flaky failure rate can be maintained with an average test flakiness rate of under 0.7%.

14.4. Flaky test mitigation strategies

Immature teams settle for marking a test as passing if it succeeds when retried. This allows flakiness to remain and grow, inevitably jeopardizing product teams and customers. Additionally, growing flakiness will eventually cause tests to fail even when retried, thus relegating developer teams to situations where they are again blocked by flaky test failures. It is critical that flakiness is

addressed.

Deleting flaky tests is often objectionable to developer teams that want to maintain a high-quality codebase by keeping test coverage high.

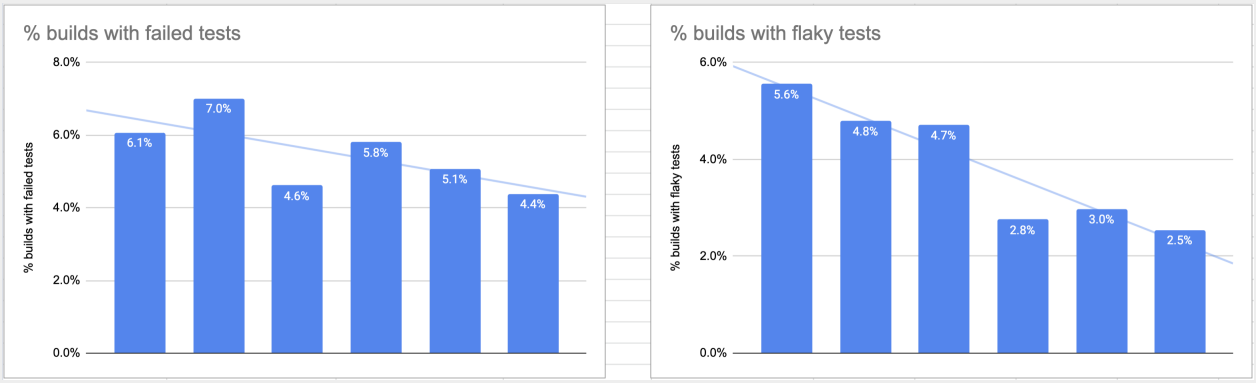
Flaky test quarantine is becoming a popular strategy for managing flaky tests. This involves establishing a primary validation process and a secondary (quarantine) process where flaky tests are run frequently but do not block code integration. Developers are then incentivized to stabilize tests to qualify them to be included as part of the primary validation pipeline.

Another effective DPE practice to measurably reduce flaky test failures is to establish a regular (monthly) Flaky Fix-IT Day. This is a day dedicated to fixing flakiness involving most or all of the engineering team. A full day is needed to provide enough time to identify elusive flakiness causes and test fixes.

The high cost of these days produces tangible benefits by reducing the number of blocked code changes and increasing the robustness of the application and test code. This practice also has intangible benefits such as helping engineers get more skilled at identifying flaky test patterns and root causes.

Visualizing the effectiveness of Flaky Fix-it Days at Gradle

To quantify the impact of Flaky Fix-it Days we track how builds with failed and flaky tests trend over time. After 2 engineering Fix-it Days, the team was able to cut the frequency of builds with flaky tests by more than half (from 5.6% to 2.5%) over a 2 month period. This also reduced the total frequency of build failures from 4.0% to 3.1%.



During these 2 days, 35 of the top 50 flaky tests were fixed and 3 bugs in production code were found and fixed.

Once flaky tests are prioritized for remediation, historical flaky test data will prove to be immensely useful when investigating the root cause.

14.5. Leveraging data to address flaky tests

DPE tools make the following data available to developers to help quickly identify the root cause of flakiness for a test or group of tests:

- Execution timestamp, execution duration, OS, and build environment data for each flaky failure
- Logs and exception details for flaky test executions
- The set of tests that are flaky or fail in the same test run
- The set of tests executed at the same time as the flaky tests

Each of these data sources allow developers to quickly rule out or isolate the type of flakiness that is occurring (e.g. asynchronous code or order-dependent tests). If possible, this data should be stored in a single place so that developers do not have to waste time aggregating data from many sources.

Once the general type of flakiness is identified, DPE-minded organizations should enable developers to tightly control test execution by allowing them to specify which tests are executed in a given test run, in what order, and in which specific environment. Such tools will allow developers to confirm fixes more quickly and confidently. Historical test data can be used to confirm that flakiness has been addressed by allowing developers to observe that no flaky test executions have occurred since a fix was implemented.

