How to measure and improve team performance

Agile Metrics IN ACTION

Christopher W. H. Davis

FOREWORD BY Olivier Gaudin

MANNING

Agile Metrics in Action

Agile Metrics in Action

How to measure and improve team performance

CHRISTOPHER W. H. DAVIS

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor:	Dan Maharry
Technical development editor:	Michael Smolyak
Copyeditor:	Linda Recktenwald
Proofreader:	Elizabeth Martin
Technical proofreader:	David Pombal
Typesetter:	Marija Tudor
Cover designer:	Marija Tudor

ISBN: 9781617292484
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – EBM – 20 19 18 17 16 15

brief contents

contents

foreword

Although it is still fairly young, the software development industry has matured considerably in the last 15 years and gone through several major transformations:

- Just a short while ago, it seems, the waterfall lifecycle was pretty much the only option for software development projects. Today, agile methodology is also frequently used.
- New development engineering practices have entered the game, such as SCM, issue tracking, build standardization, continuous integration, continuous inspection, and so on. In most organizations, it is now the norm to have a complete software factory.
- Although they started out minimalistic, modern IDEs have become a widely adopted productivity tool for developers.

This is all great news; and what's more, there is strong traction to continue these efforts and make the software development world even better. It is amazing how many development teams are seeking a common Holy Grail: continuous delivery. In other words, teams want a software development process that is predictable and repeatable, and that enables a shift to production at any time in a controlled manner.

Despite all the good things that have happened within the industry in recent years, there is a general consensus that we are not quite there yet. Software development is not yet a fully mastered science, and delivery generally still is not reliable. Projects are often delivered late, with a reduced scope of features, generating frustration at all levels in organizations and justifying their reputation for being both expensive and unpredictable.

One aspect that is missing from the recent improvements in our industry is *measurement*: measurement of what we produce, of course, but also measurement of the impact of the changes we make to improve delivery. We should be able to answer questions such as, “Did this change improve the process?” and “Are we delivering better now?” In many cases, these questions are not even asked, because doing so is not part of the company culture or because we know they are difficult to answer. If we, as an industry, want to reach the next level of maturity, we need to both ask and answer these questions. Many companies have realized this and have begun to move into the measurement area.

This is the journey that Chris will take you on in this book. It will be your steadfast companion on your expedition into measurement. Whether you are just starting out or are already an advanced “measurer,” *Agile Metrics in Action* will provide you with a 360-degree guide: from theory to practice; from defining what you should be measuring, in which area and at which frequency, to who should be targeted with which indicators; and from how to gather the numbers and which tool to use to consolidate them, to how to take action on them. The book focuses mostly on agile teams, but much of it can also be applied in other contexts. All this is done using existing tools, most of them open source and widely used.

But that’s not all! For each area of measurement, Chris presents a case study that makes it concrete and applicable, based on his own experiences. Whatever your current maturity level with regard to measuring your development process, you will learn from this book. Enjoy!

Olivier Gaudin
CEO and Cofounder
SonarSource

preface

> *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.*
>
> —agilemanifesto.org/principles.html

Development teams adopt agile practices differently based on team members, time commitments, the type of project being developed, and the software available, to name only a few factors. As quoted from the Agile Manifesto, teams should have regular check and adjust periods where they can reflect on how well they're working and how they can improve. This book demonstrates how to gather performance data to measure an agile team, interpret it, and react to it at check and adjust intervals so the team can reach their full potential.

After years of working on agile teams, I've noticed that many times teams check and adjust based on gut feelings or the latest blog post someone read. Many times teams don't use real data to determine what direction to go in or to rate their team or their process. You don't have to go far to find the data with development, tracking, and monitoring tools used today. Applications have very sophisticated performance-monitoring systems; tracking systems are used to manage tasks; and build systems are flexible, simple, and powerful. Combine all of this with modern deployment methodologies and teams shipping code to production multiple times a day in an automated fashion, and you have a wealth of data you can use to measure your team in order to adjust your process.

I've used the techniques in this book over the years, and it has been a game changer in how my teams think about their work. Retrospectives that start with conversations around data end up being much more productive and bring to light real issues to work on instead of going off of guesswork or opinion. Being able to set metrics with a team and using them in Scrums, retrospectives, or anywhere else throughout the development process helps the team focus on issues and filter out noise or celebrate parts of the process that are working well.

Finally, having this data at their fingertips typically makes managers and leadership teams happy because it gives them real insight into how the teams they're sponsoring and responsible for are really performing. They can see how their initiatives affect their teams and ultimately the bottom line.

I started using these techniques as a developer who wanted to report to leadership the true picture of the performance of my team. As I transitioned into management, I started to look at this data from another angle and encouraged my team to do the same, adding data they thought was important that reflected their day-to-day work. As I transitioned into a more senior management position, I've been able to look at this data from yet another perspective to see how strategies, initiatives, and investments affect cross-team efforts, how to bring operating efficiencies from one team to another, and how to track success on a larger scale. No matter what your role is on an agile development team, I'm sure you'll be able to apply these techniques with success in your organization.

acknowledgments

Anyone who writes a book will tell you it's a lot of work, and they're right. It's been a journey just to get to a point where I could write a book on anything, and writing itself has been an extremely rewarding experience. You wouldn't be reading this if it weren't for the love and support of many people throughout my life who have encouraged me to try new things, picked me up when I've stumbled, and given me the confidence to keep innovating.

To start, there have been my teachers through the years who noticed my love of writing and encouraged me to keep at it: my fifth-grade teacher, Mr. Rosati, who first noticed my love of writing; my seventh-grade English teacher and tennis coach, Mr. Nolan, who gave me the opportunity to continue working on my creative skills; and my tenth-grade English teacher, Ms. Kirchner, who encouraged me to publish my work. My college professors Sheila Silver, Christa Erickson, Perry Goldstein, and Daniel Weymouth all encouraged my creativity and put me on a path that combined my technical and creative abilities.

A special thank you goes out to my parents, Ward and Irene Davis, who have always stood by me and encouraged me to be myself. They gave me the freedom to grow and encouraged my efforts no matter how crazy they have been.

I'm grateful also to my lovely wife, Heather, who tolerated my long nights and weekends of typing and gave me the encouragement to keep going.

Thanks also to Grandma Davis, who taught me about the long line of inventors and writers in our family tree, which has always been a source of inspiration.

Thanks to all of the great people at Manning Publications who have helped along the way: Dan Maharry for being a great editor and Michael Stephens, Candace

Gillhoolley, and Marjan Bace for their suggestions and direction during the writing of this book. Thanks also to the production team and everyone else at Manning who worked behind the scenes.

I'd like to express my gratitude to the MEAP readers and the reviewers who took time to read my manuscript at various stages during its development and who provided invaluable feedback, especially Boyd Meier, Chris Heneghan, Enzo Matera, Ezra Simeloff, Francesco Bianchi, Hamideh Iraj, James Matlock, John Tyler, Ken Fricklas, Luca Campobasso, Marcelo Lopez, Max Hemingway, Noreen Dertinger, Steven Parr, and Sune Lomholt.

Special thanks to my technical proofreader, David Pombal, who checked the code and read the chapters one last time shortly before the book went into production, and to Olivier Gaudin for contributing the foreword to my book.

I'd also like to thank everyone who has driven me crazy by not measuring things over the years; they ultimately pushed me into exploring and mastering this topic throughout my career. Conversely, I'd like to thank everyone who has found value in these techniques or has worked on my teams that have used them, because they have helped me hone them into useful and practical ways of creating great agile teams.

about this book

In this book I hope to show you how to use the data you're already generating to make your teams, processes, and products better. The goal of the book is to teach your agile team which metrics it can use to objectively measure performance. You'll learn what data really counts, along with where to find it, how to get it, and how to analyze it. Because meaningful data may be gathered or used by any part of an agile team, you'll learn how all team members can publish their own metrics through dashboards and radiators, taking charge of communicating performance and individual accountability. Along the way, I hope you'll pick up practical data analysis techniques, including a few emerging Big Data practices.

Roadmap

This book is broken into three parts: "Measuring agile performance," "Collecting and analyzing your team's data," and "Applying metrics to your teams, processes, and software."

The first part introduces the concepts of data-driven agile teams: how to measure your processes and how to apply it to your team. Chapter 2 is an extended case study that takes the concepts from the first chapter and shows them in action on a fictional team.

The second part of this book is made up of four chapters, each focusing on a specific type of data, how to use it on your team, and what that data tells you by itself. We start off with project tracking system (PTS) data in chapter 3, move on to source control management (SCM) data in chapter 4, explore data from continuous integration (CI) and deployment systems in chapter 5, and finally in chapter 6 look at data you can get from application performance monitoring (APM) tools. Each chapter in this section ends in a case study that shows you how the data and metrics from the chapter can be applied to your team from the team's point of view.

The third part of this book shows you what you can do with the data you've learned about in the first two parts. Chapter 7 shows you how to combine different types of data to create complex metrics. Chapter 8 shows you how to measure good software and uses a variety of data and techniques to monitor your code throughout its lifecycle. Chapter 9 shows you how to report on your metrics, diving into dashboards and reports and how to use them across your organization. The final chapter in this book shows you how to measure your team against the agile principles to see how agile your team really is.

Throughout the book I use primarily open source tools to demonstrate these practices. The appendixes walk you through the code for a data-collection system called measurementor based on Elasticsearch, Kibana, Mongo, and Grails that I've used to collect, aggregate, and display data from multiple systems.

Code conventions and downloads

All the source code in the book, whether in code listings or snippets, is in a `fixed-width font like this`, which sets it off from the surrounding text. In some listings, the code is annotated to point out key concepts, and numbered bullets are sometimes used in the text to provide additional information about the code. The code is formatted so that it fits within the available page space in the book by adding line breaks and using indentation carefully.

The code for this book is available for download from the publisher's website at www.manning.com/AgileMetricsinAction and is also posted on GitHub at github.com/cwhd/measurementor.

Feel free to contribute to the project, fork it, or use the concepts to roll your own version in your language of choice. I tried to make it as easy as possible to use by employing open source tools for the bulk of the functionality. There's a Puppet script that will install everything you need and a Vagrant file so you can get up and running in a virtual machine pretty quickly.

In appendix A, I detail the architecture of the system used throughout the book.

Author Online

Purchase of *Agile Metrics in Action* includes free access to a private web forum run by Manning Publications, where you can make comments about the book, ask technical questions, and receive help from the author and from the community. To access the forum and subscribe to it, go to www.manning.com/AgileMetricsinAction. This page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialog between individual readers and between readers and the author can take place.

It's not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

About the author

Christopher W. H. Davis has been leading and working on development teams since the latter part of the twentieth century. Working in the travel, finance, healthcare, telecommunications, and manufacturing industries, he's led diverse teams in several different environments around the world.

An avid runner, Chris enjoys the beautiful and majestic Pacific Northwest in Portland, Oregon, with his wife and two children.

About the cover illustration

The figure on the cover of *Agile Metrics in Action* is captioned "Montagnard du Nord de l'Ecosse," which means an inhabitant of the mountainous regions in the north of Scotland. The mountaineer is shown wearing a long blue robe and a red hat, and is holding an older version of a traditional Scottish bagpipe.

The illustration is taken from a nineteenth-century edition of Sylvain Maréchal's four-volume compendium of regional dress customs published in France. Each illustration is finely drawn and colored by hand. The rich variety of Maréchal's collection reminds us vividly of how culturally apart the world's towns and regions were just 200 years ago. Isolated from each other, people spoke different dialects and languages. In the streets or in the countryside, it was easy to identify where they lived and what their trade or station in life was just by their dress.

Dress codes have changed since then and the diversity by region and country, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns, regions, or nations. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Maréchal's pictures.

Part 1

Measuring agile teams

Agile development has guidelines instead of hard-and-fast rules. Many teams that practice agile struggle with measuring their processes and their teams, despite having all the data they need to do the measurement.

Chapter 1 navigates through the challenges of agile measurement. You'll learn where you can get data to measure your team, how to break down problems into measureable units, and how to incorporate better agile measurement on your team.

Chapter 2 puts what you learned in chapter 1 into action through a case study where a team uses several open source technologies to incorporate better agile measurement. They identify key metrics, use the tools to collect and analyze data, and check and adjust based on what they find.

1 Measuring agile performance

This chapter covers

- Struggling with agile performance measurement
- Finding objective data for measuring agile performance
- Answering performance questions with data you're generating
- Adopting agile performance measurement

There isn't a silver-bullet metric that will tell you if your agile teams are performing as well as they can. Performance improvement is made possible by incorporating what you learn about your team's performance into how your team operates at regular intervals. Collecting and analyzing data in the form of metrics is an objective way to learn more about your team and a way to measure any adjustments you decide to make to your team's behavior.

1.1 Collect, measure, react, repeat—the feedback loop

Working with metrics in a feedback loop in parallel with your development cycle will allow you to make smarter adjustments to your team and help improve communication across your organization. Here are the steps in the feedback loop:

- *Collect*—Gather all the data you can about your team and performance. Understand where you are before you change anything.
- *Measure*—Analyze your data.
  - Look for trends and relationships between data points.
  - Formulate questions about your team, workflow, or process.
  - Determine how to adjust based on your analysis.
- *React*—Apply the adjustments based on your analysis.
- *Repeat*—Keep tabs on the data you've determined should be affected so you can continuously analyze and adjust your team.

The feedback loop depicted in figure 1.1 naturally fits into the operations of agile teams. As you're developing, you're generating and collecting data; when you pause to check and adjust, you're doing your analysis; and when you start again, you're applying lessons learned and generating more data.

Continuous delivery and continuous improvement

The word *continuous* is everywhere in agile terminology: continuous integration, continuous delivery, continuous improvement, continuous testing, continuous (choose your noun). No matter if you're doing Scrum, Kanban, extreme programming (XP), or some custom form of agile, keeping your stream of metrics as continuous as your check-and-adjust period is key.

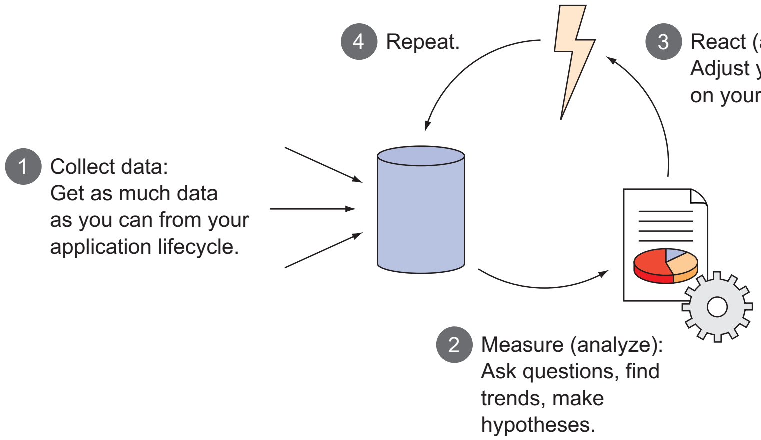


Figure 1.1 The feedback loop: collecting data from your process, asking questions, and tweaking your process

To begin you need to know where you stand. You're probably already tracking something in your development process, like what was accomplished, how much code is changing, and how your software is performing.

Your questions will drive the analysis phase by providing a lens with which to view this data. Through this lens you can identify data points and metrics that help answer your questions. These data points then become the indicators of progress as you adjust your process to get to an ideal operating model for your team. Once you have questions you want to answer, then you can start identifying data points and metrics that represent them. At that point you can adjust how your team operates and track the metrics you've identified.

1.1.1 What are metrics?

> *"A method of measuring something, or the results obtained from this."*
>
> —*metrics* defined by Google

In the scope of this book metrics will represent the data you can get from your application lifecycle as it applies to the performance of software development teams. A metric can come from a single data source or it can be a combination of data from multiple data sources. Any data point that you track eventually becomes a metric that you can use to measure your team's performance. Examples of common metrics are:

- *Velocity*—The relative performance of your team over time
- *Changed lines of code (CLOC)*—The number of lines of code that were changed over time

Metrics can be used to measure anything you think is relevant, which can be a powerful tool when used to facilitate better communication and collaboration. These metrics in effect become key performance indicators (KPIs) that help measure what's important to your team and your business.

Using KPIs and data trends to show how certain data points affect behaviors and progress, you can tweak the behavior of your team and watch how the changes you make affect data that's important to it.

1.2 Why agile teams struggle with measurement

As you drive down the road, the gauges on your dashboard are the same as the gauges in the cars around you. There are highway signs that tell you how fast you should go and what you should do. Everyone on the road has gone through the same driving tests to get a driver's license and knows the same basic stuff about driving.

Agile development is nothing like this. The people involved in delivering a software product have different roles and different backgrounds. Their idea of what *good* means can vary substantially.

- A developer might think that *good* means a well-engineered piece of software.
- A product owner might define *good* as more features delivered.
- A project manager may think *good* means it was done on time and within budget.

Even though everyone is doing something different, they're all headed down the same road.

So now picture a bunch of people driving down the same road in different vehicles with totally different gauges. They all need to get to the same place, yet they're all using different information to get there. They can follow each other down the road, but when they pull over to figure out how the trip is going, each has different ideas of what the gauges in their vehicle are telling them.

Agile is a partnership between product owners and product creators. To make the most out of that partnership you need to smooth out the communication differences by turning the data you're already generating in your development process into agreed-upon metrics that tell you how your team is doing.

Let's look at some universal problems that end up getting in the way of a common understanding of agile metrics:

- Agile definitions of measurement are not straightforward.
- Agile deviates from textbook project management.
- Data is generated throughout the entire development process without a unified view.

All of these are common problems that deserve exploring.

1.2.1 Problem: agile definitions of measurement are not straightforward

There are a few commonly accepted tenets about measuring agile that tend to be rather confusing. Let's start with common agile principles:

- *Working software is the primary measure of progress.* That statement is so ambiguous and open to interpretation that it makes it very hard for teams to pinpoint exactly how to measure progress. Essentially the point is you are performing well if you're delivering products to your consumers. The problem is the subjective nature of the term *working software*. Are you delivering something that works per the original requirements but has massive security holes that put your consumer's data in jeopardy? Are you delivering something that is so non-performant that your consumers stop using it? If you answered yes to either question, then you're probably not progressing. There's a lot more to measuring progress than delivering working software.
- *Any measurement you're currently using has to be cheap.* So what's included in the cost associated with gathering metrics? Are licenses to software included? Are you looking at the hours spent by the people collecting measures? This statement belittles the value of measuring performance. When you start measuring something, the better thing to keep in mind is if the value you get from the improvement associated with the metric outweighs the cost of collecting it. This open statement is a good tenet, but like our first statement, it's pretty ambiguous.
- *Measure only what matters.* This is a bad tenet. How do you know what matters? When do you start tracking new things and when do you stop tracking others? Because these are hard questions, metrics end up getting thrown by the wayside

when they could be providing value. A better wording would be "measure everything and figure out why metrics change unexpectedly."

1.2.2 Problem: agile focuses on a product, not a project

One of the strengths of agile development methods is the idea that you are delivering a living *product*, not completing a *project*. A project is a defined set of time within which something is developed and tracked; a product is a living thing that continues to change to meet the needs of the consumer. This is shown in figure 1.2.

A good example of a project would be building a bridge. It gets designed, built to spec, and then it stands likely unchanged for a long time. Before you start the project you design exactly what you need so you can understand the total cost, and then you track the progress against the plan to make sure you stay on schedule and budget. This process is the art of project management.

A good example of a product is a mobile application geared toward runners that shows their paths on a map and aggregates their total mileage. It's a tool; you can run with it and get lots of data about your workouts. There are several competing apps that have the same functionality but different bells and whistles. To keep up with the competition, any app that competes in that space must constantly evolve to stay the most relevant product its consumers can use. This evolution tends to happen in small iterations that result in frequent feature releases which are immediately consumed and tracked for feedback that helps shape the direction of the next feature.

Frequently, software products are built through the lens of a project, which ends up mixing project management techniques typically used for large predictive projects

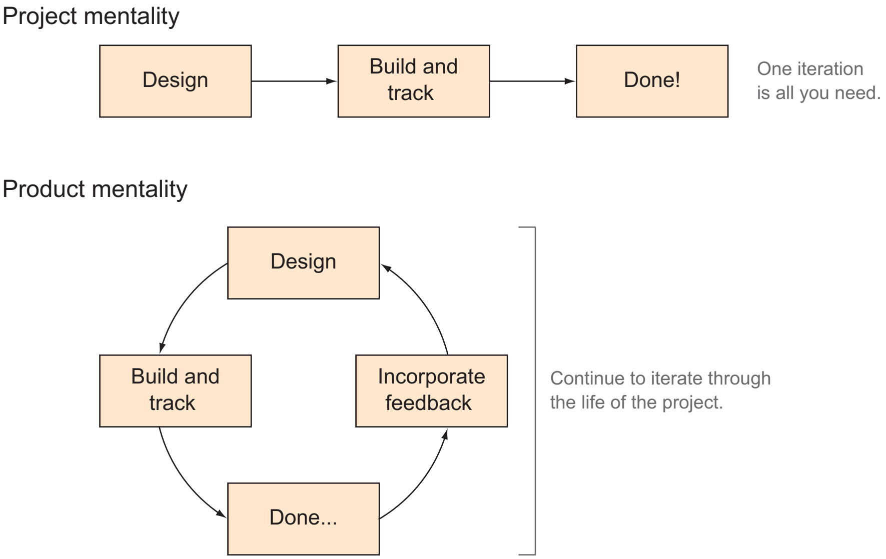
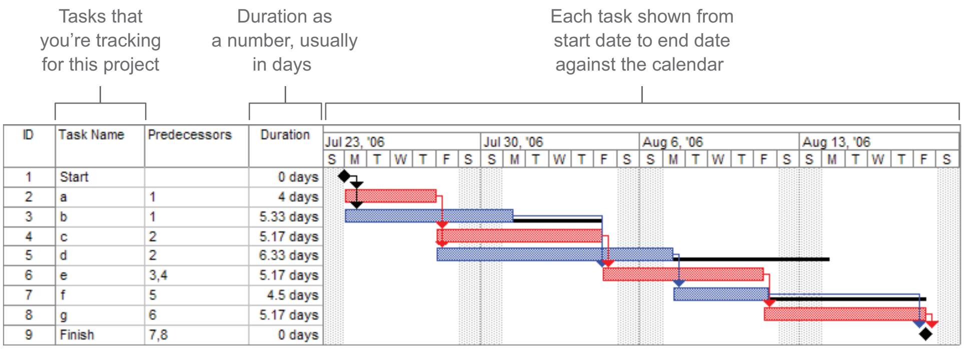


Figure 1.2 Project vs. product mentality



Tasks that you're tracking for this project

Duration as a number, usually in days

Each task shown from start date to end date against the calendar

ID	Task Name	Predecessors	Duration
1	Start		0 days
2	a	1	4 days
3	b	1	5.33 days
4	c	2	5.17 days
5	d	2	6.33 days
6	e	3,4	5.17 days
7	f	5	4.5 days
8	g	6	5.17 days
9	Finish	7,8	0 days

Figure 1.3 An example Gantt chart

with agile product management used for consistent iterative delivery. This ends up putting project constraints on agile projects that don't fit. An example would be using a Gantt chart on an agile project. Gantt charts are great for tracking long-running projects but usually cause heartache when used to track something with a definition that is regularly in flux. An example Gantt chart is shown in figure 1.3.

From a project-management perspective it would be great to have a clear prediction of the cost of a complex feature that may take several sprints to complete, which is why these tools end up tracking agile teams.

1.2.3 *Problem: data is all over the place without a unified view*

Your team is generating a lot of data throughout your entire software development lifecycle (SDLC). Figure 1.4 shows the components that are typically used by an agile software delivery team.

The first problem is there is no standard for any of these boxes. There are several project-tracking tools, plus different source control and continuous integration (CI) systems, and your deployment and application-monitoring tools will vary depending on your technology stack and what you're delivering to your consumers. Different systems end up generating different reports that don't always come together. In addition, there's no product or group of products that encompasses all of the pieces of the software development lifecycle.

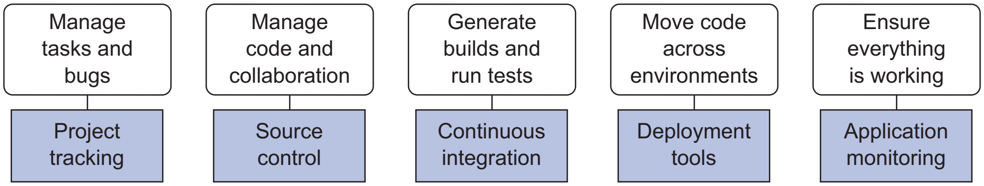

Manage tasks and bugs
Project tracking
Manage code and collaboration
Source control
Generate builds and run tests
Continuous integration
Move code across environments
Deployment tools
Ensure everything is working
Application monitoring

Figure 1.4 Groups of systems used to deliver software and what they do

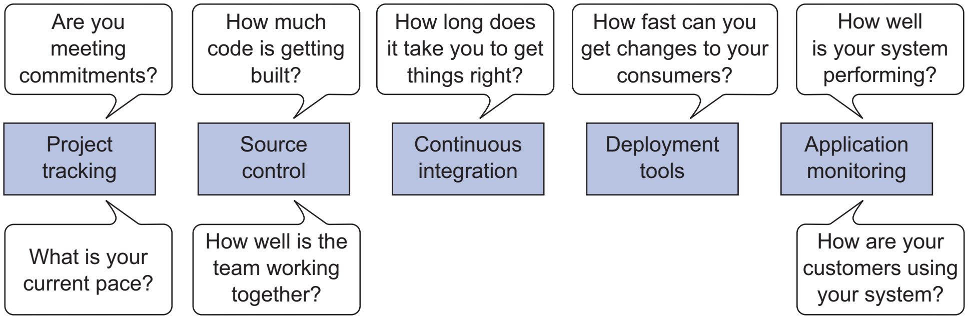

Are you meeting commitments?
How much code is getting built?
How long does it take you to get things right?
How fast can you get changes to your consumers?
How well is your system performing?
Project tracking
Source control
Continuous integration
Deployment tools
Application monitoring
What is your current pace?
How well is the team working together?
How are your customers using your system?

Figure 1.5 Questions you can answer with data from systems in your SDLC.

The second issue is that people in different roles on the team will focus on using different tools throughout the SDLC. Scrum masters are likely looking for data around your tasks in your project system, and developers pay the most attention to data in your source control and CI systems. Depending on the size and makeup of your team, you may have a completely different group looking after your application monitoring and even perhaps your deployment tools. Executives likely don't care about raw source control data and developers may not care about the general ledger, but when presented in the right way, having a unified view of all data is important in understanding how well teams are performing. No matter what angle you're looking from, having only a piece of the whole picture obviously limits your ability to react to trends in data in the best possible way.

1.3 What questions can metrics answer, and where do I get the data to answer them?

In the previous section we noted that all the data you're collecting in your SDLC is a barrier to understanding what's going on. On the flip side there's a huge opportunity to use all that data to see really interesting and insightful things about how your team works. Let's take the components of figure 1.4 and look at questions they answer in figure 1.5.

When you combine these different data points, you can start answering some even more interesting big-picture questions like the ones shown in figure 1.6.

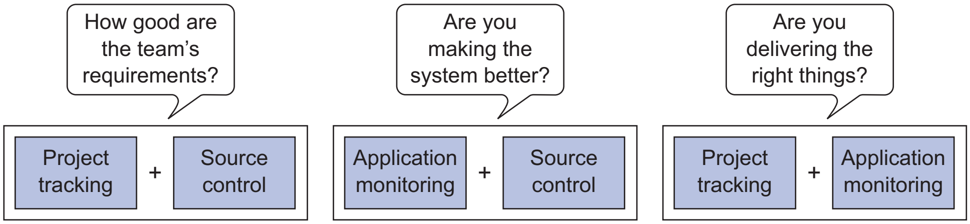

How good are the team's requirements?
Project tracking + Source control
Are you making the system better?
Application monitoring + Source control
Are you delivering the right things?
Project tracking + Application monitoring

Figure 1.6 Adding data together to answer high-level questions

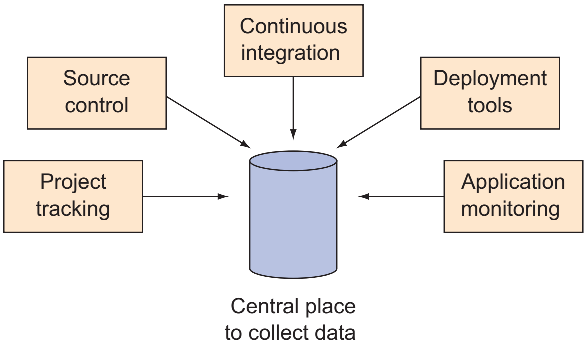


Figure 1.7 Collecting data from numerous places

Your team is already generating a ton of data through CI, your task management system, and your production-monitoring tools. These systems are likely to have simple APIs that you can use to get real data that's better for communication and for determining KPIs and that you can incorporate into your process to help your team achieve better performance. As you know, the first step in the feedback loop is to start collecting data. By putting it into a central database, you're adding the systems from figure 1.4 to the feedback loop in figure 1.1 to get figure 1.7.

There are several ways to do this:

- Semantic logging and log aggregators
- Products like Datadog (www.datadoghq.com) or New Relic Insights (newrelic .com/insights)
- An open source database like Graphite (graphite.wikidot.com/) to collect and display data
- A DIY system to collect and analyze metrics

If you're interested in building something yourself, check out appendix A, where we talk through that topic. If you have any of the other tools mentioned here, then by all means try to use them.

For now let's look at where you can get this data and what systems you should put in place if you don't already have them.

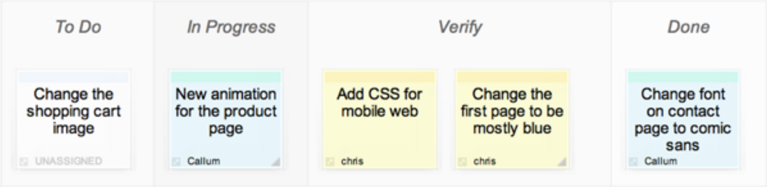
1.3.1 Project tracking

Tools like JIRA Agile, Axosoft OnTime, LeanKit, TFS, Telerik TeamPulse, Planbox, and FogBugz can all track your agile team, and all have APIs you can tap into to combine that data with other sources. All of these have the basics to help you track the common agile metrics, and although none goes much deeper, you can easily combine their data with other sources through the APIs. From these systems you can get data such as how many story points your team is completing, how many tasks you're accomplishing, and how many bugs are getting generated. A typical Scrum board is depicted in figure 1.8.

Here are questions you can answer with project-tracking data alone:

- How well does your team understand the project?
- How fast is the team moving?
- How consistent is the team in completing work?

Project-tracking data is explored in depth in chapter 3.



To Do	In Progress	Verify		Done
Change the shopping cart image UNASSIGNED	New animation for the product page Callum	Add CSS for mobile web chris	Change the first page to be mostly blue chris	Change font on contact page to comic sans Callum

Figure 1.8 A typical Scrum board to track tasks in a sprint

1.3.2 Source control

Source control is where the actual work is done and collaboration across development teams happens. From here you can see which files are changing and by how much. Some source control systems allow you to get code reviews and comments, but in other cases you need additional systems to get that type of information. Tools like Stash, Bitbucket, and GitHub have rich REST-based APIs that can get you a wealth of information about your codebase. If you're still using SVN or something even older, then you can still get data, just not as conveniently as Git- or Mercurial-based systems. In that case you may need something like FishEye and Crucible to get more data around code reviews and comments about your code.

Here are two questions you can answer from source control alone:

- How much change is happening in your codebase?
- How well is/are your development team(s) working together?

We dive deep into source control in chapter 4.

1.3.3 The build system

After someone checks something into source control, it typically goes into your build system, which is where the code from multiple check-ins is integrated, unit tests are run, your code is packaged into something that can be deployed somewhere, and reports are generated. All of this is called continuous integration (CI). From here you can get lots of great information on your code: you can see how well your team's changes are coordinated, run your codebase against rule sets to ensure you're not making silly mistakes, check your test coverage, and see automated test results.

CI is an essential part of team software development, and there are several systems that help you get going quickly, including TeamCity, Jenkins, Hudson, and Bamboo. Some teams have taken CI past the integration phase and have their system deploy their code while they're at it. This is called continuous delivery (CD). Many of the same systems can be employed to do CD, but there are products that specialize in it, like ThoughtWorks, Go CD, Electric Cloud, and Nolio.

Whether you're doing CI or CD, the main thing you need to care about is the information that is generated when your code is getting built, inspected, and tested. The more mature a CI/CD process your team has, the more data you can get out of it.

Here are questions you can answer from CI alone:

- How fast are you delivering changes to your consumer?
- How fast can you deliver changes to your consumer?
- How consistently does your team do STET work?

More details on the data you can get from this step are found in chapter 5.

1.3.4 System monitoring

Once your code goes into production, you should have some type of system that looks at it to make sure it's working and that tells you if something goes wrong, such as whether a website becomes unresponsive or if your mobile app starts to crash every time a user opens it. If you're doing a really great job with your testing, you likely are paying attention to your system monitoring during your testing phase as well making sure you don't see any issues from a support perspective before your code goes into production.

The problem with system monitoring is that it's largely reactive, not proactive. Ideally all your data should be as close to the development cycle as possible, in which case you'll be able to react to it as a team quickly. By the time you're looking at system-monitoring data in a production environment your sprint is done, the code is out, and if there's a problem, then you're usually scrambling to fix it rather than learning and reacting to it with planned work.

Let's look at ways to mitigate this problem. One way is to use system monitoring as you test your code before it gets to production. There are a number of ways your code can make it to production. Typically you see something like the flow shown in figure 1.9, where a team works in STET local development environment and pushes changes to an integration environment, and where multiple change sets are tested together and a QA environment verifies what the end user or customer will get before you ship your code to production.

Because teams typically have multiple environments in the path to production, to make system monitoring data as proactive as possible you should be able to access it as

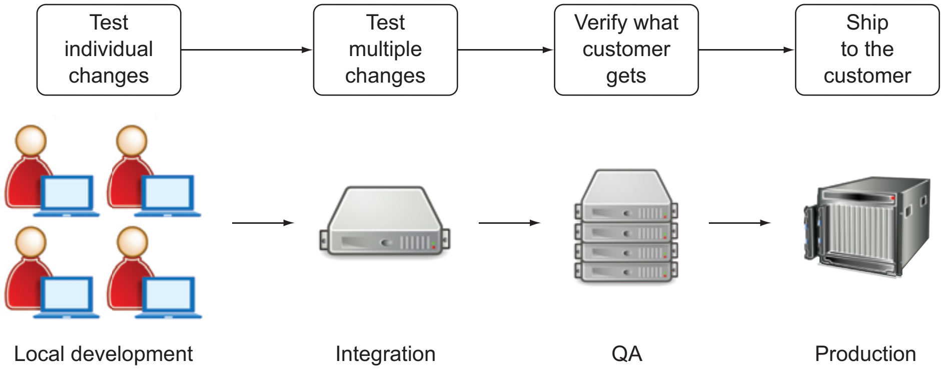


Figure 1.9 Typical environment flow in the path to production

close to the development environment as possible, ideally in the integration and/or QA stages.

A second way to mitigate this problem is to release new code to only a small number of your consumers and monitor how their activity affects the system. This is usually referred to as a *canary deploy* and is becoming more common in agile teams practicing CD.

Depending on your deployment platform, different tools are available for system monitoring. New Relic, AppDynamics, and Dynatrace are all popular tools in this space. We will go into much more detail about these in chapter 6.

All of the data we've looked at so far can tell you a lot about your team and how effectively you're working together. But before you can make this data useful, you need to figure out what is good and what is bad, especially in an agile world where your core metrics are relative.

Here are questions you can answer from your production-monitoring systems:

- How well is your code functioning?
- How well are you serving the consumer?

1.4 Analyzing what you have and what to do with the data

The second step in the feedback loop is analysis, or figuring out what to do with all the data you've collected. This is where you ask questions, look for trends, and associate data points with behaviors in order to understand what is behind your team's performance trends.

Essentially what you need to do is get all the data you've collected and run some kind of computation on it, as shown in figure 1.10, to determine what the combined data points can tell you. The best place to start is with the question you're trying to answer. When you're doing this, be careful and ask yourself *why* you want to know what you're asking. Does it really help you answer your question, solve a problem, or track something that you want to ensure you're improving? When you're trying to figure out what metrics to track, it's easy to fall into a rabbit hole of "it would be great to

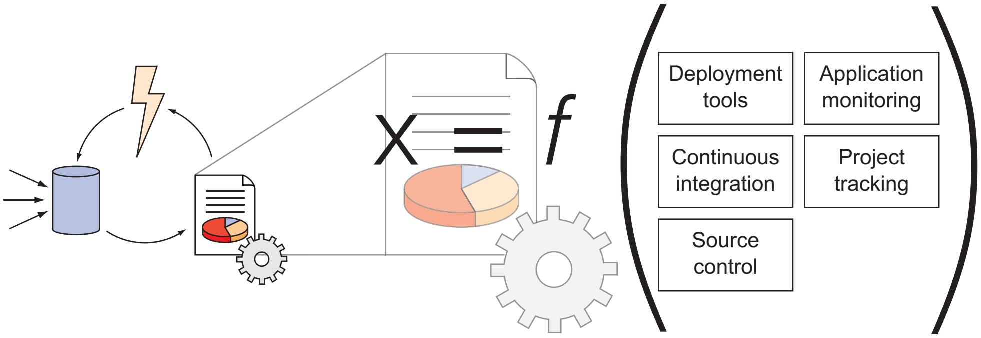


Figure 1.10 X is what you want to answer; some combination of your data can get you there.

have.” Be wary of that. Stick with “just in time” instead of “just in case” when it comes to thinking about metrics. Then plug X into figure 1.10, where X is the question you want to answer.

1.4.1 Figuring out what matters

If you’re swimming in a sea of data, it’s hard to figure out what data answers what question and which metrics you should track. One strategy I find useful when I’m trying to figure out which metrics matter is mind mapping.

Mind mapping is a brainstorming technique where you start with an idea and then keep deconstructing it until it’s broken down into small elements. If you’re not familiar with mind mapping, a great tool to start with is XMind (www.xmind.net/), a powerful and free (for the basic version) tool that makes mind mapping pretty simple.

If you take a simple example, “What is our ideal pace?” you can break that down into smaller questions:

- What is our current velocity?
- Are we generating tech debt?
- What are other teams in the company doing?

From there you could break those questions down into smaller ones, or you can start identifying places where you can get that data. An example mind map is shown in figure 1.11.

Thinking a project through, mapping, and defining your questions give you a place to start collecting the type of data you need to define your metrics.

1.4.2 Visualizing your data

Having data from all over the place in a single database that you’re applying formulas to is great. Asking everyone in your company to query that data from a command line probably isn’t the best approach to communicate effectively. Data is such a part of the fabric of how business is done today that there is a plethora of frameworks for

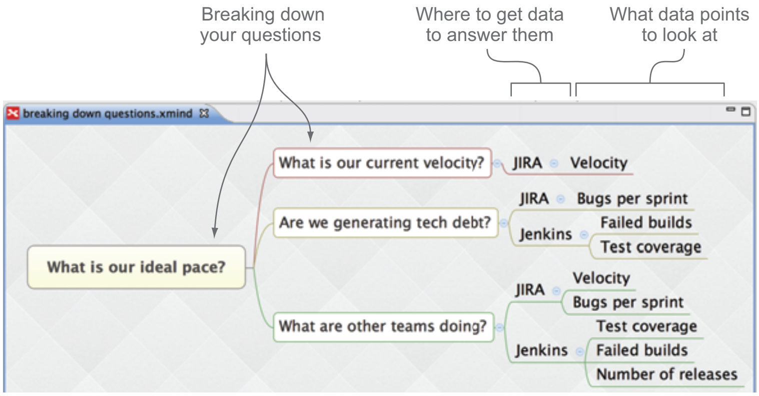

Breaking down your questions
Where to get data to answer them
What data points to look at
breaking down questions.xmind
What is our ideal pace?
What is our current velocity?
JIRA
Velocity
Are we generating tech debt?
JIRA
Bugs per sprint
Jenkins
Failed builds
Test coverage
What are other teams doing?
JIRA
Velocity
Bugs per sprint
Jenkins
Test coverage
Failed builds
Number of releases

Figure 1.11 Breaking down questions with XMind

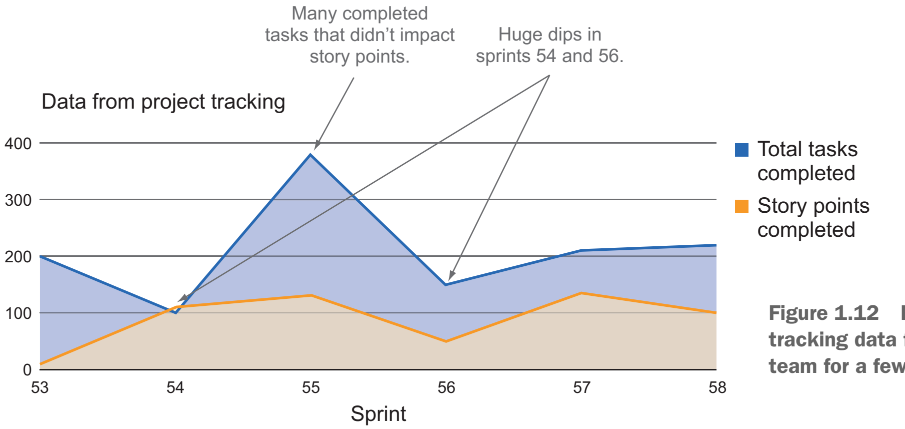


Figure 1.12 Project tracking data for a team for a few sprints

visualizing it. Ideally you should use a distributed system that allows everyone access to the charts, graphs, radiators, dashboards, and whatever else you want to show.

Keep in mind that data can be interpreted numerous ways, and statistics can be construed to prove different points of view. It's important to display the data you want to show in such a way that it, as clearly as possible, answers the questions you're asking. Let's look at one example.

If you want to see how productive your team is, you can start by finding out how many tasks are being completed. The problem with measuring the number of tasks is that a handful of very difficult tasks could be much more work than many menial ones. To balance that possibility you can add the amount of effort that went into completing those tasks, measured in figure 1.12 with story points.

Whoa! Look at those dips around sprints 54 and 56! This obviously points to a problem, but what is the problem? You know that the development team was working really hard, but it didn't seem like they got as much done in those sprints. Now let's take a look at what was going on in source control over the same time period, as shown in figure 1.13.

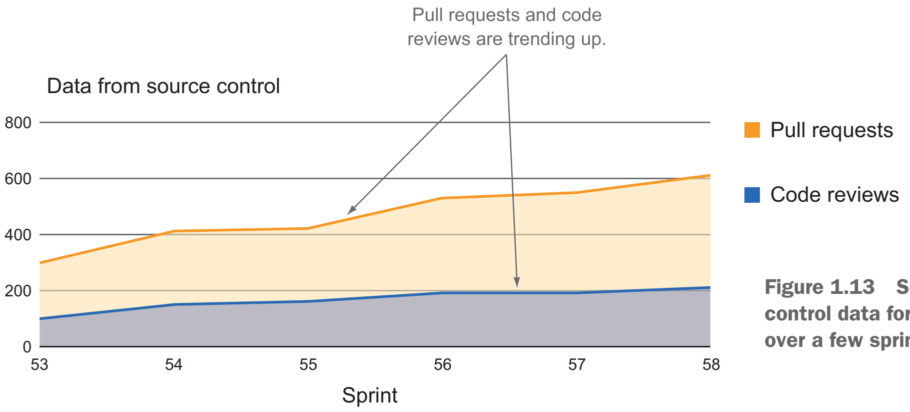


Figure 1.13 Source control data for a team over a few sprints

If anything, it looks like the development teams were doing more work over time—a lot more work! So if you're writing more code, and the number of bugs seems to be pretty constant relative to the tasks you're getting done, but you're not consistent in the amount of stuff you're delivering, what's the real issue? You may not have the answer yet, but you do have enough data to start asking more questions.

1.5 Applying lessons learned

Applying lessons learned can be the hardest part of the feedback loop because it implies behavioral changes for your team. In other words, collecting and analyzing data are technical problems; the final part is a human problem. When you're developing a software product, you're constantly tweaking and evolving code, but it's not always easy to tweak your team's behavior.

Whenever you're trying to change something because it's not good, you can easily perceive that someone is doing something bad or wrong, and who likes to be told that? The first key is always keeping an open mind. Remember that we're all human, no one is perfect, and we always have the capacity to improve. When things aren't perfect (and are they ever?), then you have an opportunity to make them better.

When you're presenting trends that you want to improve on, ensure that you're keeping a positive spin on things. Focus on the positive that will come from the change, not the negative that you're trying to avoid. Always strive to be better instead of running away from the bad.

Always steer away from blame or finger pointing. Data is a great tool when used for good, but many people fear measurement because it can also be misunderstood. An easy example of this would be measuring lines of code (LOC) written by your development team. LOC doesn't always point to how much work a developer is getting done. Something that has a lot of boilerplate code that gets autogenerated by a tool may have a very high LOC; an extremely complex algorithm that takes a while to tune may have a very low LOC. Most developers will tell you the lower LOC is much better in this case (unless you're talking to the developer who wrote the code generator for the former). It's always important to look at trends and understand that all the data you collect is part of a larger picture that most likely the team in the trenches will understand the best. Providing context and staying flexible when interpreting results is an important part of ensuring successful adoption of change. Don't freak out if your productivity drops on every Wednesday; maybe there's some good reason for that. The important thing is that everyone is aware of what's happening, what you're focused on improving, and how you're measuring the change you want to see.

1.6 Taking ownership and measuring your team

Development teams should be responsible for tracking themselves through metrics that are easy to obtain and communicate. Agile frameworks have natural pauses to allow your team to check and adjust. At this point you should have an idea of the good and bad of how you approach and do your work. Now, take the bull by the horns and start measuring!

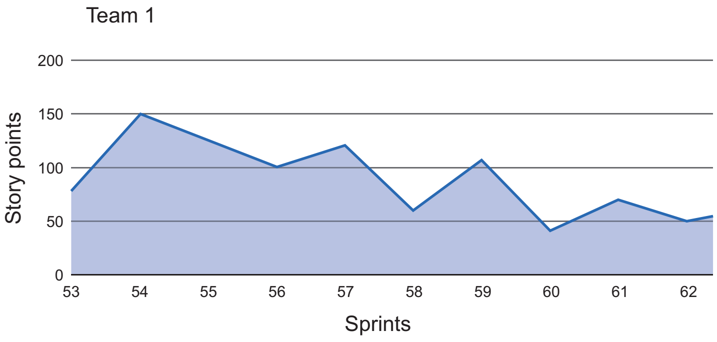


Figure 1.14 Team 1's complete story points over time

1.6.1 Getting buy-in

You can start collecting data and figuring out metrics on your own, but ideally you want to work with your team to make sure everyone is on the same page. These metrics are being used to measure the team as a whole so it's important that everyone understands what you're doing and why and also knows how to contribute to the process.

You can introduce the concepts we've talked about so far to get your team ready to do better performance tracking at any time, but perhaps the most natural time would be when your team breaks to see how STET doing. This could be at the end of a sprint, after a production release, at the end of the week, or even at the end of the day. Wherever that break is on your team is where you can introduce these concepts.

Let's look at a team which is using Scrum and can't seem to maintain a consistent velocity. Their Scrum master is keeping track of their accomplishments sprint after sprint and graphing it; as shown in figure 1.14, it's not looking good.

After showing that to leadership, everyone agrees that the team needs to be more consistent. But what's causing the problem? The development team decides to pull data from all the developers across all projects to see if there's someone in particular who needs help with their work. Lo and behold, they find that Joe Developer, a member of Team 1, is off the charts during some of the dips, as shown in figure 1.15. After digging a bit deeper, it is determined that Joe Developer has specific knowledge of the product a different team was working on and was actually contributing to multiple

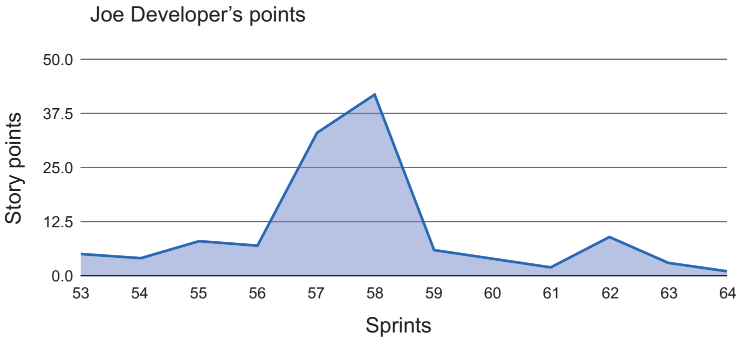


Figure 1.15 Joe Developer's contribution is off the charts, but his team is not performing well.

teams during that time. He was overworked, yet it looked as if the team were underperforming.

In this case it's pretty easy to show that the commitment to multiple projects hurts the individual teams, and apparently Joe Developer is way more productive when working on the other team's product. People are usually eager to show trends like this, especially when they indicate that individual performance is great but for some reason forces outside the team are hurting overall productivity. If you're being measured, then you might as well ensure that the entire story is being told. In this case the data ended up pointing to a problem that the team could take action on: fully dedicate Joe Developer to one team to help avoid missing goals on another team and to not burn your engineers as you're delivering product.

Collecting data is so easy that you shouldn't need buy-in from product sponsors to get it started. Remember that collecting metrics needs to be cheap, so cheap that you shouldn't have to ask for permission or resources to do it. If you can show the value of the data you're collecting by using it to point to problems and corresponding solutions and ultimately improve your code-delivery process, then you will likely make your product sponsor extremely happy. In this case it's usually better to start collecting data that you can share with sponsors to show them you know your stuff rather than trying to explain what you're planning to do before you go about doing it.

1.6.2 Metric naysayers

There will likely be people in your group who want nothing to do with measuring their work. Usually this stems from the fear of the unknown, fear of Big Brother, or a lack of control. The whole point here is that teams should measure themselves, not have some external person or system tell them what's good and bad. And who doesn't want to get better? No one is perfect—we all have a lot to learn and we can always improve. Nevertheless here are some arguments I've heard as I've implemented these techniques on various teams:

- *People don't like to be measured.* When we are children, our parents/guardians tell us we are good or bad at things. In school, we are graded on everything we do. When we go out to get a job, we're measured against the competition, and when we get a job, we're constantly evaluated. We're being measured all the time. The question is, do you want to provide the yardstick or will someone else provide it?
- *Metrics are an invasion of my privacy.* Even independent developers use source control. The smallest teams use project tracking of some kind, and ideally you're using some kind of CI/CD to manage your code pipeline. All of the data to measure you is already out there; you're adding to it every day and it's a byproduct of good software practice. Adding it all together for feedback on improvement potential isn't an invasion of privacy as much as it's a way to make sure you're managing and smoothing out the bumps in the agile development road.

- *Metrics make our process too heavy.* If anything, metrics help you identify how to improve your process. If you choose the right data to look at, then you can find parts of your process that are too heavy, figure out how to streamline them, and use metrics to track your progress. If it feels like metrics are making your process too heavy because someone is collecting data and creating the metrics manually, then you have a golden opportunity to improve your process by automating metrics collection and reporting.
- *Metrics are hard; it will take too much time.* Read on, my friend! There are easy ways to use out-of-the-box technology to obtain metrics very quickly. In appendices A and B we outline open source tools you can use to get metrics from your existing systems with little effort. The key is using the data you have and simply making metrics the byproduct of your work.

1.7 Summary

This chapter showed you where to find data to measure your team and your process and an overview of what to do with it. At this point you've learned:

- Measuring agile development is not straightforward.
- By collecting data from the several systems used in your SDLC, you can answer simple questions.
- By combining data from multiple systems in your SDLC, you can answer big-picture questions.
- By using mind mapping you can break questions down into small enough chunks to collect data.
- By using simple techniques, measuring agile performance isn't so hard.
- Showing metrics to your teammates easily demonstrates value and can easily facilitate buy-in.

Chapter 2 puts this in action through an extended case study where you can see a team applying these lessons firsthand.

2 Observing a live project

This chapter covers

- Tracking progress of multiple development efforts
- Using Elasticsearch (EC) and Kibana to track and visualize metrics over time
- Communicating progress back to leadership
- Using data to improve day-to-day operations

In chapter 1 we walked through some of the concepts around agile development and saw how a team is typically measured, looked at problems that crop up, and got an overview of how to solve those problems. Now let's look at a team that put these concepts into action, how they did it, and how it all turned out—all through a case study.

2.1 A typical agile project

Software development is flexible and can move very quickly. Every project is different, so any one project isn't going to encapsulate all of the characteristics of every project.

Let's take a look at Blastamo Music, LLC, a company that makes guitar pedals and uses an e-commerce site written by their development team. As the company grew, it built better products by replacing the electronics in the pedals with software and acquired several smaller companies that had the technology and resources needed to grow even faster. Among these technologies was an e-commerce system that had several features the leadership team wanted for their site.

2.1.1 How Blastamo Music used agile

As with many teams, the agile process used by our case study grew along with the organization, instead of being very textbook. The Blastamo team used Scrum as the cornerstone of their agile process, they used cards for their tasks that were estimated and worked on, they had a task board on which they moved the tasks along based on the status, and they had release processes that varied from team to team. The typical flow of the cards was:

- *In definition*—The card is being defined by a product owner.
- *Dev ready*—The card is defined and ready to be coded.
- *In dev*—A developer is working on it.
- *QA ready*—Development is finished and it's waiting for a member of the QA team to check it.
- *In QA*—A member of the QA team is checking it.
- *Done*—The feature is ready for production.

The system groups and technologies the team used are shown in figure 2.1.

The agile development team used JIRA (https://www.atlassian.com/software/jira) to manage the cards for their tasks that they estimated and worked on, GitHub (https://github.com/) to manage their source control, Jenkins (http://jenkins-ci.org/) for CI and deployments, New Relic (http://newrelic.com/) for application monitoring, and Elasticsearch/Logstash/Kibana (ELK) (www.elasticsearch.org/overview/) for production log analysis.

2.2 A problem arises

In one of the acquisitions, the team got a robust piece of e-commerce software that complemented a lot of their current site's functionality. Management thought the

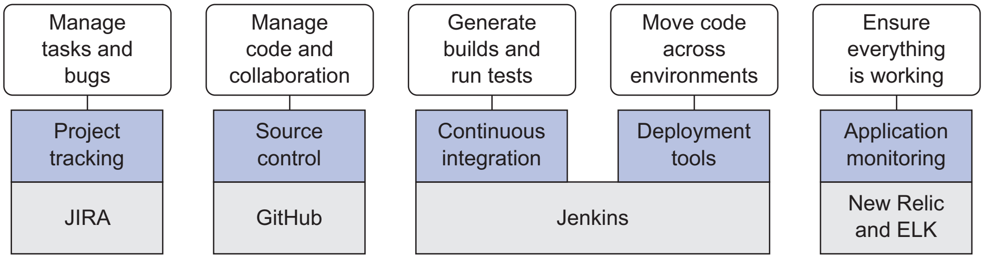
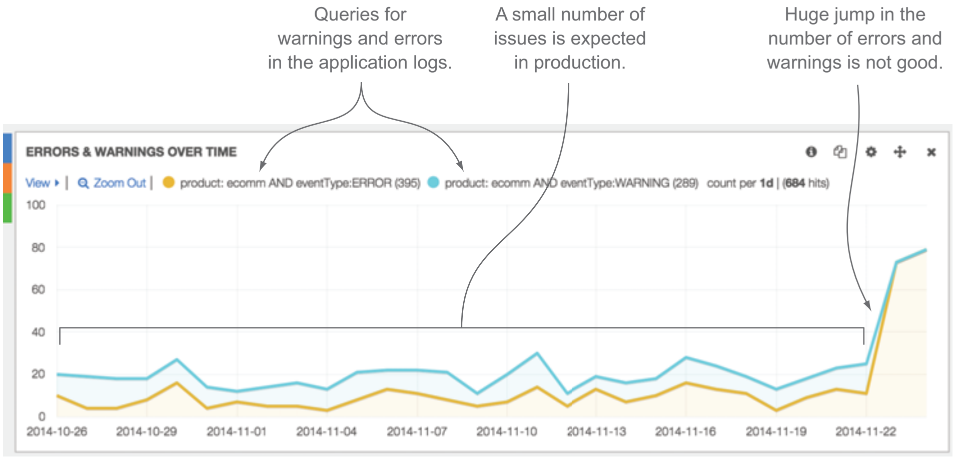


Figure 2.1 Blastamo's agile pipeline and the software it uses


Queries for warnings and errors in the application logs.

A small number of issues is expected in production.

Huge jump in the number of errors and warnings is not good.

Figure 2.2 Kibana visualizing log analysis for the case-study project. Error rates have headed off the charts for the last two days.

new software would provide a huge boost in productivity. The development teams were excited to use cool cutting-edge technology that they could develop against.

At first the integration seemed simple. The developers integrated the new system with their existing system, put the changes through testing, and got it out in production. After the launch the operations team noticed a significant uptick in errors in their production logs. The graph they saw looked something like the one in figure 2.2.

The operations team alerted the development team, which jumped in and started triaging the issue. The development team quickly noticed that the integration with the new system wasn't as straightforward as they thought. Several scenarios were causing significant issues with the application in the aftermath of the integration, which were going to need a heavy refactor of the code base to fix. The development team proposed to the leadership team that they spend time refactoring the core components of the application by creating interfaces around them, rebuilding the faulty parts, and replacing them to ensure the system was stable and reliable.

Leadership agreed that the refactor had to be done but didn't want it to adversely impact new feature development. Leadership kept a close eye on both efforts, asking the development team for data, data, and more data so that if additional problems arose, they could make the best decision possible.

2.3 Determining the right solution

Faced with the task of reflecting their accomplishments and progress with solid data, the development team started with two questions:

- How can we show the amount of work the team is doing?
- How can we show the type of work the team is doing?

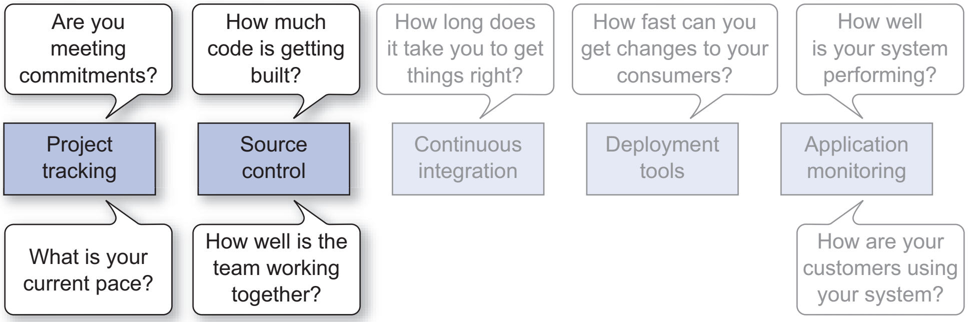

Are you meeting commitments?
How much code is getting built?
How long does it take you to get things right?
How fast can you get changes to your consumers?
How well is your system performing?
Project tracking
Source control
Continuous integration
Deployment tools
Application monitoring
What is your current pace?
How well is the team working together?
How are your customers using your system?

Figure 2.3 The type, amount, and target of work done by a team can be retrieved by querying your project-tracking and source-control management systems.

The leadership team gave the go-ahead to fix the issues with the new e-commerce components, but they wanted to get new features out at the same time. To point out progress on both tracks, the development team wanted to show what they were doing and how much work they were assigned.

The development team first needed to figure out where to get the data. If they could tap into the systems used for delivery, then they could probably get everything they were looking for. The team decided that the first two places they should focus on were project tracking and source control, as highlighted in figure 2.3.

The team decided to start looking at data aggregated weekly. While operating in two-week sprints, weekly data would show them where they were midsprint so they could adjust if necessary, and data at the end of the sprint would show them where they ended up. From their project tracking system they decided to capture the following data points:

- *Total done*—The total number of cards completed. This should show the volume of work.
- *Velocity*—Calculating the effort the tasks took over time. Before a sprint started, the team would estimate how long they thought their tasks would take and note those estimates on the cards that represented their tasks. The velocity of the team would be calculated every sprint by adding up the estimates that were ultimately completed by the end of the sprint.
- *Bugs*—If they were making progress on their refactor, then the number of bugs in the system should start to decrease.
- *Tags*—There were two types of high-level tasks: refactor work and new features. They used labels to tag each task with the type of work they were doing. This allowed them to show progress on each effort separately.
- *Recidivism rate*—The rate at which cards move backward in the process. If a task moves to done but it's not complete or has bugs, it's moved back, thus causing the recidivism rate to go up. If you use B to represent the number of times a

card moves backward and F to represent the number of times a card moves forward, then recidivism can be calculated with the formula *(B / (F + B)) * 100.*

NOTE The maximum recidivism rate for complete tasks would be 50%, which would mean that your cards moved forward as many times as they moved backward.

Those data points should give a pretty complete picture of the action in the project tracking system (PTS). The team also decided to get the following data point out of their SCM.

- *CLOC*—The total amount of change in the codebase itself.

The development team was already using ELK for log analysis, and the leadership team decided to use the same system for graphing their data. To do this they needed a way to get data from JIRA and GitHub into EC for indexing and searching.[1]

JIRA and GitHub have rich APIs that expose data. The development team decided to take advantage of those APIs with an in-house application that obtained the data they needed and sent it to EC. The high-level architecture is shown in figure 2.4.

NOTE A setup for EC and Kibana can be found in appendix A, and the code for the application that can get data from source systems and index it with EC can be found in appendix B.

Once the development team started collecting the data, they could use the Kibana dashboards they were familiar with from their log analysis to start digging into it.

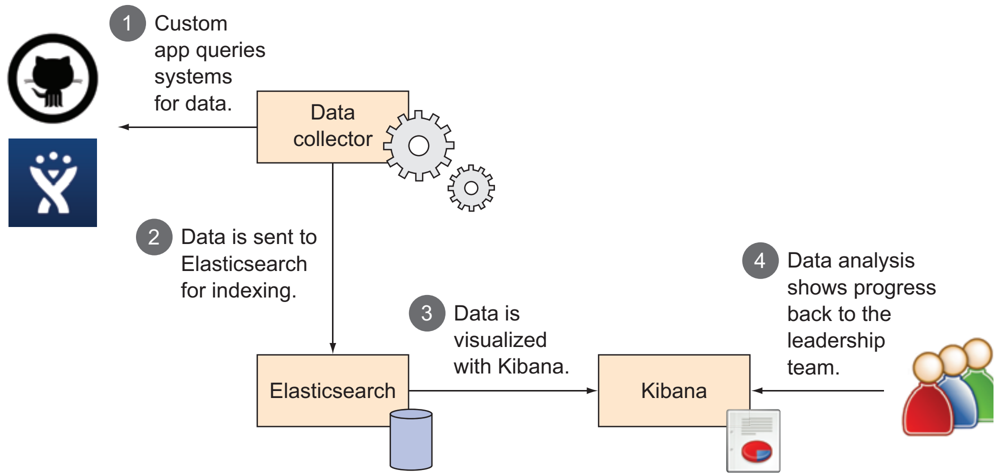

1 Custom app queries systems for data.
Data collector
2 Data is sent to Elasticsearch for indexing.
3 Data is visualized with Kibana.
4 Data analysis shows progress back to the leadership team.
Elasticsearch
Kibana

Figure 2.4 The component architecture for the system

[1] For more information on EC I recommend *Elasticsearch in Action* (Manning Publications, 2015), www.manning.com/hinman/.

They collected data for a sprint in the new operating model and went back a few sprints to get a baseline for analysis. Using this data they generated the charts shown in figures 2.5, 2.6, and 2.7.

The points are aggregated weekly for the graphs.

Bugs and recidivism were trending down towards the end of a sprint.

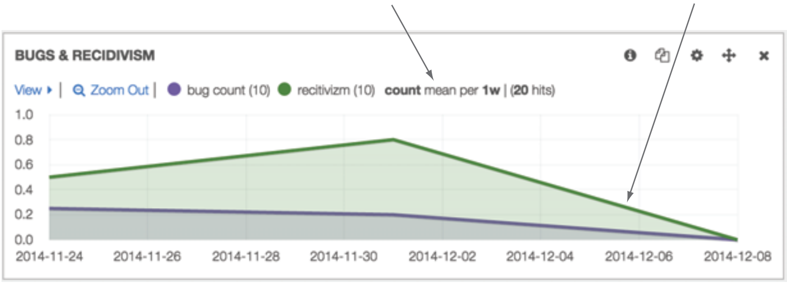


Figure 2.5 Showing recidivism and bugs in a single graph because both should trend in the same direction

Splitting the data by tags allows the velocity of each effort to be tracked separately.

After the split, velocity starts to drop just a bit.

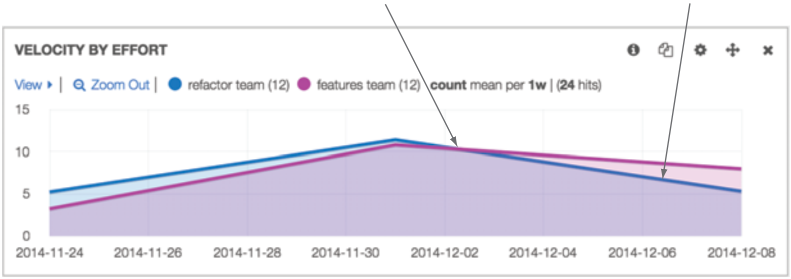


Figure 2.6 Breaking velocity out by effort after sorting data with tags

Y-axis is in thousands

The amount of code changes has been increasing a lot

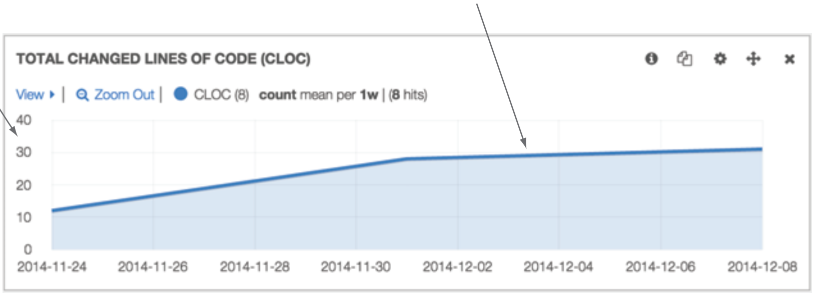


Figure 2.7 The chart graph showing how much code the team was changing over time. The Y-axis is in thousands to make it easier to read.

Now the development team was ready to start digesting the data to give leadership the objective breakdowns they wanted to see.

2.4 Analyzing and presenting the data

The first thing the development team noticed was that their velocity had dropped. That was probably to be expected because they had just refocused the team into a new working model. When they divided the efforts, they had a nice, even split in velocity between them, which was just what they wanted. If they could keep this up, they would be in great shape. But after the end of the next sprint they started seeing trends that weren't so good, as shown in figures 2.8, 2.9, and 2.10.

These are the key things the development team noticed:

- *Velocity was going down again.* This indicated that they were still being impacted by the split in work and weren't able to adjust.

Recidivism is creeping up and bugs are jumping sharply.

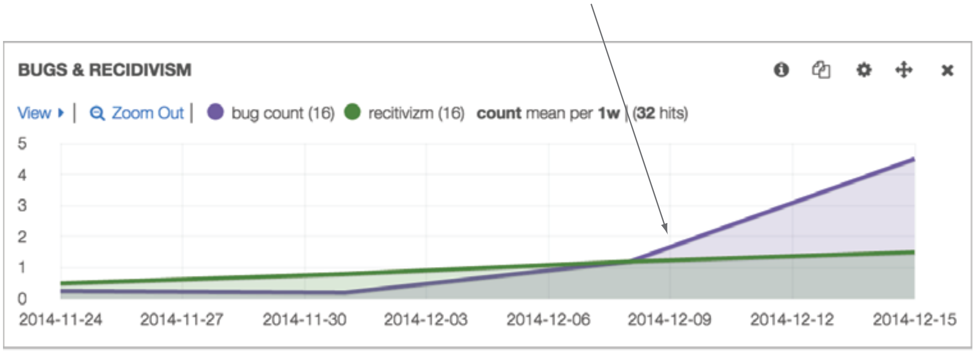


Figure 2.8 After another sprint recidivism has started creeping up and bugs are increasing sharply.

Velocity for both teams is starting to drop consistently.

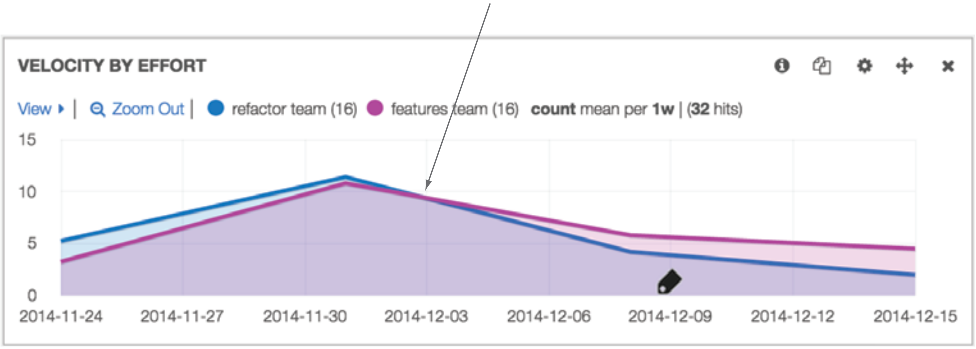


Figure 2.9 Velocity for both teams is starting to suffer.

Amount of code changed is increasing rapidly

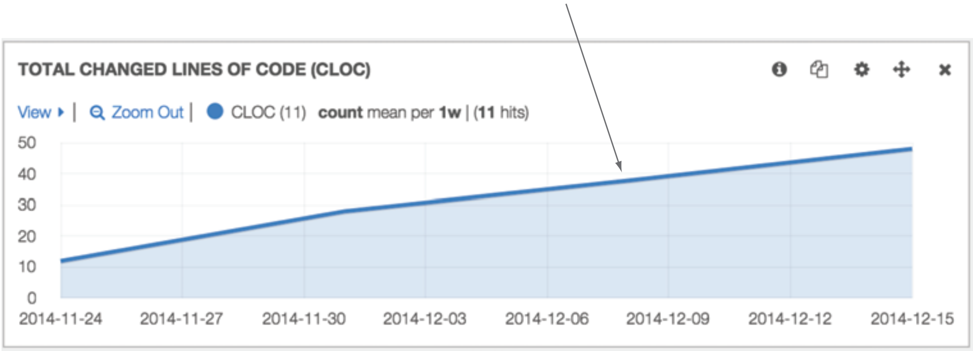


Figure 2.10 The amount of code changed across both teams is skyrocketing.

- *Velocity for new features dropped sharply*. This indicated that the work being done by the refactor was severely hurting the team's ability to deliver anything new.
- *CLOC was going up*. The team was changing a lot more code.
- *Bugs and recidivism were rising*. The overall product was suffering for the combination refactor and feature work.

According to this, the refactor was causing significant churn in the codebase and apparently leaving a host of new bugs in its wake. New feature development velocity had hit the floor because that work was being done on top of a shaky foundation.

The team needed to adjust to these findings. Team members got together and started reviewing the list of bugs to determine root cause and validate or disprove their hypotheses. They discovered that as the team dug into the refactor, changes they made in one part of the codebase were causing other parts to break. They had a general lack of unit tests across the integrated product that caused them to find bugs after the refactored code was delivered. This led to a Whac-A-Mole situation, where fixing one issue ended up creating at least one more to chase after.

2.4.1 Solving the problems

The team knew they had to make the following changes to get their project back on track:

- Ensure that they had enough automated testing across the application to prevent new bugs from popping up after code was refactored. This should help them find tangential problems as they come up and avoid the Whac-A-Mole effect.
- Stop building features for at least a sprint in order to either
  - Completely focus on the refactor to stabilize the foundation enough to build new features on.
  - Virtualize the back-end services so user-facing features could be developed against them while the refactor continued underneath it.

Service virtualization

Service virtualization is the practice of recreating the behavior of a system synthetically for isolated development and testing. Basically, you can emulate a dependency of a feature you need to build with virtualization, build and test the feature on top of the virtualized service, and deploy your new feature on top of the real service. This technique is particularly handy when consumer-facing features that need to iterate quickly depend on large back-end systems that can't change as fast.

According to the data they could see the following:

- Before the refactor the development team was averaging 12 points/developer/sprint (pds).
- The refactor team was still averaging 12 pds while changing a lot more code.
- The new-feature team was down to 4 pds.

These results led them to the following conclusions:

- Given their productivity was cut to one-third, if they could stop working for two sprints and get back to their average productivity on the third sprint, they would get the same amount of work done as if they worked at one-third productivity for three sprints.
- Based on their current velocity they could also infer that the refactor was slated to take another three sprints.

They decided that if they repurposed the entire new-feature team to tighten the team's automation for testing and virtualization for two sprints, they could easily get back to their average productivity and even ultimately become more productive by proactively finding bugs earlier in the development cycle. Now they had to convince the leadership team to let them stop working on features for two sprints.

2.4.2 *Visualizing the final product for leadership*

The development team had a strong argument to change how they were working and the data to back it up. But the charts that they found valuable would probably confuse the leadership team without a lengthy explanation on the context. The final step was to break down this data into metrics that indicated how things were going in an intuitive and concise way. To do this they decided to add panels to their dashboard that focused on the high-level data points the leadership team cared about.

To highlight their argument they broke their data down into the following points:

- The velocity of the entire team before the split
- The average drop in velocity per developer of the feature team during the refactor
- The increase in bug count every sprint after the split
- The difference in bug counts between well-tested code and poorly tested code

Once the development team had presented all of their data, they hit leadership with their proposal to drop feature development for two sprints in order to get back to their previous level of productivity. Given the current state of things, they had nothing to lose.

The leadership team gave them the go-ahead, and they all lived happily ever after.

Dashboards for the right audience

We'll get a lot deeper into creating dashboards and communicating across the organization in chapter 9 when we explore ways to visualize the same data for different audiences. In this example the team used static data for their presentation.

The development team was having success with their data visualization, so they decided to experiment to try to get an idea of the properties of their codebase without having intimate knowledge of it. As an experiment they also decided to show their codebase as a CodeFlower (www.redotheweb.com/CodeFlower/).

CodeFlowers for a new perspective

CodeFlowers are one example of visualizing your codebase in a very different way. We'll touch on them here and come back to different data-visualization and communication possibilities later in chapter 9.

First, the development team generated a CodeFlower on one of their newly built, modular, and well-tested web service projects. That generated the flower shown in figure 2.11.

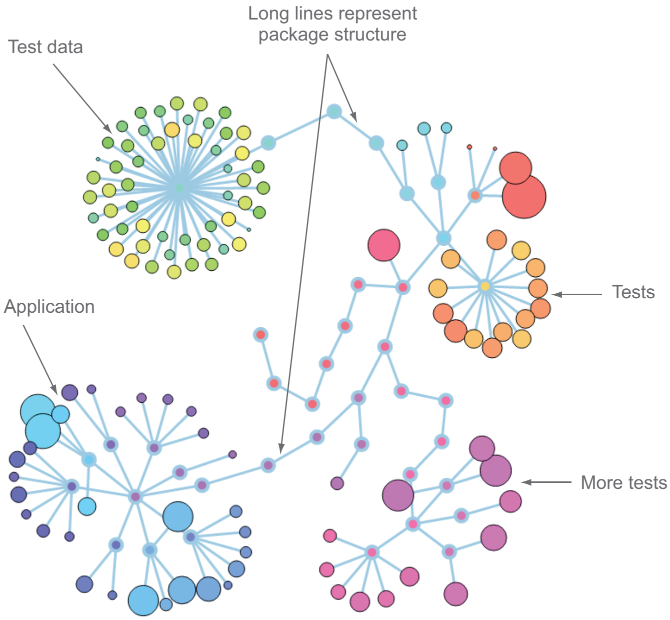


Figure 2.11 A CodeFlower of a clean, well-written codebase

Note a few interesting points from this codebase:

- The application itself has a clean package structure.
- There are as many tests as there is functional code.
- There is plenty of test data, which indicates that there are real tests for many scenarios.

Then they generated a CodeFlower on their older e-commerce application codebase. An excerpt is shown in figure 2.12.

Here are a few notes from this codebase:

- It is huge! That usually indicates it's difficult to maintain, deploy, and understand.
- There is a lot of XML and SQL, both difficult to maintain.
- There are tests, but not nearly as many tests as functional code.
- The package structure is complex and not well connected.

Of the two CodeFlower representations, which application would you rather have to modify and deploy?

The development team was now using data visualization in different ways to shape how they operated, and they were able to turn this into real-time data that the leadership team could peek at whenever they wanted.

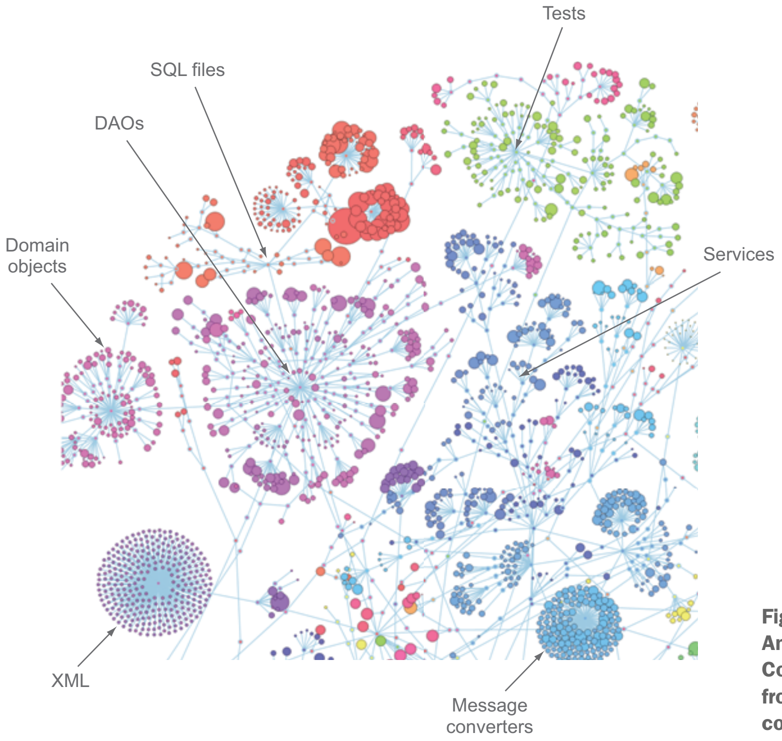


Figure 2.12 An excerpt from a CodeFlower generated from an extremely complex application

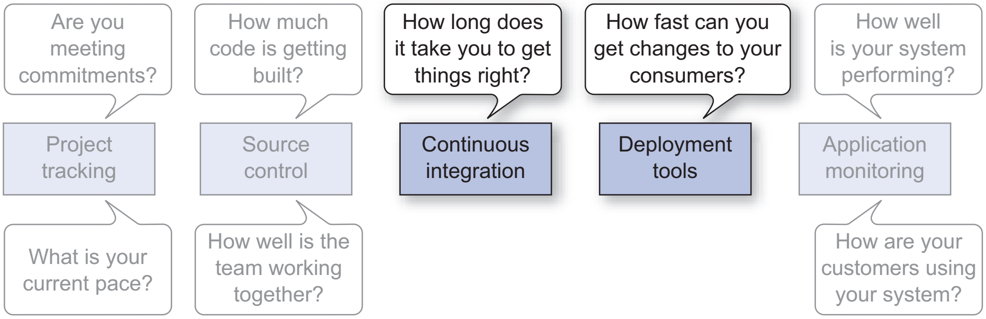

Are you meeting commitments?
How much code is getting built?
How long does it take you to get things right?
How fast can you get changes to your consumers?
How well is your system performing?
Project tracking
Source control
Continuous integration
Deployment tools
Application monitoring
What is your current pace?
How well is the team working together?
How are your customers using your system?

Figure 2.13 By adding data from the build and deployment system, the development team was able to add release dates to their analysis to track how changes they made to their process affected delivery time.

2.5 Building on the system and improving their processes

After showing how just a few pieces of data can give better insight into the application lifecycle, the development team decided to start pulling even more data into the system, expanding the set of reports it could generate. They were looking at how well their system was performing through their log analysis, and they were measuring team performance through their PTS and SCM systems. One of the main outcomes they wanted to effect by improving their velocity was to improve the time it took to deliver changes that improved the application to their consumers. To see how their deployment schedule correlated with their project tracking, they decided to add release data into the mix. Getting CI and deployment data was the next logical step, as highlighted in figure 2.13.

Because they used their CI system to release code, they could pull build information from their deploy job and add releases to their analysis. Upon doing that, their dashboard started to look like the one shown in figure 2.14.

The tag icons show releases, with info on mouse-over.

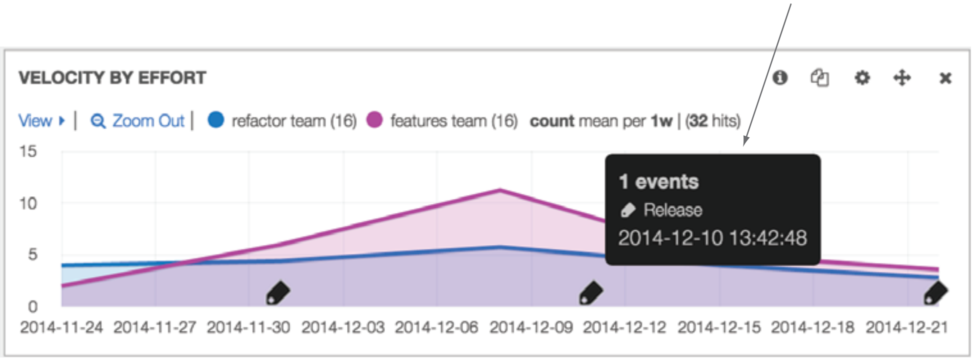


Figure 2.14 Adding releases to the dashboard. Now the team can see how their process affects the speed at which they release code.

The release data over time could show the development team how the changes in their processes affected their releases and, conversely if releases impacted the work they performed on every sprint. Now that they had a nice dashboard and were aggregating a lot of great data about how their team was performing, they started using it as a health monitor for the team.

2.5.1 Using data to improve what they do every day

Once they were collecting data and could display all of it, they wanted to ensure that every member of the development team was paying attention to it. They made sure everyone had the URL to the dashboard in their favorites, and they projected it on monitors when they weren't being used for meetings. They used the dashboards in their sprint retrospectives and frequently talked about which metrics would be affected by the proposed changes to the process.

They made it all the way around the feedback loop we talked about in chapter 1 and as shown in figure 2.15.

Now that the development team had made the loop once, the next question was how frequently they should go all the way around it. The data collection was automated so they had continuous data that they could react to at any time. But they settled on letting their sprints run their course and did their analysis and application before each new sprint started.

Before they were collecting and analyzing all of this data, they were only tracking velocity every sprint to make sure their completion rate stayed consistent. The better they got at mining their data, the more insight they obtained into how they were operating as a team and the better they became at making informed decisions on the best ways to operate. When problems came up, they looked at the data to figure out where

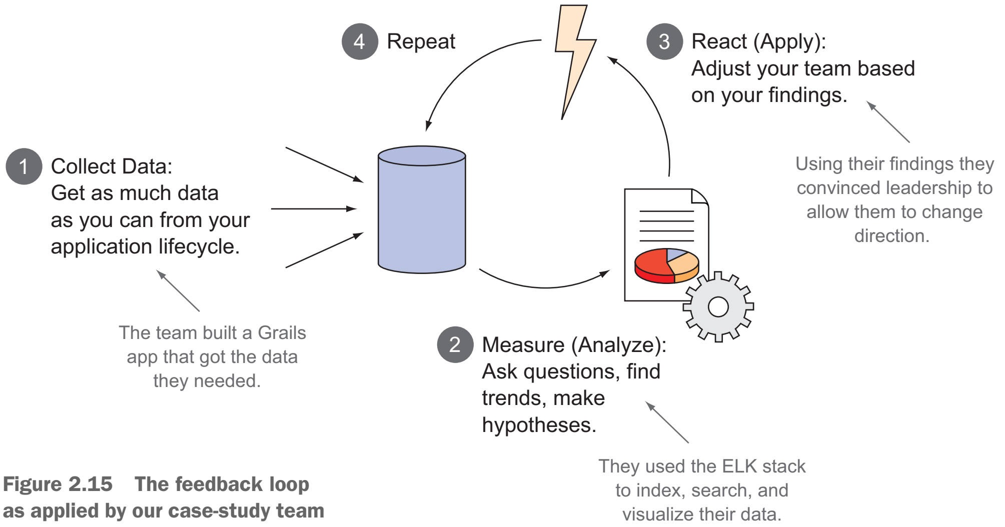

4 Repeat
3 React (Apply): Adjust your team based on your findings.
Using their findings they convinced leadership to allow them to change direction.
1 Collect Data: Get as much data as you can from your application lifecycle.
The team built a Grails app that got the data they needed.
2 Measure (Analyze): Ask questions, find trends, make hypotheses.
They used the ELK stack to index, search, and visualize their data.

Figure 2.15 The feedback loop as applied by our case-study team

the problem really was instead of making snap decisions on where they *thought* the problem was. Finally, when they wanted to make recommendations to leadership, they knew how to communicate their thoughts up the chain in a meaningful way that could effect change.

A positive side effect to this was that they felt free to experiment when it came to their development process. Because they now had a system that allowed them to measure changes as they made them, they started a new practice in which the team would adopt any changes a member suggested, as long as they had could measure their hypotheses.

2.6 Summary

In this chapter we followed a team through the implementation of an agile performance metrics system and demonstrated how they used the system. In this extended case study the development team used the following techniques that you can also use to start measuring and improving your team:

- Start with the data you have.
- Build on what you have using the frameworks you're familiar with.
- Frame data collection around questions you want to answer.
- When you find negative trends, determine solutions that have measureable outcomes.
- Visualize your data for communication at the right levels:
- The more detail an execution team has, the better equipped they will be to understand and react to trends.
- People outside the team (like leadership teams) typically need rollup data or creative visualizations so they don't require a history lesson to understand what they are seeing.

Part 2

Collecting and analyzing your team's data

In part 1 you learned the concepts of creating and using agile metrics, discovered where to get the data, and observed a team that put the metrics into practice. In part 2 you'll learn about the details of each data source in the development lifecycle, what each of these data sources can tell you, and how to start collecting and visualizing them. In each chapter you'll learn how to maximize the use of these systems to provide as much data as possible to gain insight into your team as well as what metrics you can get from each system alone.

Chapter 3 starts with the most common place to collect data about your team, your project tracking system (PTS). We'll look at different task types, estimations, and workflow metrics.

Chapter 4 shows you the data you can get from your source control management (SCM) system. We'll look at centralized versus decentralized systems, workflows to use to maximize the data you can get from your systems, and key metrics.

Chapter 5 shows you the data you can get from your continuous integration (CI) and deployment systems. We'll look at CI, deployments, and automated test results to analyze the different data you can get from various stages in your build cycle.

Chapter 6 shows you the data you can get from your application performance monitoring (APM) system. We'll look at what your production system can tell you through different types of data and instrumentation that can give you better insight into your systems.

Each chapter ends in a case study so you can see how the techniques from the chapter can be applied in a real-world scenario.

3 Trends and data from project-tracking systems

This chapter covers

- What information raw data from a project tracking system conveys
- How to utilize your PTS to collect the right data to give insight into your process
- How to get data from your PTS into your metrics-collection system
- Trends you can learn from data in your PTS
- What you're missing by relying only on project-tracking data

Perhaps the most obvious place to find data that points to your team's performance is your PTS. Common PTSs include JIRA, Trello, Rally, Axosoft On Time Scrum, and Telerik TeamPulse. This is where tasks are defined and assigned, bugs are entered and commented on, and time is associated with estimates and real work. Essentially your project's PTS is the intersection between time and the human-readable definition of your work. Figure 3.1 shows where this system lives in the scope of your application delivery lifecycle.

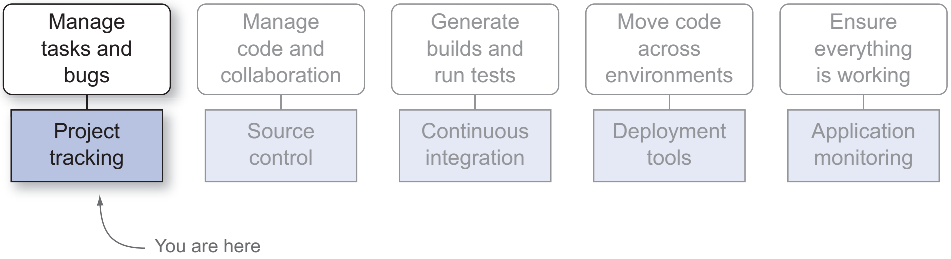


Figure 3.1 The first data source in your application lifecycle is your PTS. This is where tasks are defined and assigned, bugs are managed, and time is associated with tasks.

Trends that can be found in your PTS can go a long way in showing you how your team is performing. It's typically the only system that teams will use to track their progress, so it's a good place for us to start.

Because your PTS is tracking time and tasks, you can use it to answer the following questions:

- How well does your team understand the project?
- How fast is the team moving?
- How consistent is the team in completing work?

But some of these still end up leaving a lot to interpretation and could be clarified with additional data. Here are a few examples:

- *How hard is the team working?* Although you can partially get this information from your PTS, you should supplement it with source-control data to get a better picture.
- *Who are the top performers on the team?* Data from your PTS is only a part of the picture here; you should combine this with source-control and potentially production-monitoring data.
- *Is your software built well?* The question of quality comes mostly from other sources of data like test results, monitoring metrics, and source control, but PTS data does support these questions by showing how efficient a team is at creating quality products.

To enable you to generate the most meaningful possible analyses from your data, first I'll lay out guidelines to help you collect quality data to work with.

A word on estimates

In this chapter and throughout this book you'll notice that I talk a lot about estimating work. For the agile team, estimating work goes beyond trying to nail down better predictability; it also encourages the team to understand the requirements in greater depth, think harder about what they're building before they start, and have a greater investment in what they're building by committing to a timeframe with their peers. You'll notice that estimations are a part of the case studies at the end of several chapters and are used in combination with other data to see a broader picture of your team's operation. I encourage all of my development teams to estimate their work for all of these reasons, and I encourage you to consider estimating work if you're not doing it already.

3.1 Typical agile measurements using PTS data

Burn down and velocity are the two most commonly used metrics to track agile teams and both are derived from estimations. Burn down is the amount of work done over time, which can be derived by plotting the total number of incomplete estimates or tasks over time next to the average number of complete estimates or tasks over time. With burn down you typically have a time box that everything is supposed to be done within, and you can see how close you are to completing all the tasks you've committed to. If you simply toe the line and use your PTS out of the box, you'll get burn down and velocity as standard agile reports.

3.1.1 Burn down

An example burn down chart is shown in figure 3.2. In the figure the guideline represents the ideal scenario where your estimates burn down over time; the remaining values represent the actual number of tasks that are not closed in the current timeline.

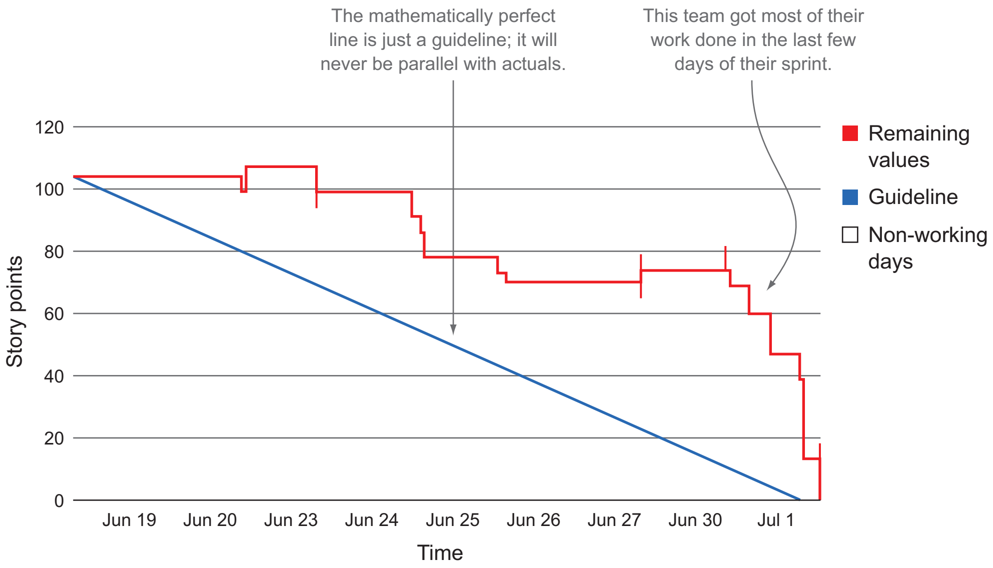


Figure 3.2 An example burn down chart

Burn down is very popular, but its value is limited due to the empirical nature of agile development. You will never have a smooth curve, and even though it's used as a guideline, its value is overestimated. So much so in fact that many Scrum coaches are deprecating the use of the burn down chart, and its use on agile teams is starting to diminish. Although it does provide a guideline to where you are against your commitment for a time period, burn down alone can't give you the whole story.

3.1.2 Velocity

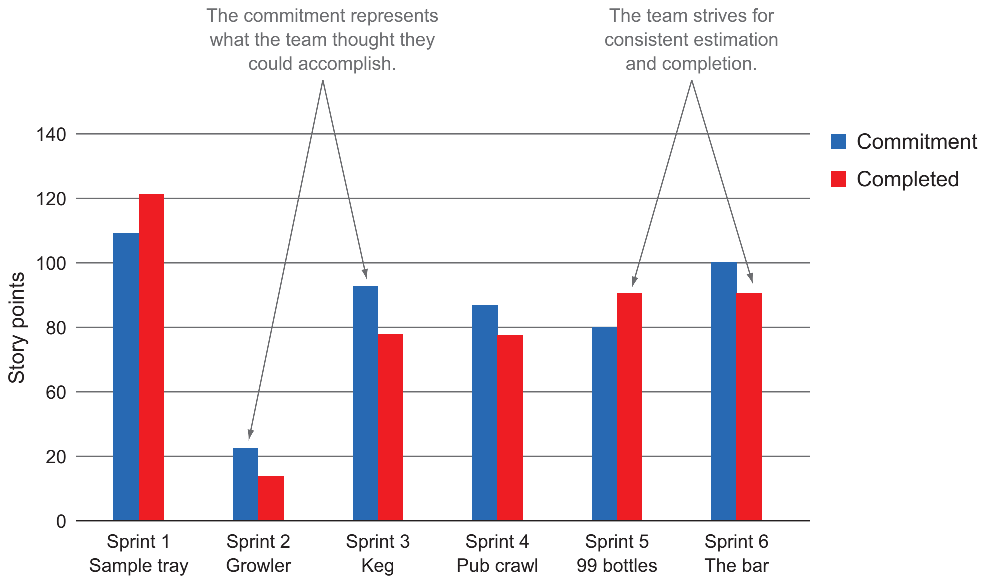
Velocity is a relative measurement that tracks the consistency of a team's completed estimates over time. The idea with velocity is that a team should be able to consistently complete their work as they define it. An example velocity chart is shown in figure 3.3.

Velocity can be calculated by graphing the amount you estimated against the amount you actually got done. Velocity ends up implying a few things about your team:

- How good your team is at estimating work
- How consistently your team gets work done
- How consistently your team can make and meet commitments

Even though velocity is a staple agile metric, its relative nature makes it difficult to pinpoint where problems really are. If velocity is inconsistent, there could be a number of issues, such as:

- Your team is not doing a good job at estimating.
- Your team has commitment issues.


The commitment represents what the team thought they could accomplish.

The team strives for consistent estimation and completion.

Figure 3.3 Velocity example

- Scope could be changing in the middle of your development cycle.
- You may be hitting troublesome tech debt.

The key data point behind velocity that we'll be working with is the estimates your team puts on their tasks. We'll be diving into estimations later in the chapter and breaking them down against other data points in chapter 7.

3.1.3 Cumulative flow

Cumulative flow shows how much work aggregated by type is allocated to your team over time. The cumulative flow diagram is typically used for identifying bottlenecks in the process by visually representing when a type of task is increasing faster than the others. In figure 3.4 you see an example cumulative flow diagram where a team is completing hardly any work and suddenly their to-do list jumps off the charts.