14.6. Summary

Flaky tests trouble developer teams of all sizes in all industries. DPE teams are able to quantify the impact of flaky tests using historical test data, and manage them effectively by proactively identifying, prioritizing, and fixing them. This practice reduces the risk of flaky production bugs and increases application and developer resilience to dubious code practices, which makes customers and developers happier.

Part 4 - ECONOMICS

Quantifying the Cost of Builds

Investing in Your Build: The ROI calculator

15. Quantifying the Cost of Builds

By Hans Dockter

This section provides a model for calculating the costs of your builds and the return you get from improving them.

Developers and build engineers are under constant pressure to ship faster and more frequently. This requires fast feedback cycles. At the same time, cloud, microservices, and mobile have made our software stacks more complex. Without taking any actions, builds will become slower and build failures will become harder to debug as your codebase grows. The strategic implications are that inefficient builds do not only affect your ability to ship fast, they also waste a lot of your R&D bandwidth.

Improving this is a huge organizational challenge for build engineers and development teams. To trigger organizational change it is important to have quantifiable data to support your arguments and this chapter provides a tool for doing so.

Fortunately, quantifying the cost of builds is straight-forward and the calculations are easy to understand. Some of our customers have more than 500,000 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.

15.1. 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 provided in this chapter reflect what we see typically in the wild. But the numbers can vary a lot. For some example scenarios, we have better data averages than for others. At any rate, the example numbers are helpful to get a feeling for the potential magnitude of the hidden costs of builds. Your numbers might be similar or very different. You should collect and use

your own data to assess your situation and understand your priorities. The primary purpose of this chapter is to provide a model with which you can quantify costs and apply 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 code that is built from a single repository, which then results in longer build times. But as a rule of thumb, successful software teams have many builds per day and aim to increase that number. To better reflect this, as we evolve our model we plan to convert to a lines-of-code build-per-day metric.

15.2. 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%

15.3. Local builds

When developers execute local builds, waiting for the build to finish is largely idle time. Anything shorter than 10 minutes does not allow for meaningful context switching. That is why we assume W_L as 80%. It could even be more than 100%. Let's say people 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:

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

Conversely, every saved minute is worth $400,000. Every saved second is worth $6667. A 17 seconds faster local build is the equivalent of one additional engineer per year. A 5-minute improvement in build speed, which is often possible for teams of that size, is equivalent to the cost of 18 engineers or 9% of your engineering team or $2,000,000 in R&D costs.

The reality for most organizations is that builds take longer and longer as codebases grow, 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 their output. Furthermore, talent is hard to come by and anything done to make existing talent more productive is invaluable. If local builds are so expensive, it is not unreasonable to ask why they are done at all. Actually, some organizations have come to the conclusion that it is not worth running local builds. But without the quality gate of a local build (including pull request builds), the quality of the merged commits substantially deteriorates, leading to a debugging and stability nightmare on CI and many other problems for teams that consume output from upstream teams. The effect of kicking the can down the road is even higher costs.

Our experience is that the most successful developer teams build very often, both locally and on CI. The faster builds are, the more often one builds and gets feedback, thus the more often you can release.

STORIES FROM THE TRENCHES: 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 (e.g. "We do 1,300,128 local builds per year and the virus scanner introduces an additional 39 seconds of waiting time per execution, which is the equivalent of the cost of 8 wasted engineering years") that discussion will most likely have a completely different dynamic.

15.4. 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 factor in the build cost, so it should 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:

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

Long CI feedback is very costly beyond the waiting cost:

- Context switching from fixing problems on CI will be more expensive
- The number of merge conflicts from 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 these 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.

15.5. 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)

15.6. The cost of debugging build failures

One of the biggest time sinks for developers is to determining the root cause of failures. When we say the build failed, we discussed in Chapter 10 that it can mean two different things. Something might be wrong with the build itself (e.g., an out of memory exception when running the build). We will cover this in Section 15.7. In this section, we talk about the second interpretation which is that the build failed due to a problem with the code (e.g., a compile, test or code quality failure). The data used below is an estimate based on what we typically see for teams 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	What percent of the failed local builds require an investigation	5%
I_{CI}	What percent of the failed CI builds require an investigation	20%
T_L	Average investigation time for failed local builds	20 minutes
T_{CI}	Average investigation time for failed CI builds	60 minutes

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, low test coverage and low development activity are likely causes.