As with other charts, this tells only a part of the picture. It's possible that tasks were simply too big and the team started breaking them down into smaller chunks. It's also possible that this team is struggling to complete work and they end up getting more work piled on as an unrealistic task.

Cumulative flow can be useful when you use the correct task types for your work. If you have tasks for integration and you see that part of your graph starts to get fat, you know there's a bottleneck somewhere in your integration.

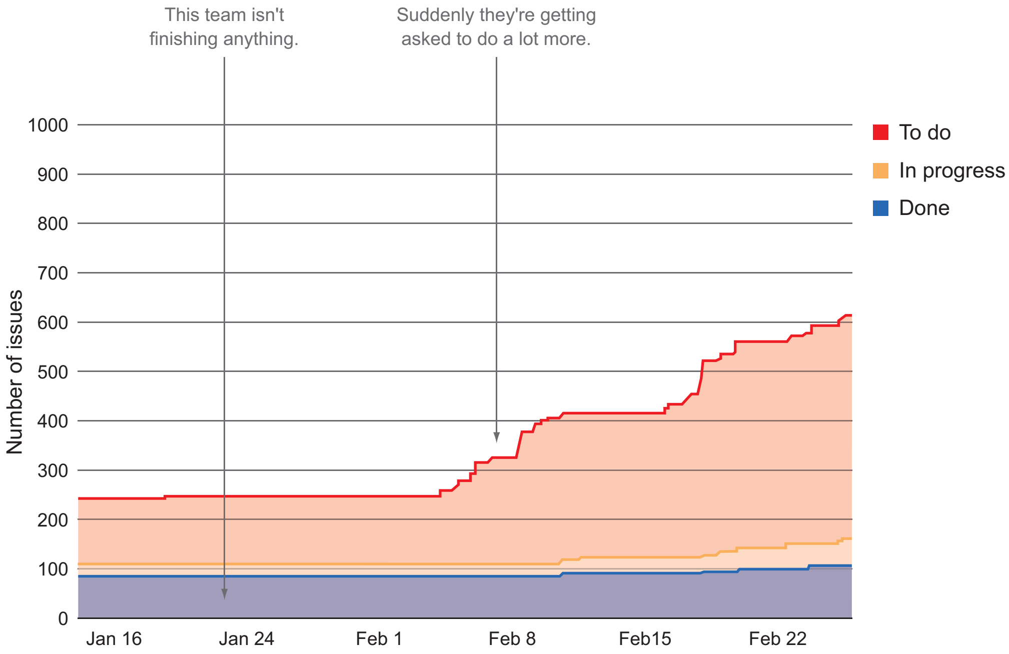


Figure 3.4 An example cumulative flow diagram showing a team getting asked to do a lot more than they have been able to accomplish historically

3.1.4 Lead time

Lead time is the amount of time between when a task is started and when it is completed. Teams that practice Kanban focus on throughput, and as a result they usually put a heavy emphasis on lead time; the faster they can move tasks through the pipes, the better throughput they can achieve. Some Scrum teams use lead time as well as velocity because the two metrics tell you different things; velocity tells you how much your team can get done in relation to what they think they can get done, and lead time tells you how long it takes to get things done.

The problem with lead time alone is that often teams don't pay close attention to what composes it. Lead time includes creating a task, defining it, working on it, testing it, and releasing it. If you can't decompose it, then it's hard to figure out how to improve it. We'll explore decomposing lead time in later chapters.

For teams that are simply receiving small tickets and pushing them through the system, lead time is a good indicator of how efficient the team can be. But if you have a team that has ownership of the product they're building and is vested in the design of that product, you usually don't have such a simple delivery mechanism. In fact, with the state of automation you could even argue that if your tasks are so small that you don't have to estimate them, then maybe you don't need a human to do them.

Lead time becomes a very valuable metric if you're practicing CD and can measure all of the parts of your delivery process to find efficiencies. It's a great high-level efficiency indicator of the overall process your team follows.

3.1.5 Bug counts

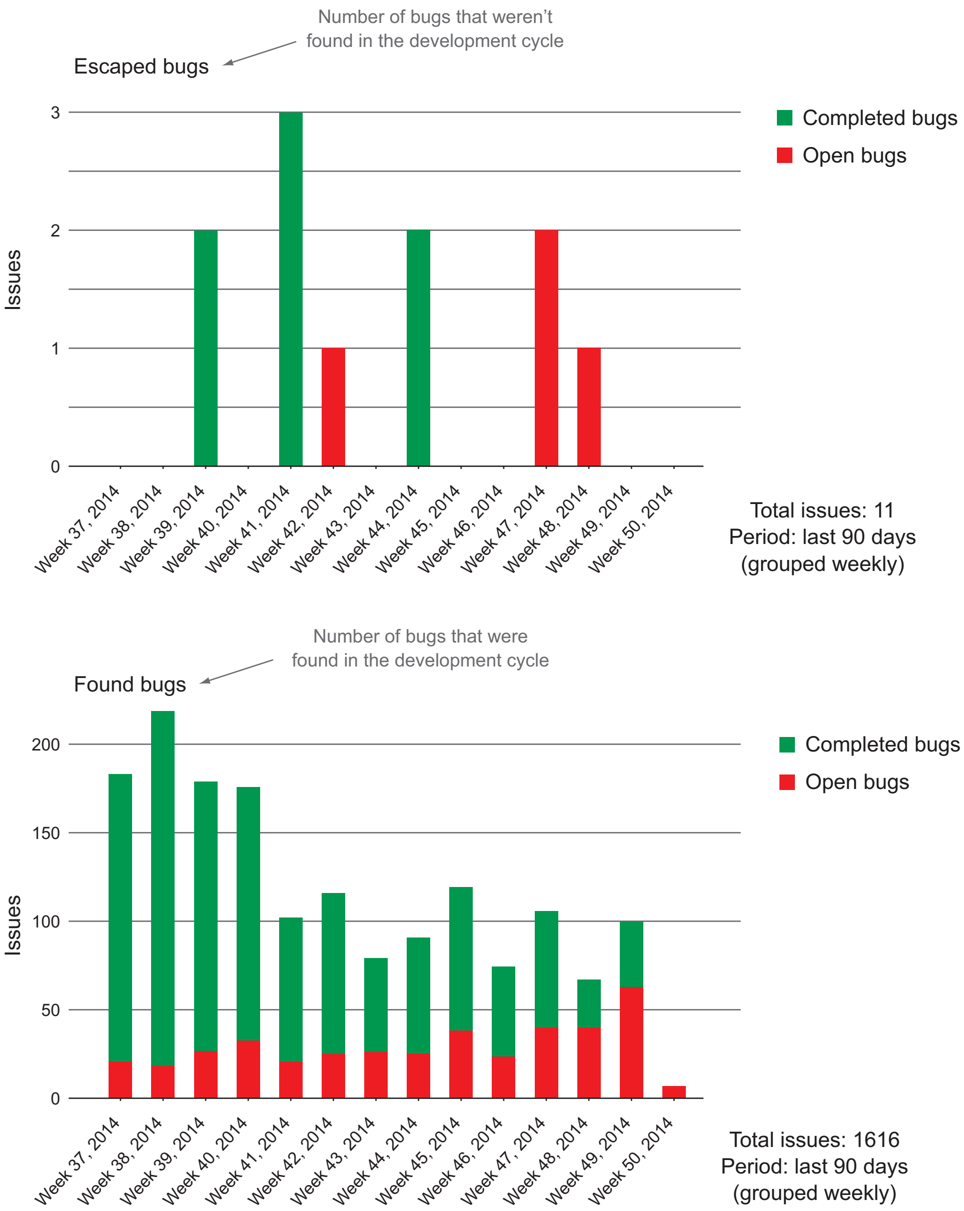
Bugs represent inconsistencies in your software. Different teams will have different definitions of what a bug is, ranging from a variation from your application's spec to unexpected behaviors that get noticed after development is complete.

Bugs will pop up at different times depending on how your team works together. For example, a team with embedded quality engineers may not generate bugs before a feature is complete because they're working side by side with feature engineers and they work out potential problems before they're published. A team with an offshore QA team will end up generating tickets for bugs because testing is staggered with feature development.

Figure 3.5 shows two different charts that track bugs over time. Because bugs represent things you don't want your software to do, teams strive for low bug counts as an indicator of good software quality.

Bugs can also be broken down by severity. A noncritical bug might be that you don't have the right font on the footer of a website. A critical or blocker bug may be that your application crashes when your customers log into your application. You might be willing to release software that has a high bug count if all the bugs are of a low severity.

The inconsistency with bugs is that *severity* is also a relative term. To a designer the wrong font in the footer may be a critical bug, but to a developer concerned with



Number of bugs that weren't found in the development cycle

Total issues: 11
Period: last 90 days
(grouped weekly)

Number of bugs that were found in the development cycle

Total issues: 1616
Period: last 90 days
(grouped weekly)

Figure 3.5 Two charts showing bug counts and at what point they were found

functionality, maybe it shouldn't even be a bug in the first place. In this case, severity is in the eye of the beholder.

The charts and graphs you've seen up to this point are all useful in their own right even though they measure relative data. To go beyond the standard PTS graphs there are some best practices you can follow to ensure you have data that's enriched for analysis and ultimately for combination with other data sources.

3.2 Prepare for analysis; generate the richest set of data you can

As you collect data, keep in mind that your analysis will only be as good as the data you collect; how your team uses your PTS will translate into what you can glean from the data in the system. To set yourself and your team up for success by getting the most meaningful analysis from this data, this section offers tips on working with your PTS to help ensure you're generating as much useful data as possible from it.

Most APIs will give you something like a list of work items based on the query you make, and each work item will have many pieces of data attached to it. The data you get back will give you information such as:

- Who is working on what
- When things get done in a good way

Those two statements are general and loaded with hidden implications. We break them down in figure 3.6.

You should look for the following pieces of data in your tasks so you can index them, search them, and aggregate them for better analysis:

- User data.
- Who worked on the task?
- Who assigned the task?
- Who defined the task?
- Time data.
- When was the task created?
- When was it complete?
- What was the original estimate of the task?

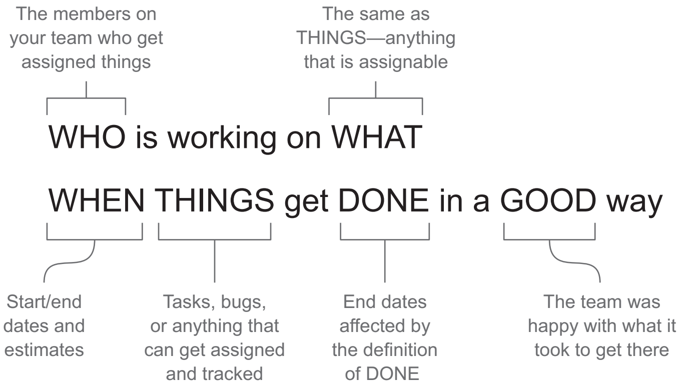

The members on your team who get assigned things
The same as THINGS—anything that is assignable
WHO is working on WHAT
WHEN THINGS get DONE in a GOOD way
Start/end dates and estimates
Tasks, bugs, or anything that can get assigned and tracked
End dates affected by the definition of DONE
The team was happy with what it took to get there

Figure 3.6 Parsing the loaded words that tell you how PTS is used

Who, what, and when are the building blocks that we're going to focus on in this chapter. Some raw response data is demonstrated in the following listing.

Listing 3.1 Excerpts from the raw data in a typical API response

```
{
    "issues": [
        {
            "key": "AAA-3888",
            "fields": {
                "created": "2012-01-19T14:50:03.000+0000",
                "creator": {
                    "name": "jsmit1",
                    "emailAddress": "Joseph.Smith@blastamo.com",
                    "displayName": "Smith, Joseph",
                },
                "estimate": 8,
                "assignee": {
                    "name": "jsmit1",
                    "emailAddress": "Joseph.Smith@nike.com",
                    "displayName": "Smith, Joseph",
                },
                "issuetype": {
                    "name": “Task”,
                    "subtask": false
                },
…
```

What

When

Who

Who

What

3.2.1 Tip 1: Make sure everyone uses your PTS

This first tip may seem obvious if you want to ensure you can get data from your system: make your PTS your one source of truth. Even though many teams use web-based project tracking, you'll still often see sticky notes on walls, email communication around issues, spreadsheets for tracking tasks and test cases, and other satellite systems that help you manage your work. All of these other systems generate noise and confusion and detract from your ability to get data that you can work with. If you've invested in a PTS, use it as the single source of truth for your development teams if you really want to mine data around how your team is functioning. Otherwise, the effort of collecting all the data from disparate sources becomes a tedious, manual, and time-consuming effort. It's best to spend your time analyzing data rather than collecting it.

Using your PTS is especially relevant when you have multiple teams that aren't colocated. Agile tends to work best with teams that are working together in the same place; it makes communication and collaboration much easier and allows everyone to contribute during planning, estimation, and retrospective meetings. But it's not always possible to have teams in the same place, and many companies have development teams spread around the world. If this is the case, try to ensure that everyone is using the same system and using it in the same way.

To be able to track everything your team is doing, you should have a task for everything. That doesn't mean you should require cards for bathroom breaks, but you do need to identify meaningful tasks and ensure you have the ability to create tasks in your system for them.

When creating a task, the most important thing to think about is that it should never take more than a few days to accomplish. If you find your team estimating a task at more than a few days of work, that typically means they don't understand it enough to put real time on it or it's too big to be considered a single task. I've seen that when teams estimate their work to take more than a few days, those tasks tend to be underestimated more than 50% of the time.

Whatever you're working on needs to be broken down into small chunks so that you can track the completion of the work in a meaningful way and so that the development team understands what it will take to move tasks over the line.

3.2.2 Tip 2: Tag tasks with as much data as possible

Task types should be pretty simple; you have a task either for new work or for fixing a bug in something that's already done. Getting any fancier than that is a great way to confuse your team, so it's best to keep it simple. Tagging is a great way to add data that you may want to use for analysis later on that isn't part of the default data of a task. By default every task should have a description, start and end dates, the person to whom it's assigned, an estimate and the project to which it belongs. Other potential, useful data for which you may have no fields, include a campaign with a feature, team name, product names, and target release. Just as you can create hash tags for anything you can think of #justfortheheckofit on multiple social media platforms, you can do the same in your PTS.

The alternative to adding tags in most systems is creating custom fields. This is the equivalent of adding columns to a flat database table; the more fields you add, the more unwieldy it becomes to work with. Teams are forced into using the structure that exists, which may not translate into how they work. When new people join the team or when new efforts spin up, it becomes cumbersome to add all the data required in the system just to create tasks. This is visualized in figure 3.7.

Adding data in the form of tags allows you to add the data you need when you need it and allows you to return to tasks once they're complete to add any kind of data you want to report on. This is valuable because it helps you figure out what works well

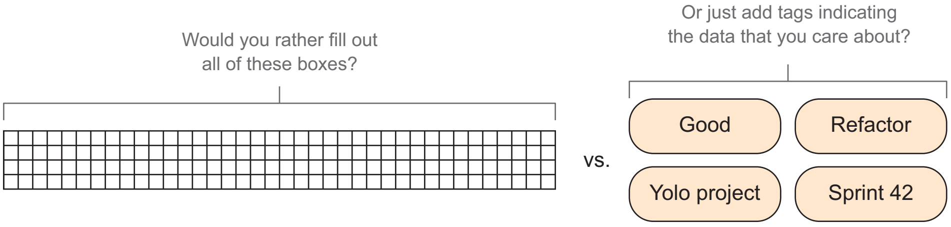

Would you rather fill out all of these boxes?
vs.
Or just add tags indicating the data that you care about?
Good
Refactor
Yolo project
Sprint 42

Figure 3.7 The difference between organizing flat project data and adding relevant metadata with tags

for your team even if it's contrary to normal trends. For example, the number of comments on a ticket could mean that your team is communicating well, but it could also mean that the original requirements were very poor and they were flushed out through comments on the card. By going back and marking cards as "worked well" or "train wreck," for example, you can find trends that may not have been visible before.

In addition to the other good reasons for using tags, this method of organizing your tasks allows you to sort, aggregate, or map-reduce your data much easier. Instead of modifying queries and domain objects to get good analysis, you can simply add more tags, and you'll be able to see clear trends in your data, as you'll see later in this chapter.

Some PTSs like JIRA support tags or labels and have explicit fields for them. In JIRA the data comes back in a format that's nice and easy to work, as shown in the next listing.

Listing 3.2 Example labels JSON block in a JIRA API response

```
labels:
[
    "Cassandra",
    "Couchbase",
    "TDM"
],
```

Each element represents a separate tag

Other PTSs may not have this feature, but that doesn't have to stop you. You can get into the habit of hashtagging your cards to ensure they can be grouped and analyzed later on. You may end up getting data like that shown in the following listing.

Listing 3.3 Example comments JSON block from an API call with hashtags in the text

```
comments: [
    {
        body: "To figure out the best way to test this mobile app we should
     first figure out if we're using calabash or robotium. #automation
     #calabash
     #poc #notdefined"
    }
]
```

Hashtags used to note details for later analysis

If this is the data you have, that's fine. This may not be as simple to parse as the previous listing, but there are hash tags that you can parse and reduce into meaningful data that can be used later.

3.2.3 Tip 3: Estimate how long you think your tasks will take

Estimating how long tasks will take can be pretty tricky in an agile world. It can be tough to figure out how long something can take when definitions are often in flux and products are constantly changing. Estimates are important from a project management perspective because they will eventually allow you to predict how much work you can get done in a certain amount of time. They also show you if your team really understands the work that they're doing.

One thing that seems to always challenge teams is what estimations mean. Story points are supposed to relate to effort, yet every team I've ever been on ends up translating those points to time, so in most cases I find it easier for teams to use a time-based estimation. There are a plethora of blogs, articles, and chapters on the association of time and story points, but if there's one thing I've learned it's this: embrace whatever comes most naturally to your team. If your team estimates in time, then use days or half-days as story points. If they like the concept of effort, then go with it. If you try to fight whatever people understand as estimates, you'll end up with numbers whose relativity constantly changes as you try to change their association of time to estimations. By embracing what comes naturally to your team, you may not end up with a textbook way of estimating your tasks, but you will end up with more reliable data over time.

For teams I've managed it normally takes three to four iterations before they can start to trust their estimations. The best way to see if your estimations are on track is to look at them over time next to the total number of tasks that you complete over time.

Even if you get to a point where your estimations and completed work are consistent, using estimations alone can't tell you if you are under- or overestimating. Overestimating work means your team is underachieving for some reason; underestimating typically leads to team burnout and an unsustainable pace of development. The tricky thing here is that estimations alone don't give you that insight into your team; you have to combine estimations with other data to see how well your team is estimating. Figure 3.8 may look great to you if you're trying to hit a consistent goal over time.

But how do you know if your estimates are any good? All you know is that you're getting done what you thought you could get done, not if you did as much as you could or if you were doing too much.

Flat-line trends occur when you see the same value over and over so your chart is just a flat line. Your team might be overestimating if you see a flat-line trend in your estimates over time, or they might be overestimating if you get done a lot more than

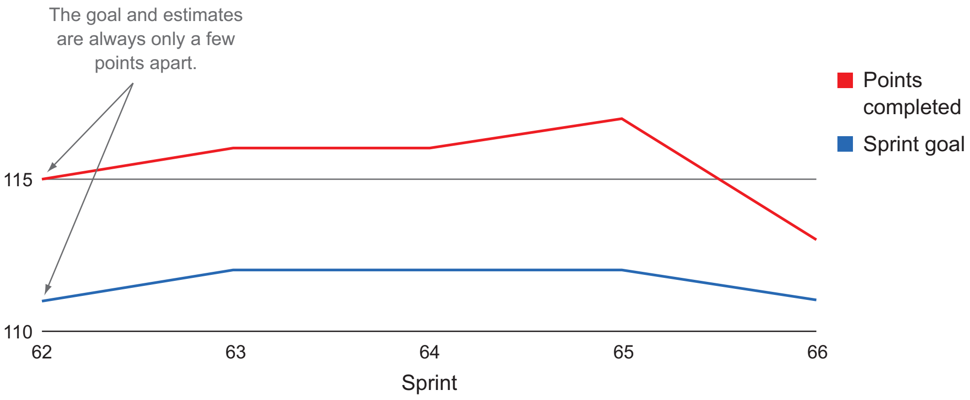


Figure 3.8 Estimates are spot on every time!

you commit to over time. At a glance this may not seem so bad; after all, you're either being consistent or you're beating your commitments—project managers of the world rejoice! But you don't want your team to get bored, and you want to be as efficient and productive as possible; overestimating equals underachieving. As I continue, we'll add more data to help you spot potential overestimating.

It's also possible to see the same flat-line trend if your team is overestimating. If you have a team that is overestimating and you get the chart shown in figure 3.8, it could mean that your team is pushing too hard to get their work done, which is typically unsustainable. Once the trend breaks, it will result in much less work completed or people leaving the team in frustration. In this case relying on estimation to tell the whole picture could mean devastation is looming even though the trends look good.

As you can see, estimates alone can give you an idea of how your team is performing, but they can also hide things that you need your attention. Next, we'll add volume and bug count to our estimates to show how you can better identify how your team is doing.

3.2.4 Tip 4: Clearly define when tasks are done

Done means a task is complete. Despite the seemingly simple definition, *done* can be a tough thing for the business and the agile team to figure out and agree on. Because the data in your PTS associates time with tasks, the criteria for which tasks are complete are key because they determine when to put an end time on a task. Often, done means that the task in question is ready to deploy to your consumers. If you group lots of changes into deployments and have a separate process for delivering code, then measuring the time it takes to get ready for a deployment cycle makes sense. If you move to a model where you're continuously deploying changes to your consumers, it's better to categorize tasks as done after they've been deployed. Whatever your deployment methodologies are, here are some things to consider when defining done for your team's tasks:

- Whatever was originally asked for is working.
- Automated tests exist for whatever you built.
- All tests across the system you're developing against pass.
- A certain number of consumers are using the change.
- You are measuring the business value of the change.

Make sure you collaborate with your team and stakeholders on the definition of done for a task, and keep it simple to ensure that everyone understands it. Without a clear understanding and agreement, you're setting yourself up for unhappiness down the line when people don't agree when something is complete and you don't have data that you can trust.

Once a task is marked as done it becomes history, and any additional work should become a new task. It's often tempting to reopen tasks after they've been completed if the requirements change or a bug is found in a different development cycle. It's much better to leave the task as complete and open another task or bug fix. If you have a

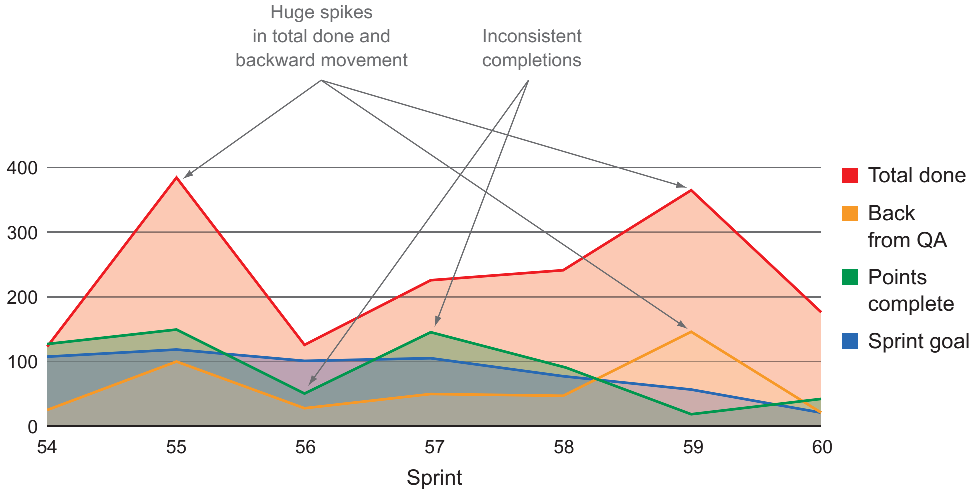


Figure 3.9 Bad trends that you can see when completion criteria are not well defined

good agreement on what it means to be done and a task meets those criteria, there really is no good reason to make it undone.

If you don't have a solid agreement on what it means to complete your tasks and your team doesn't mark tasks as done in a consistent way, then you won't be able to trust most of your data. This is because a lot of what you want to know from your PTS centers on how long tasks take to complete versus how long you thought they were going to take. If you don't have an accurate way to determine when things are complete, it's hard to know with any certainty how long tasks actually take.

You can tell when your completion criterion is not well defined when you start to see unusual spikes in volume and backward movement, as shown in figure 3.9.

This happens for either of the following reasons:

- Tasks that were complete end up back in the work stream.
- Tasks aren't validated and are sent back to the development team for further work.

Another trend you can see from figure 3.9 is that the completions of the team jump up and down. With unplanned work making its way back to the team, it becomes tough to complete the original commitment. In this case you can see the sprint goal start to drop more and more over time as the team tries to get to a consistent velocity. In reality they need to figure out why things are moving back from QA and why the number of tasks entering the work stream is spiking every few sprints.

3.2.5 Tip 5: Clearly define when tasks are completed in a good way

Good and *bad* are the epitome of relative terms. Each team has a different idea of what they think is good based on their history, the product they're working on, how they're implementing agile, and the opinions of its members. Go through your list of completed tasks and start tagging your cards as good, bad, or in between. As demonstrated time and again, different patterns mean different things based on the team; absolute patterns simply don't exist in a relative world. By tagging your cards based on how well you think the process worked in addition to adding other tags that describe the task, you can identify patterns that are contributing to the performance of the team. As you'll see as you start working with your data, tags will become the metadata around your work; the power of working with data in this way is that you can find trends that tell you how you're working now and help you identify what needs to be tweaked in order to improve.

This relates directly to section 3.2.2 when we talked about tagging tasks and working with that data later. If you map tasks as good or bad and track that against volume, you'll end up with a pretty good indicator of how happy your team is. In this case *good* means that the task worked well, there were no problems around it, and everyone is happy with how it played out. *Bad* would mean that the team was not happy with the task for some reason; examples could be that they weren't happy with how the requirements were written, there was another problem that made this task more difficult to complete, or the estimation was way off. In other environments a Niko-niko Calendar may be used for this, where every day team members put an emoticon representing how happy they are on the wall. Mapping out how satisfied you are with the completion of individual tasks is a much easier and more subtle way to figure out how the team is feeling about what they're working on. It can also help you gain insight into your estimate trends. As we mentioned earlier, flat-line estimates may make project managers happy, but that doesn't mean everything is going well on the team. Figure 3.10 is an example of a team that is flat-lining estimates and completion and is also tagging cards as "happy" or "sad" based on how well the developer thought it was done.

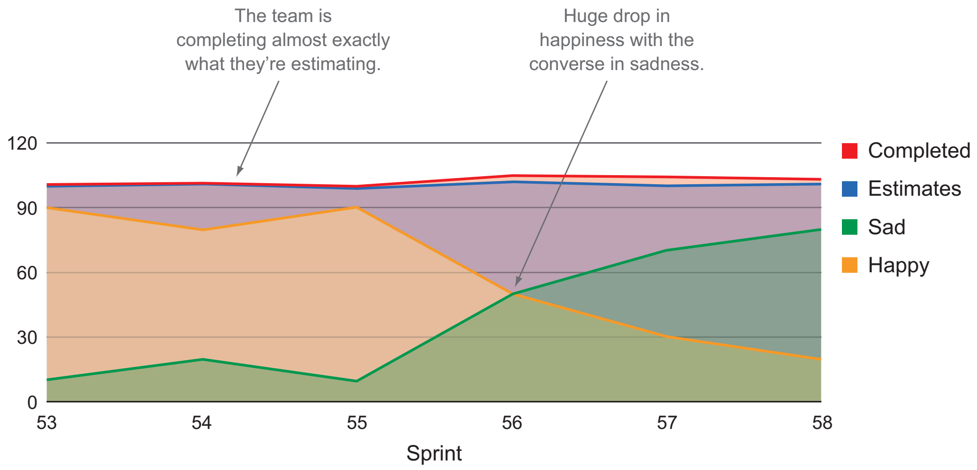


Figure 3.10 Estimates and completion mapped with how the developer thought the card was completed

I've seen the chart shown in figure 3.10 on tight deadline projects where a team needed to get an almost unrealistic amount of work done. They started off very happy because they were finishing more than they thought they could and the thrill of it made them feel good. But this lasted for only a few sprints before the team started to get tired. Maybe they were still completing their work, but they weren't as excited about it, and as a result they were tagging their cards as "sad." This is a great example of how additional data can give you real insight into your team outside the normal measurement metrics.

3.3 Key project management metrics; spotting trends in data

Instead of focusing on the typical agile metrics, we'll look at what some key PTS data means when combined. The four metrics we're going to combine to help you get a broader picture of the performance of the team are:

- *Estimates*—The perceived amount of effort that a team gives to a task before they work on it
- *Volume*—The number of tasks that are completed
- *Bugs*—The number of defects that are created and worked by a team
- *Recidivism*—Tasks that someone said were good enough to move forward in the process but ended up getting moved back

As we take a more in-depth look at interpreting this data, you'll see the inconsistencies and ambiguity that result from looking at small bits of all the data you generate in your software development lifecycle.

3.3.1 Task volume

The data we've charted in this chapter so far is the amount of estimated effort completed over time. Measuring estimates is a good place to start because it's already typical practice on agile teams and provides a familiar starting place.

Volume is the number of work items your team is getting done. This is a bit different than measuring estimates, where you're evaluating the amount of estimated effort regardless of the number of tasks you complete. Tracking volume adds a bit more depth to the estimates and bugs because it helps you determine a few key items:

- How big are your tasks?
- What is the ratio of new work to fixing things and revisiting old work?
- Are tasks coming in from outside the intake process?

The delta between estimation and actual time is a valuable metric because it shows you a few potential things:

- How well your team understands the product they're working on
- How well the team understands the requirements
- How well your requirements are written
- How technically mature your team is

Adding velocity to your estimates will show you how many tasks are getting completed versus how much estimated effort you put into tasks. There should be a significant delta between the two because each task would have an estimate greater than 1 point. If your team takes on 10 tasks each with an estimate of 3 points, then your volume will be 10 and your velocity target will be 30 (10 tasks * 3 estimate points).

When you notice the delta between volume and estimation shrinking, that's usually an indication that something is wrong. Usually that means you're doing a lot of work you didn't plan for. If your task volume is 30 and your target estimate is 30, then you either have 30 tasks all estimated at 1 point or you have tasks creeping into your work stream that you never estimated. To figure out where that is coming from you'll have to dig into your data a bit more.

3.3.2 Bugs

The previous data will help you dig into your estimates to figure out how accurate they are, but there's still more data you can add to further hone your understanding. Adding bugs to the picture will give you an idea of what kind of tasks your team is working on. This is an especially important metric for teams that don't estimate bugs, because that work would have been only partially visible to you so far.

A bug represents a defect in your software, which can mean different things to different people. If a feature makes its way to done, all the necessary parties sign off on completion, and it turns out that there's a defect that causes negative effects to your consumers, you have a bug. Bugs are important to track because they point to the quality of the software your team is producing. You should pay attention to two bug trends bug creation rate and bug completion rate.

Bug creation rate can be calculated by getting the count of all of the tasks of type "bug" or "defect" by create date. Bug completion rate is calculated by the count of all tasks of type "bug" or "defect" by completion date.

Usually you want to see the exact opposite from bug creation and bug completion; it's good when bugs are getting squashed—that means bug completion is high. It's bad when you're generating a lot of bugs—that means bug creation is high. Let's take a look at bug completion in the scope of the big picture. Figure 3.11 is an aggregate of bugs, volume, and story points, the data that we've collected and put together so far.

Figure 3.11 shows that this team doesn't output many bugs, and estimates and volume seem to be fairly closely aligned, which at a glance is pretty good. But there are some instances where volume shows that work is being completed, but no points are associated with those tasks and no bugs are completed at the same time. This shows that the team is picking up unplanned work, which is bad because:

- It's not budgeted for.
- It impacts in a negative way the deliverables the team committed to.

Notice how around 09-11 even though volume shows work is getting completed, no story points are completed. This is a great example of how unplanned work can hurt the deliverables that a team commits to. After seeing this trend, this team needs to

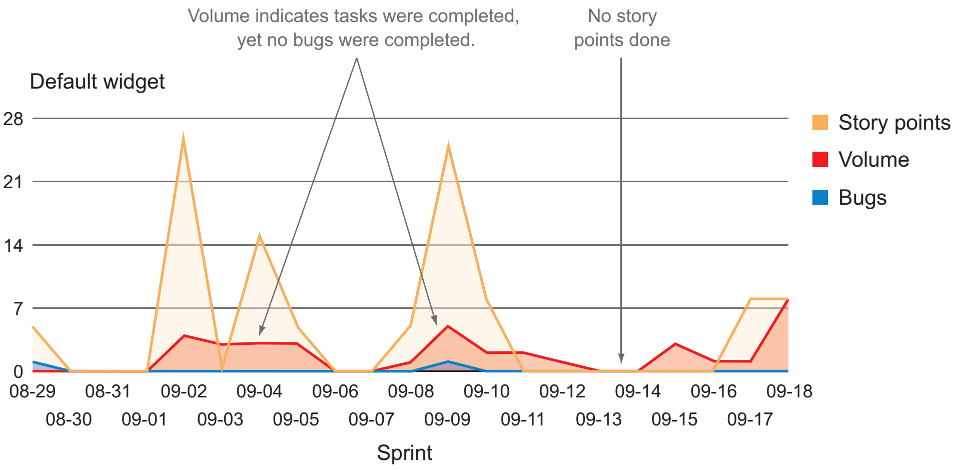


Figure 3.11 Bugs, volume, and estimates tracked together

find the root of the unplanned work and get rid of it. If they must complete it, then they should estimate it and plan for it like everything else in the work stream.

3.3.3 Measuring task movement; recidivism and workflow

The final piece of data to add to this chart is the movement of tasks through your team's workflow. This data gives you great insight into the health of your team by checking how tasks move through the workflow.

For this we'll look at recidivism, which is the measurement of tasks as they move backward in the predefined workflow. If a task moves from development to QA, fails validation, and moves back to development, this would increase the recidivism rate.

Spikes in this data point can indicate potential problems:

- There's a communication gap somewhere on the team.
- Completion criteria (a.k.a. done) are not defined clearly to everyone on the team.
- Tasks are being rushed, usually due to pressure to hit a release date.

If the number of tasks that move backward isn't spiking but is consistently high, then you have an unhealthy amount of churn happening on your team that should be addressed.

Figure 3.12 adds this data point to the charts you've been looking at so far in this chapter.

At a glance this may be too much data to digest all at once. If you look at only the tasks that came back from QA, which is measuring cards that moved backward, you can see that spikes end up around bad things, such as huge spikes in tasks complete or huge dips in productivity. The large spikes in backward tasks along with the large spikes in total tasks complete and a flat-line sprint goal representing estimations tell

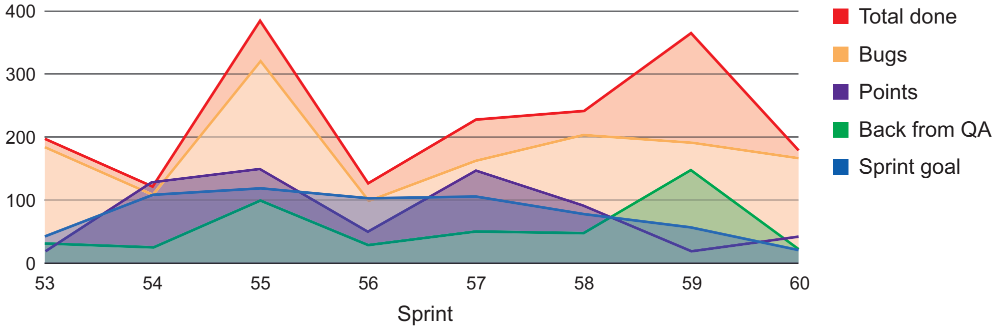


Figure 3.12 An example of all the data we have so far

you that your team must be opening up tasks that were previously marked as complete and putting them back into the work stream for some reason. That behavior results in a lot of unplanned work, and your team should discuss this trend to figure out why they're doing this and how to fix the problem.

3.3.4 Sorting with tags and labels

Tags and labels help you associate different tasks by putting an objective label on them. This helps you see associations between properties of tasks that you may not have thought of slicing up before. Figure 3.13 shows a simple aggregation of tags for a specific project. In this example all you can see is what the most common tags are and the count of the different tags.

By labeling or tagging your data you can find trends in other data that relate to it.

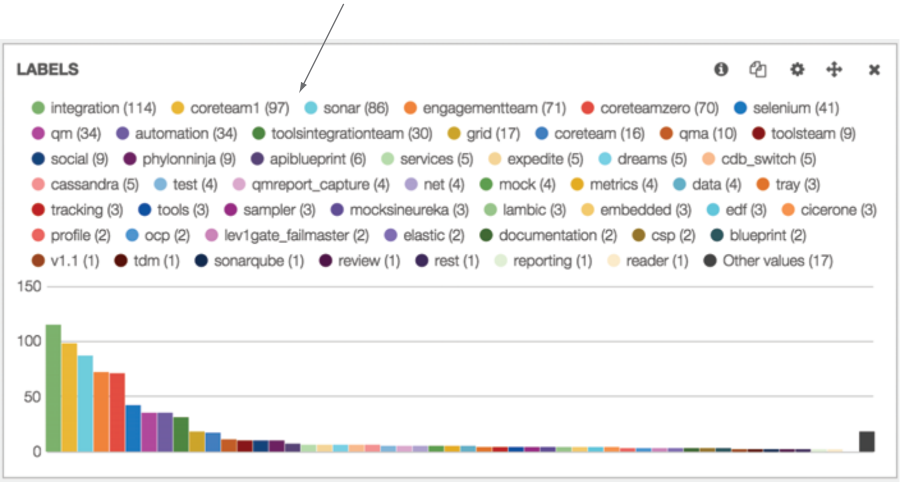
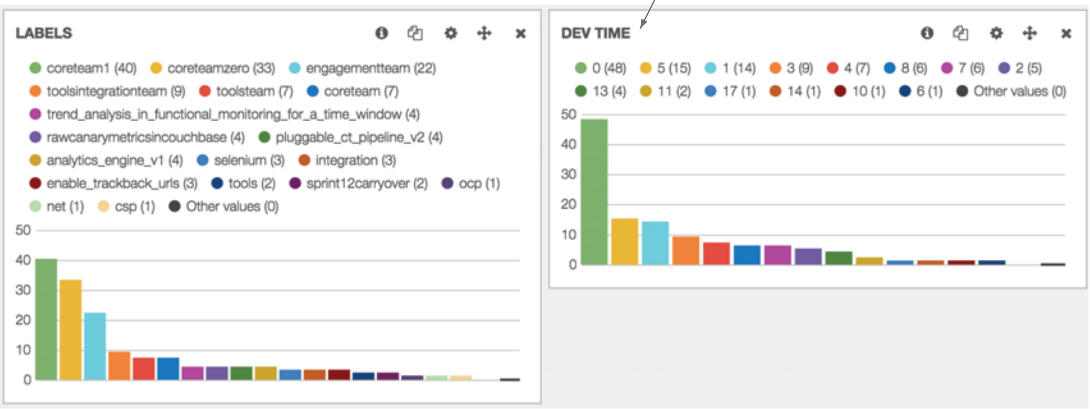


Figure 3.13 A breakdown of tags for a specific project


LABELS
● coreteam1 (40) ● coreteamzero (33) ● engagementteam (22)
● toolsintegrationteam (9) ● toolsteam (7) ● coreteam (7)
● trend_analysis_in_functional_monitoring_for_a_time_window (4)
● rawcanarymetricsincouchbase (4) ● pluggable_ct_pipeline_v2 (4)
● analytics_engine_v1 (4) ● selenium (3) ● integration (3)
● enable_trackback_urls (3) ● tools (2) ● sprint12carryover (2) ● ocp (1)
● net (1) ● csp (1) ● Other values (0)
50
40
30
20
10
0
DEV TIME
● 0 (48) ● 5 (15) ● 1 (14) ● 3 (9) ● 4 (7) ● 8 (6) ● 7 (6) ● 2 (5)
● 13 (4) ● 11 (2) ● 17 (1) ● 14 (1) ● 10 (1) ● 6 (1) ● Other values (0)
50
40
30
20
10
0

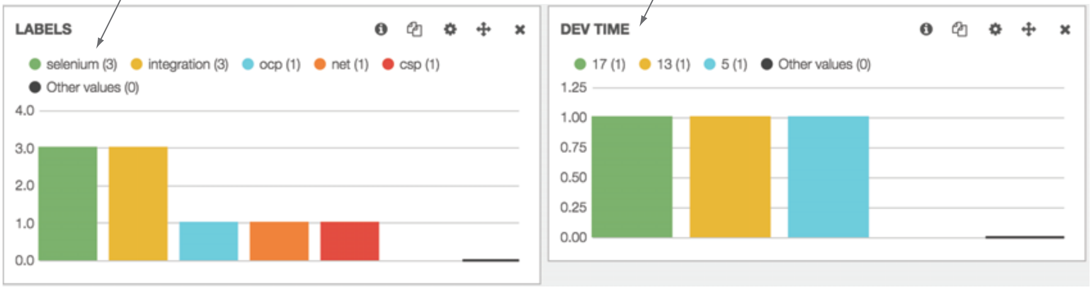
Adding another metric against labels will allow you to see correlations between tags and that metric—here, development time.

Figure 3.14 Viewing development time side by side with tags

In figure 3.14 we add another metric into the mix, development time.

Now you can sort the two charts by filtering the data behind these charts by your labels. The Kibana dashboard allows you to click any chart to filter the rest of the charts on a dashboard by that data. You can begin by clicking tasks labeled "integration" in the Labels panel from figure 3.15.

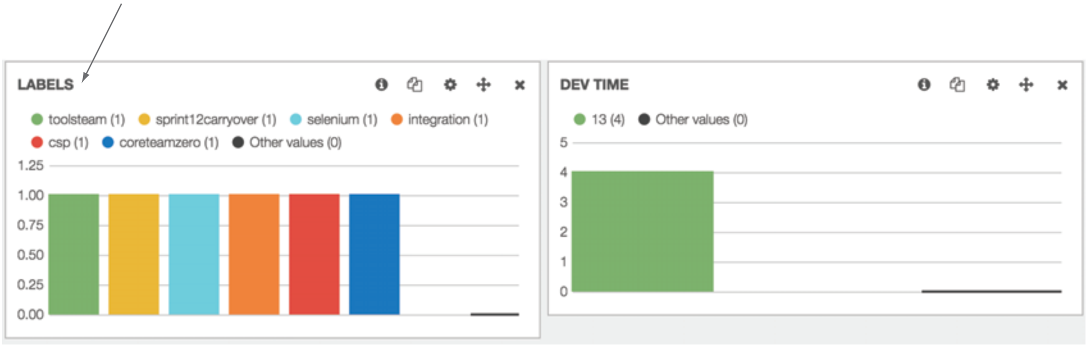
In this case you can see that tasks labeled "integration" take a particularly long time: 17, 13, or 5 full days. Conversely, you can also sort on development time to see which labels take a long time, as shown in figure 3.16.


LABELS
● selenium (3) ● integration (3) ● ocp (1) ● net (1) ● csp (1)
● Other values (0)
4.0
3.0
2.0
1.0
0.0
DEV TIME
● 17 (1) ● 13 (1) ● 5 (1) ● Other values (0)
1.25
1.00
0.75
0.50
0.25
0.00

If you sort by tasks labeled "integration" you see the other associated tags.

Development time seems to take a long time when you sort by tasks labeled "integration."

Figure 3.15 Sorting data by labels and seeing the effect on development time


LABELS
● toolsteam (1) ● sprint12carryover (1) ● selenium (1) ● integration (1)
● csp (1) ● coreteamzero (1) ● Other values (0)
1.25
1.00
0.75
0.50
0.25
0.00
DEV TIME
● 13 (4) ● Other values (0)
5
4
3
2
1
0

Sorting by development time you can see all labels that take a long time to complete.

Figure 3.16 Sorting labels by development time

In chapter 7 we take a closer look at exploring data and the relationship between data points.

3.4 Case study: identifying tech debt trending with project tracking data

Now that we've talked about this data and have it populating our graphs, I'll share a real-world scenario where this data alone helped a team come to terms with a problem, make adjustments, and get to a better place.

Our team was using Scrum and had a two-week sprint cadence. We were working on a release that had a rather large and interrelated set of features with many stakeholders. As with most teams, we started with tracking velocity to ensure that we were consistently estimating and completing our work every sprint. We did this so we could trust our estimates enough to be able to deliver more predictably over time and so stakeholders could get an idea of approximately when the whole set of features they wanted would be delivered.

In this case we couldn't hit a consistent number of story points at the end of every sprint. To complicate things we didn't have a typical colocated small agile team; we had two small teams in different time zones that were working toward the same large product launch.

In summary the current state of the team was:

- Geographically distributed team working toward the same goal
- Agile development of a large interrelated feature set

We were asking these questions:

- Why are we not normalizing on a consistent velocity?
- What can we do to become more predictable for our stakeholders?

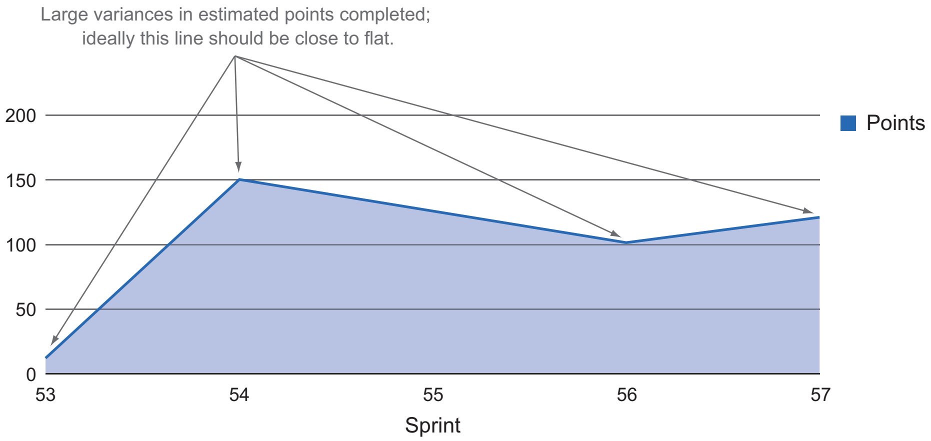


Figure 3.17 Story points are jumping all over the place.

The goal was to maintain a consistent velocity so we could communicate more effectively to our stakeholders what they could expect to test by the end of every sprint. The velocity of the team is shown in figure 3.17.

In this case we were tracking data over sprints and using it in our retrospectives to discuss where improvements could be made. From sprints 53–57 we were seeing significant inconsistency in the estimates we were completing. To get a handle on why our estimates were not consistently lining up, we decided to also start tracking volume to compare what we thought we could get done against the number of tasks we were completing. In theory, estimates should be higher than volume but roughly completed in parallel; that got us the graph shown in figure 3.18.

Whoa! That doesn't look right at all! Volume and estimates are mostly parallel, but according to this we're getting the same amount of tasks completed as points. Because

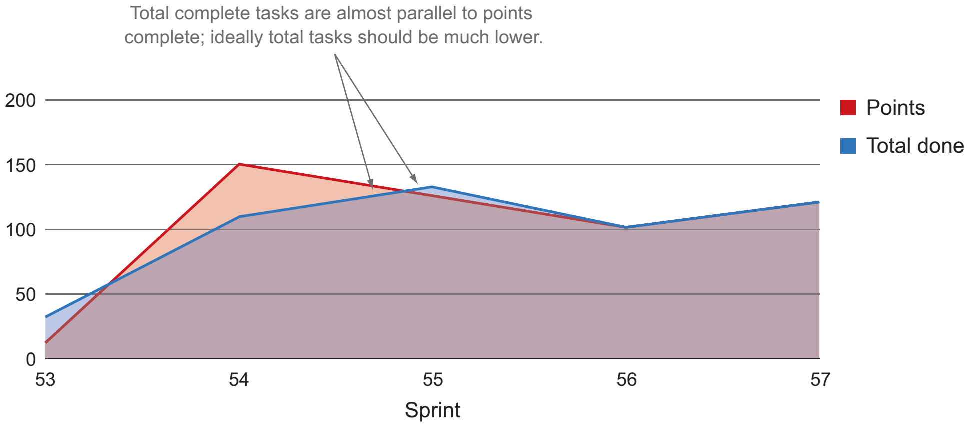


Figure 3.18 Adding volume to the picture

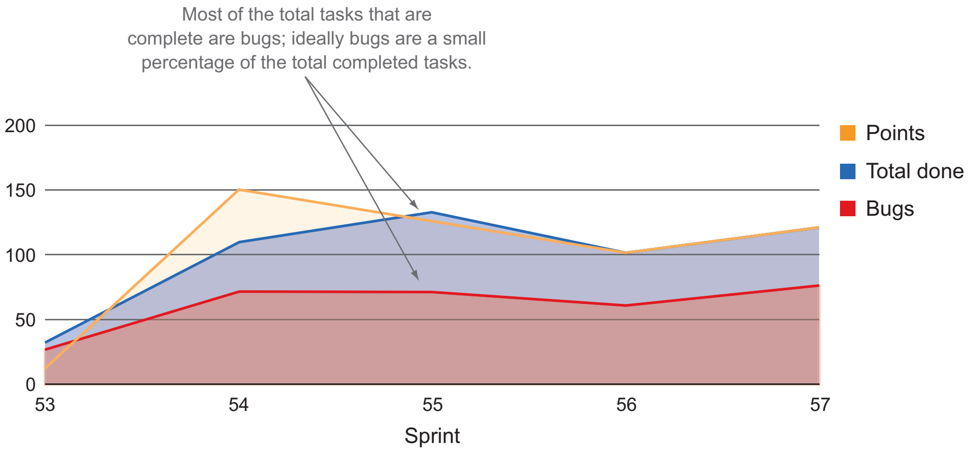


Figure 3.19 Adding bugs

each task should have an estimate attached to it, points should be much higher than tasks. After a closer look at the data we noticed that a lot of bugs were also getting completed, so we added that to our chart (figure 3.19).

At this point we were much closer to the full picture. The huge volume of bugs that we had to power through every sprint was causing volume to go way up, and because the team wasn't estimating bugs we weren't getting the number of estimated features done that we thought we could. We all discussed the problem, and in this case the development team pointed to the huge amount of tech debt that had been piling up over time. Most of the things we completed ended up having edge cases and race conditions that caused problems with the functionality of the product. To show the amount of work that was moving backward in sprint, we also started mapping that out, as shown in figure 3.20.

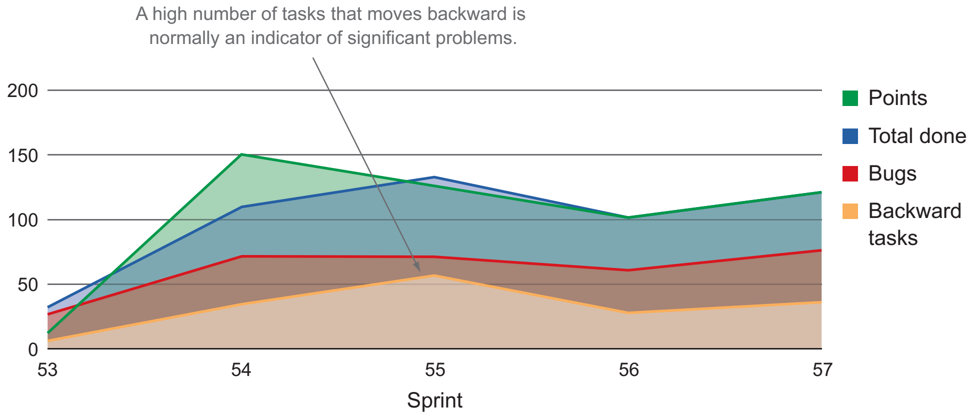


Figure 3.20 Adding tasks that move backward to the chart

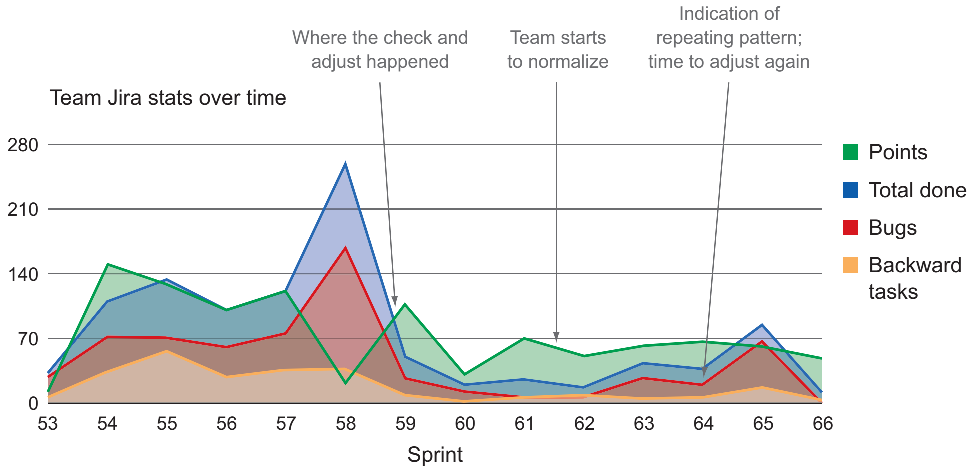


Figure 3.21 The complete data over time

Most of the tech debt stemmed from decisions that we made in earlier sprints to hit an important release date. To combat the issue we decided to have a cleanup sprint where we would refactor the problem code and clean up the backlog of bugs. Once we did that, the number of bugs in sprint was drastically reduced and the number of bug tasks that moved backward also went way down. After using this data to point to a problem, come up with a solution, try it, and watch the data change, we started watching these charts sprint over sprint. Eventually tech debt started to build again, which was clearly visible in our metrics collection system and shown in figure 3.21, starting in sprint 62.

We saw the same trend starting to happen again, so we took the same approach; we had a cleanup sprint in 65 and managed to get the team stable and back in control with a much smaller hit than back in sprint 58.

Even though this data paints a pretty good picture of a problem and the effects of a change on a team, it's still only the tip of the iceberg. In the next chapter we'll start looking at even more data that you can add to your system and what it can tell you.

3.5 Summary

The project tracking system is the first place most people look for data to measure their teams. These systems are designed to track tasks, estimates, and bugs in your development lifecycle and alone they can give you a lot of insight into how your team is performing. In this chapter you learned:

- Your PTS contains a few main pieces of raw data: what, when, and who.
- The richer the raw data in the system, the more comprehensive the analysis. To enable this your team should follow these guidelines:
  - Always use the PTS.

  - Tag tasks with as much data as possible.
  - Estimate your work.
  - Clearly define the criteria for completion (done).
  - Retroactively tag tasks with how happy you were about them.
- Tagging tasks in your PTS allows you to analyze your data in novel ways.
- Adding a narrative gives context to changes over time.
- Velocity and burn down are a good starting point for analysis, but adding more data gives you a clearer picture of the trends you notice in those metrics.
- Pulling key data out of your PTS into a time-data series makes it easier to see how your team is performing.
- Just about anything can be distilled into a chart in your analytics system.

4

Trends and data from source control

This chapter covers

- Learning from your SCM system's data alone
- Utilizing your SCM to get the richest data possible
- Getting data from your SCM systems into your metrics collection system
- Learning trends from your SCM systems

Project tracking is a great place to start when you're looking for data that can give you insight into the performance of your team. The next data mine that you want to tap into is your source control management (SCM) system. In the scope of our application lifecycle, this is highlighted in figure 4.1.

SCM is where the action is; that's where developers are checking in code, adding reviews, and collaborating on solutions. I've often been on teams where it was like pulling teeth to get developers to move cards or add comments in the project tracking system, which ends up leading to a huge blind spot for anyone outside the development team interested in the progress of a project. You can eliminate that blind spot by looking at the data generated by the use of your SCM.

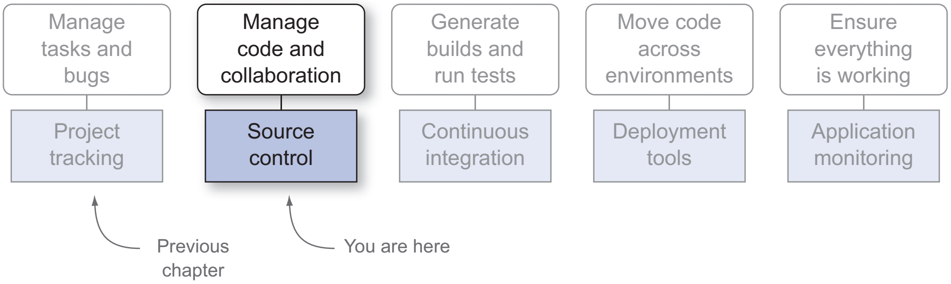


Figure 4.1 You are here: SCM in the scope of the application lifecycle.

If we revise the questions we asked in chapter 3, we get the types of who, what, and how through data from SCM, as shown in figure 4.2.

Here are two questions you can answer from your SCM systems:

- How much change is happening in your codebase?
- How well is/are your development team(s) working together?

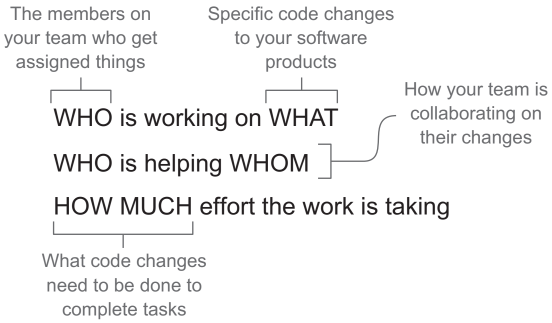

The members on your team who get assigned things
Specific code changes to your software products
How your team is collaborating on their changes
WHO is working on WHAT
WHO is helping WHOM
HOW MUCH effort the work is taking
What code changes need to be done to complete tasks

Figure 4.2 The who, what, and how from SCM

If you combine SCM data with your PTS data from chapter 3, you can get really interesting insights into these issues:

- Are your tasks appropriately sized?
- Are your estimates accurate?
- How much is your team really getting done?

We take a much closer look at these questions in chapter 7 when we dive into combining different data sources. Our system is ready to go, but before we start taking a look at our data, let's make sure you're clear on the basic concepts of SCM.

4.1 What is source control management?

If you're working on a software production, you should already know what SCM is. But because one of my golden rules is never to assume anything, I'll take this opportunity to tell you just in case.

The software product that you're working on ends up getting compiled and deployed at some point. Before that happens, your team is working on the *source,* or the code that comprises the functionality you want to ship to your consumers. Because you likely have more than one person working on this source, you need to *manage* it somewhere to ensure that changes aren't getting overwritten, good collaboration is happening, and developers aren't stepping on each other's toes. The place where this is managed and the history and evolution of your code are stored is your SCM system.

Popular SCM systems today include Subversion (SVN), Git, Mercurial, and CVS. SCM systems have evolved significantly over time, but switching from one type of SCM to another isn't always easy. This is because if all of your code and history are in one SCM system, moving all of it and asking your development team to start using a new system usually takes quite a bit of effort. There are tools to help you migrate from one system to another, and following those tools leads you down the path of the evolution of source control. For example, a tool called cvs2svn will help you migrate from CVS to the newer and more popular SVN. Google "migrating SVN to CVS"; most of the links will point you in the other direction. There are also tools that help you migrate from SVN to the newer Git or Mercurial, but going in the opposite direction isn't so common.

If you're not using SCM for your project, I strongly recommend you start using it. Even teams of one developer benefit from saving the history of their codebase over time; if you make a mistake or need to restore some functionality that you took out, SCM makes it easy to go back in time and bring old code that you got rid of back to life.

4.2 Preparing for analysis: generate the richest set of data you can

The most obvious data you can get from these systems is how much code is changing in CLOC and how much individuals on the team are changing it. As long as you're using source control, you have that data. But if you want to answer more interesting questions like the ones I posed at the beginning of the chapter:

- How well is your team working together?
- Are your estimations accurate?
- Are your tasks appropriately sized?
- How much is your team really getting done?

then you should use the following tips to ensure you're generating the richest set of data you can.

Changing SCM in the enterprise

As I talk about generating rich data in the next few sections, I'll be talking through potential changes to how your team operates that might not seem realistic. If you're working at a nimble startup or are about to start a new project, then choosing or changing the type of SCM you use may not be a big deal. If you're from a large enterprise company that has used CVS forever and refuses to hear anything about these newfangled SCM systems, a good data point to use to justify changing systems is the amount of data you can get out of them and how they can improve collaboration on your team. If you find yourself in that situation, keep that in mind as you read through the following sections.

4.2.1 Tip 1: Use distributed version control and pull requests

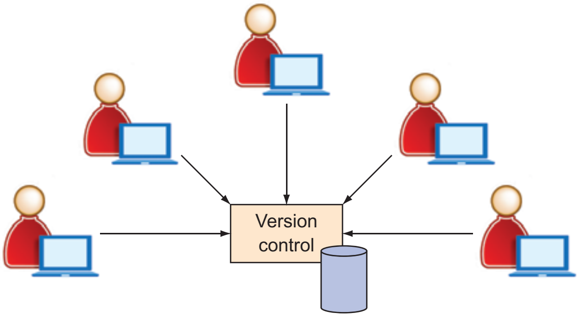


Figure 4.3 Centralized SCM system: there is one place everyone must get to.

There are plenty of options when it comes to choosing a version control system to use. When you do choose a system, think about the type of data that you can get out of it as part of its overall value. The first choice to make when choosing an SCM system is whether to go with a distributed or a centralized system. A centralized system is based on a central server that's the source of truth for all code. To make changes and save the history of them, you must be connected to the central repository. A centralized SCM system is shown in figure 4.3.

In a centralized SCM model the type of metadata that's stored along with the commit is usually just a comment. That's better than nothing, but it isn't much.

With a distributed system everyone who checks out the project then has a complete repository on their local development machine. Distributed version control systems (DVCSs) typically have individuals or groups who make significant changes to the codebase and try to merge it later. Allowing everyone to have their own history and to collaborate outside of the master repo gives a team a lot more flexibility and opportunity for distributed collaboration. A DVCS is illustrated in figure 4.4.

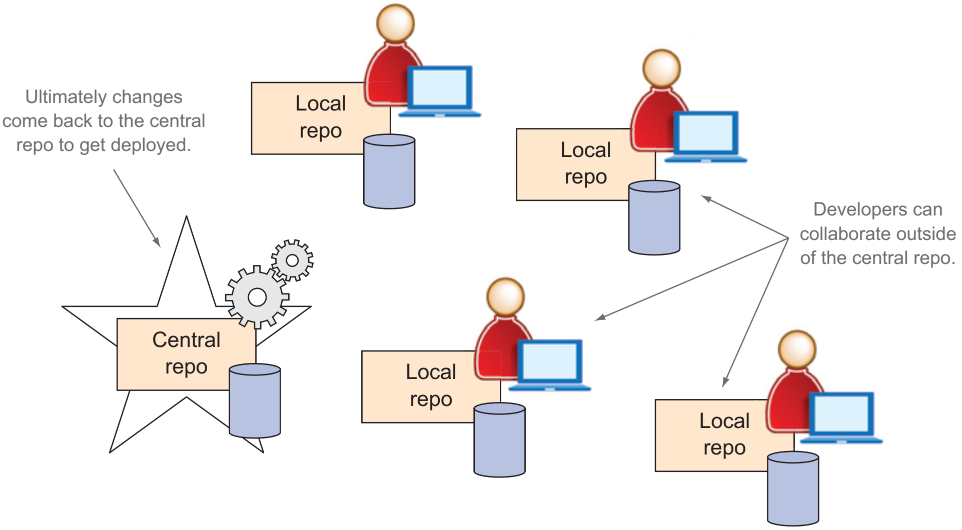

Ultimately changes come back to the central repo to get deployed.
Local repo
Local repo
Developers can collaborate outside of the central repo.
Central repo
Local repo
Local repo

Figure 4.4 Distributed version control system (DVCS): everyone has a repo.

Note in figure 4.4 that there's still a central repo where the master history is stored and the final product is deployed from.

SCM systems have two primary focal points:

- Saving the history of everything that happens to the code
- Fostering great collaboration

Centralized SCM systems have all the history in one place: the central repo. DVCSs save the history of changes no matter where they happen and as a result have a much richer set of data available around code changes. In addition, DVCSs typically have RESTful APIs, which make getting that data and reporting on it with familiar constructs plug and play at this point.

From the perspective of someone who wants to get as much data about your team's process as possible, it's much better to use a DVCS because you can tell a lot more about how the team is working, how much collaboration is happening, and where in your source code changes are happening.

A common practice in teams that use DVCSs is the idea of a pull request. When a developer has a change or a set of changes that they want to merge with the master codebase, they submit their changes in a *pull request*. The pull request includes a list of other developers who should review the request. The developers have the opportunity to comment before giving their approve/deny verdict. This is illustrated in figure 4.5.

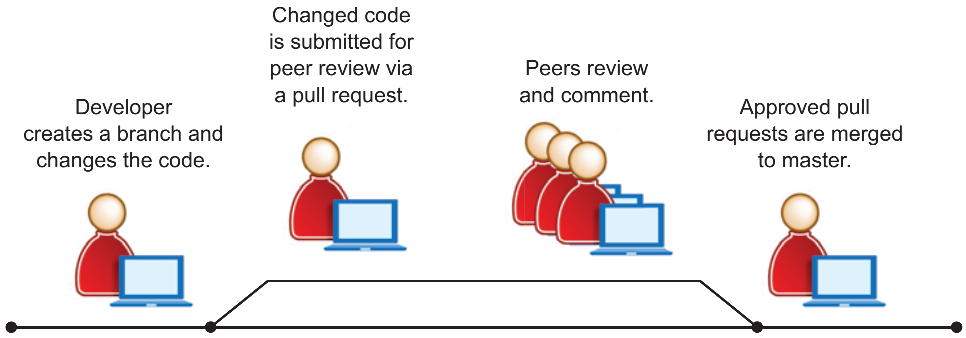

Developer creates a branch and changes the code.
Changed code is submitted for peer review via a pull request.
Peers review and comment.
Approved pull requests are merged to master.

Figure 4.5 The pull request workflow for source control usage

DVCS workflows

The most popular DVCS workflows are the feature-branch workflow and gitflow.

The feature-branch workflow is the simpler one. The overall concept is that all new development gets its own branch, and when development is done, developers submit a pull request to get their feature incorporated into the master.

Gitflow also focuses on using pull requests and feature branches but puts some formality around the branching structure that your team uses. A separate branch, used to aggregate features, is usually called develop or next. But before merging a feature into the develop branch, there's typically another pull request. Then, after all features are developed and merged, they are merged into the master branch to be deployed to the consumer.

For information about Git and using workflows, check out *Git in Practice* by Mike McQuaid (Manning, 2014), which goes into great detail about these workflows and how to apply them.

This collaboration among developers is then saved as metadata along with the change. GitHub is an example system that has a rich API with all of this data that will help you get better insight into your team's performance. The following listing shows an abridged example of data you can get from the GitHub API about a pull request.

Listing 4.1 Abridged example response from the GitHub API

```
{
  "state": "open",
  "title": "new-feature",
  "body": "Please pull these awesome changes",
  "created_at": "2011-01-26T19:01:12Z",
  "updated_at": "2011-01-26T19:01:12Z",
  "closed_at": "2011-01-26T19:01:12Z",
  "merged_at": "2011-01-26T19:01:12Z",
  "_links": {
    "self": {
      "href": "https://api.github.com/repos/octocat/Hello-World/pulls/1"
    },
    "html": {
      "href": https://github.com/octocat/Hello-World/pull/1
    },
    "issue": {
      "href": "https://api.github.com/repos/octocat/Hello-World/issues/1"

    },
    "comments": {
      "href": https://api.github.com/repos/octocat/Hello-World/issues/1
      ➥ /comments
    },
    "review_comments": {
      "href": "https://api.github.com/repos/octocat/Hello-World/pulls/1
      ➥ /comments"
      "merge_commit_sha": "e5bd3914e2e596debea16f433f57875b5b90bcd6",
      "merged": true,
      "mergeable": true,
      "merged_by": {
      "login": "octocat",
      "id": 1,
```

General information about the pull requests

Dates are great for charting

Follow the link for details on this node

Links to all of the comments

Links to the comments for reviews

The ID of the parent commit

If this has already been merged

If this can be merged

Who merged the pull request

```
    },
    "comments": 10,
    "commits": 3,
    "additions": 100,
    "deletions": 3,
    "changed_files": 5
  }
```

Key stats; very useful for charting

As you can see, there's a lot of great data that you can start working with from this request. The key difference between the data you can get from DVCS over centralized VCS is data around collaboration; you can not only see how much code everyone on your team is changing, but you can also see how much they're participating in each other's work by looking at pull request activity.

Linking in RESTFUL APIs

As shown in listing 4.1, GitHub uses a technique called linking in its API. In the case of listing 4.1, you'll notice that several sections of the response are HTTP URLs. The idea with linking is that the API itself should give you all the information you need to get more. In this case, if you want to get all of the data about the different comments, follow the link in the comments block. Listing 4.1 is an abridged example to show you generally what type of data you can get back from the API, but in a full response there will be many links that point to the details inside each node. RESTful APIs often use linking this way to make pagination, discovery of data, and decoupling different types of data generally easier.

Let's look at the metrics you can pull out of this data and how to combine it with the metrics you already have from our PTS in chapter 4.

4.3 The data you'll be working with; what you can get from SCM

If you use the system that we built in appendix A, you can create a service that plugs in and gets data from your DVCS. Because the data that you can get from a DVCS is so much richer than that from centralized SCM systems and because they have great APIs to work with, this section will focus on getting data from DVCSs. We'll also explore getting data from a centralized SCM at a high level.

4.3.1 The data you can get from a DVCS

The simplest structure you can get from a DVCS is a commit, as shown in figure 4.6.

Figure 4.6 contains a message, an ID that's usually in the form of an SHA (secure hash algorithm), parent URLs that link to the previous history of the commit, and the name of the person who created the code that was committed. Keep in mind that who wrote the code can be someone different than who commits it to the repo. If you're following good gitflow or feature-branch workflows, you'll see this often; a developer will submit a pull request and typically the last reviewer or the owner repo will accept and usually commit the change.

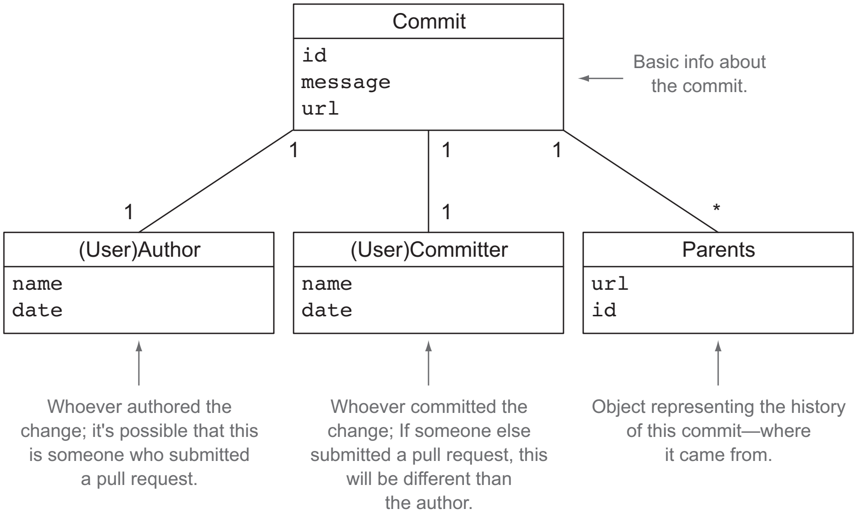

Commit
id
message
url
Basic info about the commit.
1
1
1
1
1
*
(User)Author
name
date
(User)Committer
name
date
Parents
url
id
Whoever authored the change; it's possible that this is someone who submitted a pull request.
Whoever committed the change; If someone else submitted a pull request, this will be different than the author.
Object representing the history of this commit—where it came from.

Figure 4.6 The object structure of a commit in a typical DVCS

Speaking of pull requests, the typical structure for them is shown in figure 4.7.

In figure 4.7 you have the author of the pull request and a link to all the reviewers who took part in the pull request workflow. You can also get all the comments for that pull request and the history of where it came from and where it was submitted.

Commits tell you all the code that's making it into the repo. Pull requests tell you about all the code that *tried* to make it into the repo. Pull requests are essentially a

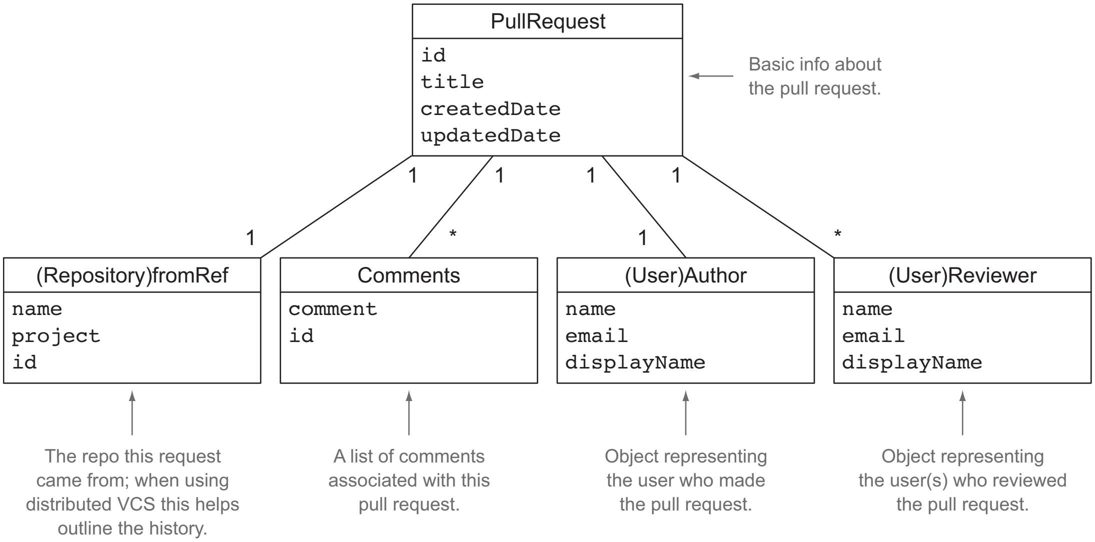

PullRequest
id
title
createdDate
updatedDate
Basic info about the pull request.
1
1
1
1
1
*
1
*
(Repository)fromRef
name
project
id
Comments
comment
id
(User)Author
name
email
displayName
(User)Reviewer
name
email
displayName
The repo this request came from; when using distributed VCS this helps outline the history.
A list of comments associated with this pull request.
Object representing the user who made the pull request.
Object representing the user(s) who reviewed the pull request.

Figure 4.7 The data structures you can get from a pull request in DVCS

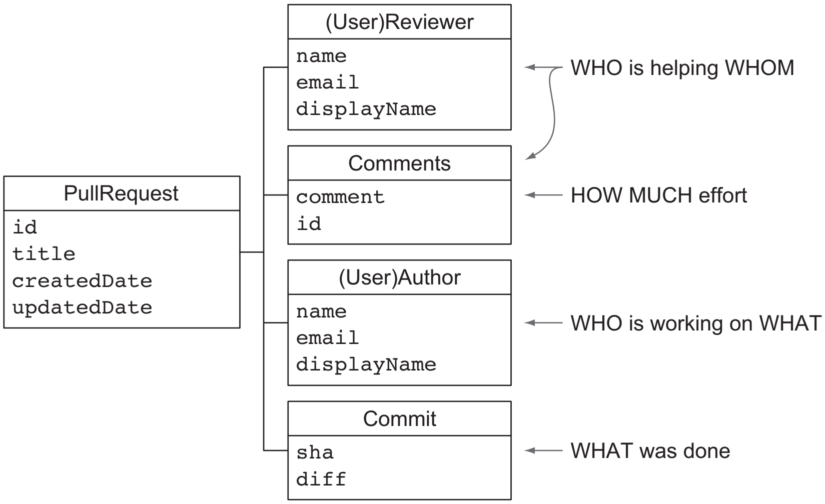


Figure 4.8 Mapping your questions back to the objects you can get from the DVCS API

request to put code into the repo, and by their nature they're much richer in data than commits. When a user submits a pull request, they ask their peers for comments, reviews, and approval. The data that is generated from the pull request process can tell you not only who is working on what and how much code they're writing but also who is helping them and how much cooperation is going on across the team.

Now that you see what kind of data you get back, you can start mapping that data back to the questions you want answered, as shown in figure 4.8.

Here's what you know so far:

- Who is doing what from the `User` objects that you get back for reviewers and authors
- Who is collaborating through the `User` objects representing reviewers
- How much collaboration is going on through the comments

All of this is great if you're looking for data about your team. In addition, you can parse the information from the code changes to find out which files are changing and by how much. With this information you can answer additional questions like the following:

- Where are the most changes happening (change hot spots in your code)?
- Who spends the most time in which modules (hot spots by user)?
- Who is changing the most code?

Hot spots in your code point to where the most change is happening. This could mean that your team is developing a new feature and they're iterating on something new or changing something—this is what you'd expect to see. Things you might not expect are spots that you have to iterate every time you change something else—these are usually indications of code that's too tightly coupled or poorly designed. Spots like this are time-consuming, often simply because of tech debt that needs to be cleaned up. Showing the amount of time spent on problem spots is good evidence that you should fix them for the sake of saving time once the change is made.

4.3.2 Data you can get from centralized SCM

Even though I keep trash-talking SVN and CVS, both are common SCM systems in use by many developers today. This goes back to the fact that changing your SCM system is going to be a disruption to your team at some level. To move from SVN to Git, for example, your team will have to learn Git, stop working while the code gets migrated, migrate the code, and then get used to working in a different workflow. You can definitely expect this to slow your team down temporarily, but the benefits you'll get out of the data alone are worth it. This data will give you such greater insight that it gives you the potential to find problems and shift direction much faster than the data you can get from centralized SCM. The move to DVCS also encourages and fosters better collaboration through pull requests and, in my opinion as a developer, is nicer and easier to use.

If I haven't convinced you yet, you can still get some data out of your centralized SCM system. Basically, you can get who made the change and what the change was, as demonstrated in figure 4.9.

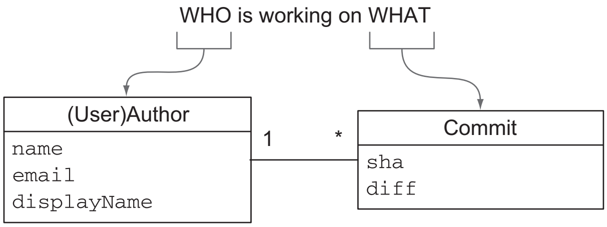


Figure 4.9 The data you can get out of centralized SCM

This data alone is often misleading. You can tell how many lines of code were changed and by whom, but that doesn't give you an idea of effort or productivity. A junior programmer can make many commits and write a lot of bad code, whereas a senior developer can find a much more concise way to do the same work, so looking at volume doesn't indicate productivity. Additionally, some people like to start hacking away at a problem and hone it as they go; others like to think about it to make sure they get it right the first time. Both methods are perfectly viable ways to work, but the data you'll get from them in this context is totally different.

The data from centralized SCM can be useful when combined with other information, such as how many tasks are getting done from your PTS or whether tasks are moving in the wrong direction. Using those as your indicators of good and bad will help you determine what your SCM data means.

4.3.3 What you can tell from SCM alone

The minimum set of data you can get from SCM is the amount of code everyone on your team is changing. With no other data you can get answers to questions like these:

- *Who is changing the code?*—Who is working on what parts of the product? Who is contributing the most change?

- *How much change is happening in your codebase?*—What parts of your product are undergoing the most change?

At its core SCM tracks changes to the codebase of your product, which is measured in CLOC. Just as velocity is a relative metric whose number means something different to every team that tracks it, CLOC and LOC are the epitome of relative metrics. LOC varies based on the language your code is written in and the coding style of the developers on the team. A Java program will typically have a much higher LOC than a Rails, Grails, or Python program because it's a much wordier language, but it's even difficult to compare two Java programs without getting into discussions on whose style is better. An example is the following listing.

Listing 4.2 Two identical conditional statements

```
if(this == that) {
doSomething(this);
else {
doSomethingElse(that);
}

(this == that) ? doSomething(this) : doSomethingElse(that);
```

Statement 1 = 5 LOC

Statement 2 = 1 LOC

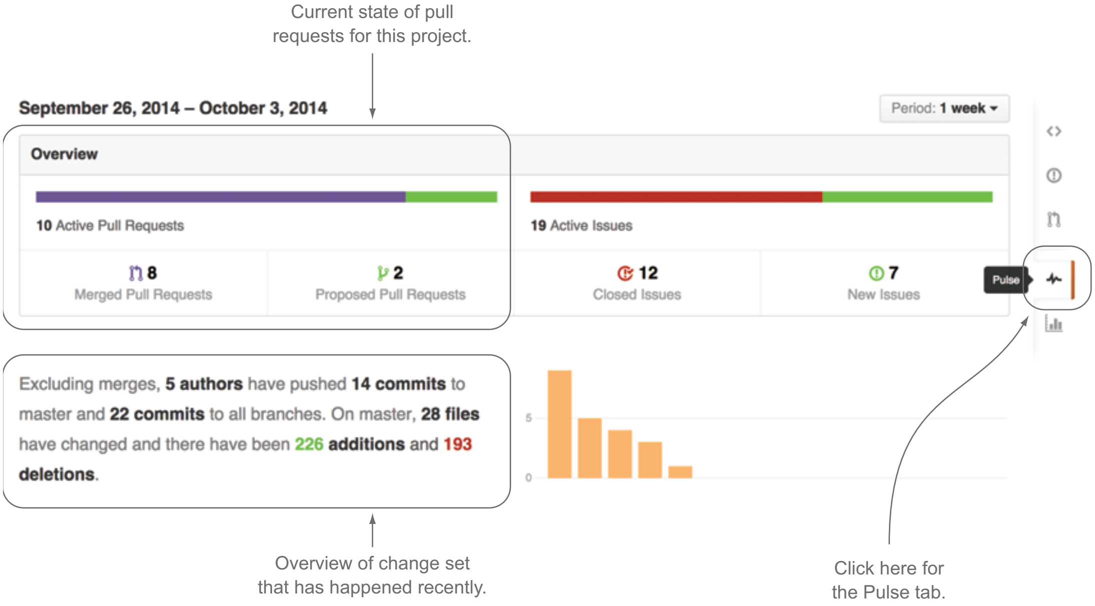
In this simple case two statements do the same thing, yet their LOC is very different. Both perform the same and produce the same result. The reason to choose one over the other is personal preference, so which is better? Examples like this show that LOC is a metric that can't tell you if changing code is good or bad without taking into account the trends over time.

There are a number of standard charts for SCM data for trending change. If you look at any GitHub project, you'll find standard charts that you can get to through the Pulse and Graph sections. FishEye produces the same charts for SVN-based projects. As an example in anticipation of our next chapter, we'll look at the GoCD repo in GitHub, which is the source for a CI and pipeline management server.

Figure 4.10 shows the GitHub Pulse tab, which gives you an overview of the current state of the project. The most valuable section here is the pull request overview on the left side of the page.

The Pulse tab is pretty cool, but just like checking your own pulse, it only tells you if the project you're looking at is alive or not. In the context of GitHub, where open source projects tend to live, it's important to know if a project is alive or dead so potential consumers can decide if they want to use it. To get data that's more relevant for your purposes, check out the Graphs section. It has the typical breakdowns of CLOC and LOC that you'd expect to see from your source control project. Let's start with contribution over time, shown in figure 4.11.

The first chart shows you total contribution in CLOC. Below that you can see the developers who have contributed to the codebase and how much they've contributed. This is interesting because you can see who is adding, deleting, and committing the most to your codebase. At a glance this may seem like a good way to measure developer productivity, but it's only a piece of the puzzle. I've seen bad developers committing and changing more code than good developers, and I've seen great developers at the top of the commit graph in others.



Current state of pull requests for this project.

September 26, 2014 – October 3, 2014

Period: 1 week

Overview

10 Active Pull Requests

19 Active Issues

8 Merged Pull Requests

2 Proposed Pull Requests

12 Closed Issues

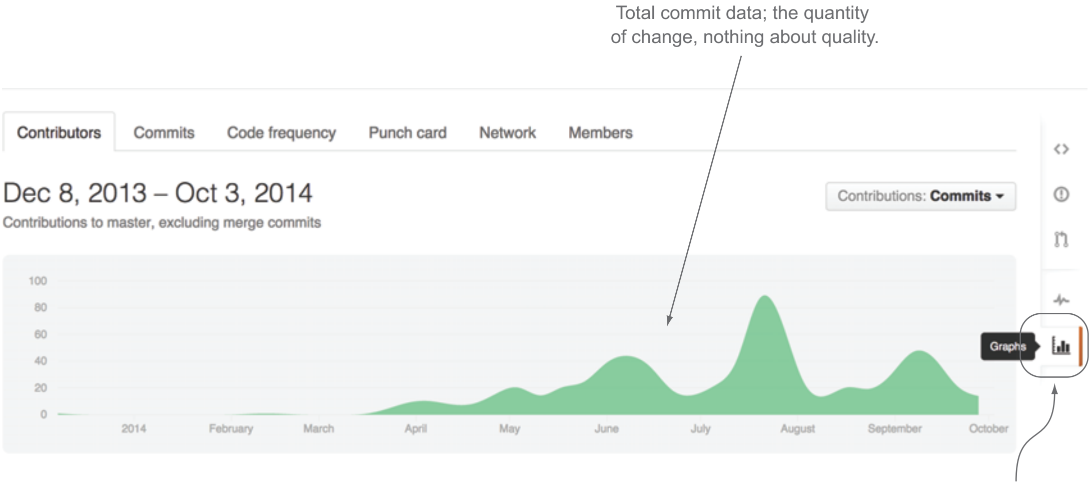
7 New Issues

Excluding merges, **5 authors** have pushed **14 commits** to master and **22 commits** to all branches. On master, **28 files** have changed and there have been **226 additions** and **193 deletions**.

Overview of change set that has happened recently.

Click here for the Pulse tab.

Figure 4.10 The Pulse page on GitHub gives you info on pull requests, issues, and amount of change. This is from the GoCD repo (github.com/gocd/gocd/pulse/weekly).



Total commit data; the quantity of change, nothing about quality.

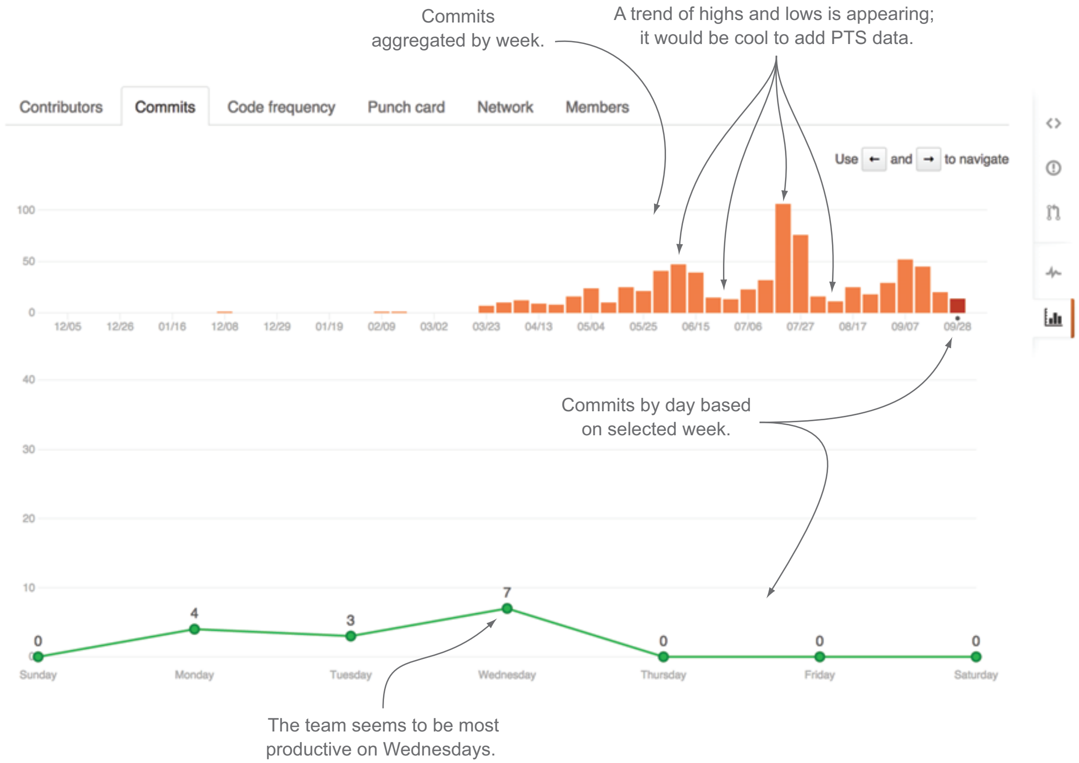
Dec 8, 2013 – Oct 3, 2014

Contributions to master, excluding merge commits

Contributions: Commits

Figure 4.11 The graph of contributors over time. From GoCD (github.com/gocd/gocd/graphs/contributors).

Click here for the Graphs tab.



Commits aggregated by week.

A trend of highs and lows is appearing; it would be cool to add PTS data.

Commits by day based on selected week.

The team seems to be most productive on Wednesdays.

Figure 4.12 Total commits to this repository over time show the amount of code changed from week to week and day to day. From GoCD (github.com/gocd/gocd/graphs/commit-activity).

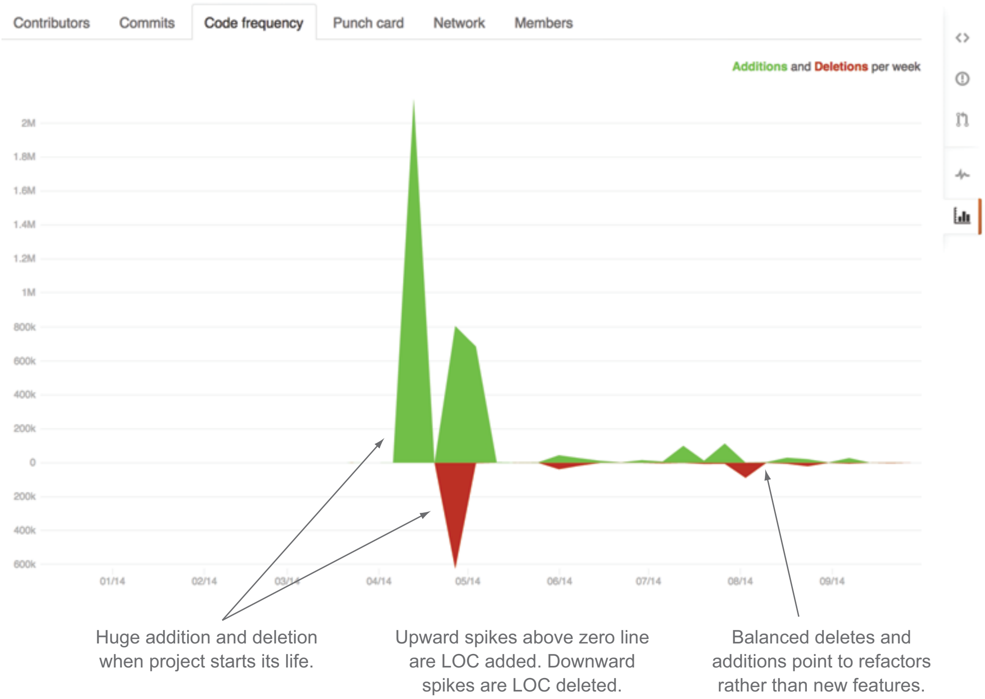
Next up are the commits over time, as shown in figure 4.12. This shows you how many code changes there are and when they happen.

Figure 4.12 shows the number of commits. To get to CLOC, click the next tab: Code Frequency.

In figure 4.13 you can see the huge addition, which was the birth of this project in GitHub. After bit of intense CLOC, it normalizes somewhat.

The last standard chart in GitHub is the punch card, shown in figure 4.14.

This is valuable in showing you when the most code change occurs in your project. In figure 4.14 you can see that between 10 a.m. and 5 p.m. the most change occurs, with a bit of extra work bleeding early or later than that—pretty much what you'd expect if your team is putting in a normal workweek. If you start to see the bubbles getting bigger on Fridays or at the end of sprints, that's typically a sign that your team is waiting until the last minute to get their code in, which is usually not good. Another bad sign is seeing days your team should have off (typically Saturday and Sunday) starting to grow. These trends may not point to the problem, but they at least indicate that there is a problem.



Huge addition and deletion when project starts its life.

Upward spikes above zero line are LOC added. Downward spikes are LOC deleted.

Balanced deletes and additions point to refactors rather than new features.

Figure 4.13 The Code Frequency tab shows how much code is changing over time in CLOC. From GoCD (github.com/gocd/gocd/graphs/code-frequency).

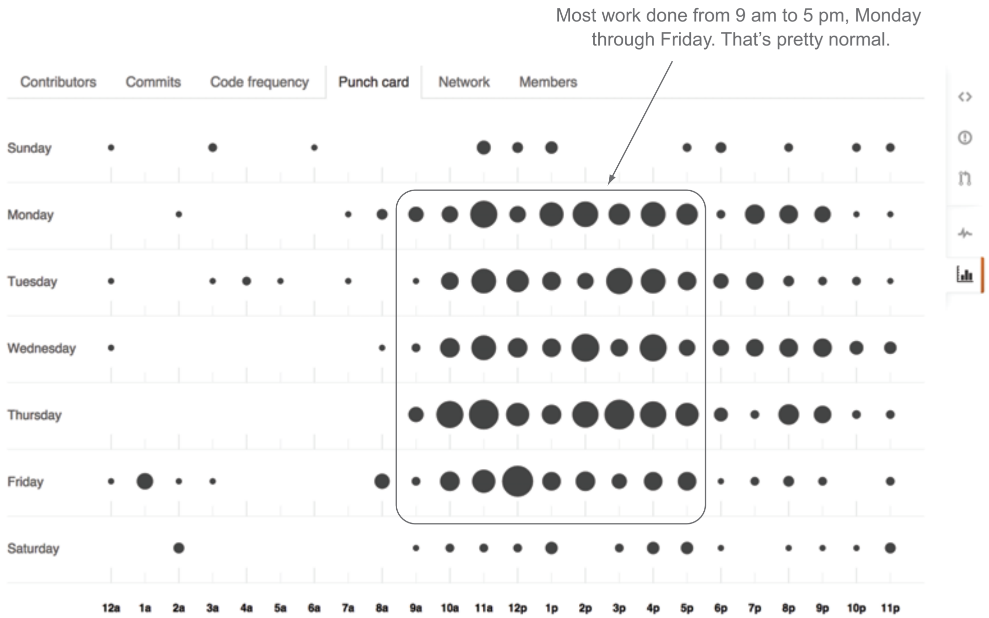
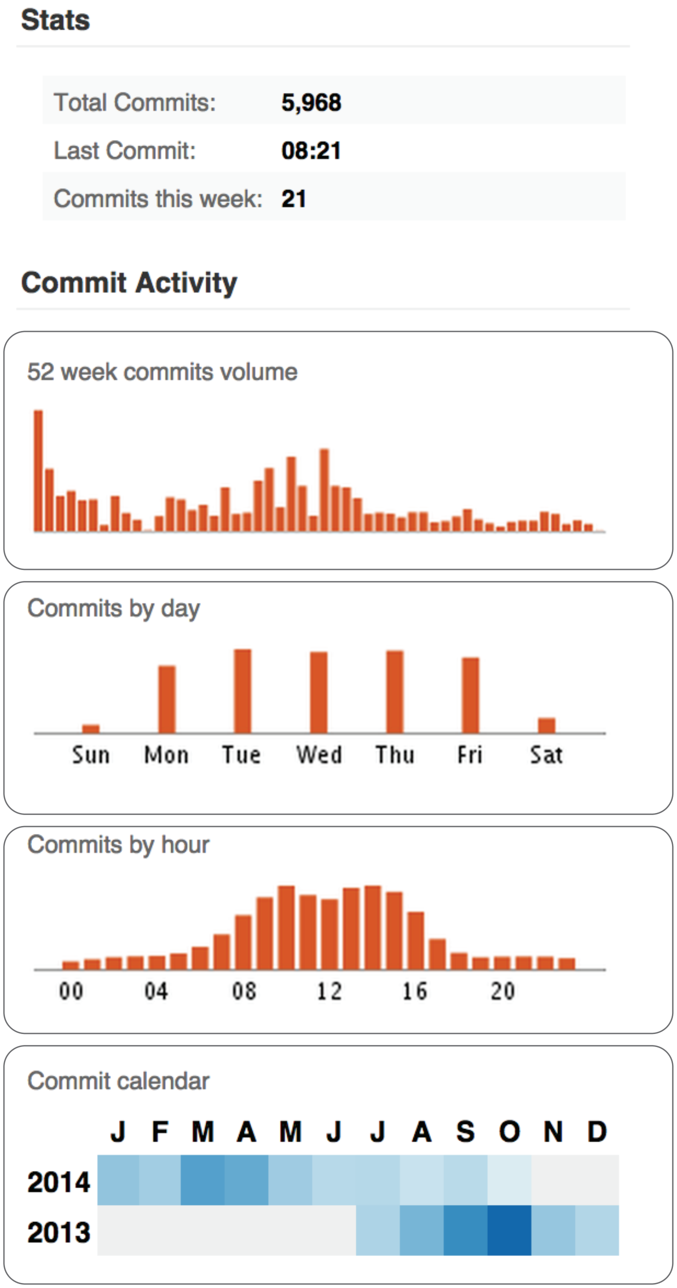

Most work done from 9 am to 5 pm, Monday through Friday. That's pretty normal.