For many failed builds the root cause is obvious and does not require an investigation, but there are many situations where investigations are needed. These are expressed by I_L and I_{CI}. CI builds usually include changes from multiple sources. They are harder to debug and multiple resource may 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.

Many underestimate their actual failure rate. At the same time, there is quite a bit of variation in those numbers in the wild. Some teams have very long-running builds and because builds are frequently slow, developers don't run them as often and there are also less CI builds. Fewer builds means 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. Since any one of many changes could be responsible for the failure, the investigation is more complex and the average investigation times go up. We have seen many companies with average investigation times for CI failures of a day or more. This debugging is expensive but the costs of such long-lived CI failures go beyond that. It kills your capability to ship software regularly and in a timely manner.

The basic rule is that the investigation time grows exponentially with the time it takes for the failure to show up.

If developers don't do a pre-commit build, it will push up the failure rate and investigation time on CI. Everything is connected. With 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

15.7. Faulty build logic

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

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	What percent of the failed local builds require an investigation	100%
I_{CI}	What percent of the failed CI builds require an investigation	100%
T_L	Average investigation time for failed local builds	240 minutes
T_{CI}	Average investigation time for failed CI builds	90 minutes

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 (for Gradle builds). If not properly managed they often introduce some instability. Such problems usually require an investigation which is why the investigation rate is set at 100%. Such problems are hard to debug, even more so for local builds since most organizations don't have any durable records for local build execution. As a result, build

engineers need to work together with a developer to reproduce and debug the problem. For CI builds there is at least some primitive form of durable record that might provide an idea of what happened, like the console output. We have seen organizations with much higher rates for F_L and F_{CI} than 0.2% and 0.1%. But since this is currently very hard to measure we don't have good average data, so the numbers we assume for the example team are conservative.

Cost

Debugging local build failures:

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

Debugging CI build failures:

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

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, 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

15.8. 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 per year.

Potential investments to reduce CI infrastructure cost

- Reuse build output across machines (build cache)
- Collect build metrics to optimize performance

15.9. 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 minutes
A_{CI}	Average build time for CI builds	8 minutes

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 is the equivalent to the cost of 15 engineers for our example team of 200. And keep in mind that if nothing is done about it, the cost will increase year by year as your codebases and the complexity of your software stacks grow.

There are a lot of other costs that are not quantified in the scenarios above. This includes, 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.

15.10. 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 that “build performance and efficiency is not a problem.” That often means that developers simply do not complain about build performance. For those teams, unless the developers are screaming, nothing is a priority. While developer input is very important for build engineering, anecdotal input from developers should not be the sole driver for prioritizing investments in developer tooling. You might leave millions of dollars of lost R&D on the table. Build engineering should operate more professionally and be more data-driven.

Benefit is understated

For other teams, there is a lot of pain awareness (e.g., around long build times) but the impact of incremental steps is still often underestimated since they are not addressing the pain completely. 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.