Figure 4.14 The SCM punch card chart shows the days when the most change in your code occurs.



Stats

Total Commits:	5,968
Last Commit:	08:21
Commits this week:	21

Commit Activity

Commits by different frequencies; still only counting LOC

Figure 4.15 SCM data rendered through FishEye

If you're using SVN, then you'll have to use a third-party tool that gets the data from your VCS and displays it for you. Figure 4.15 shows the previous data as visualized by FishEye (www.atlassian.com/software/fisheye).

A note about FishEye

If you're using centralized SCM, specifically SVN, then FishEye is the standard commercial tool used to depict SCM data. There are also open source tools that you can get to run these stats locally, but FishEye is a standard commercial off-the-shelf (COTS) product that teams use to get this data into pretty charts on web pages that everyone can see.

If you want to get this data from the API to show your own graphs, check out the Repository Statistics API: GET /repos/:owner/:repo/stats/contributors.[1] This will return data in the structure shown in figure 4.16.

If you're starting with the basics, this is a pretty convenient API to work with.

4.4 Key SCM metrics: spotting trends in your data

Instead of focusing on LOC, we're going to look at some of the richer data you can get if your team is submitting pull requests and using CI to build and deploy code. We're going to combine that with the data we're already looking at from chapter 3.

We'll look at the following data from our SCM:

- Pull requests
- Denied pull requests
- Merged pull requests
- Commits
- Reviews

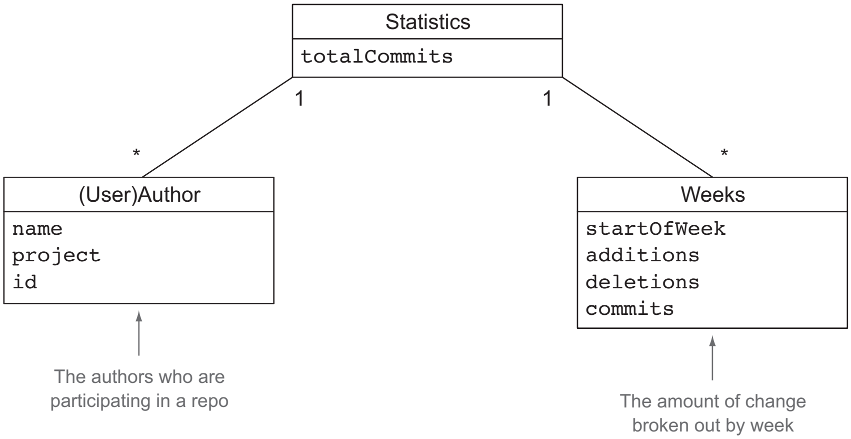


Figure 4.16 GitHub Repository Statistics data structure

[1] "Get contributors list with additions, deletions, and commit counts," developer.github.com/v3/repos/statistics/#contributors.

- Comments
- CLOC (helps calculate risk)

4.4.1 Charting SCM activity

If you're using standard SCM systems, then you probably already have the basic LOC stats that we discussed earlier. To get some information that can really help you, we'll start off by looking at pull request data to see what that tells you.

If you start off with something simple like the number of pull requests, you simply have to count the requests.

As shown in figure 4.17, if all is well you should have a lot more comments than pull requests. This delta is dependent on how you do pull requests on your team. If you have a team with four to six developers, you may open the pull request to the entire team. In this case you'll probably see something like two to three members of the team commenting on pull requests. If you have a bigger team, then opening the pull request to the entire team may be a bit heavy-handed, so a developer may open it to just four to six people with the intention of getting feedback from two or three peers. However you do it, you should see comments parallel pull requests, and you should see comments and reviews be at least double the value of the number of pull requests, as you saw in figure 4.17.

One thing to note is that figure 4.17 is aggregating data every sprint. That means you're seeing the total pull requests and the total comments after the sprint period (usually two to three weeks) is complete. If you're practicing Kanban and/or tracking these stats with greater granularity, then another trend you might want to look for is the difference between when pull requests are open, when comments are added, and when commits are finalized. Ideally you want pull requests to be committed soon after they're submitted. If for some reason you see the trend shown in figure 4.18, you may have an issue.

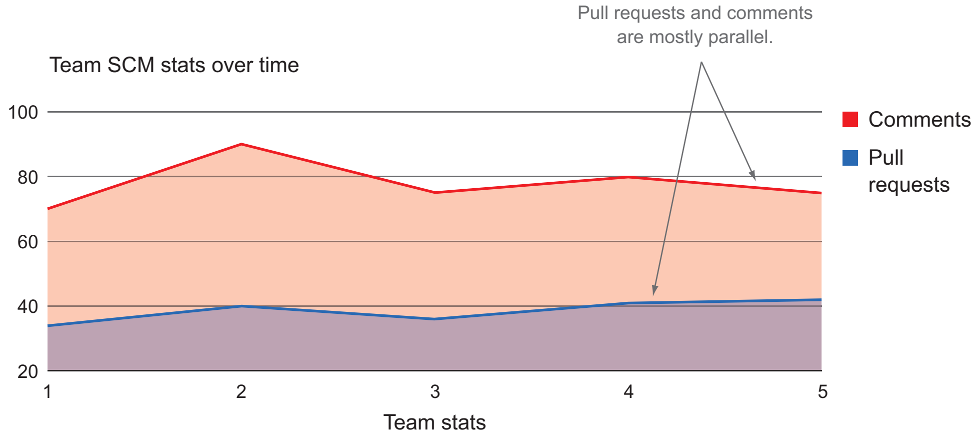


Figure 4.17 Pull requests charted with pull request comments

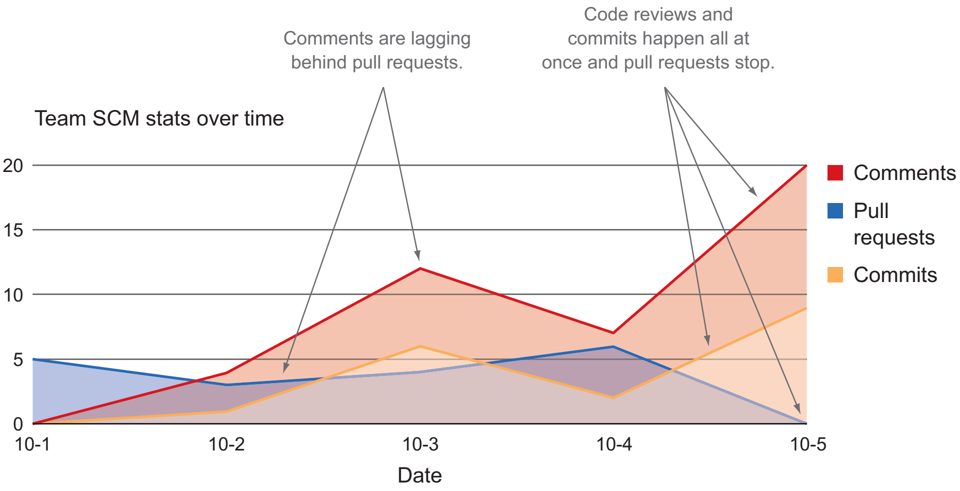


Figure 4.18 Commits and comments lagging behind pull requests is usually a bad pattern.

For some reason your team isn't doing code reviews when they're submitted. This increases the time it takes to complete issues. The biggest danger in leaving code reviews open is that after a while they lose context or developers will be pressured to close the issues and move them along. Ideally you want that gap to be as short as possible.

Depending on the SCM system, you can also see some pull requests denied. This is an interesting data point as a check to your pull request quality. Typically even the best team should have pull requests denied due to mistakes of some kind—everyone makes mistakes. You can add denied pull requests to your chart and expect to see a small percentage of them as a sign of a healthy process. If your denied pull requests are high, that's likely a sign that either the team isn't working well together or you have a bunch of developers trying to check in bad code. If you don't have any denied pull requests, that's usually a sign that code reviewers aren't doing their jobs and are approving pull requests to push them through the process.

4.5 Case study: moving to the pull request workflow and incorporating quality engineering

We've been through how to collect and analyze SCM data, so it's time to show it in action. In the case study that follows, we demonstrate how to move to the pull request workflow and integrate quality engineers on the team for better software quality. This real-world scenario shows you to apply this to your process.

The team in question was generating a lot of bugs. Many of them were small issues that should have been caught much earlier in the development process. Regardless, every time the team cut a release they would deploy their code, turn it over to the quality management (QM) team, and wait for the bugs to come in. They decided to make a change to improve quality.

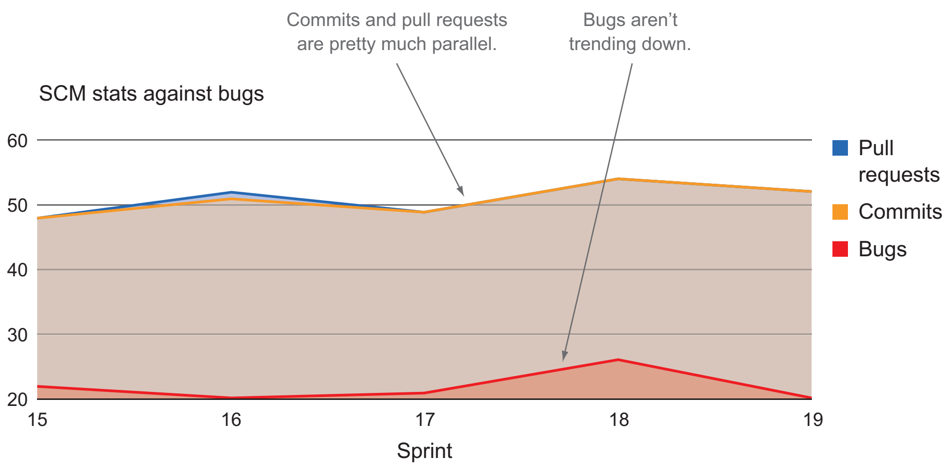


Figure 4.19 Bugs aren't trending down as the team starts doing pull requests.

After discussing the issue, the team decided to try out the pull request workflow. They were already using Git, but the developers were committing their code to a branch and merging it all into the master before cutting a release. They decided to start tracking commits, pull requests, and bugs to see if using pull requests decreased their bug count. After a few sprints they produced the graph shown in figure 4.19.

To make trends a bit easier to see, we divided the pull requests and commits by two so there wasn't such a discrepancy between the metrics. The result is shown in figure 4.20.

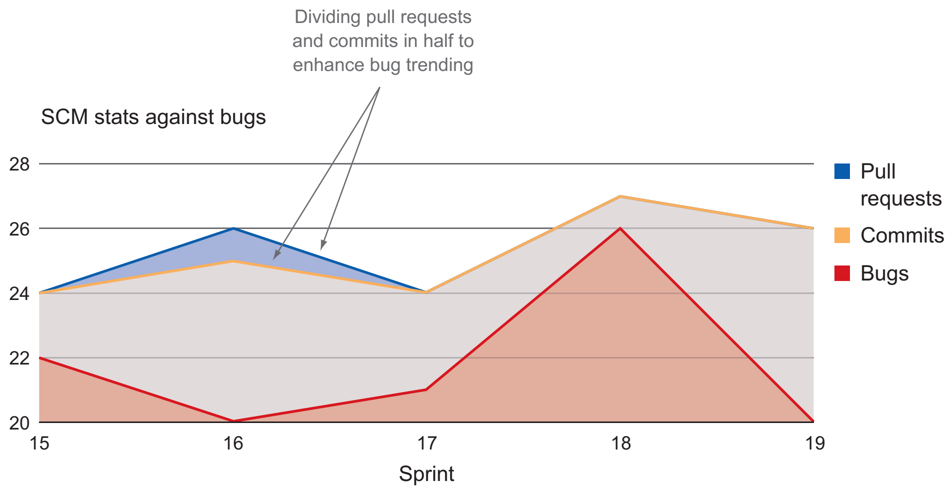


Figure 4.20 The same data with variance decreases between bugs and other data

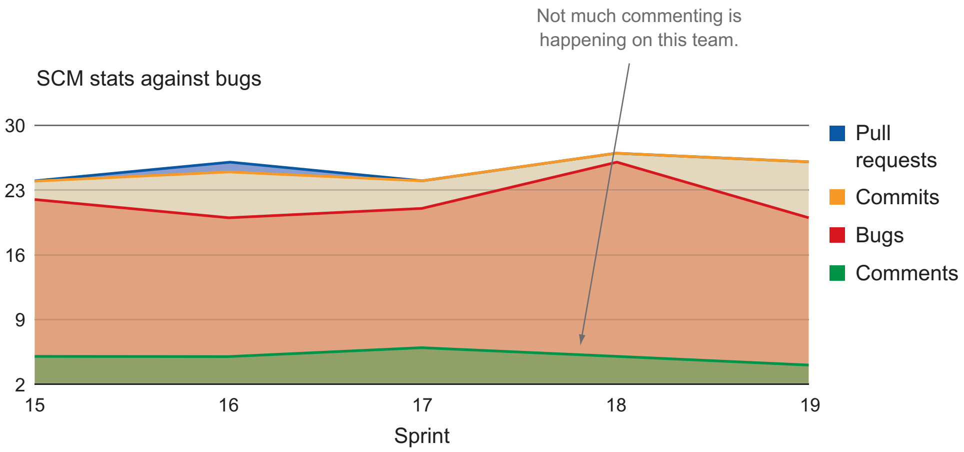


Figure 4.21 Adding comments to the graph and finding an ugly trend

That makes it a lot easier to see the variance. As you can tell from figure 4.20, there wasn't much change. Even though there's a big dip in bugs from sprint 18 to 19, bugs weren't decreasing over time; there was just a big jump in bugs in sprint 18. After discussing the situation, the team decided to add more data points to the mix. To see how much collaboration was happening in the pull requests, they began adding comments to their graphs. That resulted in the chart shown in figure 4.21. To keep things consistent, we divided comments by two.

Figure 4.21 shows that there weren't many comments along with the pull requests, which implies there wasn't much collaboration going on. Because the bug trend wasn't changing, it looked like the changes to their process weren't having any effect yet. The workflow itself wasn't producing the change the development team wanted; they needed to make a bigger impact on their process. To do this they decided that developers should act like the QM team when they were put on a pull request. The perspective they needed wasn't just "is this code going to solve the problem?" but "is this code well built and what can go wrong with it?" There was some concern about developers accomplishing less if they had to spend a lot of time commenting on other developers' code and acting like the QM team. They moved one of their QM members over to the development team as a coach, and the team agreed that if this would result in fewer bugs then the effort spent up front was time well spent. They started taking the time to comment on each other's code and ended up iterating quite a bit more on tasks before checking them in. A few sprints of this process resulted in the graph shown in figure 4.22.

Figure 4.22 shows that as collaboration between development and quality increased—in this case shown through comments in pull requests—the number of bugs went down. This was great news for the team, so they decided to take the process

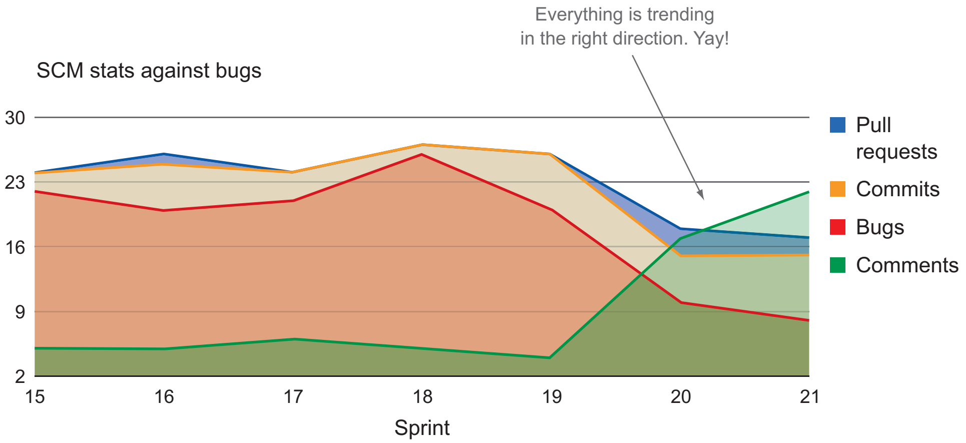


Figure 4.22 Everything is trending in the right direction!

one step further. The development managers brought in another member of the QM team to work with the developers on code reviews and quality checks to avoid throwing code over the wall to the whole QM team.

Test engineers

For a long time the role of the quality department in software engineering has involved checking to make sure features were implemented to spec. That's not an engineering discipline, and as a result many people in the QA/QM space were not engineers. To truly have an autonomous team, quality engineering has to be a significant part of the team. The role of the quality engineer, a.k.a. QE, a.k.a. SDET, a.k.a. test engineer, has started to become more and more popular. But as quality moves from one state to another in the world of software engineering, this role isn't clearly defined, and often you have either someone with an old quality background who recently learned to write code or you have an expert in testing technology. Neither of these works; you need a senior engineer with a quality mindset. This topic could fill another book, so we'll leave it at that.

Over time, commits and pull requests started increasing as well. As the development team started thinking with a quality mindset, they started writing better code and producing fewer bugs. The combined QM and development teams found and fixed many issues before deploying their code to the test environment.

4.6 Summary

Source control is where your code is being written and reviewed and is a great source to complement PTS data for better insight into how your team is operating. Using the pull request workflow and distributed version control can give you a lot more data than non-distributed SCMs. Often web-based DVCS systems, like GitHub, will have built-in charts and graphs you can use to get a picture of how your team is using them.

- Teams use source control management to manage their codebase.
- Here are some things you can learn from SCM data alone:
  - Who is changing the code?
  - How much change is happening in the codebase?
- Here are some questions you can answer with just SCM data:
  - Who is working on what?
  - Who is helping whom?
  - How much effort is going into the work?
- Use pull requests with DVCS to obtain the richest data out of your SCM.
- Look for these key trends from SCM:
  - Relationship between pull requests, commits, and comments.
  - Denied pull requests versus merged pull requests.
  - CLOC over time.
  - SCM data against PTS data to see how they affect each other.
- DVCSs are superior to centralized ones for a variety of reasons:
  - They provide much richer data than centralized SCM systems.
  - They can improve the development process by using recommended flows.
  - They tend to have RESTful APIs for easier data collecting.
- Pull requests combined with comments and code reviews add another dimension to the team's collaborative process.
- GitHub with its Pulse and Graph tabs contains a lot of useful information about the health of the project.
- Visualization techniques are available for centralized VCS through commercial products.

5 Trends and data from CI and deployment servers

This chapter covers

- What can you learn from your CI systems, CT, and CD data alone
- How to utilize your CI to get the richest data possible
- How the addition of data from CI and deployment servers enhances and fills gaps in PTS and SCM data

Now that we've been through project tracking and source control, we're moving to the next step, which is your CI system. In the scope of our application lifecycle, CI is highlighted in figure 5.1.

Your CI server is where you build your code, run tests, and stage your final artifacts, and, in some cases, even deploy them. Where there are common workflows when it comes to task management and source control, CI tends to vary greatly from team to team. Depending on what you're building, how you're building it is the biggest variable at play here.

Before we dive into collecting data, we'll talk about the elements of continuous development so you know what to look for and what you can utilize from your

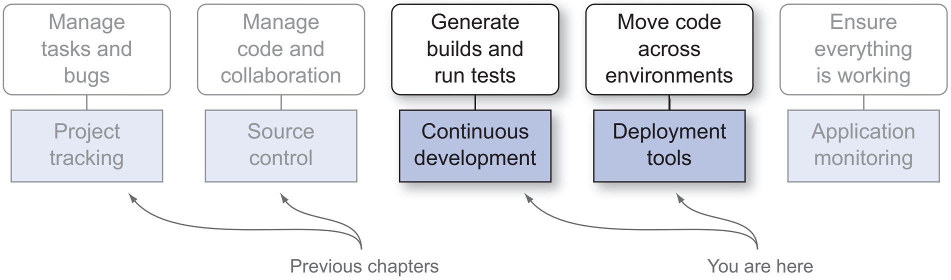


Figure 5.1 You are here: continuous integration in the application lifecycle.

development process. If we revise the questions we asked in chapters 3 and 4 a bit, we get the types of *who*, *what*, and *when* shown in figure 5.2 through data from CI.

Here are questions you can answer from your CI systems:

- How fast are you delivering changes to your consumer?
- How fast can you deliver changes to your consumer?
- How consistently does your team do their work?
- Are you producing good code?

If you combine CI data with your PTS data from chapter 3 and SCM data from chapter 4, you can get some really interesting insights with these questions:

- Are your tasks appropriately sized?
- Are your estimates accurate?
- How much is your team actually getting done?

We're going to take a much closer look at combining data in chapter 7. For now, we'll make sure you're clear on what continuous (fill in the blank) means.

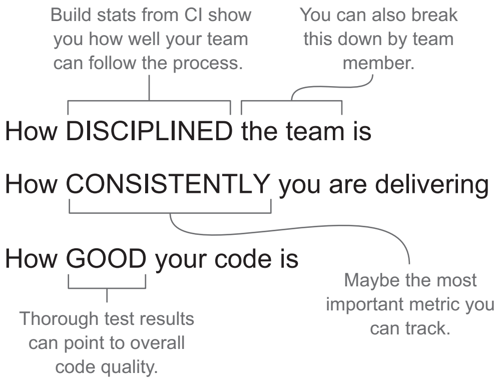

Build stats from CI show you how well your team can follow the process.
You can also break this down by team member.
How DISCIPLINED the team is
How CONSISTENTLY you are delivering
How GOOD your code is
Maybe the most important metric you can track.
Thorough test results can point to overall code quality.

Figure 5.2 The *hows* from CI: the basic questions you can answer

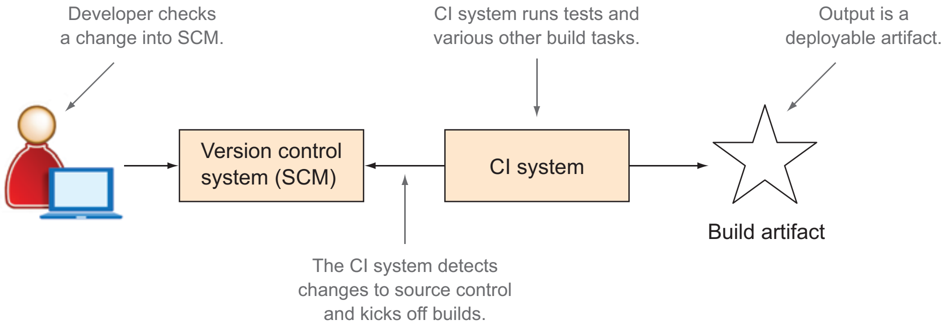


Figure 5.3 The simplest possible CI pipeline. When developers check in changes, the CI system builds an artifact.

5.1 What is continuous development?

In today's digital world consumers expect the software they interact with every day to continuously improve. Mobile devices and web interfaces are ubiquitous and are evolving so rapidly that the average consumer of data expects interfaces to continually be updated and improved. To be able to provide your consumers the most competitive products, the development world has adapted by designing deployment systems that continuously integrate, test, and deploy changes. When used to their full potential, continuous practices allow development teams to hone their consumer's experience multiple times a day.

5.1.1 Continuous integration

Continuous development starts with CI, the practice of continuously building and testing your code as multiple team members update it. The simplest possible CI pipeline is shown in figure 5.3.

In theory, CI allows teams to collaborate on the same codebase without stepping on each other's toes. SCM provides the source of truth for your codebase; multiple developers can work on the same software product at the same time and be aware of each other's changes. The CI system takes one or more of the changes that are made in SCM and runs a build script that at its simplest will ensure that the code you're working on compiles. But CI systems are basically servers that are designed to run multiple jobs as defined in the build script for an application and therefore are capable of extremely flexible and potentially complex flows. Typical CI jobs include running tests, packaging multiple modules, and copying artifacts.

Common CI systems include Jenkins (jenkins-ci.org/), Hudson (hudson-ci.org/), Bamboo (atlassian.com/software/bamboo), TeamCity (www.jetbrains.com/teamcity/), and Travis CI (travis-ci.org/). Although each has slightly different features, they all do the same thing, which is take a change set from your SCM system, run any build scripts you have in your codebase, and output the result somewhere.

A bit about Jenkins

Jenkins, formerly called Hudson, is an open source CI system that is very popular, has a plethora of plugins for additional functionality, is extremely easy to set up and administrate, and has a very active community. When Hudson became commercial software, the open source version continued to evolve as Jenkins. Because Jenkins is by far the most popular build system, we'll be using it for our examples in this book. If you're not familiar with it and want more information, check out the Jenkins home page: jenkins-ci.org/. Note that everything we do with Jenkins in this book could easily be applied to other CI systems.

Build scripts are the brain of your CI process. A build script is code that tells your CI system how to compile your software, put it together, and package it. Your build script is essentially the instructions that your CI system should follow when it integrates code. Common build frameworks are Ant, Nant, Ivy, Maven, and Gradle. Build scripts are powerful tools that manage dependencies, determine when to run tests and what tests to run, and check for certain conditions in a build to determine whether it should continue or stop and return an error. Figure 5.4 shows graphically what a typical build script will control.

Think of your build script as a set of instructions that describes how to build and package your code into something you can deploy.

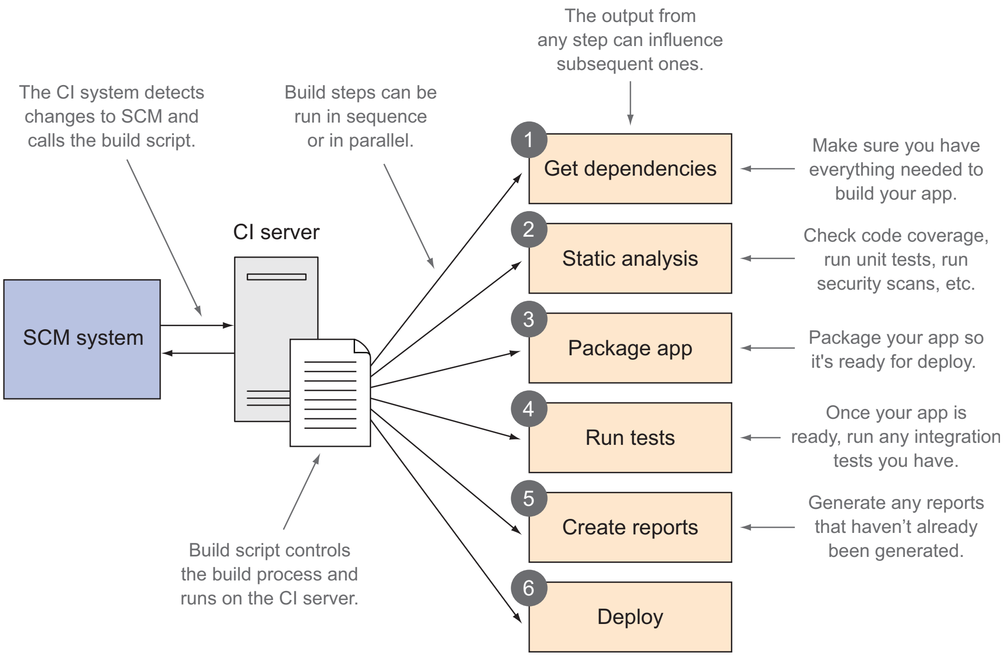

The output from any step can influence subsequent ones.
The CI system detects changes to SCM and calls the build script.
Build steps can be run in sequence or in parallel.
CI server
SCM system
1 Get dependencies
2 Static analysis
3 Package app
4 Run tests
5 Create reports
6 Deploy
Make sure you have everything needed to build your app.
Check code coverage, run unit tests, run security scans, etc.
Package your app so it's ready for deploy.
Once your app is ready, run any integration tests you have.
Generate any reports that haven't already been generated.
Build script controls the build process and runs on the CI server.

Figure 5.4 A build script running on a CI server controlling a multistep build process

5.1.2 Continuous delivery

Interestingly enough, the very first principle in the Agile Manifesto calls out "continuous delivery."

> *"Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."*[1]
>
> —Principles behind the Agile Manifesto

The term *continuous delivery* was thoroughly explored by Jez Humble and David Farley in *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* (Addison-Wesley, 2010). CD builds on CI by taking the step of orchestrating multiple builds, coordinating different levels of automated testing, and in advanced cases moving the code your CI system built into a production environment for your consumers. Although you can do CD with CI systems, there are specialized CI systems that make CD a bit easier. The idea is built on a few facts:

- Consumers demand and expect constant improvement to their products.
- It's easier to support a bunch of small changes over time than large change sets all at once.
- It's easier to track the value of change when it's small and targeted.

Examples of CD systems are Go (www.go.cd), Electric Cloud (electric-cloud.com), Ansible (www.ansible.com/continuous-delivery), and Octopus (octopusdeploy.com). Some of these systems can be used for CI as well but have additional capabilities to coordinate multiple parts of a build chain. This becomes particularly useful when you have complex build chains such as the one in figure 5.5.

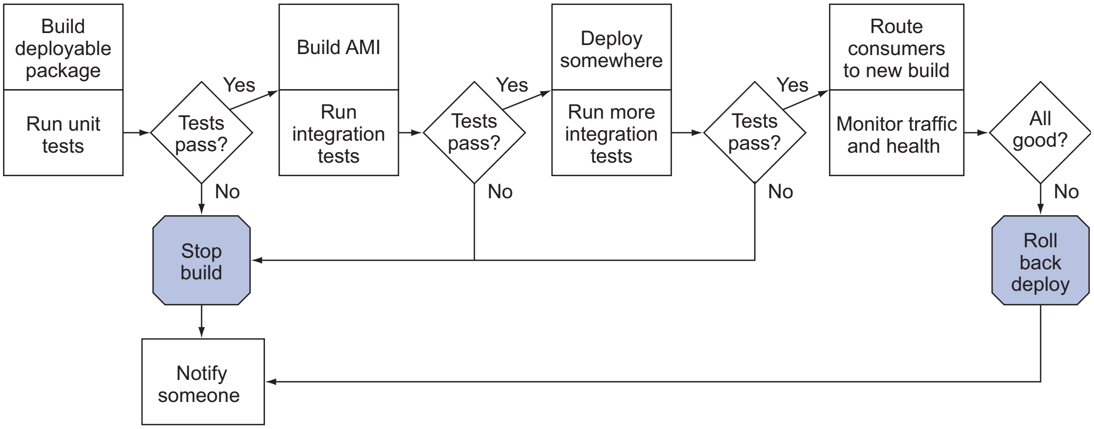


Figure 5.5 An example of a complex build chain

[1] "We follow these principles," agilemanifesto.org/principles.html.

In figure 5.5 getting code all the way to production is completely automated and relies heavily on testing at every stage of the deployment. It starts off with unit tests, builds the machine image that will get deployed, handles the deployment, and makes sure nothing breaks when it finally makes it to the consumer. Complexity and orchestration like this are what make the difference in using a tool for CI or CD.

Moving to continuous delivery

Although most software projects have CI, getting your team to CD is first a cultural shift. Having your entire team understand and accept that small changes will be deployed to your consumer automatically and frequently can be a bit scary for teams that release even as frequently as every few weeks. Once you come to grips with the fact that you can change your consumer's experience multiple times a day, you're likely going to have to make significant changes to your build systems to accommodate the types of automation you'll need. You'll also likely need to make big changes in how you think about testing, because all testing in a CD world should rely on trustworthy automation rather than manual button pushing. What should be obvious (but rarely is) is to get the most out of CD you should have a good sense of why you're doing it and how you're monitoring the success of your changes. Fortunately, if you're implementing what you're learning in this book, monitoring should be the easy part.

5.1.3 Continuous testing

Continuous testing (CT) is also a part of CI; it's the practice of having tests continuously running on your codebase as you make changes to it. Teams that take the time to automate their testing process, write tests while they're writing their code, and run those tests constantly throughout the build process and on their local development environments end up with another rich set of data to use in the continuous improvement process.

To clarify where this testing happens and what this testing is, we take another look at figure 5.5 but highlight the testing in figure 5.6.

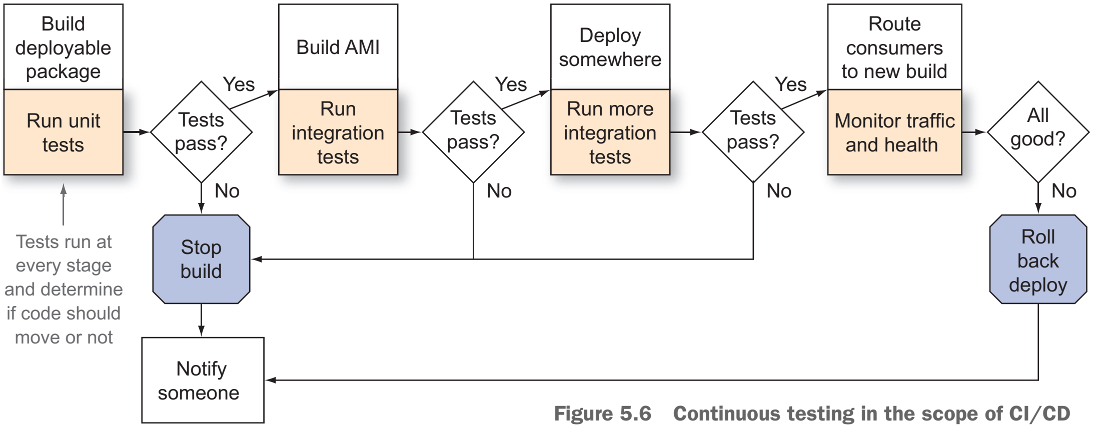


Figure 5.6 Continuous testing in the scope of CI/CD

Figure 5.6 highlights the fact that every stage in the CD pipeline relies on automated tests (CT) to inform the decision on moving to the next stage in the deployment. If this data is rich enough to inform a decision on moving through the automated deployment process, it can also provide good insight into how your team is operating. A couple of things that this can help point to, especially when paired with VCS, are:

- *Specific trouble spots in your code that are slowing you down*—By identifying sections of your codebase that have significant technical debt, you can anticipate updating those sections to take longer than expected. You can use this information to adjust estimates or set expectations with stakeholders.
- *Your team's commitment to automation*—Having great automation is key to allowing your team to move faster and get changes out to consumers. Gauging this helps determine the potential delivery speed of your team.

Getting data from CT is as easy as tapping into the test results that are published in your CI system because the CI/CD system publishes and parses these results.

5.2 Preparing for analysis: generate the richest set of data you can

The current health of your build and the build history are default data points your CI system provides. You can also get results from test runs or whatever build steps you define. But if you want to answer more interesting questions like the ones I posed at the beginning of the chapter like

- How fast can you deliver code to the consumer?
- Are your estimations accurate?
- Are you getting tasks done right the first time?
- How much is your team actually getting done?

then you should use the following tips to ensure you're generating the richest set of data you can.

5.2.1 Set up a delivery pipeline

A delivery pipeline goes beyond your build scripts. Once your product is tested, built, and staged, other coordination typically has to happen. You may need to kick off tests for dependent systems, deploy code with environment-specific parameters based on test results, or combine parallel builds of different parts of your larger system for a complete deployment. Automating more complex build and deploy scenarios can be tricky and is exactly what your pipeline specializes in. By automating your complex build process, you can get data out of your pipeline that tells you how fast you're delivering products to your consumers. When you break down that data point, you can also find where you can make the most improvement to deliver products faster and more efficiently.

If you have a CI system in place, that's great; everything you did in the last section should just plug in and start generating reports for you. If you don't, we recommend taking a look at Jenkins or GoCD (www.go.cd/) for your pipeline. GoCD is a CI system like Jenkins or TeamCity, and all of them are capable of building your code, running your tests, and generating reports that you can bring into your metrics collection system. The cool thing about GoCD is the idea of a pipeline as a first-class citizen, so breaking your build/deploy into steps that are easy to visualize is completely standard and normal. Jenkins is extremely popular and can do pipeline management using plugins.

Once your pipeline is set up you should consider the following:

- Using SonarQube (www.sonarqube.org) for static analysis gives you a very rich data point on the quality of your code.
- If you can't use SonarQube, tools you can build into your build process include Cobertura (cobertura.github.io/cobertura/), JaCoCo (www.eclemma.org/jacoco), or NCover (www.ncover.com/).
- A standard test framework that gives you reports that are easily digestible is TestNG (testng.org/doc/index.html), which uses ReportNG (reportng.uncommons.org) for generating useful reports in your build system that are available through an API.

Using these technologies in your pipeline will give you better insight into the quality of your code and help you find potential problems before they affect your consumers. This insight into quality will also give you insight into your development process.

5.3 The data you'll be working with: what you can get from your CI APIs

Because CI server data is so flexible, the data you can get out of it depends greatly on how you set up your system. For simple builds it's possible to get very little data, and for more complex builds it's possible to get a wealth of data on quality, build time, and release information. If you set up your CI system with data collection in mind, you can get great insight into your development cycle.

5.3.1 The data you can get from your CI server

Your build scripts define how your project is being built and what happens throughout your build process. In your build scripts you can create reports that are published through your CI system that give you details on every step you report on. The most common build servers have great APIs, which should be no surprise because they're at the heart of automation. If you use Jenkins, you can communicate completely through REST simply by putting /api/json?pretty=true at the end of any URL that you can access. The following listing shows some of the data you'd get back from the main Jenkins dashboard by examining Apache's build server. For the entire response you can look at Apache's site: builds.apache.org/api/json?pretty=true.

Listing 5.1 Partial response from the Jenkins dashboard for Apache's build server

```
{
"assignedLabels" : [
{
}
```

```
],
"mode" : "EXCLUSIVE",
"nodeDescription" : "the master Jenkins node",
"nodeName" : "",
"numExecutors" : 0,
"description" : "<a href=\"http://www.apache.org/\"><img
     src=\"https://www.apache.org/images/asf_logo_wide.gif\"></img></a>\r\n<p>\r\
     nThis is a public build and test server for <a
     href=\"http://projects.apache.org/\">projects</a> of the\r\n<a
     href=\"http://www.apache.org/\">Apache Software Foundation</a>.
     All times on this server are UTC.\r\n</p>\r\n<p>\r\nSee the <a
     href=\"http://wiki.apache.org/general/Jenkins\">Jenkins wiki page</a> for
     more information\r\nabout this service.\r\n</p>",
"jobs" : [
{
"name" : "Abdera-trunk",
"url" : "https://builds.apache.org/job/Abdera-trunk/",
"color" : "blue"
},
{
"name" : "Abdera2-trunk",
"url" : "https://builds.apache.org/job/Abdera2-trunk/",
"color" : "blue"
}
…
```

General information about your build server

```
    "mode": "EXCLUSIVE",
    "nodeDescription": "the master Jenkins node",
    "nodeName": "",
    "numExecutors": 0,
    "description": "This is a public build and test server for projects of
     the Apache Software Foundation”,
```

Lists jobs available through this URL

```
    "jobs": [
        {
            "name": "Abdera-trunk",
            "url": "https://builds.apache.org/job/Abdera-trunk/",
            "color": "blue"
        },
        {
            "name": "Abdera2-trunk",
            "url": "https://builds.apache.org/job/Abdera2-trunk/",
            "color": "blue"
        },
…
```

One key piece of data is missing from here that we've been depending on from every other data source we've looked at so far; did you notice it? Dates are missing! That may seem to put a kink in things if you're talking about collecting data and mapping it over time, but don't worry; if you dig deeply enough you'll find them. Jenkins assumes that when you hit a URL you know the current date and time. If you're going to take this data and save it somewhere, you can simply schedule the data collection at the frequency at which you want to collect data and add the dates to the database yourself. Additionally, if you look for something that references a specific time, you can get the data from that if you want. The next listing shows data from a specific build that has a date/time in the response.

Listing 5.2 Jenkins response from a specific build

```
{
    "actions": [
        {},
        {
            "causes": [
                {
                    "shortDescription": "[URLTrigger] A change within the
                     response URL invocation (<a href=\"triggerCauseAction\
                     ">log</a>)"
                }
            ]
        },
        {},
        {
            "buildsByBranchName": {
                "origin/master": {
                    "buildNumber": 1974,
                    "buildResult": null,
                    "marked": {
                        "SHA1": "54ad73e1adb22fd84fdd1dfb5c28175f743d1960",
                        "branch": [
                            {
                                "SHA1":
     "54ad73e1adb22fd84fdd1dfb5c28175f743d1960",
                                "name": "origin/master"
                            }
                        ]
                    },
                    "revision": {
                        "SHA1": "54ad73e1adb22fd84fdd1dfb5c28175f743d1960",
                        "branch": [
                            {
                                "SHA1":
     "54ad73e1adb22fd84fdd1dfb5c28175f743d1960",
                                "name": "origin/master"
                            }
                        ]
                    }
                }
            },
            "lastBuiltRevision": {
                "SHA1": "54ad73e1adb22fd84fdd1dfb5c28175f743d1960",
                "branch": [
                    {
                        "SHA1": "54ad73e1adb22fd84fdd1dfb5c28175f743d1960",
                        "name": "origin/master"
                    }
                ]
            },
```

An array of actions from this build

Details about what code is getting built

```
            "remoteUrls": [
                "https://git-wip-us.apache.org/repos/asf/mesos.git"
            ],
            "scmName": ""
        },
        {},
        {},
        {}
    ],
    "artifacts": [],
    "building": false,
    "description": null,
    "duration": 3056125,
    "estimatedDuration": 2568862,
    "executor": null,
    "fullDisplayName": "mesos-reviewbot #1974",
    "id": "2014-10-12_03-11-40",
    "keepLog": false,
    "number": 1974,
    "result": "SUCCESS",
    "timestamp": 1413083500974,
    "url": "https://builds.apache.org/job/mesos-reviewbot/1974/",
    "builtOn": "ubuntu-5",
    "changeSet": {
        "items": [],
        "kind": "git"
    },
    "culprits": []
}
```

Links back to the Git code repository

Lists generated artifacts

Is this building now?

How long did this build take?

How long should a build take?

Generated build number

Did the build pass or fail?

When did this build complete?

Ah, there's that date! If you're collecting data moving forward, then it's not a big deal, but if you want to find trends from the past, you'll have to do a bit more querying to get what you want.

What you want from CI is the frequency at which your builds are good or bad. This data will give you a pretty good indication of the overall health of your project. If builds are failing most of the time, then a whole host of things can be going wrong, but suffice it to say that something is going wrong. If your builds are always good, that's usually a good sign, but it doesn't mean that everything is working perfectly. This metric is interesting by itself but really adds value to data you collect from the rest of your application lifecycle.

The other interesting data you can get from CI is the data you generate yourself. Because you control the build system, you can publish pretty much anything you want during the build process and bring that data down to your analytics. These reports can answer the question "Are you writing good code?" Here are some examples of tools and frameworks you can use to generate reports:

- TestNG—Use it to run many test types; ReportNG is its sister reporting format.
- SonarQube—Run it to get reports including code coverage, dependency analysis, and code rule analysis.
- Gatling—Has rich reporting capabilities for performance benchmarking.
- Cucumber—Use it for BDD-style tests.

Let's take a closer look at some of these test frameworks and the data that they provide.

TESTNG/REPORTNG

TestNG is a popular test framework that can be used to run unit or integration tests. ReportNG formats TestNG results as reports that are easy to read and easy to interface with through XML or JSON. These reports give you the number of tests run, passed, and failed and the time it takes for all of your tests. You can also dig into each test run to find out what's causing it to fail.

SONARQUBE AND STATIC ANALYSIS

SonarQube is a powerful tool that can give you a lot of data on your codebase, including how well it's written and how well it's covered by tests. There are books written on SonarQube, so we'll just say you should use it. A good source is *SonarQube in Action* by G. Ann Campbell and Patroklos P. Papapetrou (Manning, 2013; www.manning.com/papapetrou/). We'll talk a lot more about SonarQube in chapter 8 when we're measuring what makes good software.

GATLING

Gatling is a framework used for doing stress testing and benchmarking. You can use it to define user scenarios with a domain-specific language (DSL), change the number of users over the test period, and see how your application performs. This type of testing adds another dimension to the question "Is your software built well?" Static analysis and unit tests can tell you if your code is written correctly, but stress testing tells you how your consumers will experience your product. Using Gatling you can see the response times of your pages under stress, what your error rate looks like, and latency.

BDD-STYLE TESTS

Behavior-driven development (BDD) is a practice in which tests in the form of behaviors are written with a DSL. This makes understanding the impact of failing tests much easier because you can see what scenario your consumer expects that won't work.

At the end of the day the output of your BDD tests will be some form of test results or deployment results, so the reports you add to your build process can answer the following questions:

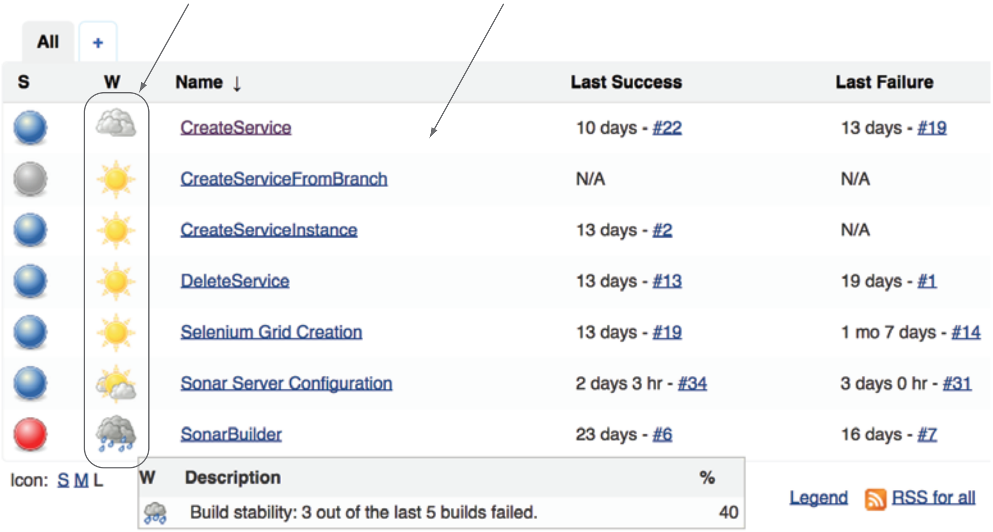
- How well does your code work against how you think it should work?
- How good are your tests?
- How consistent is your deploy process?

All of these questions point to how mature your development process is. Teams with great development processes will have rock-solid, consistent tests with complete test coverage and frequent deploys that consistently add value to the consumers of their software.

If you want to read all about BDD, I recommend *BDD In Action: Behavior-Driven Development for the whole software lifecycle,* by John Ferguson Smart (Manning, 2014; www.manning.com/smart/).

The weather report for a build shows how healthy it is: sunshine is good, clouds and rain are bad.

Each row represents a defined build. Within each build there are multiple jobs over time.