15.11. Conclusions

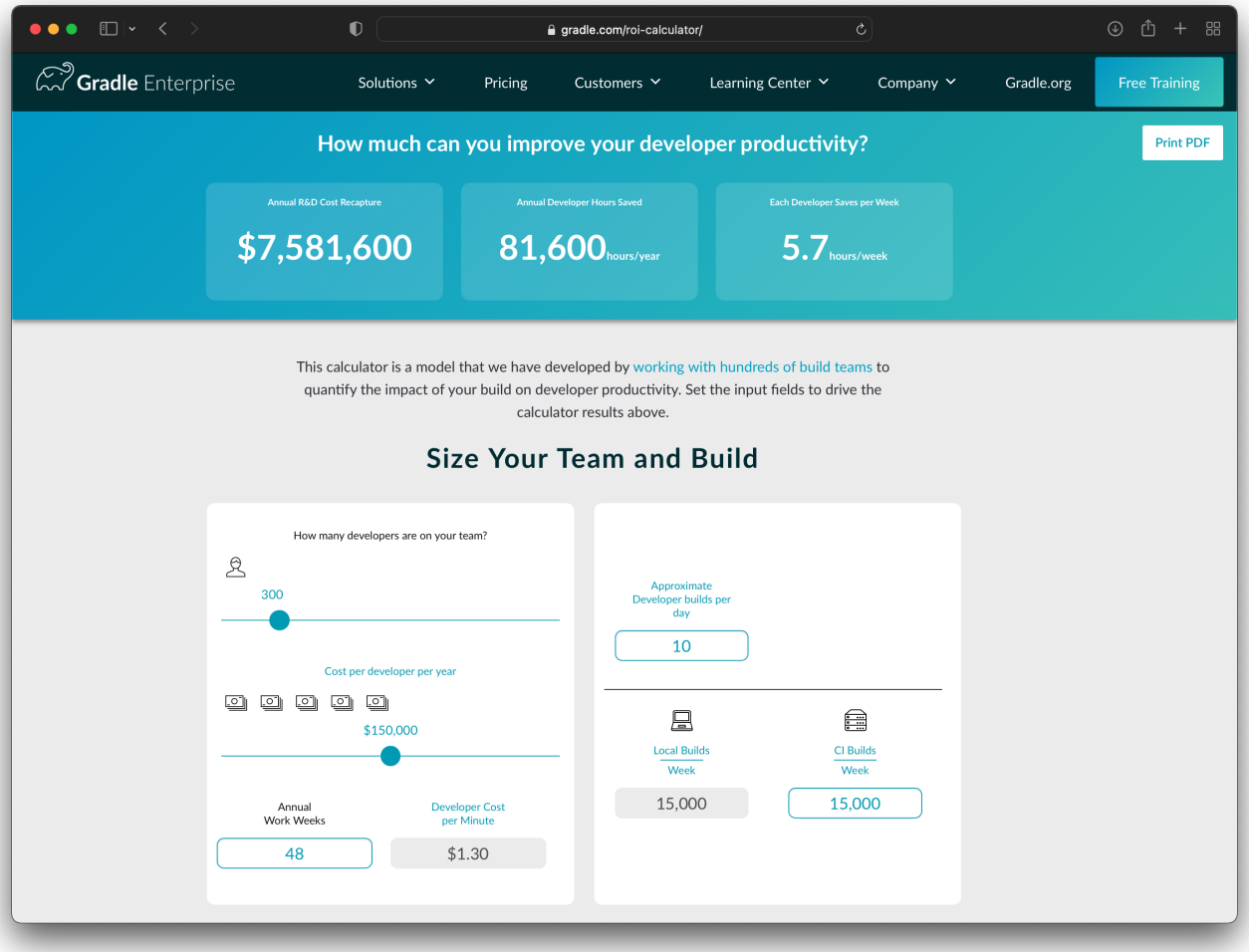
We haven’t yet come across a company where investing in more efficient builds did not lead to a significant return on investment. The most successful software teams on the planet are the ones

with an efficient build infrastructure.

16. Investing in Your Build: The ROI calculator

By Hans Dockter

The previous chapter described 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.)



16.1. 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:

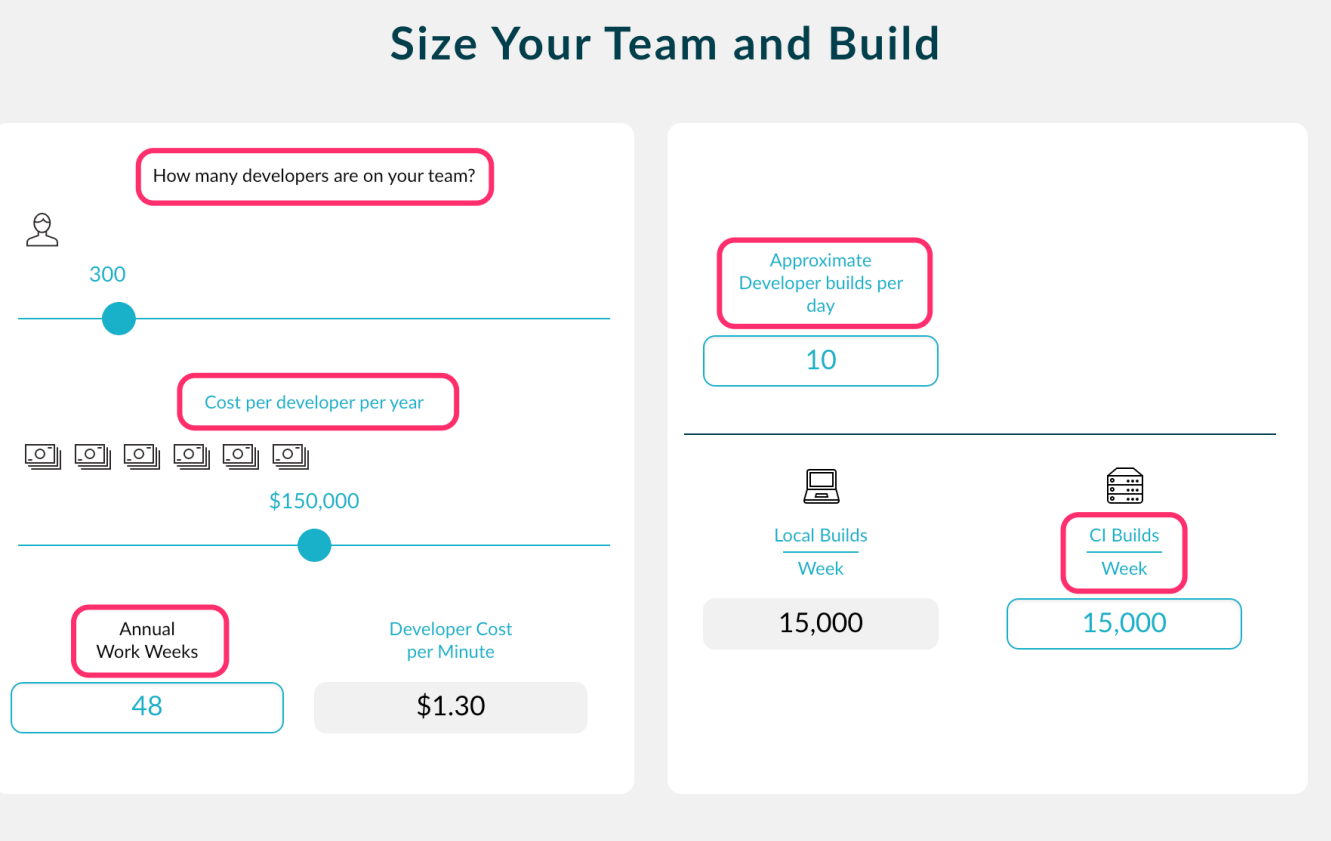
- Speeding up slow builds
- Preventing build regressions over time
- Faster build debugging

The savings figures for each of these drivers are summarized and depicted in the top banner. You input the values for each parameter referenced below to derive an ROI estimate for your team.

Step 1: Size your team and build

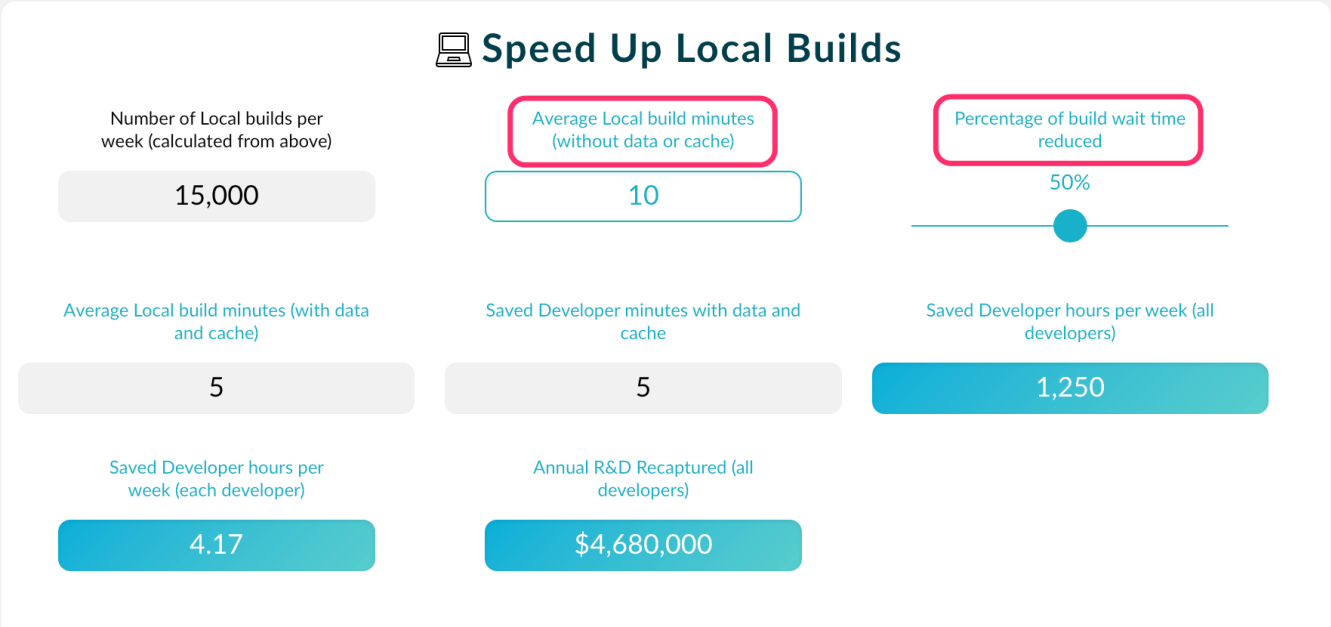
Start with the top section of the calculator entitled "Size your Team and Build." On the left side, use the sliders to estimate the number of developers running local and CI builds on your team and the total annual compensation per developer (including benefits and other employee costs). On the right side, update the blue input boxes for the number of weekly developer and CI builds.