All +

S	W	Name ↓	Last Success	Last Failure
●	☁	CreateService	10 days - #22	13 days - #19
●	☀	CreateServiceFromBranch	N/A	N/A
●	☀	CreateServiceInstance	13 days - #2	N/A
●	☀	DeleteService	13 days - #13	19 days - #1
●	☀	Selenium Grid Creation	13 days - #19	1 mo 7 days - #14
●	⛅	Sonar Server Configuration	2 days 3 hr - #34	3 days 0 hr - #31
●	🌧	SonarBuilder	23 days - #6	16 days - #7

Icon: S M L

W	Description	%
🌧	Build stability: 3 out of the last 5 builds failed.	40

Legend RSS for all

Figure 5.7 The weather report from Jenkins

5.3.2 What you can tell from CI alone

Your CI system is where code is integrated, tests are run (CT happens), and potentially things are deployed across environments (CD). For our purposes we'll show some things from Jenkins, by far the most popular CI environment.

The first report we're going to talk about is the Jenkins dashboard weather report. Because CI runs jobs for you, the first thing you see when you log in is the list of jobs it's responsible for and how healthy they are. On the weather report you'll see that the healthier the build, the sunnier the weather, as shown in figure 5.7.

The weather report tells you how frequently your build is broken or passing. By itself this data is interesting because you ultimately want passing builds so you can get your code into production and in front of your consumers. But it's not so bad to break builds from time to time because that could indicate that your team is taking risks—and calculated risks are good a lot of the time. You definitely don't want unhealthy or stormy builds because that could indicate a host of problems. As with CLOC from SCM, you should combine this with the other data you've collected so far to help get to the root of problems and understand why you're seeing the trends that you are.

You can get a lot more data from your CI system if you're generating reports from the different parts of your build process and publishing results. Earlier we talked about generating reports from your build script; this is where those reports become accessible and useful in the context of CI.

5.4 Key CI metrics: spotting trends in your data

The most basic information you can get from CI is successful and failed builds. Even though these seem like straightforward metrics, there's some ambiguity in interpreting them.

The most obvious problem is if your build fails all or most of the time; if that happens there's obviously a big problem. Conversely, if your build passes all the time, that might mean you have an awesome team, but it could also mean any of the following:

- Your team isn't doing any meaningful work.
- You don't have any tests that are running.
- Quality checks are disabled for your build.

When you run up against this issue, it's not a bad idea to pull a bit more data to sanity-check your build history. In this case you can take a look at the details from your CI builds to explore some of the following:

- Test reports
- Total number of tests
- Percentage of passing and failing tests
- Static analysis
- Test coverage percentage
- Code violations

If you're using your CI system to also handle your deployments, you can get build frequency out of there too, which is really the key metric that everything else should affect: how fast are you getting changes to your consumers?

We'll look at some of the data you can get from your CI system so you can track these metrics along with the rest of the data you're already collecting.

5.4.1 Getting CI data and adding it to your charts

The easiest thing to get out of your CI system is build success frequency. The data you'll look at for this will be

- Successful versus failed builds
- How well is your code review process working?
- How good is your local development environment?
- Is your team thinking about quality software?
- Deploy frequency
- How frequently do you get updates in front of your consumers?

What you can get greatly depends on how you're using your CI system. In our examples we'll look at what you can get if you have a job for integrating code and a job for deploying code.

Starting off with the job that integrates code, we'll look at the difference between successful and failed builds, as shown in figure 5.8.

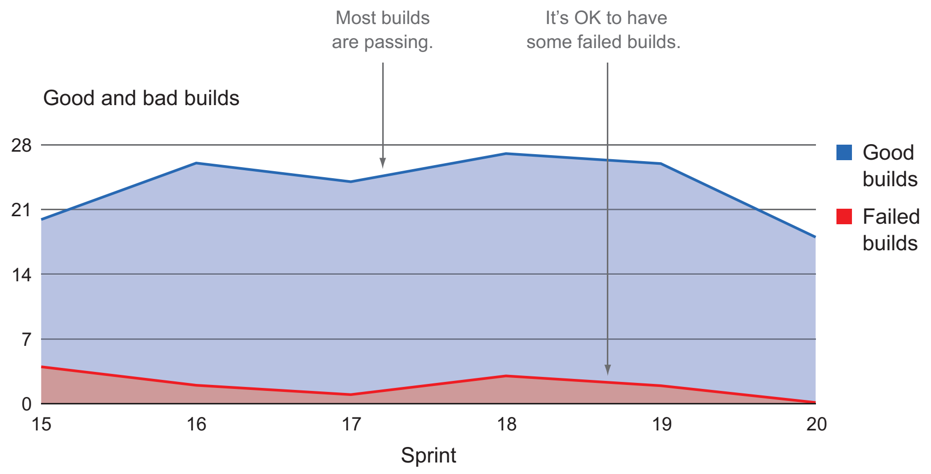


Figure 5.8 A healthy-looking project based on good and bad builds

In figure 5.8 you can see some failed builds; that's okay, especially on complex projects. You can start getting worried when your failed builds start to become a high percentage of your total builds, as shown in figure 5.9; that's usually a signal that something is pretty wrong and you need to dig in.

If figure 5.9 is bad, then you might think something like figure 5.10 is good.

But as we mentioned earlier in the chapter, that may not always be the case. If you pull more data into the picture like test coverage or test runs, as shown in figure 5.11,

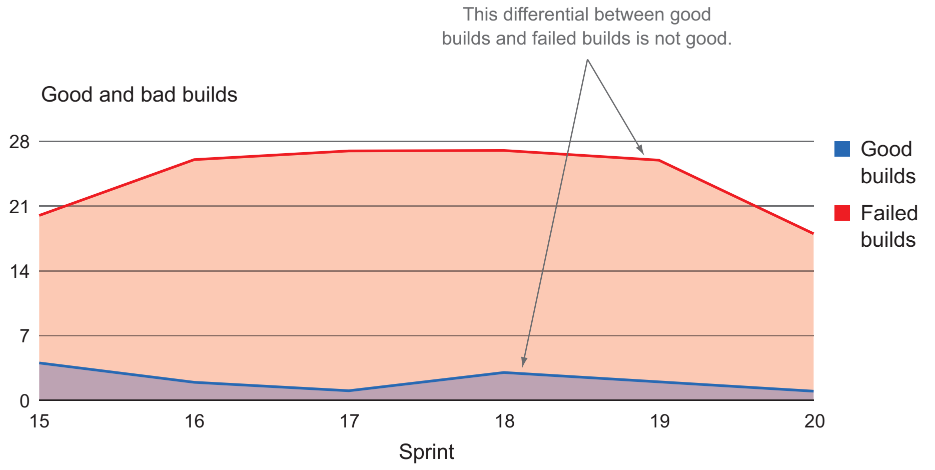


Figure 5.9 A worrisome trend: lots of bad builds

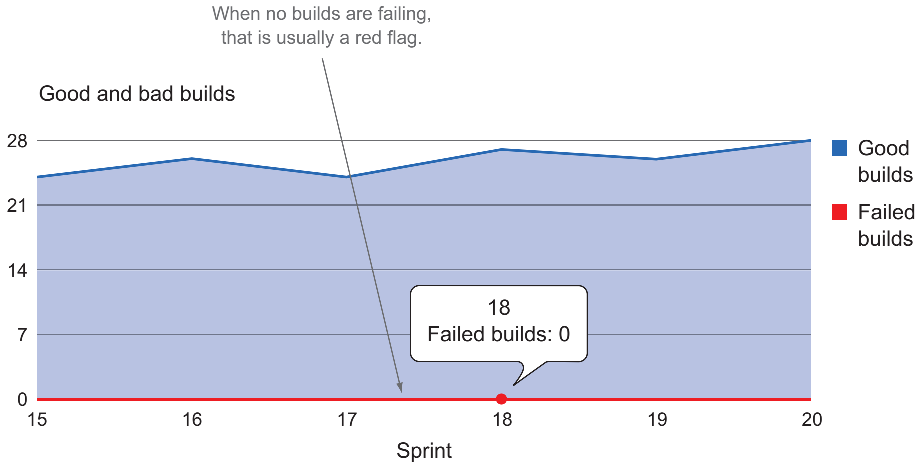


Figure 5.10 It appears that everything is great: no builds are failing.

you can get a sanity check on whether your builds are worthwhile and doing something of value, or if they're just adding one code delta to the next without checking anything.

Figure 5.11 shows a team that's generating good builds but not testing anything. In this case what appeared to be a good trend is actually meaningless. If you're not testing anything, then your builds need only compile to pass. You want to see something more like figure 5.12.

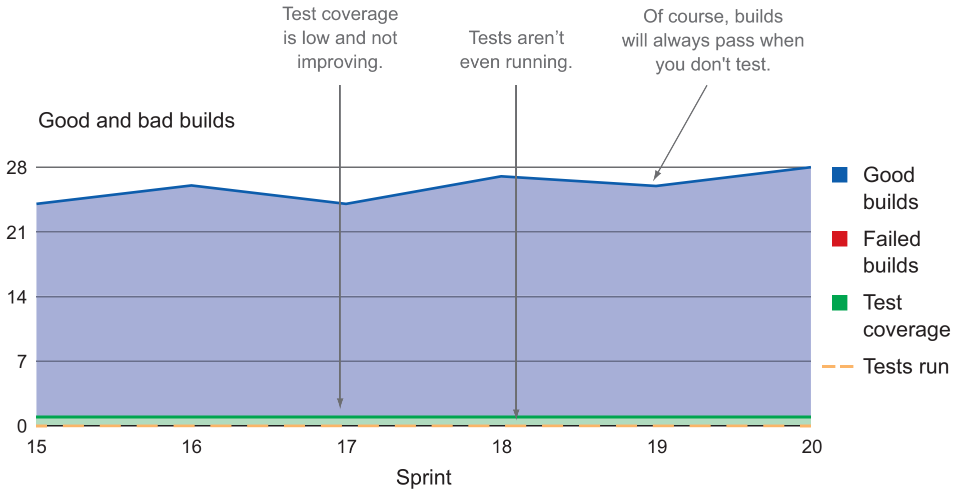


Figure 5.11 This is really bad. Builds always pass but there are no tests running and barely any coverage.

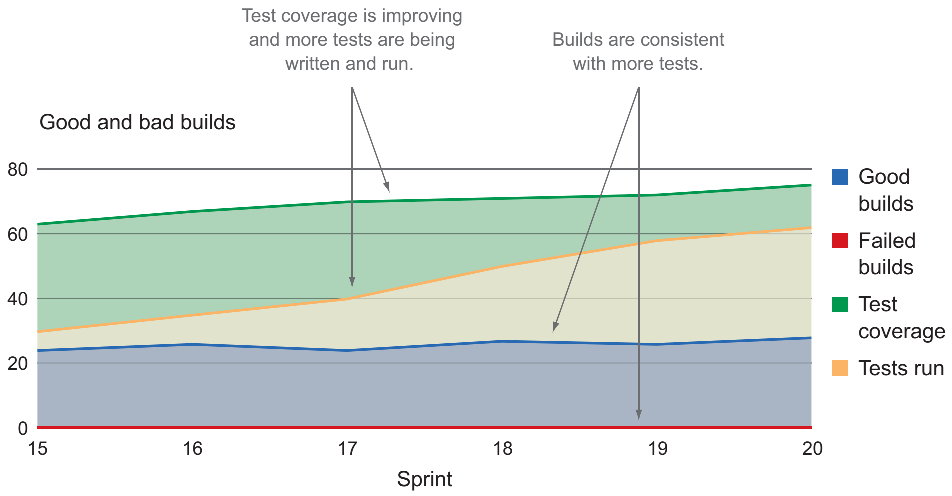


Figure 5.12 There are no failing builds, code coverage is going up, and more tests are running every time. It looks like this team has it together.

Another way to visualize your build data is to look at releases over time. If you're releasing at the end of every sprint or every few sprints, it might be more useful to show release numbers as points on your other graphs, like in figure 5.13.

But if you have an awesome team practicing CD and putting out multiple releases a day, putting each release on the graph would be tough to interpret. Instead, show the number of good releases and bad releases (releases that caused problems or had to be

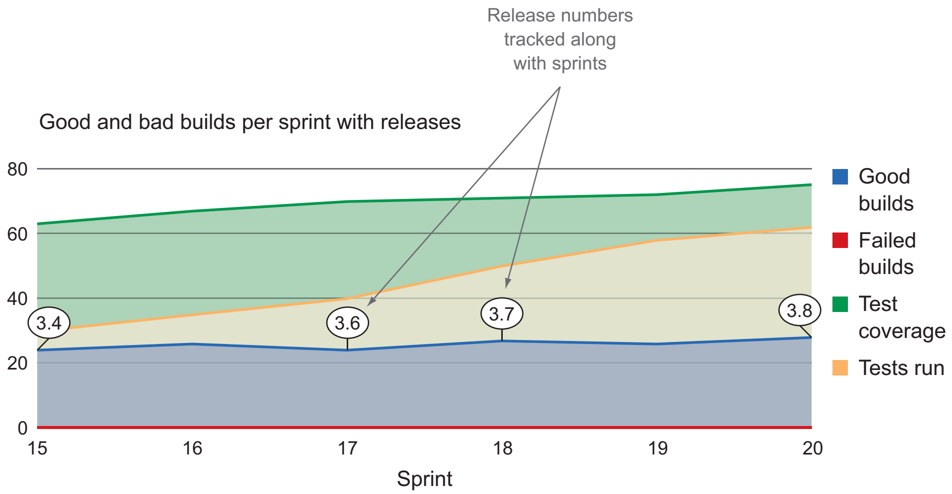
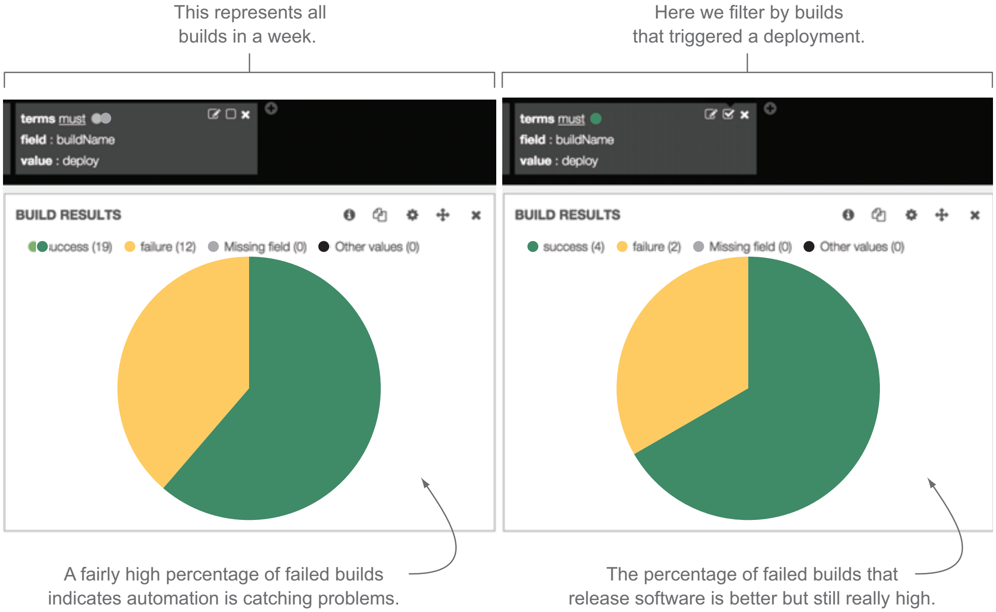


Figure 5.13 Releases charted with other data


terms must
field : buildName
value : deploy
BUILD RESULTS
● uccess (19) ● failure (12) ● Missing field (0) ● Other values (0)
terms must
field : buildName
value : deploy
BUILD RESULTS
● success (4) ● failure (2) ● Missing field (0) ● Other values (0)

This represents all builds in a week.

Here we filter by builds that triggered a deployment.

A fairly high percentage of failed builds indicates automation is catching problems.

The percentage of failed builds that release software is better but still really high.

Figure 5.14 Showing all build pass/fail percentages for a team deploying code multiple times a day and the percentage filtered by builds that trigger a release. By hovering over the charts in Kibana, you can see the percentage associated with the pie slices.

rolled back) along with your other build data. In this case, if your automation is really good you'll see a percentage of builds failing before releases and a much smaller percentage failing that actually release code. An example of this is shown in figure 5.14.

The team depicted in figure 5.14 is building 31 times a week and releasing software 6 times a week. Of their releases 33% fail, and of total builds 38.7% fail. Despite having a high failure rate for releases, the team still has four successful deployments a week, which is pretty good. Now imagine a team that releases software twice a month with a 33% failure rate; that would mean the team would sometimes go for an entire month without releasing anything to their customers. For teams that release daily, the tolerance for failure ends up being much higher due to the higher release frequency. Teams that release every two weeks usually are expected to have a successful deployment rate of close to 100%, a much harder target to hit.

5.5 Case study: measuring benefits of process change through CI data

We ended chapter 4 with a happy team that made some measurements, reacted to them, and had great results. They moved to the pull request workflow on their development team and moved quality into the development process. They were able to measure some key metrics that were being generated from their development process to track progress and use those metrics to prove that the changes they were making

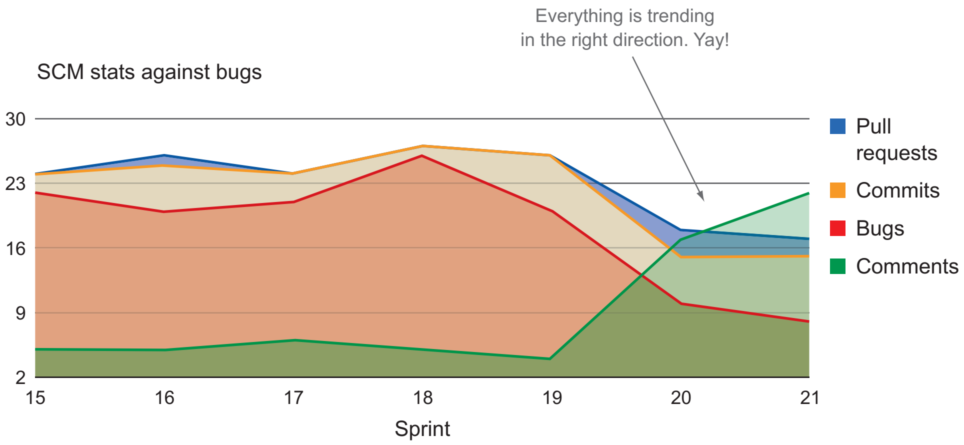


Figure 5.15 The graph the team ended with in the last chapter. Everything is looking good.

were paying dividends. As a reminder, they were looking at the chart in figure 5.15 and feeling pretty good about themselves.

The team was so excited that they brought their findings to their leadership to show them the great work they were doing. There they were confronted with these questions:

- Are you releasing faster now?
- Are you getting more done?

If we compare those to the questions we asked at the beginning of the chapter, they line up pretty well:

- How fast are you delivering or can you deliver changes to your consumer? = Are you releasing faster?
- How consistently does your team do their work? = Are you getting more done?

The team decided to add more data to their graphs. The first question was “Are you releasing faster?” Their deployments were controlled from their CI system; if they could pull data from there, they could map their releases along with the rest of their data.

The second question was “Are you delivering more consistently?” In chapter 3 we talked about the fairly standard measure of agile consistency: velocity. Another data point that can be used to track consistency is good builds versus failed builds from the CI system. They added that to their charts as well, which output the graph shown in figure 5.16.

Based on the data shown here, there’s some bad news:

- Velocity isn’t consistent.
- Releases are far apart.

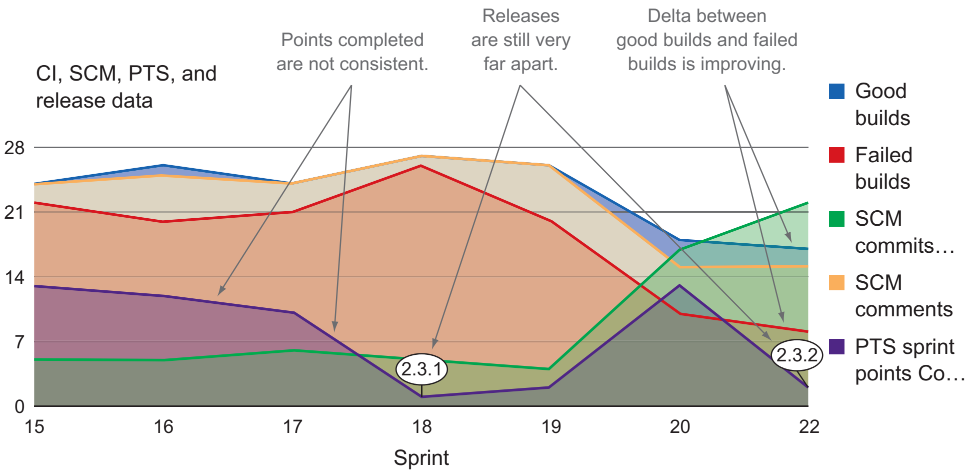


Figure 5.16 Adding velocity and releases to our case study

But there is good news too; the delta between good and failed builds is improving. The total number of builds is decreasing, which would be expected because the team is committing less code and doing more code reviews. Of the total builds, the percentage of failed builds has gone down significantly. This is an indicator of better consistency in delivering working code and also an indicator of higher code quality.

So if their CI system was indicating that they were delivering quality code more consistently, why were they not hitting a consistent velocity? At a closer look they realized that even though their SCM data was fairly consistent, their velocity was not. Based on that, the team hypothesized that perhaps something was wrong with their estimations. To try to dig deeper into that, they decided to look at the distribution of estimations for a sprint. They used base 2 estimating with the maximum estimation being a 16, so the possible estimation values for a task were 1, 2, 4, 8, and 16. The typical distribution across sprints looked something like figure 5.17.

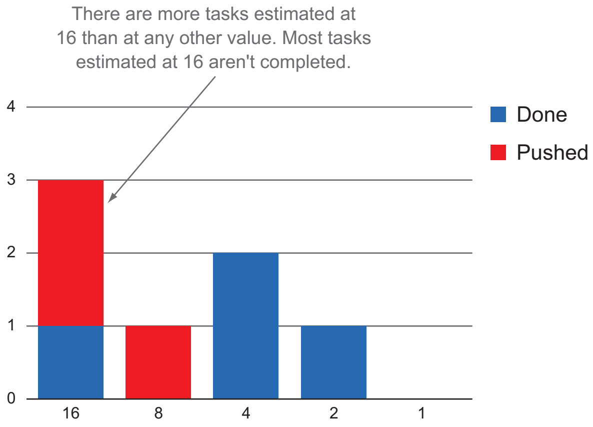


Figure 5.17 The distribution of estimates for a sprint: how many are completed and how many get pushed to the next sprint?

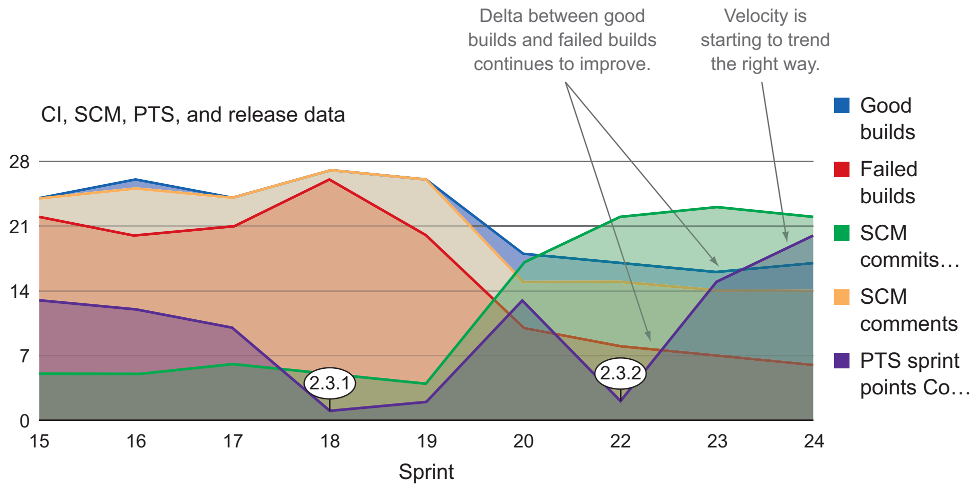


Figure 5.18 Velocity isn't dipping anymore and the good/bad build trend is continuing to improve.

As shown in their analysis, most of their tasks were pretty big and the bigger tasks tended to not always get done. These big features were also pushing releases farther apart because the team was missing their sprint goals and had to postpone a release until the tasks they wanted to release were done. If they could get their tasks back to an appropriate size, then perhaps they could get their velocity on track and have more frequent, smaller releases.

The team started to break their stories down further; anything that was estimated as a 16 was broken down into smaller, more manageable chunks of work. After a few sprints they noticed that their velocity started to increase, as shown in figure 5.18.

In addition, their task-point distribution has changed dramatically for the better, as shown in figure 5.19.

They were getting more done and bad builds were decreasing, but their stakeholders still wanted a lot of features out of each release. They made a team decision to start getting their smaller pieces of work out to production more frequently. In the previous chapter, the introduction of the pull request

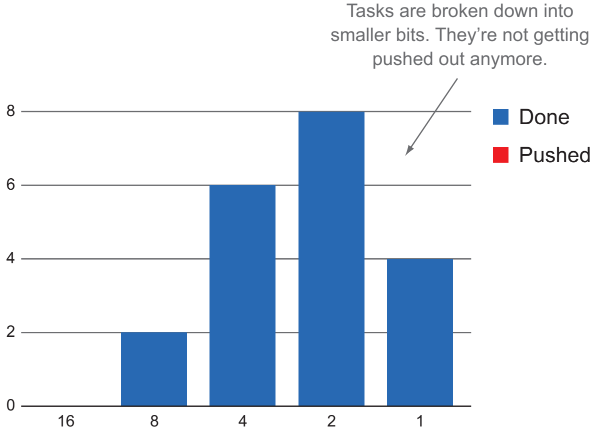


Figure 5.19 A much better distribution of estimates for a sprint

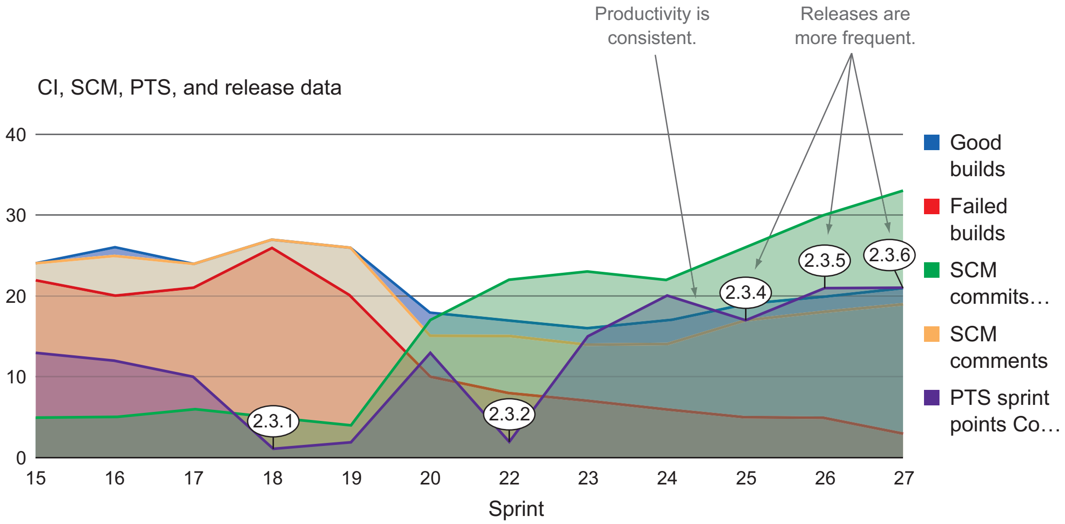


Figure 5.20 Adding releases to our graph to see frequency showed trends continuing to improve over time.

workflow and quality engineering into their process gave them a much higher degree of confidence in the quality of their work. As a result of those process improvements, they could see their build success rate improving. Their confidence level of getting more frequent releases was at an all-time high, so they started getting their changes out to their consumers at the end of every sprint; the result is shown in figure 5.20.

The team finally got to a point where their releases were happening a lot more frequently because of their commitment to quality. Their confidence was at an all-time high because of their ability to make informed decisions and measure success. Now they had a complete picture to show their leadership team that demonstrated the improvements they were making as a team.

5.6 Summary

The large amount of data you can get from your CI system through CD/CI practices can tell you a lot about your team and your processes. In this chapter we covered the following topics:

- Continuous integration (CI) is the practice of integrating multiple change sets in code frequently.
- Continuous delivery (CD) is an agile practice of delivering change sets to your consumer shortly after small code changes are complete.
- Continuous testing (CT) makes CD possible and is usually run by your CI system.
- Continuous development generates lots of data that can help you keep track of your team.

- CI/CD/CT data can tell you the following:
  - How disciplined your team is
  - How consistently you're delivering
  - How good your code is
- Setting up a delivery pipeline will enable you to get better data in your application lifecycle.
- You can learn a lot from the following CI data points:
  - Successful and failed builds
  - Tests reports
  - Code coverage
- You can use multiple data points from your CI system to check the true meaning of your build trends.
- Combining CI, PTS, and SCM data gives you powerful analysis capabilities.

6 Data from your production systems

This chapter covers

- How the tasks you're working on provide value back to the consumer
- Adding production monitoring data to your feedback loop
- Best practices to get the most out of your application monitoring systems

Figure 6.1 shows where the data covered in this chapter can be found in the software delivery lifecycle.

Once your application is in production, there are a couple more types of data that you can look at to help determine how well your team is performing: application performance monitoring (APM) data and business intelligence (BI) data.

- APM data shows you how well your application is performing from a technical point of view.
- BI data shows you how well your application is serving your consumer.

Data from your production systems mainly produces reactive metrics. Your release cycle is behind you; you thought your code was good enough for the consumer and

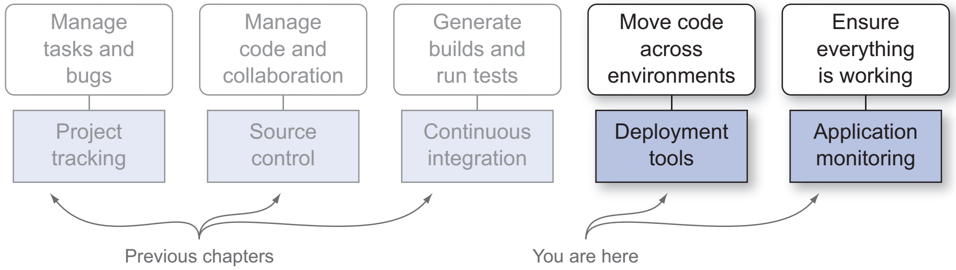


Figure 6.1 You are here: application monitoring's place in the application lifecycle

you put it out there. Now you have to watch to make sure you understand how it's working and if the changes you made are good—or not. Your sponsor will also likely care how well the features you're delivering are serving the consumer.

Up to this point we've looked only at data that you can gather in your development cycle, most of which is typically used on agile teams. APM and BI data is also common, but not something that the development team typically works with. In many cases APM data is owned and watched by operations teams to make sure the systems are all working well and that nothing is going to blow up for your consumers. Once the application is up and running, a separate BI team mines the consumer data to ensure your product is meeting the business need for which it was built. This responsibility is shown in figure 6.2.

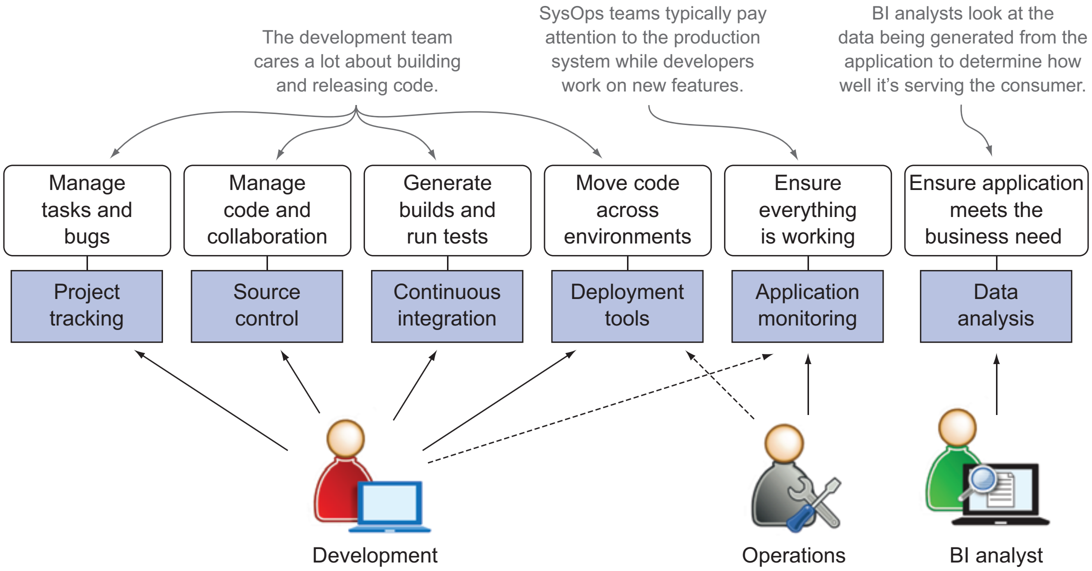

The development team cares a lot about building and releasing code.
SysOps teams typically pay attention to the production system while developers work on new features.
BI analysts look at the data being generated from the application to determine how well it's serving the consumer.
Manage tasks and bugs
Manage code and collaboration
Generate builds and run tests
Move code across environments
Ensure everything is working
Ensure application meets the business need
Project tracking
Source control
Continuous integration
Deployment tools
Application monitoring
Data analysis
Development
Operations
BI analyst

Figure 6.2 The data from the application lifecycle and the division of development and operations teams

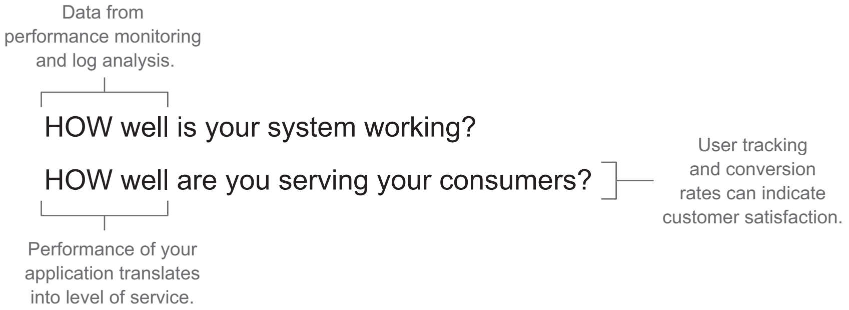

Data from performance monitoring and log analysis.
HOW well is your system working?
HOW well are you serving your consumers?
User tracking and conversion rates can indicate customer satisfaction.
Performance of your application translates into level of service.

Figure 6.3 Questions that you can answer from production monitoring alone

Realistically, the data that you can get from your consumer-facing production system is the most valuable that you can collect and track because it tells you how well your system is working and if your consumers are happy. This is broken down in figure 6.3.

If you tie this back to the rest of the data you've been collecting, you now have a complete picture of the product you're building and enhancing from conception to consumption. We'll look more deeply into combining data from the entire lifecycle in chapter 7. For now we'll focus on the data you can get out of your APM systems.

There are plenty of tools that can do production monitoring for you, but if you don't think about how your system should report data when you're building it, you're not going to get much help from them. Let's start off talking about best practices and things you can do to ensure you're collecting the richest set of data you can to get the whole picture on monitoring your system and using that data to improve your development process.

A bit about DevOps

On a team that's practicing DevOps, there isn't much separation between who writes the code (Dev) and who supports the systems the code runs on (Ops). That's the whole point; if the developers write the code, they should also know better than anyone else how to support it and how to deploy it. DevOps is heavily associated with CD, which we talked about in chapter 5. DevOps teams tend to have more control over their production monitoring systems because the same team owns the code and supports it in production.

6.1 Preparing for analysis: generating the richest set of data you can

The nature of production monitoring systems is to do real-time analysis on data for quick feedback on software systems so you don't have to do much to get data from them. Even with the most low-touch system, there are techniques you can apply to collect better data that provides insight into its performance and how it's being used by your consumers.

A lot of the advice that you're about to read crosses the line into BI. Understanding how your consumer is using your site is key to ensuring you're building the right product and adding the right features at the right time. I've been on several teams where BI was a separate organizational function that didn't always communicate directly with the development team, and that feedback certainly didn't come back to the team directly after they released updates to the consumer. If possible, BI should be integrated tightly with the development team; the closer the two functions are to one another, the better the tactics described in the next subsections will work.

> **The difference between BI and business success metrics**
>
> Even though I'm advocating keeping the BI effort closely tied to your development effort, it makes sense to have BI as a separate function from application development. BI teams exist to crunch the data your application generates, looking for trends and figuring out how different things affect each other. BI teams generate reports that show trends and relationships that reflect how the data being generated affects the success of the business. Success metrics reflect how your application is being used and are the indicators of consumer behavior that are unique to your application.

6.1.1 *Adding arbitrary metrics to your development cycle*

It is not uncommon that a team will get caught up in building, and even testing, a feature and then leave any possible instrumentation or tagging for production monitoring up to another team—if it's done at all. As you're building your features you should be thinking about your consumer, how they're using your product, and what you need to know to improve their experience. This mindset will help you do the right type of work to get good data that can help you determine if you're hitting your true goal: giving your consumer what they want.

Two example frameworks that help with this concept are StatsD (github.com/etsy/statsd/) and Atlas (github.com/Netflix/atlas/wiki). The concept behind them fits precisely with what we've been discussing so far and a mentality that shouldn't be much of a stretch: "measure anything, measure everything." StatsD and Atlas allow developers to easily add arbitrary metrics and telemetry in their code. These libraries are embedded into your code with the intention of pushing the ownership of arbitrary monitoring onto the development team.

These systems work by installing an agent into your application container or a daemon on your server that listens for metrics as they're being sent from your application. That component then sends the metrics back to a central time-data series server for indexing. This process is illustrated in figure 6.4.

The most popular example for this is StatsD. The StatsD daemon runs on a node.js server, but there are clients you can use to send data to that daemon in many different

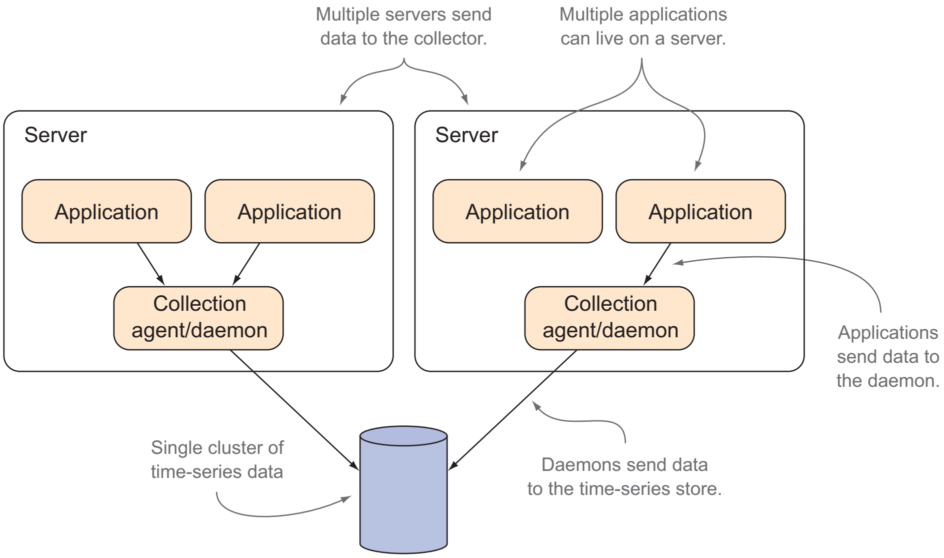

Multiple servers send data to the collector.
Multiple applications can live on a server.
Server
Application
Application
Collection agent/daemon
Server
Application
Application
Collection agent/daemon
Applications send data to the daemon.
Single cluster of time-series data
Daemons send data to the time-series store.

Figure 6.4 An example architecture to collect arbitrary metrics

languages. Once you have a time-series data store set up and get the daemon running, using StatsD is as simple as the code shown in the following listing.

Listing 6.1 Using StatsD

```
private static final StatsDClient statsd = new
     NonBlockingStatsDClient("ord.cwhd", "statsd-host", 8125);

statsd.incrementCounter("continuosity");
statsd.recordGaugeValue("improve-o-meter", 1000);
statsd.recordExecutionTime("bag", 25);
statsd.recordSetEvent("qux", "one");
```

Creates the static client

Increments a counter

Sets a gauge value

Records a timer

Saves an event with a timestamp

As shown in the listing, using this library is a piece of cake. Create the client object and set the metrics as you see fit in your code. The four different types of data give you lots of flexibility to record metrics as you see fit. On my teams I like to make this type of telemetry a part of quality checks that are enforced by static analysis or peer review. If you're building code, ensure that you have the proper measurement in place to show the consumer value of the feature you're working on.

Let's say you have a site that sells some kind of merchandise and you want to track how many users utilize a new feature in your code during the checkout process. You could use the database of orders to sort through your data, but there is usually personal data as well as financial data in that database, which requires special permissions and security procedures to access. By using an arbitrary metric library, you can collect any statistics you want on your consumer's usage separately from the rest of the application. Because the data pertains to how your collective consumers use the site rather than individual transactions with customer details, the security around it doesn't have to be as stringent, which opens up the possibility of collecting and analyzing this data in real time.

Another use case is as an indicator of quality—is your application doing what it's supposed to do? If it is, then you should see your arbitrary metrics behaving as you'd expect. If you start to see those metrics displaying unusual patterns, that may indicate that something isn't working correctly.

StatsD is designed to write to a server called Graphite (graphite.wikidot.com/faq), a time-data series database with a front end that shows arbitrary charts rather simply. That looks something like figure 6.5.

In figure 6.5 you can see metrics that the team made up and are tracking to see how their consumers are using the site.

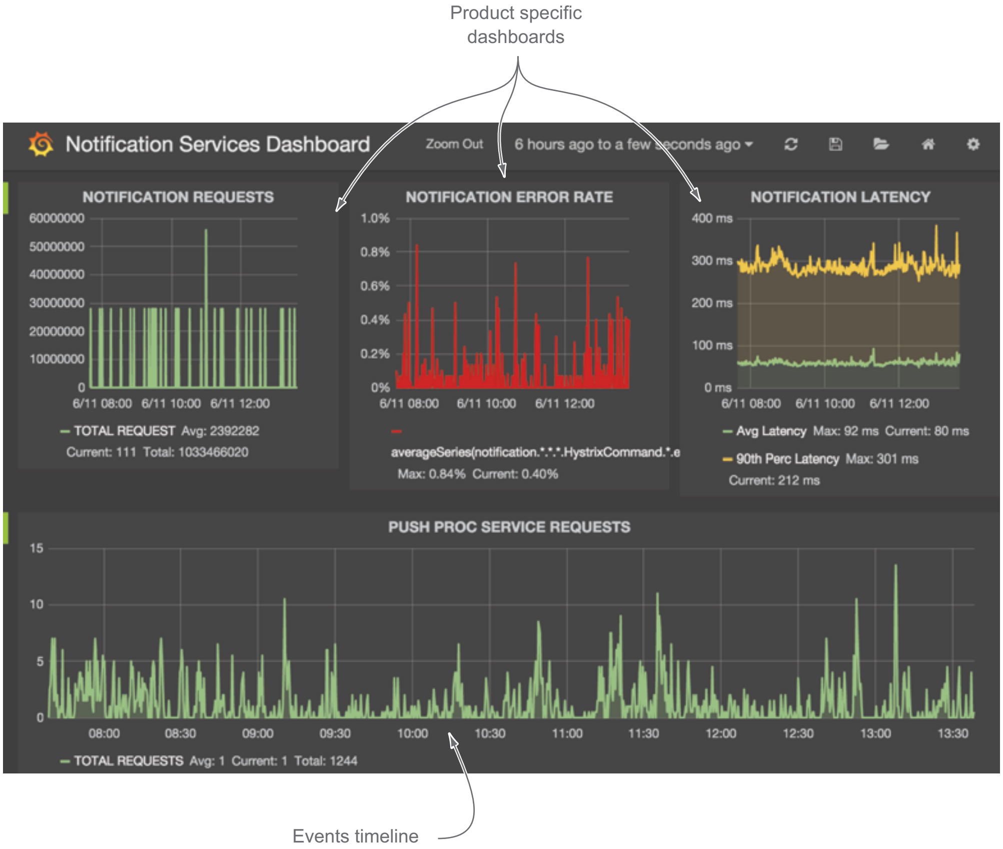

Product specific dashboards
Notification Services Dashboard
Zoom Out
6 hours ago to a few seconds ago
NOTIFICATION REQUESTS
60000000
50000000
40000000
30000000
20000000
10000000
0
6/11 08:00
6/11 10:00
6/11 12:00
TOTAL REQUEST Avg: 2392282
Current: 111 Total: 1033466020
NOTIFICATION ERROR RATE
1.0%
0.8%
0.6%
0.4%
0.2%
0%
6/11 08:00
6/11 10:00
6/11 12:00
averageSeries(notification.*.*.*.HystrixCommand.*.e
Max: 0.84% Current: 0.40%
NOTIFICATION LATENCY
400 ms
300 ms
200 ms
100 ms
0 ms
6/11 08:00
6/11 10:00
6/11 12:00
Avg Latency Max: 92 ms Current: 80 ms
90th Perc Latency Max: 301 ms
Current: 212 ms
PUSH PROC SERVICE REQUESTS
15
10
5
0
08:00
08:30
09:00
09:30
10:00
10:30
11:00
11:30
12:00
12:30
13:00
13:30
TOTAL REQUESTS Avg: 1 Current: 1 Total: 1244
Events timeline

Figure 6.5 A screenshot of Grafana, the web front end for the Graphite time-series database. Note that the metrics displayed are arbitrary metrics defined in the code of the application.

Using Graphite

Graphite is a time-data series database[a] that different frameworks can write into. The web-based front end in our example is Grafana, which charts anything users can define. Throughout this book we frequently use Kibana as our front end for charting and metrics aggregation; Grafana is actually a fork of Kibana, so the two look very similar.

[a] A time series database is a software system optimized for handling time series data, arrays of numbers indexed by time (a datetime or a datetime range), en.wikipedia.org/wiki/Time_series_database.

6.1.2 Utilizing the features of your application performance monitoring system

The data you can get from your application performance monitoring system helps you troubleshoot issues that you can trace back to your code changes. This connection allows you to tie your application performance to your development cycle. By utilizing the features of your APM system, you'll be able to make the connection between your product's performance and your development cycle much easier.

The tools we've looked at for analyzing arbitrary data can also be used for APM, but they require a lot of setup and maintenance. There are some production-ready tools that specialize in monitoring the performance of your application with minimal setup and a host of features. Two popular systems that fall into this category are New Relic (newrelic.com/) and Datadog (www.datadoghq.com).

Typical monitoring systems don't care about what your consumers are doing or examples like the previous one, but they do look at an awful lot of things relating to the health of the system. These allow you to monitor things like the following:

- Network connections
- CPU
- Memory usage
- Transactions
- Database connections
- Disk space
- Garbage collection
- Thread counts

New Relic is a hosted service that you can use for free if you don't mind losing your data every 24 hours. It has a great API so it's not so tough to pull out data you need to store and use the rest of it for alerting and real-time monitoring. New Relic also gives you the ability to instrument your code with custom tracing so you can track specific parts of your code that you want to ensure are performing appropriately. An example of a trace annotation is:

```
@Trace(dispatcher=true)
```

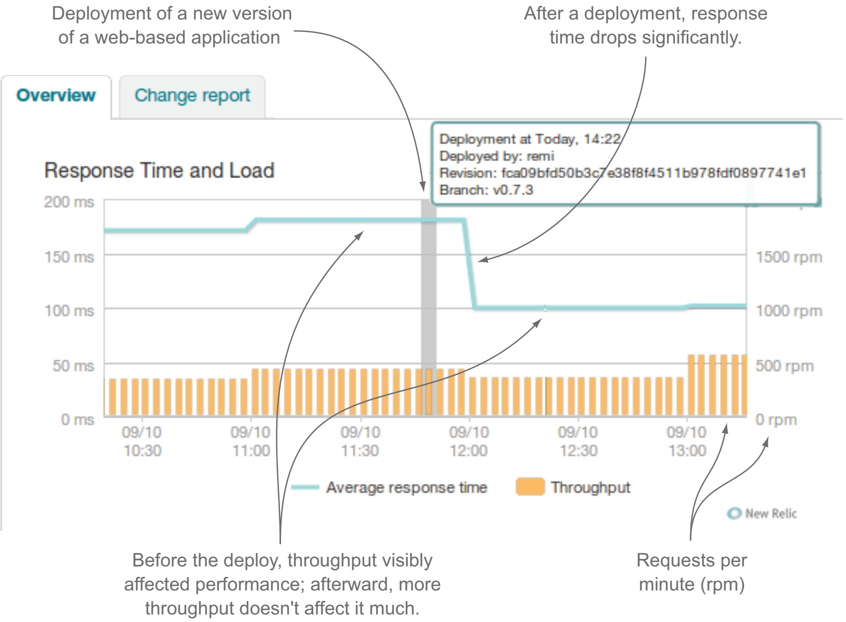
It's a good idea to put a trace annotation on anything that you want to get monitoring data about. Just about anything is a good candidate, but here are particular parts of the code where you should definitely monitor:

- Connections to other systems
- Anything that you communicate with through an API
- Working with databases
- Document parsing
- Anything that runs concurrently

Using annotations in New Relic can give you a lot more data to be able to dig into and understand how your code is functioning. You should at a minimum set up traces on critical business logic and any external interfaces. From the list just shown, if you're going to annotate anything, ensure you measure the following:

- Database connections
- Connections to other APIs or web services

Another example of tying your data back to the development cycle is including releases in your APM data. In New Relic you have to set up deployment recording,[1] which varies based on what type of application you're running. Figure 6.6 demonstrates how New Relic shows this data.


Deployment of a new version of a web-based application

After a deployment, response time drops significantly.

Deployment at Today, 14:22
Deployed by: remi
Revision: fca09bfd50b3c7e38f8f4511b978fdf0897741e1
Branch: v0.7.3

Before the deploy, throughput visibly affected performance; afterward, more throughput doesn't affect it much.

Requests per minute (rpm)

Figure 6.6 New Relic charting data over time with a clear delineation of a code release

[1] "Setting up deployment notifications," docs.newrelic.com/docs/apm/new-relic-apm/maintenance/setting-deployment-notifications.

As shown in figure 6.6 it's very clear how the deployment affected the application's performance. In this case average response time went way down, which at a glance looks pretty good. It would be interesting to map that back to the iteration that produced that code to see if anything related to how the team operated affected this performance improvement.

6.1.3 Using logging best practices

Your app should be writing something out to logs; at a minimum it's likely writing out errors when they happen. Another way to collect arbitrary metrics would be to write them in your logs and use something like Splunk (www.splunk.com) or the ELK (EC [www.elasticsearch.org], Logstash [logstash.net], and Kibana [www.elasticsearch.org/guide/en/kibana/current/]) stack to aggregate logs, index them for quick searching, and turn your searches into nice graphs that you can use for monitoring. If you're looking at your logs from that perspective, log as much as possible, particularly anything that would be of value when aggregated, charted, or further analyzed. Regardless of your favorite way to collect data, you should use some best practices when you do logging to ensure that the data you're saving is something you can work with. After all, if you're not doing that, what's the point of writing logs in the first place?

USE TIMESTAMPS, SPECIFICALLY IN ISO 8601

Anything you log should have a timestamp. Most logging frameworks take care of this for you, but to help make your logs even more searchable and readable use the standard date format ISO 8601 (en.wikipedia.org/wiki/ISO_8601). ISO 8601 is a date/time format that's human readable, includes time zone information, and best of all is a standard that's supported in most development languages. The following code shows the difference between an ISO 8601 date and a UNIX epoch timestamp, which unfortunately many people still use.

```
2014-10-24T19:17:30+00:00
1414277900
```

A date in ISO 8601 format

The same date as a UNIX epoch timestamp

Which would you rather read? The most annoying thing about the UNIX format is that if you are at all concerned with time zones you need an additional offset to figure out when the date happened relative to UTC time.

USE UNIQUE IDS IN YOUR LOGS, BUT BE CAREFUL OF CONSUMER DATA

If you want to be able to track down specific events in your logs, you should create IDs that you can index and search on to track transactions through complex systems. But if you do this, make sure you're not using any kind of sensitive data that could enable hackers to find out more about consumers, like their Social Security number or other personal information.

USE STANDARD LOG CATEGORIES AND FRAMEWORKS

Log4J and other Log4 frameworks make adding logging to your application a breeze. Instead of trying to reinvent the wheel, use these common frameworks. In addition, use the standard log levels of INFO, WARN, ERROR, and DEBUG so you can set your logs to output the data you need when you need it.

- *INFO*—Use this when you want to send information back to your logs.
- *WARN*—Use this when something isn't going well but isn't blowing up yet.
- *ERROR*—Something broke; always pay attention to these.
- *DEBUG*—Typically you don't look at DEBUG logs in a production environment unless you're trying to find a problem you can't figure out from the rest of your logs. Of course, this is great to have in nonproduction environments.

PAY ATTENTION TO WHAT YOUR LOGS ARE TELLING YOU PROACTIVELY

Although this may sound obvious, I've seen teams completely ignore their logs until the consumer was reporting something they couldn't reproduce. By that time it was too late; there were so many errors in the logs that trying to find the root of the problem became like searching for a needle in a haystack. Be a good developer and write data to your logs; then be a good developer and pay attention to what your logs tell you.

USE FORMATS THAT ARE EASY TO WORK WITH

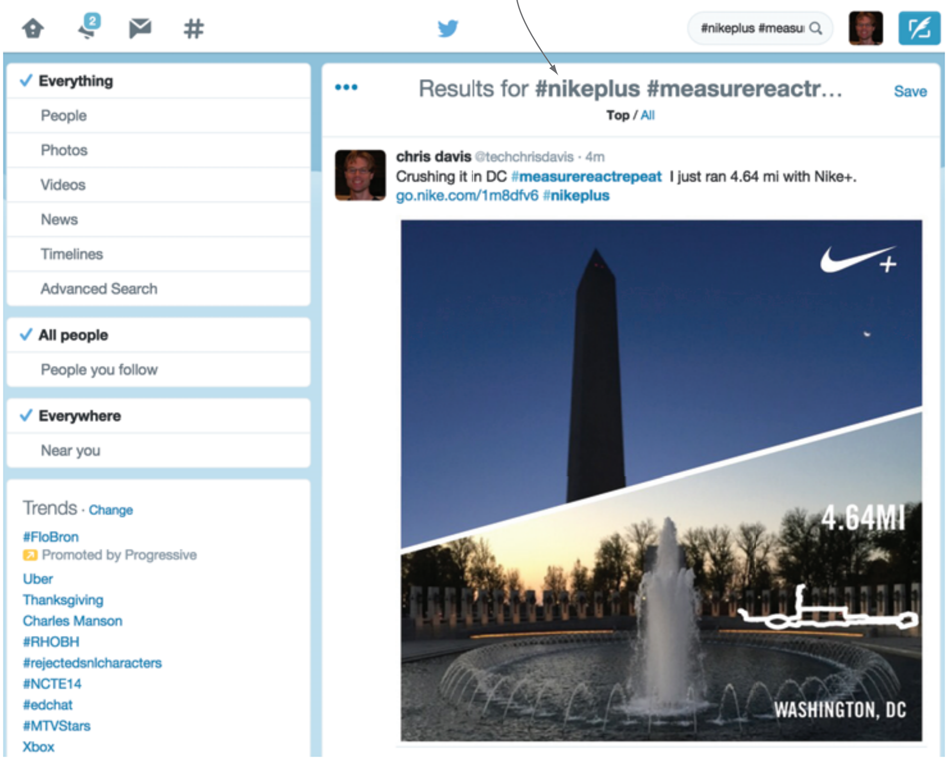
To be able to analyze the data in your logs, you'll want to make sure you're writing them in a format that's easy to parse, search, and aggregate. The following guidelines give details on how to do this:

- *Log events in text instead of binary format.* If you log things in binary, they're going to require decoding before you can do anything with them.
- *Make the logs easy for humans to read.* Your logs are essentially auditing everything your system does. If you want to make sense of that later, you had better be able to read what your logs are telling you.
- *Use JSON, not XML.* JSON is easy for developers to work with, easier to index, and a more human-readable format. You can use standard libraries to write JSON out into your logs through your logger.
- *Clearly mark key-value pairs in your logs.* Tools like Splunk and ELK will have an easier time parsing data that looks like key=value, which will make it easier for you to search for later.

6.1.4 Using social network interaction to connect with your consumers

One easy way to stay in touch with your consumers is to have a solid social network presence. Depending on what your team is building, perhaps it doesn't make sense to use Twitter, but even an internal social network like Yammer, Convo, or Jive can take advantage of that in the same way. If you can create hashtags that your consumers use to talk about you, it's very easy to get that data out of Twitter, Yammer, or other social networks to find out how well changes you make to your software affect your consumers.

Our app posted this custom hashtag when users tweeted their runs.



Results for #nikeplus #measurereactr...

chris davis @techchrisdavis · 4m
Crushing it in DC #measurereactrepeat I just ran 4.64 mi with Nike+.
go.nike.com/1m8dfv6 #nikeplus

Figure 6.7 Results for #nikeplus from the Nike+ Running app

A simple example would be to promote a new feature by creating a hashtag that allows your consumers to talk about it on their social network of choice. If you're using Twitter, this becomes really easy to do. When I was working on the Nike+ Running app, we even built hashtags into the social posting feature when someone finished a run; then we knew exactly how people were using the app and what they thought. Figure 6.7 shows searching Twitter for the tags we implemented in our app.

The Nike+ Running app was fun to work on because it has a strong fan base whose members are very vocal about how they use it. Building in social interaction and then following what your users are doing gives you tremendous insight into how well your applications are serving your consumers. Because we appended the hashtag #nikeplus on every Twitter post, it was really easy for us to see exactly how our consumers were using the app and what they were thinking when they used it. This kind of data helped us shape new features and modify how the app worked to enhance the consumer experience even further.

NOTE Twitter has a great API that allows you to get this data into your data analysis system, but the API isn't public; you have to register an app to use it. In addition, there are rate limits to it, so if you use it be careful to stay within allowable request limitations.

6.2 The data you'll be working with: what you can get from your APM systems

Now that I've given you myriad tips on how to maximize your data, let's look at getting the data out and how you can use it to hone your team's performance. There are two big categories of data that you can get from production:

- *Application monitoring*—The data your application is generating regardless of what your application does:
  - Server and application health
  - General logging
- *BI data*—Application-specific data that tells you how your consumers are interacting with your application:
  - Captured as arbitrary metrics that you can make up as you go along
  - Semantic logging, using strongly typed events for better log analysis

6.2.1 Server health statistics

Your server health statistics are the indicators of how well your system is built. Looking at crash rates, stack traces, and server health shows you if your code is running well or if it's eating resources and generating poor experiences. Typical things you can look at are:

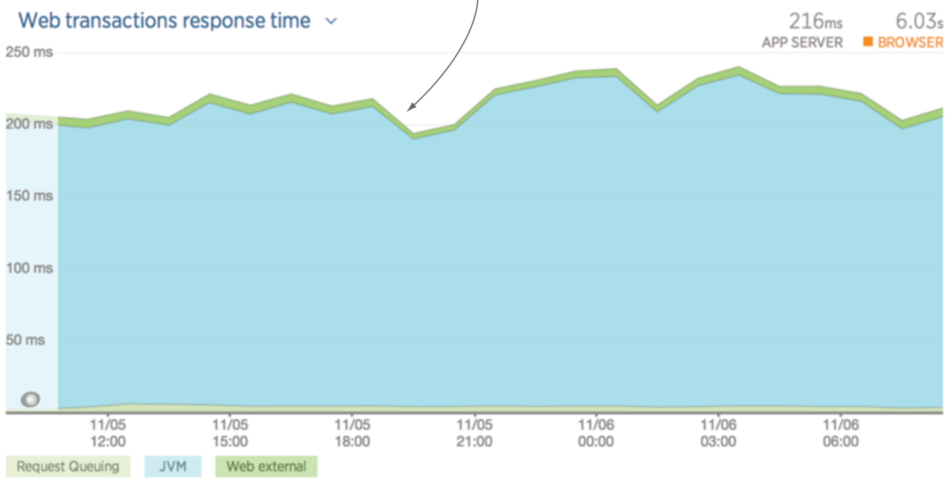
- CPU usage
- Heap size
- Error rates
- Response times

The first important bit of data you can learn from New Relic dashboards is shown in figure 6.8, which shows how long web responses take and where the response time is spent.

Because New Relic is looking at how much time consumers are spending on your site, it can also tell you what the most popular pages are, as shown in figure 6.9.

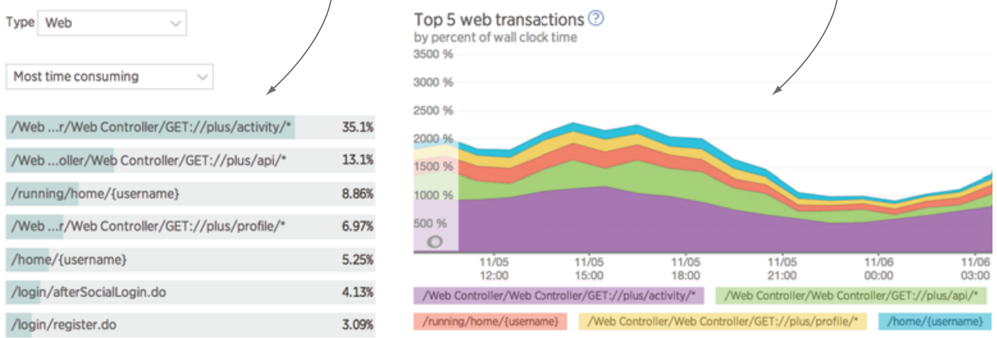
These metrics are interesting because they help inform you about the experience your consumers are having with your applications. It's important to set service-level agreements (SLAs) for your applications to help you define what you think is a good experience. When you do this, don't look at how your application performs now; think of what your consumers expect. Are they okay with a web page that loads in four seconds or should the SLA be under one second? Do they expect to get results from entering data of some kind immediately, or do they just expect to dump some data in and see results later?

Watching how your performance trends toward your SLAs becomes interesting as you compare it to the data we've already looked at from project tracking, source control, and CI. Are you improving team performance, at the expense of application



The breakdown of the response time

Figure 6.8 New Relic dashboard overview showing the performance of the different layers of a web application



The top pages broken down by percentage

The amount of time consumers spend on your website

Type: Web

Most time consuming

/Web ...r/Web Controller/GET://plus/activity/*	35.1%
/Web ...oller/Web Controller/GET://plus/api/*	13.1%
/running/home/{username}	8.86%
/Web ...r/Web Controller/GET://plus/profile/*	6.97%
/home/{username}	5.25%
/login/afterSocialLogin.do	4.13%
/login/register.do	3.09%

Top 5 web transactions
by percent of wall clock time
3500 %
3000 %
2500 %
2000 %
1500 %
1000 %
500 %
11/05 12:00
11/05 15:00
11/05 18:00
11/05 21:00
11/06 00:00
11/06 03:00
/Web Controller/Web Controller/GET://plus/activity/*
/Web Controller/Web Controller/GET://plus/api/*
/running/home/{username}
/Web Controller/Web Controller/GET://plus/profile/*
/home/{username}

Figure 6.9 The New Relic transactions view showing the pages on your website where consumers spend the most time

performance, or is application performance a first-class citizen in your development cycle? When development teams are completely focused on new features and don't pay attention to how their product is performing in the wild, it's easy to lose sight of application performance.

REACTING TO APPLICATION HEALTH

Keeping your application healthy is an important factor in consumer satisfaction. Ideally you want to have great response times and an app that can scale up to whatever your potential consumer base is. A good APM system will tell you when performance is getting bad and will also usually tell you where your problems are. When you see performance problems, you should take the time in your current development cycle to fix or mitigate them.

6.2.2 Consumer usage

Another production monitoring strategy is to watch how your consumers are using your site. There are a number of ways to track page hits and site usage; two examples of off-the-shelf products are Google Analytics (www.google.com/analytics/) and Crittercism (www.crittercism.com/). These solutions not only track the number of people who are using your products, but also how long they spent on certain pages and how they flowed through the app, as well as something called *conversion rate*.

Conversion rate is determined by dividing whatever you decide is success criteria for using your application by total users. Success may be selling something, clicking on an advertisement, or making a connection with another user. Conversion tells you how successful you are at turning users into consumers.

REACTING TO CONSUMER USAGE

Trends in consumer usage help you determine how your application needs to evolve to better serve your consumer. If you have features that your consumers don't use, then why are they even in your product? Hone the most popular features in your application to ensure your consumers are getting the best possible experience you can offer.

Try to move your team to a model where you can do small, frequent releases. In each release be sure to measure the impact of the change you just deployed to help determine what feature to work on next.

6.2.3 Semantic logging analysis

How your consumers are using your system is key to continuously improving it. If you're collecting arbitrary metrics and page tracking, then there's no limit to what you can learn about your consumers and how they use your system.

If you're practicing semantic logging and logging custom metrics, then you can get just about any kind of data you want. Perhaps you have search functionality on your site and you want to know what people are searching for so you can better tune your content of products based on what consumers are looking for. You could save every search to the database and then do some kind of data-mining exercise on it every so often, or you could log a strongly typed event with every search.

Using the frameworks or logging that we talked about earlier in the chapter to view this data in real time makes tracking metrics that you define as success criteria for your features much easier. Let's say that you have a gardening app where users enter the plants in their garden and search for advice on how to make them as healthy as possible. If your business model is based on creating the most valuable content for your users, then the searches people plug into your app are invaluable information because it tells you what your user base wants to read.

The key to semantic logging analysis is that it tells you what you set your system up to tell you. If you're logging metrics that are valuable to your business and watching them as you update your software products, they become some of the most valuable metrics you can collect.

REACTING TO DATA IN YOUR LOGS

You should strive for zero errors in your logs. For those of you laughing right now, at least try to clean up your errors with every release. If you're seeing large numbers of errors or warnings in your log analysis, then you have a strong data point you can use to lobby for time to clean up tech debt.

6.2.4 Tools used to collect production system data

Table 6.1 is summary of the tools we've talked about and used in this chapter. Keep in mind that open source systems typically have a lot of setup you have to do yourself (DIY).

Table 6.1 APM and BI tools used in this chapter and the data they provide

Product	Type of system	Data It provides	Cost model
Splunk	Cloud-based with agents installed on your servers for data collection	APM and log aggregation and analysis. Splunk allows you to search through anything you send to it.	Pay for the amount of data you store in it.
EC, Logstash, and Kibana (ELK)	DIY	APM and log aggregation and analysis. The ELK stack allows you to search through anything you send to it. This is commonly referred to as "open source Splunk."	Labor costs for setup and maintenance along with the cost of infrastructure and storage.
New Relic	Cloud-based with agents stored on your servers for data collection	Using instrumentation, New Relic gives you lots of performance data ranging from CPU and memory analysis to site usage statistics and a detailed breakdown of where your code spends the most of its time.	Free if you don't store data for longer than a day; then you pay based on the amount of data you store.
Graphite and Grafana	DIY	This is the epitome of a DIY system. Graphite and Grafana, like the ELK stack, show nice charts on any type of time-data series you send to it.	Labor costs for setup and maintenance along with the cost of infrastructure and storage.
Open Web Analytics	DIY	Collects data on how consumers navigate through your site along with page hit counts.	Labor costs for setup and maintenance along with the cost of infrastructure and storage.
Google Analytics	Cloud-based	The standard for collecting usage statistics for web applications. It tracks how consumers navigate your site, counts page hits, and helps track conversion rate.	Free for most cases; the premium version has a flat fee.

Table 6.1 APM and BI tools used in this chapter and the data they provide ***(continued)***

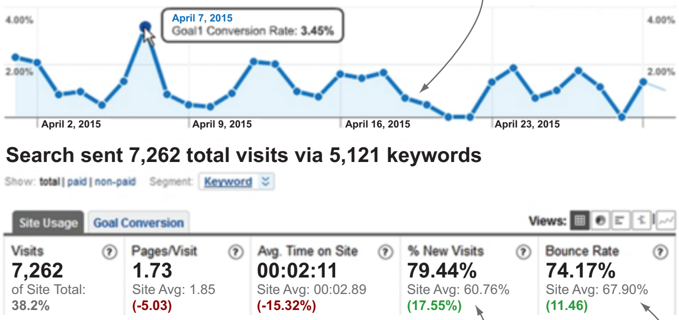
Product	Type of system	Data It provides	Cost model
Data Dog	Cloud-based	Aggregates data from all over the place.	Free for small installations; you pay for how long you want to retain your data and the size of your installation.
Crittercism	Mobile library with cloud-based data collectors	You can get crash rates and usage statistics that help you figure out what your consumers are seeing and how to fix their issues.	License fees

6.3 Case study: a team moves to DevOps and continuous delivery

Our case study team has a lot of the pieces in place to start utilizing the data from their production environment. They've been working to transition to a DevOps model, and the development team has started paying close attention to their production systems. They have log analysis, are using New Relic to monitor the health of their systems, and are looking at key metrics in their development cycle to ensure their process is working well. They've even improved their CI system so much that they were able to start deploying small changes to production every day. They thought they had achieved continuous delivery and were pretty excited about it; now their consumers were getting added value every day! Or so they thought.

After a few weeks of operating like this, they got their biweekly report from the BI team showing how consumers were using the system, and it hadn't changed much at all. The BI team was tracking the conversion rate of consumers on the site, total visits for the different pages across the site, and how long unique visitors stay on the site. The data they were getting back looked like the dashboard in figure 6.10.

Conversion rate; how many consumers do what you want them to



Search sent 7,262 total visits via 5,121 keywords

Visits	Pages/Visit	Avg. Time on Site	% New Visits	Bounce Rate
7,262	1.73	00:02:11	79.44%	74.17%
of Site Total: 38.2%	Site Avg: 1.85	Site Avg: 00:02.89	Site Avg: 60.76%	Site Avg: 67.90%
	(-5.03)	(-15.32%)	(17.55%)	(11.46)

New visitors to your site

How many people come to your site and leave right away

Figure 6.10 The dashboard the BI team paid the closest attention to

The delivery team took a look at the BI report and realized that there was no direct link between the work they were doing and the metrics being used to determine if features were successful. Because they were delivering features, the team needed to figure out how those features improved conversion and stickiness on their site. The delivery team decided to start by bringing the BI team into their sprint planning to help them close the gap.

For every new feature they were going to implement they asked, "What is the value to the consumer of this change, how do we measure it, and how does it affect conversion?" The next tweak the development team made was to add a More Info button that had a Buy Now button on it. The feature was designed with the following theories in mind:

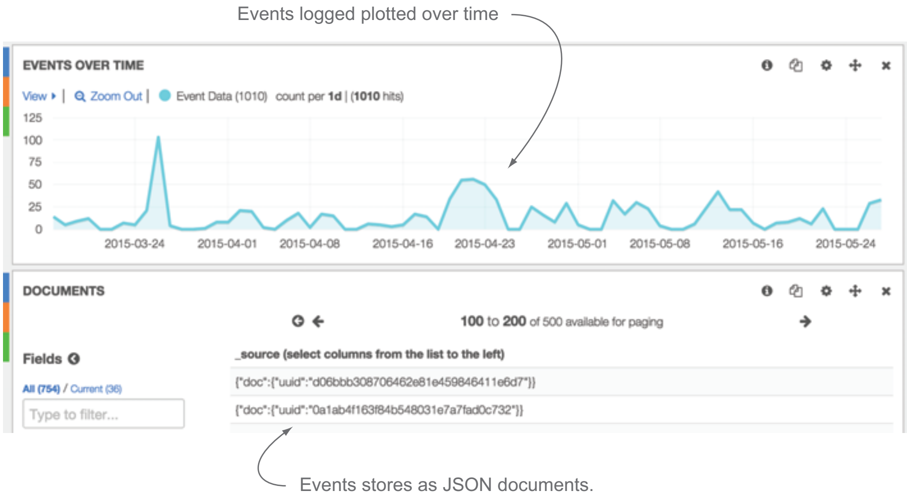
- They wanted to show many products on the page to a consumer at once, so they would save space by showing info in a pop-up instead of inline.
- They thought that with that info consumers would be more likely to buy a product and therefore should click the Buy Now button from that pop-up.

To figure out how this affected conversion, they added a few custom metrics:

- Buy Now button clicks
- More Info clicks

If their theories were correct, they should see a significant number of clicks on the More Info link and a high percentage of clicks on Buy Now after clicks on More Info, and conversion should go up.

The team was already using ELK for their logging, so they simply added strongly typed messages to their logs for the events they wanted to track. These started showing up in their dashboard, as shown in figure 6.11.



```
100 to 200 of 500 available for paging
_source (select columns from the list to the left)
{"doc":{"uuid":"d06bbb308706462e81e459846411e6d7"}}
{"doc":{"uuid":"0a1ab4f163f84b548031e7a7fad0c732"}}
```

Events logged plotted over time

Events stores as JSON documents.

Figure 6.11 The Kibana dashboard the team generated to track their statistics

Now the team was completing the picture. They could see how they were developing, what they were developing, and how it affected their consumers when they deployed changes to the consumer. Now they could use that data to shape what features they should work on next and how they could tweak their development process to ensure they were delivering things the right way.

6.4 Summary

The final piece of the data picture is the data your application generates as your consumers interact with it. In this chapter you learned:

- Application performance monitoring and business intelligence typically aren't measured by development teams.
- APM gives you insight into how well your application is built, and BI tells you how your consumers use it:
  - Server health statistics show you how well your application is performing.
  - Arbitrary stats and semantic logging give you measureable stats on how your application is being used.
  - Teams using a DevOps model are more likely to have access to APM data.
- Add arbitrary metrics collection to your code to collect application specific data. Netflix Servo and StatsD are popular open source libraries to help with this.
- Use logging best practices to get as much data as you can from your log analysis:
  - Use ISO8601-based timestamps.
  - Use unique IDs in your logs, but be careful of consumer data.
  - Use standard log categories and frameworks.
  - Pay attention to what your logs are telling you proactively.
  - Use formats that are easy to work with.
- Using social networks allows you to better connect with your consumers.
- Use arbitrary monitoring or semantic logging to provide feedback to the development team based on BI data to let them gauge effectiveness of new features.
- A variety of open source and commercial tools are available for application monitoring and collecting BI.

Part 3

Applying metrics to your teams, processes, and software

Using the concepts from part 1 and the rich set of data from part 2, you're ready to take your metrics collection, analysis, and reporting to the next level. In part 3 you'll learn how to combine the data you've been collecting into complex metrics, get the full picture on the quality of your software, and report the data throughout your organization.

Chapter 7 shows you how to combine data across several data sources to create metrics that fit into your processes and your team. You'll learn how to explore your data, determine what to track, and create formulas that output your own custom metrics.

Chapter 8 shows you how to combine your data to determine how good your software products really are. You'll learn how to measure your software from two key perspectives: usability and maintainability.

Chapter 9 shows you how to publish metrics effectively across your organization. You'll learn how to build effective dashboards and reports that communicate the right level of information to the right people. We'll also look at some pitfalls that can cause your reports to fail and how to avoid them.

Chapter 10 breaks down the agile principles and shows you how to measure your team against them.

As in part 2, each chapter ends in a case study so you can see the techniques you learned in the chapter applied in a real-world scenario.

7 Working with the data you're collecting: the sum of the parts

This chapter covers

- Identifying when to use custom metrics
- Figuring out what you need to create a metric
- Combining data points to create metrics
- Building key metrics to track how well your team is working

Metrics are measurements or properties that help in making decisions. In agile processes metrics can be created from the data your team is generating to help determine where you need to take action to improve.

7.1 Combining data points to create metrics

To create a metric you only need two things:

- Data to generate the metric from
- A function to calculate the metric

In previous chapters we've been focusing on the data you can collect from the different systems in your application lifecycle and what you can learn from it alone or

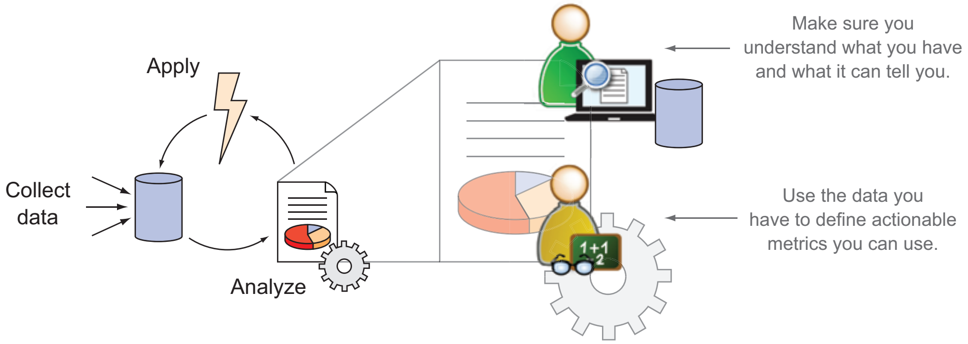

Apply
Collect data
Analyze
1+1
2
Make sure you understand what you have and what it can tell you.
Use the data you have to define actionable metrics you can use.

Figure 7.1 Building metrics in the context of collecting, analyzing, and applying metrics

combined with other data. Now you can start combining data points to create your own metrics using the following steps.

- *Explore your data.* Ensure that you know what you have.
- *Break down your data to determine what to track.* Using the knowledge you have about your data, pick out the most useful and telling data points to build metrics with.
- *Build your functions around data points.* Adding together multiple data points that are related to a behavior will give you metrics that tell you a lot with a simple measurement.

Figure 7.1 shows these steps in the context of the big picture.

In the analysis phase you want to spend enough time understanding what data you have and how it's connected so that you can define actionable metrics that become useful in your development cycle.

As you've seen throughout this book, there's so much data generated throughout the application lifecycle that no one or even two data points can give you a clear indicator of performance for your team. To get the best picture in previous chapters, we looked at data points from different systems alone or next to one another. An alternative to that technique is to combine these elements into metrics that shed light on larger questions that can't be answered by a single data point.

An example of a calculated metric that we've already looked at is recidivism. Recidivism tells you how frequently your team is moving the wrong way in your workflow. It's calculated purely with PTS data with the following formula:

```
Recidivism = bN / (fN + bN)
```

- `N` = number of tasks
- `f` = moving forward in the workflow
- `b` = moving backward in the workflow

In many cases you can do some interesting calculations on data from a single system to get new insight into recidivism. An example using only SCM data is the Comment To

Commit Ratio. If a team is using a workflow that includes code reviews, sometimes developers, usually the leads, will end up spending most of their time reviewing other people's code rather than writing any code themselves. We call this PR Paralysis. This is usually a bad sign that there isn't enough senior technical leadership on the team or pull requests aren't being distributed across enough of your team. You can calculate Comment To Commit Ratio with the following formula:

```
Comment To Commit Ratio = r / (m + c)
```

- `m` = merged pull requests
- `c` = commits
- `r` = reviews

In chapter 8 we'll look at the elements of good software, and one of the measures we'll talk about is Mean Time To Repair (MTTR). This is another important metric that can be calculated simply with data from your APM if you just want it at a high level. The most simplistic view of MTTR would be

```
MTTR = f - s
```

- `s` = start date time when an anomaly is identified
- `f` = date time when the anomaly is fixed

Later in this chapter we'll be looking at estimate health, or how accurate your team's estimations are. The algorithm for estimate health is outlined in listing 7.1. In a few words it compares the amount of time a task took to complete against the estimated effort, and gives a rating of 0 when estimates line up, a rating of greater than 0 when tasks are taking longer than estimated (underestimating), and a rating of less than 0 when tasks take less time than estimated (overestimating).

A rather complex metric that we'll dive into in our case study is release health. This is a combination of PTS, SCM, and release data to find out how healthy releases are for a team practicing continuous deployment and releasing software multiple times a day.

7.2 Using your data to define "good"

There are three "goods" that we'll be looking at for the remainder of the book.

- *Good software*—This is covered in the next chapter. Is what you're building doing what it's supposed to do and is it built well?
- *A good team*—Good teams usually build good software. Because different teams operate differently and use their data-collection tools differently, the metrics to measure how good a team is are often relative to that team. Thus, you need to have the next "good" to measure accurately.
- *Good metrics*—You must have good indicators for your team and software that provide trustworthy and consistent data. These are used to measure your team and your software.

While exploring data in earlier chapters you probably started drilling into data that seemed either good or bad based on how you perceived your team's behaviors. While querying for data points that you know represent good behavior, it may be surprising to find how other data points relate. These relationships are important to hone in on and discuss across the team to understand why the relationships behave the way they do. Through this exploration and discussion you take action on metrics you know are important and observe related metrics until patterns emerge that you want to affect.

While measuring your teams and processes it's important to always keep an open mind about the results you get back. Don't try to jam data into an explanation because it's what you want to see. Assessing what you're seeing openly and honestly is key to getting your team to their highest possible level of efficiency and productivity.

7.2.1 Turning subjectivity into objectivity

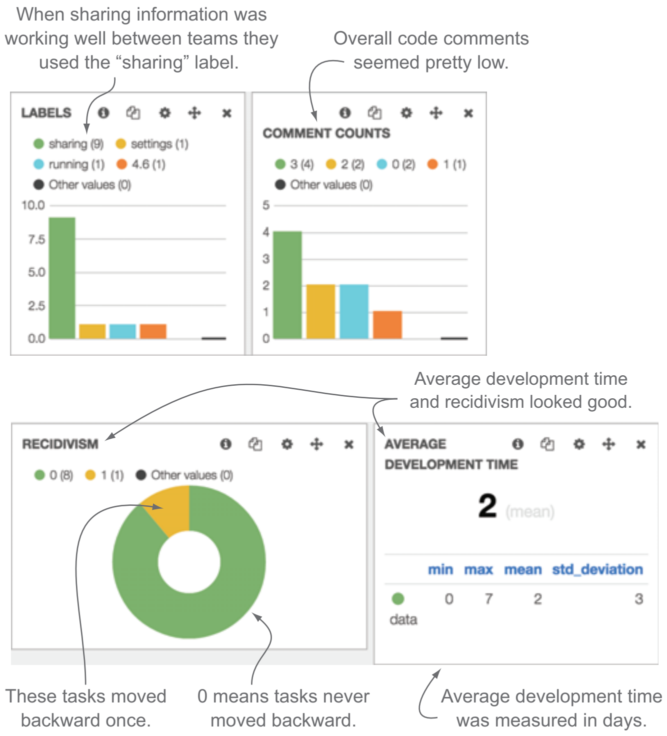
In chapter 3 we looked at tagging the tasks in your PTS system to subjectively indicate whether or not they went well. If you did this, you can now turn that subjective data into objective data.

Agile development is all about the team. You want to be able to create a happy and efficient team that has some ownership of the product they're working on. Asking the team to mark their tasks with how well they think they went is as good an indicator as any to tell you how well your team is doing. A happy team will more likely be a consistently high-performing team. Sad teams may perform well for a time until burnout and attrition destroy them.

In this example a team repeatedly brought up in retrospectives that they thought different roles on the team weren't working together closely enough and it was impacting delivery. They wanted to see more information sharing and collaboration between teams to be able to deliver their products more efficiently and consistently. They decided that they would have more face-to-face conversations and warm hand-off of tasks instead of just assigning tickets to someone else when they thought a task was complete. To track their success they started labeling tasks in their PTS and SCM with the tag "sharing" if they thought that everyone was working together well through the development and delivery cycle. To them *sharing* meant that people were sharing their time and information and generally working together well. In this case they knew development time and recidivism should be low but they didn't have a good idea of how much they should expect their team to comment on their pull requests in source control; they called this "code comment count." Labels, code comment count, recidivism—or the percentage of time tasks move backward in the workflow—and average development time for tasks labeled "sharing" are shown in figure 7.2.

In this case recidivism and development time looked great. Looking at the same data for tasks where the team didn't think information sharing worked well, they produced the graphs shown in figure 7.3.

In figure 7.3 you see that recidivism and average development time went up significantly. The really interesting thing is the comment counts; they also went way up. Building on this trend, the team checked to see what happened when comment counts



When sharing information was working well between teams they used the "sharing" label.

Overall code comments seemed pretty low.

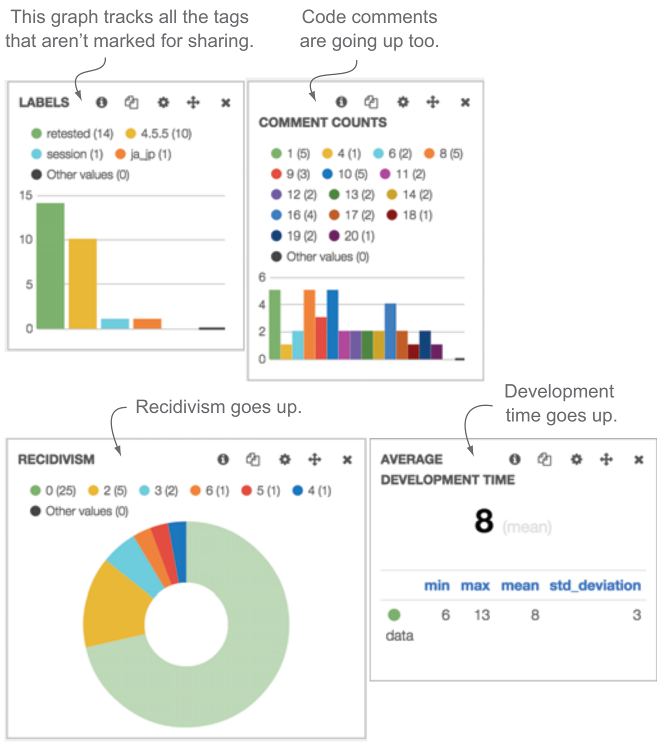
Average development time and recidivism looked good.

These tasks moved backward once.

0 means tasks never moved backward.

Average development time was measured in days.

Figure 7.2 Labels, code comment counts, recidivism, and development time for tasks labeled "sharing" over the course of a single sprint



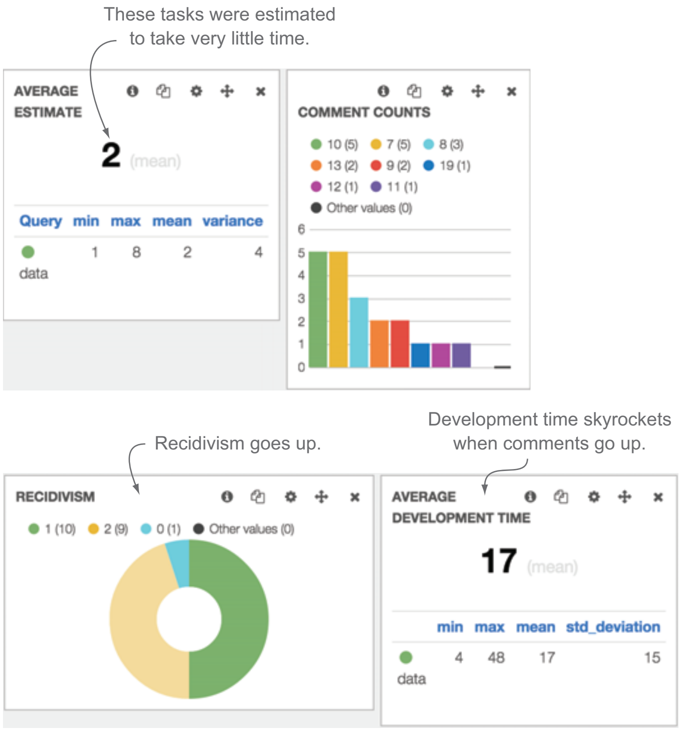
This graph tracks all the tags that aren't marked for sharing.

Code comments are going up too.

Recidivism goes up.

Development time goes up.

Figure 7.3 Labels for tasks not marked with "sharing" over the course of a sprint



These tasks were estimated to take very little time.

Recidivism goes up.

Development time skyrockets when comments go up.

AVERAGE ESTIMATE

2 (mean)

Query	min	max	mean	variance
● data	1	8	2	4

COMMENT COUNTS

RECIDIVISM

AVERAGE DEVELOPMENT TIME

17 (mean)

	min	max	mean	std_deviation
● data	4	48	17	15

Figure 7.4 The relationship among recidivism, development time, code comments, and estimates

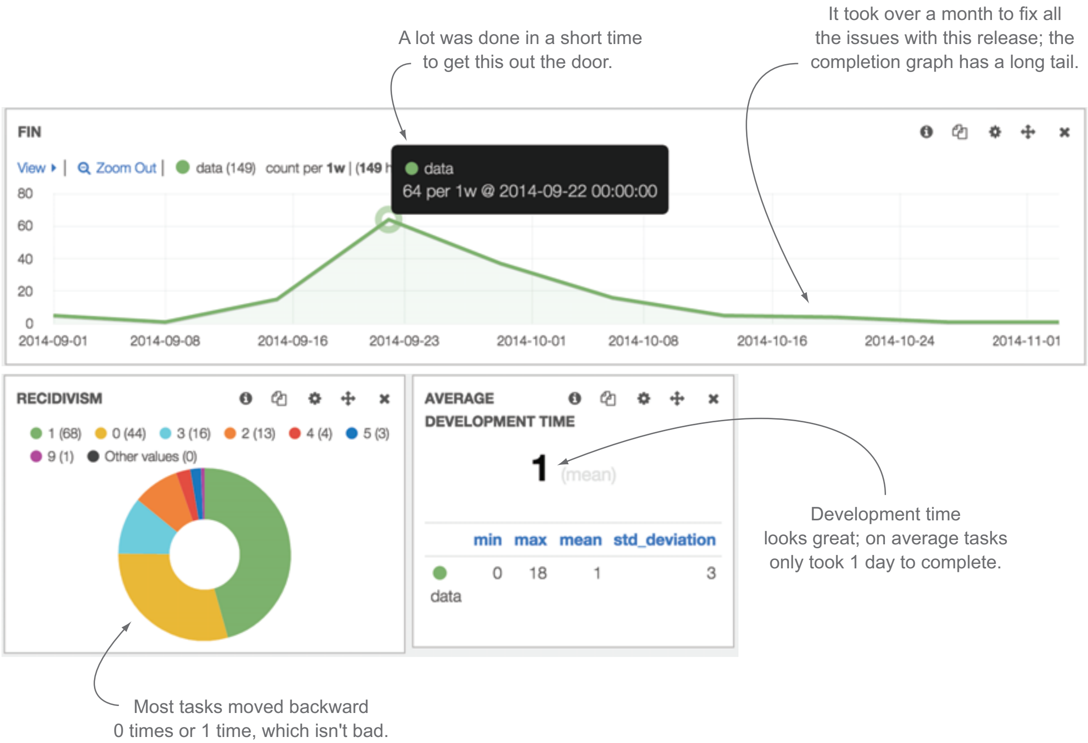
were high in general. That is reflected in figure 7.4. Note that estimates were added into this dashboard as well to show the correlation between how much work the team thinks something will take and the amount of time it actually takes.

If the estimates were high, then maybe it would make sense to see the huge jump in development time, but in this case the team noticed the opposite: estimates were very low.

For this team, having a lot of code comments seemed to point to poor information sharing, tasks taking a lot longer than estimated, and an increase in recidivism. In this case the team was able to turn something that was very subjective—whether they thought that sharing was working well—into an objective data relationship they could use as an indicator to see if tasks were going off the rails.

7.2.2 Working backward from good releases

You want a happy team and you want a team that can deliver. Another way to find out what good thresholds are for your team is to focus on what went into good software releases to figure out how to tune your process to replicate that success. In this case you can filter your data by anything that went into a release that worked well to find the key data points. As we did with tags in the previous examples, you can then look at the resulting watermarks to see what behavior went into a particular release.



A lot was done in a short time to get this out the door.

It took over a month to fix all the issues with this release; the completion graph has a long tail.

	min	max	mean	std_deviation
● data	0	18	1	3

Development time looks great; on average tasks only took 1 day to complete.

Most tasks moved backward 0 times or 1 time, which isn't bad.

Figure 7.5 Stats around a painful release. Even though development time looked great, there were nagging bugs that caused hot fixes over a long period.

In the next example we'll look at the tale of two releases: a pretty good release and a painful release. First, we'll look at the painful release.

The painful release looked great during the development cycle. The team was focusing on development time and trying to keep it to a minimum. They had a huge push to get the release out the door, and when they did the bugs started rolling in. Nagging issues caused a long support tail of nearly two months of fixing problems. The resulting charts are shown in figure 7.5.

Here are a few notable characteristics of figure 7.5:

- Average development time of one day looks awesome. If we were to look at only that, it would seem that this team was really good at breaking down their tasks.
- Even though the team managed to complete over 100 tasks in a three-week period to hit the release, there were nagging issues that made up the long tail of support after the release, which kept the team from completely focusing on the next release.
- Tasks were moving backward in the workflow around 70% of the time; those are all the tasks with recidivism of greater than zero. 20% of tasks got moved backward more than once.

Given these observations, maybe an average development time of one day isn't good after all. None of the related metrics around it seem very good, so perhaps this team should start focusing on different metrics to make sure their project is healthy.

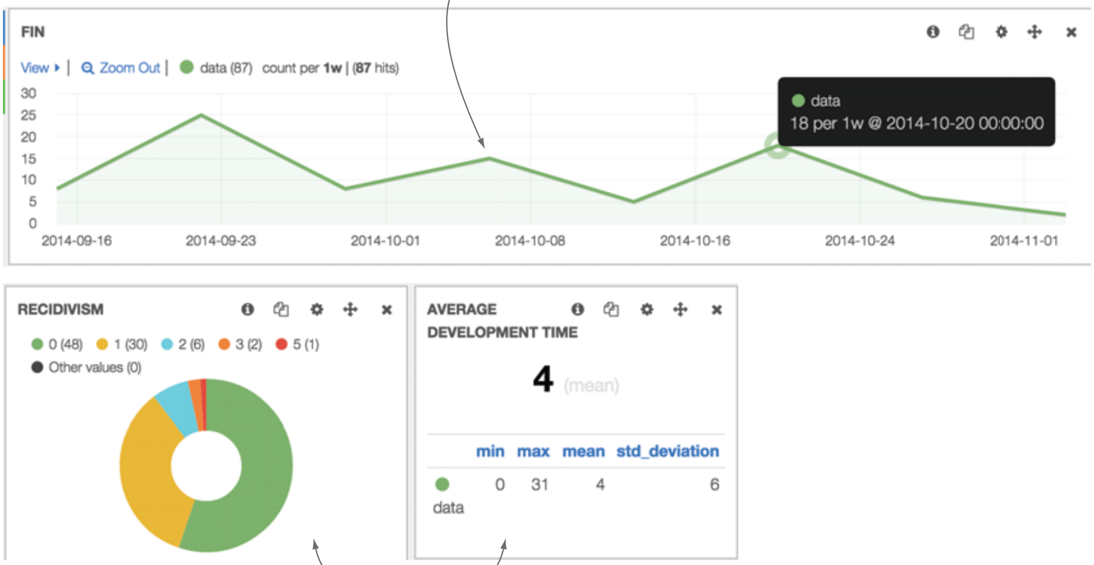
In this example, average development time was not enough to help the team get a good release out the door. They paid for shorter development times later with lots of bug fixes and additional releases.

As a comparison, in a completely different and much better release the team's average development time was four full days, much higher than the first example. But recidivism was down and the support tail for the release was much shorter and deliberately phased, as shown in figure 7.6.

In figure 7.6, notice:

- Tasks were not moving backward in the workflow as frequently. Here they never moved backward 60% of the time, and less than 10% of tasks moved backward more than once.
- Compared to figure 7.5 the releases in this chart were smaller, between 15 and 25 tasks instead of the 60 in figure 7.5. Along with the smaller releases came less support and more consistency.
- Average development time of tasks went up a lot, from one to four days.

The spikes indicate releases and show the team is able to consistently get things out the door with only a week of downtime between.



	min	max	mean	std_deviation
● data	0	31	4	6

Development time and recidivism look reasonable but could be better.