The summary calculations shown at the top automatically update to reflect your changing inputs. 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 build cache.



Repeat the process in the next section called "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%.

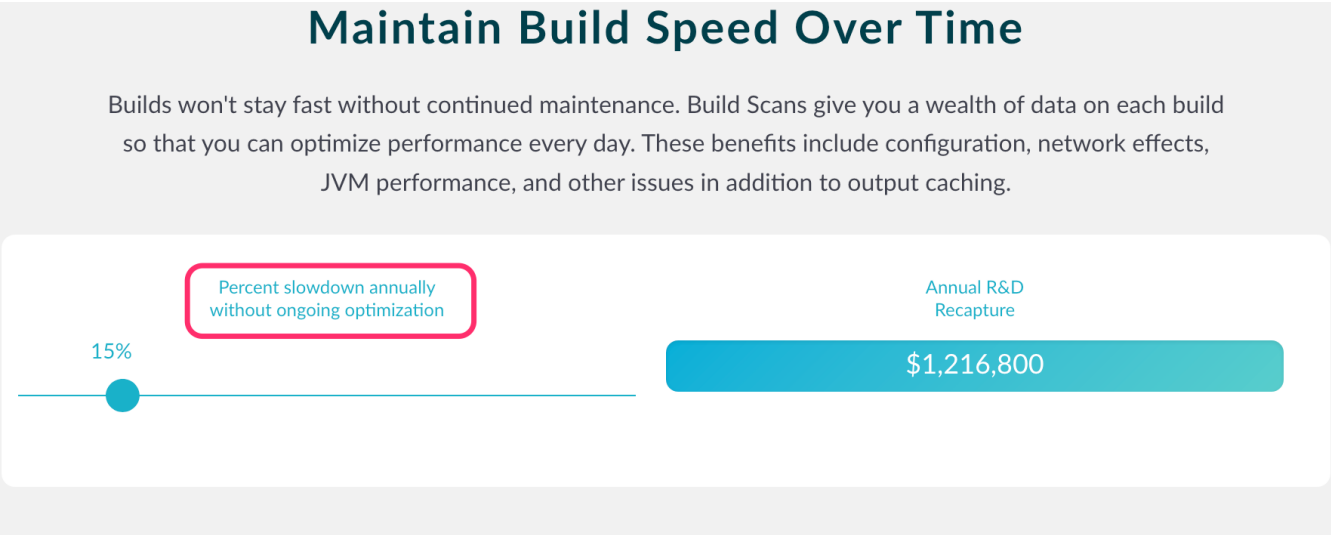
Notice how changing these settings updates both the total savings in these sections and the aggregate results shown in the top banner.

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 continuously to 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 15% or more each year.

Navigate to the section “Maintain Build Speed Over Time.” Use the slider to estimate how much savings your team could achieve by reducing or eliminating these regressions.



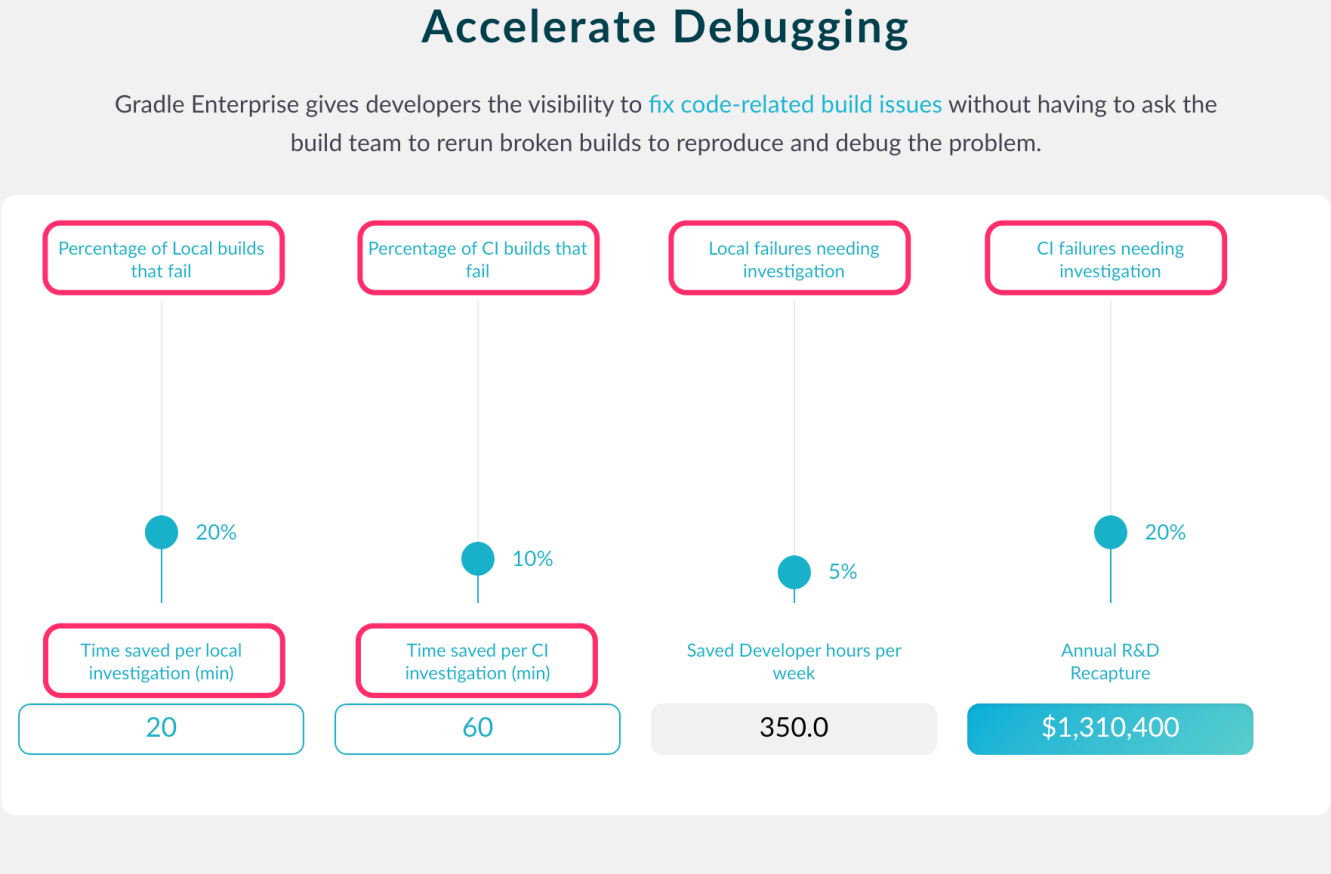
Step 4: Accelerate debugging

Debugging builds is a complex 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 that almost 25% of their engineering time is spent debugging builds!

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

Navigate downward to the final section entitled “Accelerate Debugging.” Set the sliders to the approximate percentage of time both developer and CI builds fail and 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 Failure Analytics.



Create a PDF of the calculation

To save your estimate, use the “Print PDF” button in the banner to create a PDF summary of your ROI calculation based on current inputs.

You can expect a more precise calculation by using your own actual 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

To learn more about Developer Productivity Engineering please visit the DPE Learning Center [https://gradle.com/learning-center-by-objective/]. There you will find a plethora of content that you can navigate and filter by your specific learning objective, by phase in your DPE journey, and by key topics you are most interested in. You will also find quick links to training and events, our video library, solution documentation and tutorials, and more.

If you need any help in planning your next move, please don't hesitate to contact us here.