Figure 7.6 A very different release with a better support tail, lower recidivism, and longer development time

If the team is looking for consistency in their release cycle, certainly the pattern in figure 7.6 is the pattern they want to replicate. If that's the case, then a longer development time isn't a bad thing and perhaps the team should consider extremely low average development times a red flag.

7.3 How to create metrics

> *"A model is a formal representation of a theory."*
>
> —Kenneth A. Bollen

The end goal is to get our team members to perform as well as they can. To get there we want to be able to track what matters most in the context of our team and combine that data into a form that's easy to track and communicate. In chapter 1 I talked about how to find what matters most for your team by asking questions and mind mapping. Now that we've gone through all the data that you can use, we're going to start putting it together to answer questions that lead to better team performance.

Before we start creating metrics, we should lay ground rules for what makes a good metric. The following guidelines are a good place to start:

- Metrics should create actionable insight:
  - Do track metrics that you can respond to should they identify a problem.
  - Don't track metrics you can't do anything about or haven't taken the time to understand the meaning of.
- Metrics should align with core business and team tenets:
  - Do pick data to track what's relevant to your end goal. Perhaps security is your number-one goal; for others, feature delivery is more important. Teams should prioritize the most important indicators of their process as it relates to what they're delivering.
  - Don't track metrics that you can't somehow track back to something your team thinks is important.
- Metrics should be able to stand alone:
  - Do create metrics that will give you a clear indication of the health of some part of your process or team by itself.
  - Don't create a metric that forces you to look at more data to determine if it's good or bad.

With these points in mind let's look at how to figure out how well your team is estimating work. You'll start by breaking the problem down into pieces and identifying all the data points that you can use to track the pieces. You'll do that by exploring your data to make sure you understand what you have and defining metrics that help you see it clearly. Once you have good actionable metrics, you can incorporate them into your development cycle.

We've already looked at what you can do with your data when you save it in a central place, index it, and create dashboards out of it. As you spend quality time with the data your team is generating, you'll start to think up questions, and more data will lead to more questions. Eventually you'll get to a point where you start to see pieces of the larger puzzle and how they affect each other.

An aspect worth measuring is how well your team decomposes tasks. You can get more done for your customers if you're frequently delivering lots of small tasks instead of delivering large swaths of functionality at a time. Smaller changes are also easier to deliver with tests, to test in general, and to troubleshoot. To measure this let's start with the question "Is your team breaking their tasks into small enough chunks?" If they are, then you'd expect to see:

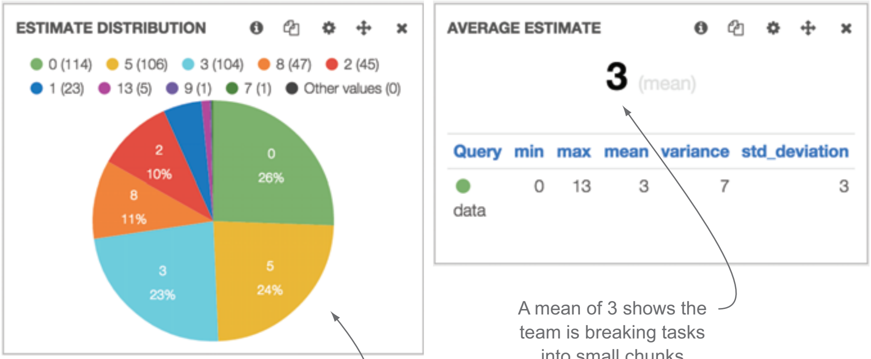
- *Distribution of estimates leaning toward smaller numbers and a fairly low average estimate*—If your tasks are defined well with the intention of delivering small pieces of functionality, this should also be reflected in the team's estimates of the work.
- *Decreased lead time*—Delivery of individual tasks measured by lead time should be short.

That will lead you to:

- How long do these tasks really take?
- Are your estimates accurate?

7.3.1 Step 1: explore your data

To figure out if your estimates are accurate, you can start by tracking the size of estimates with estimated distribution and average estimate by the amount of time it takes to get a task all the way through the development process with lead time. You'd expect that tasks that are well defined and broken down into small pieces will be well understood by the development team and as a result should be able to make it through the development process expediently, thus resulting in fast lead times. Using the tools we've been using so far produced the data shown in figure 7.7.



Query	min	max	mean	variance	std_deviation
● data	0	13	3	7	3

A mean of 3 shows the team is breaking tasks into small chunks.

Most estimates are 3 or 5; apparently the team is breaking tasks down well.

Figure 7.7 The historical estimate distribution and average estimate dashboards for a team

This team is using the Fibonacci series of estimation with a minimum estimate of 1 and a maximum estimate of 13 in a two-week, or 10-working-day, sprint. The potential values in the Fibonacci series are 1, 2, 3, 5, 8, and 13, so an average estimate of 3 is fairly low in the spectrum, which is good. Lower estimates show that tasks have been broken down to a workable level.

Overall estimation data looks pretty good; the team seems to be breaking their tasks into small, manageable chunks. With that in mind, the next questions that arise are these:

- How long do these tasks really take?
- Are our estimations accurate?

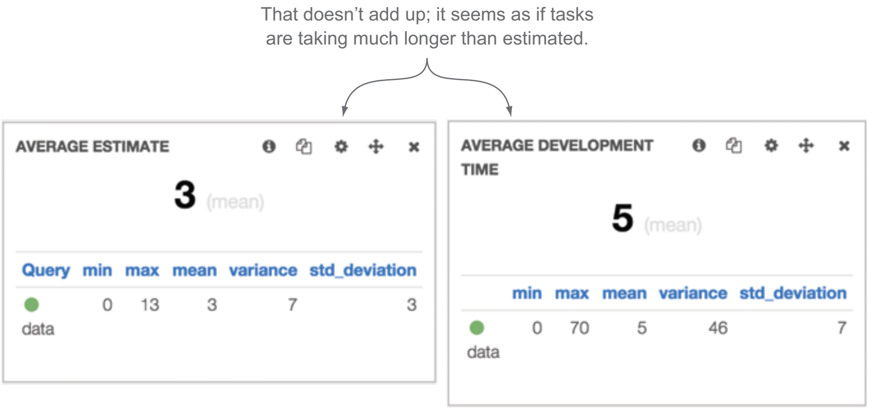
Estimating with effort vs. time

Story points are often used to estimate tasks in project tracking systems, but they don't always have a 1:1 translation to time. For example, 16 story points could mean 9 days of development on one team and it could mean 14 days on another team. No matter how you estimate, you'll have to translate them to time to figure out if they're accurate.

In this case, based on the estimation system used by the team, tasks estimated at 3 points should take around 2–3 days to complete. The next thing we can pull in is the average amount of time tasks actually take. That is shown in figure 7.8.

According to our data our average estimate is 3, but on average tasks take 5 days to complete. In this case a sprint is two weeks or 10 working days and the maximum estimate is 13. That would mean that our average estimate should be closer to 8 than 3. The next question that comes to mind is

- Why are tasks taking longer than we think they should?



That doesn't add up; it seems as if tasks are taking much longer than estimated.

AVERAGE ESTIMATE

3 (mean)

Query	min	max	mean	variance	std_deviation
● data	0	13	3	7	3

AVERAGE DEVELOPMENT TIME

5 (mean)

	min	max	mean	variance	std_deviation
● data	0	70	5	46	7

Figure 7.8 Adding average time a task takes to get to done. This doesn't look right; an average estimate of 3 should be 2–3 days.

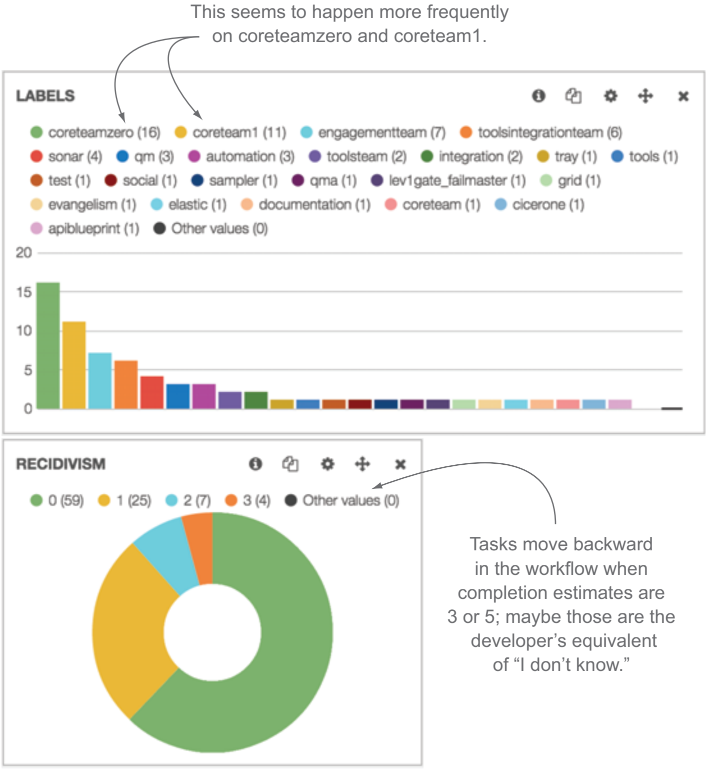
Because this question is a bit more open-ended we can start by querying for the data that we think doesn't look right to see what properties it has. To do that we can search for all data that has an estimate of 3 and a development time greater than 3 and data that has an estimate of 5 and a development time greater than 5. The query in EC/Lucene looks like this:

```
((devTime:[3 TO *] AND storyPoints:3) OR (devTime:[5 TO *] AND storyPoints:5))
```

As mentioned earlier, if you're tagging or labeling your tasks with as much data as possible, then at this point you can see what tags come back from your search. In this case, a few things jump out at us.

Figure 7.9 shows us that coreteamzero and coreteam1 seem to have this happen more than other teams and that estimates tend to be low when tasks move backward in the workflow.

This data will probably lead to even more questions, but by now you should get the idea. Rarely do you start off with all the questions that lead you to where you want to go, but spending time exploring your data to see how things are related will help you determine what to track.


This seems to happen more frequently on coreteamzero and coreteam1.

Tasks move backward in the workflow when completion estimates are 3 or 5; maybe those are the developer's equivalent of "I don't know."

Figure 7.9 Labels show what has been tagged in the cards where estimations are off, and recidivism shows that tasks tend to move backward in the workflow more frequently when estimates are 3 or 5.

7.3.2 Step 2: break it down—determine what to track

Having a big data mine is great, but sometimes it's tough to figure out what data points to watch. After exploring your data (a habit that can be very addictive), you should start to get an idea of what data points speak most to how your team works. If you have a small team, whose members sit next to each other, having a high number of comments on PTS tickets probably indicates a communication problem, but if you have teams in multiple geographic locations, it could indicate great cross-team communication. If your team doesn't comment on tickets because they sit next to each other and discuss issues, then that data point may never move and therefore will be of no use to you. If yours is a globally distributed team that uses comments to push things forward, then that data point will be extremely important to you.

A productive way to get to the bottom of what you want to track is to build a mind map that helps you identify the questions you're trying to answer. Follow these steps:

1. Ask the question.
2. Think of all the dimensions of the answer.
3. Note where you can obtain data to get your answer.

Using the scenario in section 7.2.1, the team would have created the mind map shown in figure 7.10. Another example of breaking down a big question into smaller, measureable chunks is shown in figures 7.11 and 7.12, which illustrate how to check if your team is estimating accurately.

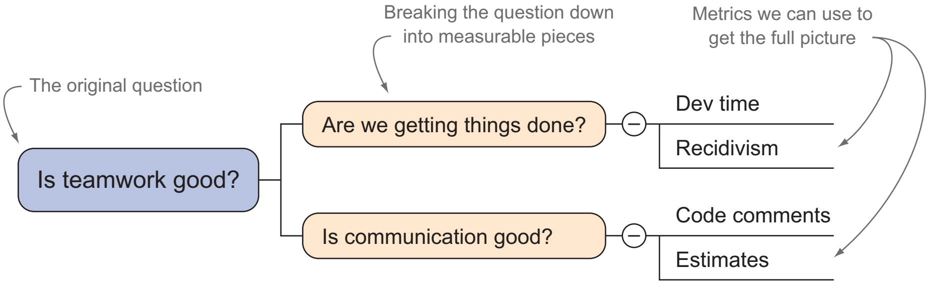

Breaking the question down into measurable pieces
Metrics we can use to get the full picture
The original question
Is teamwork good?
Are we getting things done?
Is communication good?
Dev time
Recidivism
Code comments
Estimates

Figure 7.10 Breaking down how to measure teamwork with a mind map

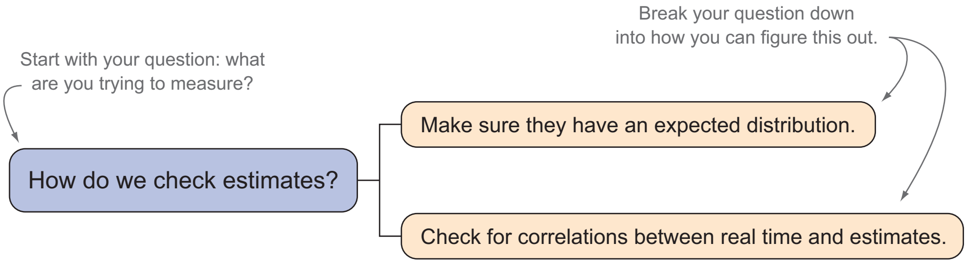

Break your question down into how you can figure this out.
Start with your question: what are you trying to measure?
How do we check estimates?
Make sure they have an expected distribution.
Check for correlations between real time and estimates.

Figure 7.11 Breaking down a problem with a mind map; starting with your question and then breaking it down one level

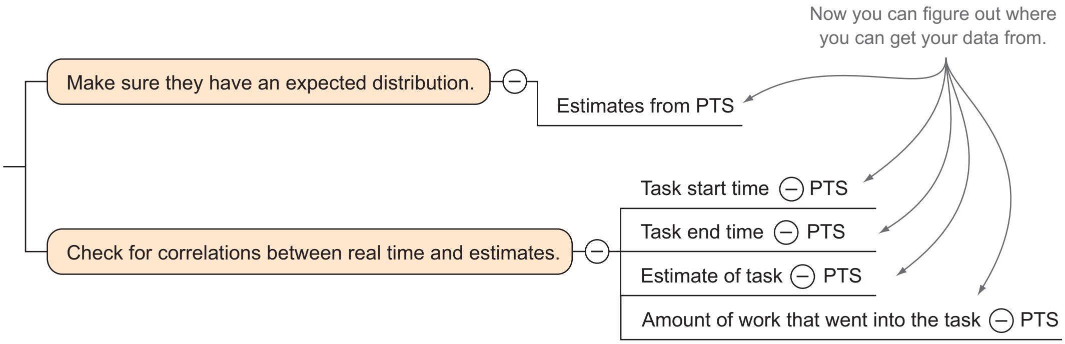

Now you can figure out where you can get your data from.
Make sure they have an expected distribution.
Estimates from PTS
Task start time
PTS
Task end time
PTS
Estimate of task
PTS
Amount of work that went into the task
PTS
Check for correlations between real time and estimates.

Figure 7.12 Using the mind map to break it down another level and then figuring out where to get data from

Once you have all the data points that you can use to answer the questions, you're ready to start using those data points to build out your metric by adding them together.

7.3.3 Step 3: create formulas around multiple data points to create metrics

Once you have the relevant data points, you can always stop there and track the individual data. This might work for some people, but it's useful to have a single data point that allows you to keep a bird's-eye view on groups of data without having to micromanage every data point. This will also lead to much cleaner, more succinct dashboards that can be shared outside the team and make more sense at a glance.

Now it's time to do a bit of math. If we stick with the examples shown in figures 7.11 and 7.12, we have several data points that we can roll up to give us an indication on how good our team's estimates are. We want to know at a glance if the team is over- or underestimating and by how much. If we visualize the association of estimates to time, we'll see something like figure 7.13.

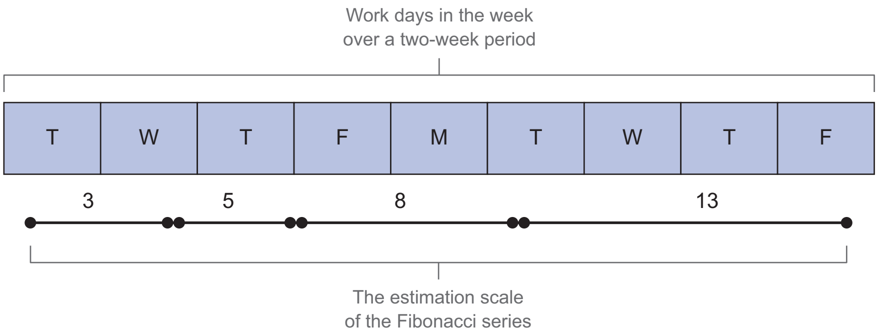

Work days in the week over a two-week period
T
W
T
F
M
T
W
T
F
3
5
8
13
The estimation scale of the Fibonacci series

Figure 7.13 A visual representation of a series of estimates next to time using the Fibonacci series of estimates over a two-week period

First, we'll name our metric to track estimation accuracy. Let's call this one estimate health. Estimate health is some combination of the following data points:

- Estimates
- Task start time
- Task end time
- Amount of work

Breaking these data points down further, we really care about the amount of time a task took, so we can start by combining start time and end time into elapsed time by subtracting one from the other:

Elapsed time (t_{actual}) = task end time - task start time

Because estimates don't equal days, the second thing we'll have to do is correlate time to estimates.

Two weeks equals 10 working days, but 1 day in a sprint is used for retrospectives and planning, which gives us a total of 9 working days. To deduce time from estimates, first we'll take the highest possible estimate and equate it to the highest possible amount of time, in this case the highest possible estimate of 13 for this team equals 9 working days. Using that as our maximum, we can break down the rest of the possible estimations. To figure out the exact correlation, use the following formula:

$$max(estimate_{actual}) = (estimate_{workdays}/max(estimate_{workdays}))*max(t_{actual})$$

Once you have the maximum amount of time, you can figure out the rest of the estimate sweet spots with the following formula:

$$correlation\text{-}value = max(t_{actual})*(max(estimate_{actual})/estimate_{workdays})$$

When you know the correlation between your estimates and time, you can plug in the data you have to find out if your estimates are high or low.

Some example estimation-time mappings using these formulas are shown in table 7.1. If you're using the system outlined in appendix A, which uses EC and Lucene, I've added the queries you can use to hone in on specific data. These will become more useful when you see them in action in listing 7.1.

Note that in table 7.1 I've put the Fibonacci series of estimates along with the series of power of 2 estimates, another common estimation series. In the power of 2 series, each possible estimate is double the previous estimate.

Table 7.1 Mapping estimates to time ranges and validating with Lucene

Estimate	Exact time	Time range	Query
Power of 2 estimations with two-week sprints			
16 points	9 days	7–9 days	`devTime:[7 TO 9] AND storyPoints:16`
8 points	4.5 days	4–7 days	`devTime:[4 TO 7] AND storyPoints:8`

Table 7.1 Mapping estimates to time ranges and validating with Lucene ***(continued)***

Estimate	Exact time	Time range	Query
Power of 2 estimations with two-week sprints			
4 points	2.25 days	2–4 days	`devTime:[2 TO 4] AND storyPoints:4`
2 points	1.125 days	1–2 days	`devTime:[1 TO 2] AND storyPoints:2`
1 point	.56 days	less than 1 day	`devTime:[0 TO 1] AND storyPoints:1`
Fibonacci series estimations with two-week sprints			
13 points	9	6–9 days	
8 points	5.53	4–6 days	
5 points	3.46	3–4 days	
3 points	2.07	2–3 days	
2 points	1.38	1–2 days	
1 points	.69	less than a day	

This is helpful, but it would be better to have a scale that told you if your estimations are high or low and how off they are. For example, a 0 means that estimations meet the time equivalents, a number greater than 0 means your team is underestimating, and a value less than 0 means your team is overestimating. Using a scale like this has benefits:

- It's actionable.
- If you see your estimate health go under 0, you can start adding more time to your estimates to get them back to a level of accuracy.
- If you see your estimate health go over 0, you can start cutting your estimates a bit to get back to a better level of accuracy.
- It's easy to communicate.
- You can easily share this outside your team without a long explanation.
- It allows you to digest several data points at a glance.
- You can put this on your dashboards to keep everyone aware of the latest data.

To create a rating number you'll need to write an algorithm that figures out the time-estimate ratio of the estimate in question and checks it against the other estimates in your estimation scale to see if the amount of time a task took matches the corresponding time window for the estimate.

To map estimates to time you'll need a function that can find the time bounds you're estimating within, the time scale, and the estimate scale. Figure 7.14, shows how this can work as a function.

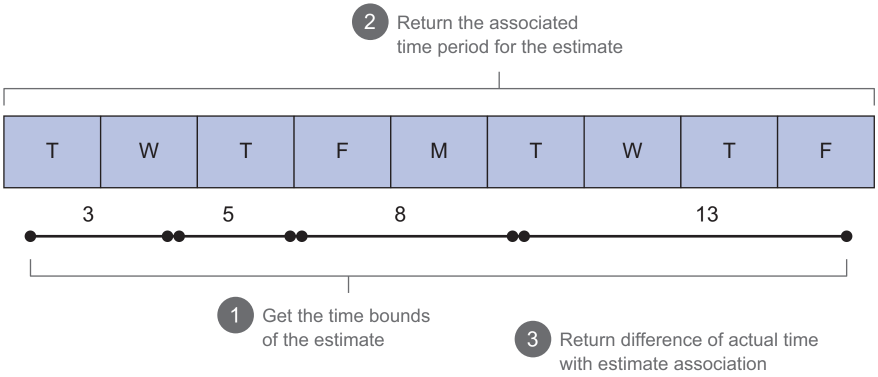

2 Return the associated time period for the estimate
T
W
T
F
M
T
W
T
F
3
5
8
13
1 Get the time bounds of the estimate
3 Return difference of actual time with estimate association

Figure 7.14 A visual representation of the algorithm to determine estimate health

An algorithm to do this is outlined in the following listing.

Listing 7.1 Algorithm for checking estimate health

```
static def estimateHealth(estimate, actualTime, maxEstimate, maxTime,
     estimationValues) {
  def result
  def timeEstimateRatio =  maxTime / maxEstimate
  def estimateTime = estimate * timeEstimateRatio
  def upperTimeBound = maxTime
  def lowerTimeBound = 0

  def currentEstimateIndex = estimationValues.findIndexOf { it ==
  ➥ estimate}

  if(currentEstimateIndex == 0) {
    lowerTimeBound = 0
  } else {
    lowerTimeBound = estimateTime - ((estimateTime -
    ➥ (estimationValues[estimationValues.findIndexOf { it == estimate} - 1]
    ➥ * timeEstimateRatio)) / 2)
    }

  if (currentEstimateIndex == estimationValues.size() -1) {
    upperTimeBound = maxTime
  } else {
    upperTimeBound = estimateTime +
    ➥ (((estimationValues[estimationValues.findIndexOf { it == estimate} + 1]
    ➥ * timeEstimateRatio) - estimateTime) / 2)
  }

  //Calculate the result
  if(upperTimeBound < actualTime) {
    def diff = actualTime - upperTimeBound
    result = 0 + diff
```

Initializes the variables

Finds the input's index in the array

Lower bound of 0 for the lowest estimate

Calculates the lower time bound

The highest estimate uses the upper bound

Calculates the upper time bound

Underestimated; it will be greater than 0

```
  } else if(lowerTimeBound > actualTime) {
    def diff = lowerTimeBound - actualTime
      result = 0 - diff
  } else {
    result = 0
  }

  return [ raw:result, result:result.toInteger() ]
}
```

Overestimated; it will be less than 0

Returns 0 within a day of the bounds

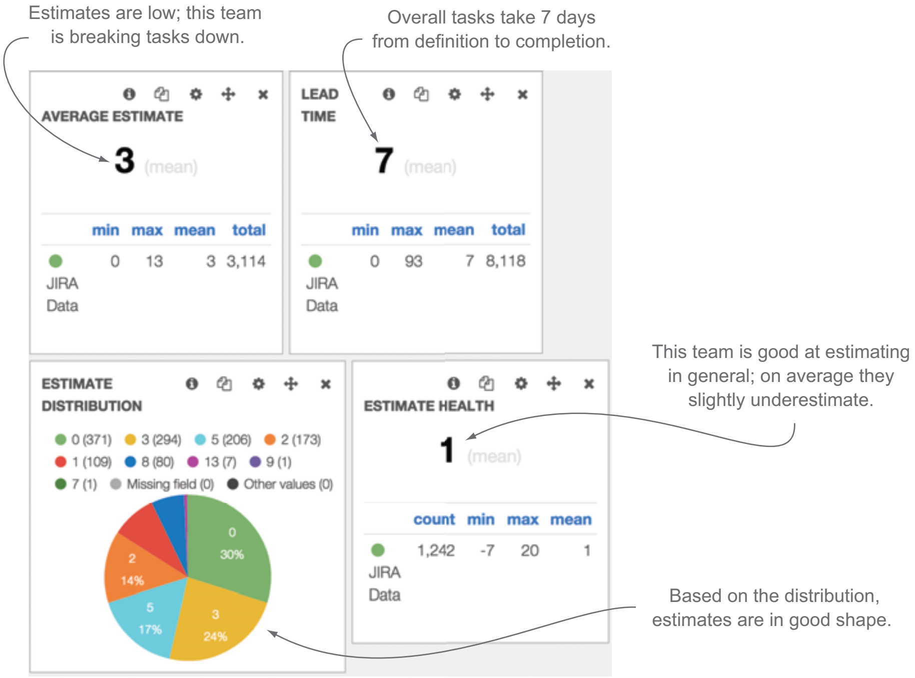
Raw result for in-depth analysis

This is a simple example of adding together a few data points to create an actionable and easy-to-understand metric that has real value to your team. Estimate health is shown alongside lead time, average estimate, and estimate distribution in figure 7.15.

Using this algorithm you can hone your estimations and thus get to a more predictable cadence of delivery. Using these techniques you can generate a host of useful and easy-to-use metrics. Some of the more useful are the ones that help evaluate how good your development team is doing.

7.4 Case study: creating and using a new metric to measure continuous release quality

Our case study for this chapter concerns a team that practices CD and is deploying code to production multiple times a day. Before they made this change they were



Estimates are low; this team is breaking tasks down.

Overall tasks take 7 days from definition to completion.

This team is good at estimating in general; on average they slightly underestimate.

Based on the distribution, estimates are in good shape.

AVERAGE ESTIMATE

3 (mean)

	min	max	mean	total
● JIRA Data	0	13	3	3,114

LEAD TIME

7 (mean)

	min	max	mean	total
● JIRA Data	0	93	7	8,118

ESTIMATE DISTRIBUTION

● 0 (371) ● 3 (294) ● 5 (206) ● 2 (173) ● 1 (109) ● 8 (80) ● 13 (7) ● 9 (1) ● 7 (1) ● Missing field (0) ● Other values (0)

ESTIMATE HEALTH

1 (mean)

	count	min	max	mean
● JIRA Data	1,242	-7	20	1

Figure 7.15 Adding estimate health to our other metrics for predictability

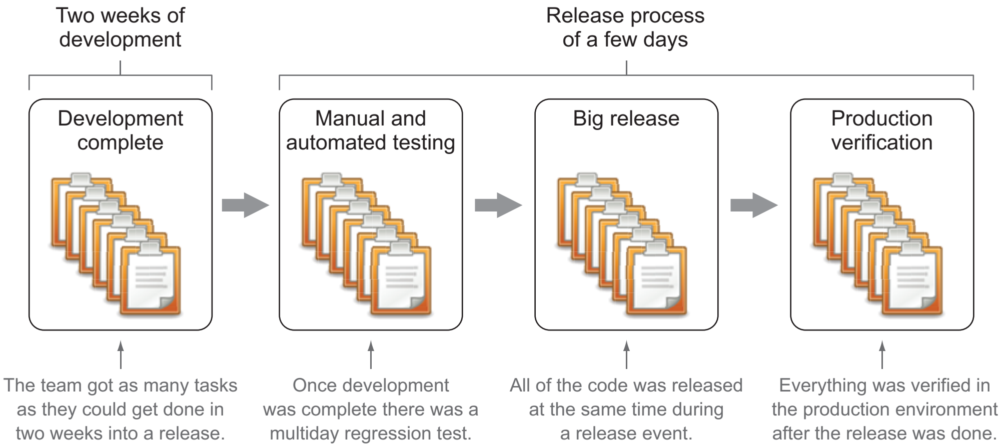

Two weeks of development
Release process of a few days
Development complete
Manual and automated testing
Big release
Production verification
The team got as many tasks as they could get done in two weeks into a release.
Once development was complete there was a multiday regression test.
All of the code was released at the same time during a release event.
Everything was verified in the production environment after the release was done.

Figure 7.16 A representation of the release process before the team moved to CD

releasing every few weeks and were able to put a rating on a release based on a few factors:

- How many bugs they found in the production environment that they didn't find in the test environment.
- How big the feature they released was, or how many tasks went into the release.
- How long the release took measured in hours. Good releases would typically take a few hours to complete, but if there were issues during the deployment, releases could take 8–12 hours.

In moving to a CD model these metrics didn't mean much anymore. Deployments were always a single task, they took minutes instead of days, and the team didn't run a full regression test suite after each release. The before-and-after release process is shown in figures 7.16 and 7.17.

Figure 7.16 shows how things were done before CD. The team was agile, but even in two-week increments a lot of change went into a release. They modified their process to the representation shown in figure 7.17.

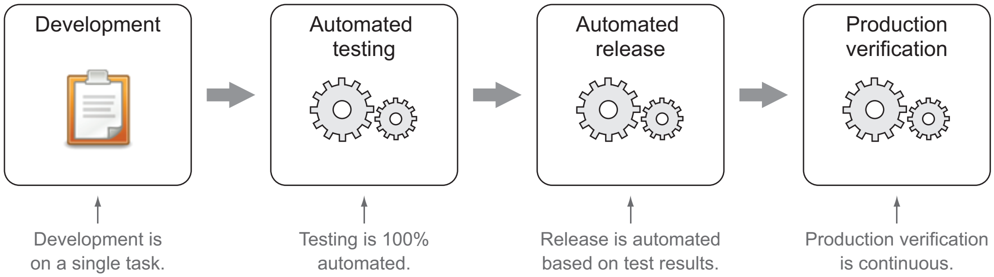

Development
Automated testing
Automated release
Production verification
Development is on a single task.
Testing is 100% automated.
Release is automated based on test results.
Production verification is continuous.

Figure 7.17 A representation of the new process they used to deploy code continuously

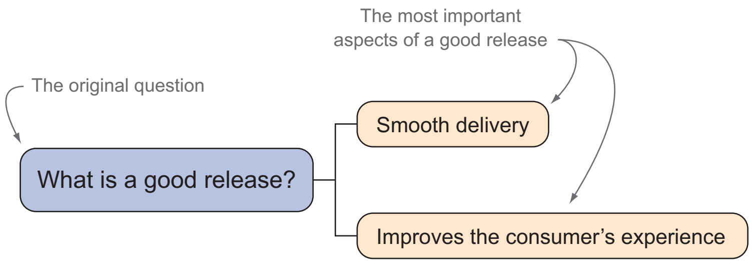

The most important aspects of a good release
The original question
What is a good release?
Smooth delivery
Improves the consumer's experience

Figure 7.18 Part 1 of the mind-mapping exercise

Now that a release was a single task, they had to change their point of view from the efficiency of a team to get a group of tasks out to the consumer to the aggregation of the individual health of tasks as they were being worked and deployed. Because they were releasing multiple times a day, they needed a simple indicator that could tell them if their delivery process was working well.

They started off with a mind map to get to the bottom of what they had and what they wanted to track. In the context of their new delivery model, they needed to determine what made up a good release. They started with the elements shown in figure 7.18.

The two most important things the team could think of were these:

- *The release was smooth.* Using CD if a release didn't work well, it slowed down the whole development cycle. A smooth release indicated the team was on track.
- *The consumer's experience improved.* Instead of asking if a release broke anything, they changed to the consumer's perspective: whether or not the product was improving.

Then they broke it down further, as shown in figure 7.19.

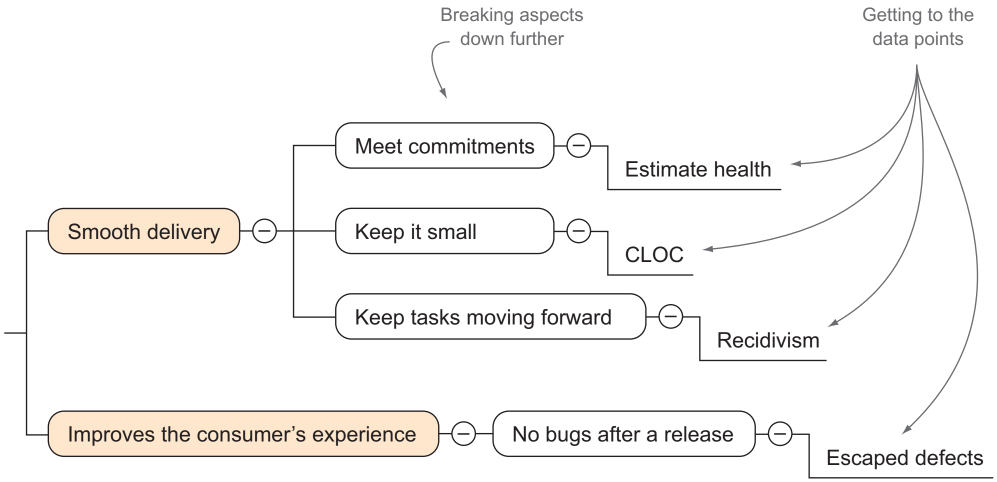

Breaking aspects down further
Getting to the data points
Smooth delivery
Meet commitments
Estimate health
Keep it small
CLOC
Keep tasks moving forward
Recidivism
Improves the consumer's experience
No bugs after a release
Escaped defects

Figure 7.19 Getting to the individual data points for this metric

For each release they believed the following indicators would tell them if things were good or bad:

- *They could make and meet their commitments.* Even though we're talking about a delivery time in days, the team still needs to be able to trust their estimates so they can have some level of predictability.
  - Measured by estimate health.
- *Tasks continued to move forward.* Anything that goes backward in the workflow could be linked to a number of problems, some of which are not understanding requirements, poor quality of work, or lack of tests. It was important that overall tasks were progressing and not regressing in the workflow.
  - Measured by recidivism.
- *Work items were kept intentionally small.* To mitigate the risk of integration issues the team had a goal of keeping changes to the codebase minimal in each release.
  - Measured by CLOC.
- *Releases don't introduce bugs.* The release process was very fast and relied heavily on automated testing. As a result, any defects that escaped into the production environment needed to be triaged so the team could figure out how to harden their testing cycle.
  - Measured by escaped defects.

They decided to come up with a formula that combined all of these data points into a single metric that would indicate the overall health of releases across their development teams, a code health determination (CHD). If this metric started to trend in the wrong direction, they could put the brakes on and dig into the problem. If the metric stayed in a good place, then teams could continue developing and deploying at will.

In their metric each of the four elements would make up 25% of the total value, and each metric would have equal weight in the total calculation. If `F` represents a formula that normalized each metric, a high level of the formula would be represented like this:

```
F1(CLOC) + F2(Estimate Health) + F3(Recidivism) + F4(Escaped Defects)
```

They decided to create a final number that was a score between 0 and 100 where 0 meant everything was going to pieces and 100 meant everything was perfect. To create this number they had to do some math to normalize the inputs and the outputs.

NORMALIZING CHANGED LINES OF CODE

After exploring their data the team concluded that each small change they made was around 50 CLOC. To come to their indicator they decided to divide CLOC by 50, chop off the decimal, and then multiply it to magnify the effect. Because 25 would be the best possible value, they subtracted the result from 25. To handle the potential of a negative number, they took the maximum between the result and 0 to ensure they would stay within the bounds of 0–25. This gave them the following function:

```
MAX((25 - ABS((int)(cloc/50)-1)*5), 0)
```

With that formula, example inputs and outputs are shown in table 7.2.

Table 7.2 Example inputs and outputs after normalizing the ideal number of CLOC

Input	Output	Result
50	25	Perfect.
135	20	Over the sweet spot range of 0–100 CLOC results in a lower score.
18	25	Under 50 CLOC but by getting the absolute value it still yields 25.
450	0	An input of 450 results in -15, which is then normalized to 0.

NORMALIZING ESTIMATE HEALTH

As you saw earlier, estimate health was a 0 if everything was good, and as the value became larger or smaller than 0, it was getting worse. To normalize this, the team had to get the absolute value, compare the result to 0, and determine what the maximum tolerance was for over- or underestimating. If the absolute value was greater than 0, it should count against the total maximum value of 25. The team decided that if they were over- or underestimating by more than three days, then there was a big problem. They made the multiplier 7, which would bring the value really close to 0 if they were over or under by three days. For anything more than that, they made the value 0, as shown in this formula:

```
MAX((25 – (ABS(Estimate Health)* 7)), 0)
```

Example inputs and outputs are shown in table 7.3.

Table 7.3 Example inputs and outputs for normalizing estimate health data

Input	Output	Result
0	25	Perfect score; estimates are right on track.
1	18	Even a day will have a significant effect on the total.
-1	18	Because we take the absolute value, -1 and 1 have the same result.
3	4	At this point it's guaranteed the total rating will be under 80.
4	0	Anything greater than 3 results in 0.

NORMALIZING RECIDIVISM

Recidivism results in a percentage or a decimal. The team was striving to keep tasks moving forward in their workflow so they wanted to have a low recidivism rate. Remember that tasks that are completed can have a maximum recidivism of 50% or .5, because they would have moved backward as many times as they moved forward. To take the output of the recidivism formula and equate that to a number between 0 and 25 where 0 is the worst and 25 is the best, they normalized the result by multiplying recidivism by 50 (25 * 2). Using that formula the highest possible rate of recidivism of completed tasks would be 0, and the lowest possible result would be 25.

As a reminder, we used the following formula earlier for recidivism:

```
Recidivism = Backwards Tasks / (Forward Tasks + Backwards Tasks)
```

Or

```
Recidivism = 25 - ((bN / (fN + bN)) *50)
```

With that formula some example inputs and outputs are shown in table 7.4..

Table 7.4 Example inputs and outputs for a normalized recidivism

Input	Output	Result
bN = 5, fN = 100	22.62	Not bad
bN = 100, fN = 100	0	The worst possible output
bN = 0, fN = 125	25	Perfect score

NORMALIZING ESCAPED DEFECTS

An escaped defect was a bug that wasn't caught in the release process; it was found after the release was considered a success. Finding any escaped defects should make the team stop and find out what happened so they could continue to improve their automated test and release. Because of this even a single escaped defect should count against the total. The formula for escaped defects was pretty simple: multiply them and subtract from 25. As with estimate health, we take the larger of 0 or the result to accommodate for potentially negative numbers.

```
MAX((25 - (Escaped Defects * 10)), 0)
```

Using this formula, even a single escaped defect would have a big impact on the overall rating. Some example inputs and outputs are shown in table 7.5.

Table 7.5 Example inputs and outputs from the formula used to normalize escaped defects

Input	Output	Result
0	25	Perfect.
1	15	A significant impact on the total rating.
2	5	This is the highest tolerance before the number goes below 0.
3	-5	Negative numbers have a huge impact on the total rating.

ADDING THE ELEMENTS TOGETHER

To get a number from 0 to 100 with four equally important elements, the calculation should be simply adding the four numbers together. But one final part of the calculation is to create a minimum and maximum for each element. The min and max of each number should represent the absolute limits of when the team needs to take corrective action. If there are two or more escaped defects each release, the team needs to stop and figure out where the problem is. Everything running smoothly would represent the minimum measurement of each input.

To calculate the min and max, the team decided to use a scale of 0–5 for each input. This would give more weight to each input if they started to trend in the wrong direction. The final metric was calculated with the algorithm shown in the next listing.

Listing 7.2 The algorithm as Groovy code

```
static int calculateCHD(cloc, estimateHealth, recidivism, escapedDefects) {
  def chd = 0

  def nCloc = ((int)(cloc/50) - 1) * 5
  def nEstimateHealth = Math.abs(estimateHealth)
  def nRecidivism = recidivism * 50
  def nEscapedDefects = escapedDefects * 10

  chd = (minMax(nCloc) + minMax(nEstimateHealth) +
  ➥ (minMax(nRecidivism) + minMax(nEscapedDefects)) * 5

  return chd
}

private static int minMax(val){
  def mm = { v ->
    if (v >= 5) {
      return 5
    } else if (v <= 0) {
      return 0
    } else {
      return v
    }
  }
  return 5 - mm(val)
}
```

Normalizes CLOC

Gets the absolute value

Multiplies it by 10 to get an integer

Escaped defects gets multiplied

Returns a number between 0 and 100

Normalizes the outputs of the individual inputs by making the maximum 5 and the minimum 0. Because the ideal of all the inputs is 0, you subtract the result from 5.

Using this algorithm the team successfully created a metric that gave them an indication of how releases were faring. If they noticed CHD going below 80, they knew they had to check in to see what was not working as expected. If they were above 80, then they could let teams continue to release at will.

Some example inputs and outputs from this formula are shown in table 7.6.

Table 7.6 Example inputs and outputs from the CHD formula with the corresponding ratings in parentheses

Estimate Health	CLOC	Recidivism	Escaped Defects	CHD
0 (25)	49 (25)	0 (25)	0 (25)	100
0 (25)	100 (20)	0 (25)	0 (25)	95

Table 7.6 Example inputs and outputs from the CHD formula with the corresponding ratings in parentheses

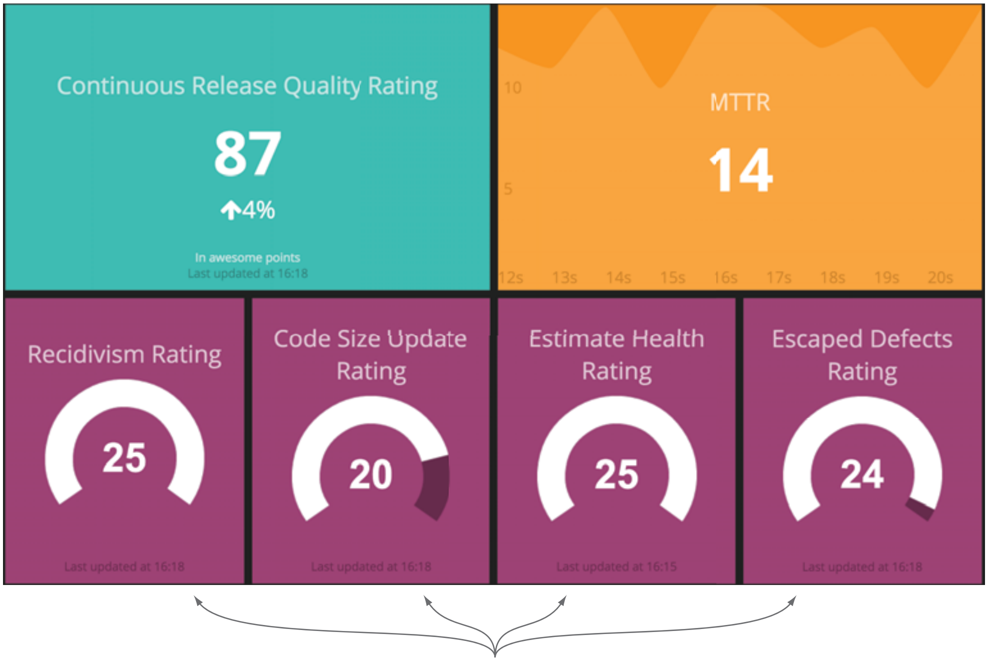
Estimate Health	CLOC	Recidivism	Escaped Defects	CHD
1 (18)	65 (25)	10% (20)	0 (25)	88
0 (25)	33 (25)	20% (15)	1 (15)	80
-2 (11)	150 (15)	0 (25)	0 (25)	76
3 (4)	350 (0)	50% (0)	0 (25)	29
-1 (18)	45 (25)	50% (0)	2 (0)	43

When the CHD rating dipped below 80 in the span of a 48-hour window, the team would immediately stop releasing code and work to get to the root of the problem.

The final step was to put the data into a dashboard to easily display it to the team. They decided to use Dashing, an open source dashboarding framework, to aggregate and display their data easily. The end result is shown in figure 7.20.

The key metric is displayed at the top of the page.

MTTR over time is also shown; if there is a problem, leadership wants to know how long it would take to react.



Each building block metric is broken out below to easily identify the source of a problem.

Figure 7.20 Using Dashing to create an easy-to-read dashboard for the larger team

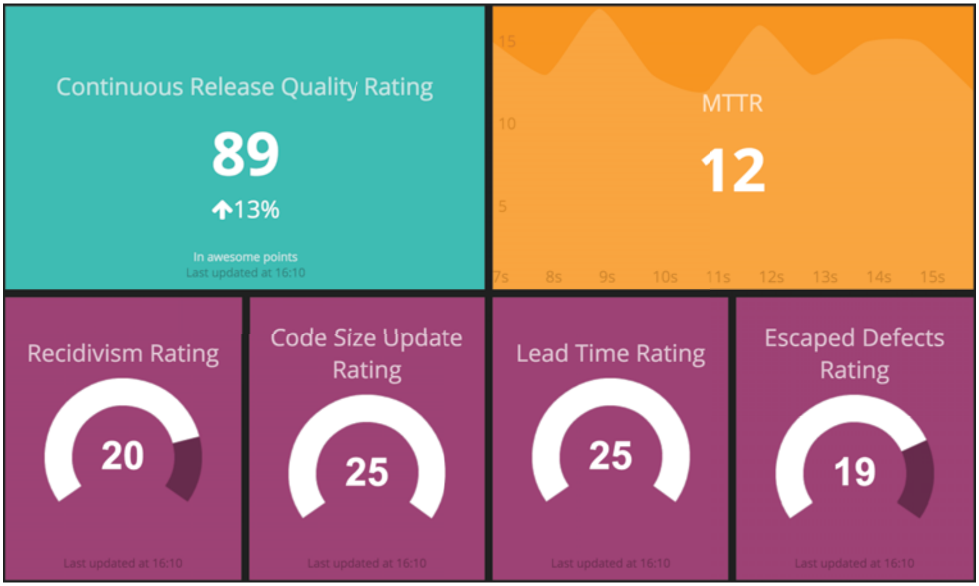
Leaders and managers could focus on the top-line metrics, and teams could focus on the next level down to make sure they were focusing on the right parts of their lifecycle. In figure 7.20 their estimate health is lower than the other components, so the team could work on their estimation accuracy sprint over sprint.

After this team started publishing their dashboard, other teams wanted to do the same. Another team that wanted to adopt this rating used lead time instead of estimates to track the predictability of their work. They decided to use this rating mostly as is; they just swapped out the estimate health rating with the lead time rating. They had an ideal lead time of seven working days, so they crafted their lead time rating to fit into the formula:

```
Lead Time Rating = 25 – ((Lead Time – 7) * 2.5)
```

Because 7 was the ideal lead time, they subtracted that from the current lead time to get the difference. When lead time started to go over by a few days, there was usually a problem, so they used a multiplier of 2.5 to enhance the effect of longer lead time relative to the maximum of 25. Their dashboard is shown in figure 7.21.

Note that the rating has gone up by 13%; that indicates that they dipped under 80 but were able to check and adjust.



The only difference in the dashboard is the rating used to measure consistency.

Figure 7.21 A second team decided to use the same rating system but tweak the composition of the metric to fit their process.

In this case they were able to take a customized metric, tweak it to meet their needs, but use the same terminology and high-level metrics that other teams in the company were using. This metric was a combination of several data points and was a call to action. Using these same techniques you can create metrics of your own based on how your team works and values that have been determined good to keep a pulse on the progress and consistency of your development team.

7.5 Summary

Using the data collected through earlier chapters, we crafted custom metrics and used them to give big-picture indicators of complex interactions. In this chapter you learned the following:

- You can create simple metrics from single data points, or you can use formulas and algorithms to combine data for more complex and insightful metrics.
- Some example metrics used in this chapter:
  - *Recidivism*—Backward Tasks / (Forward Tasks + Backward Tasks)
  - *Comment To Commit Ratio*—code reviews / (merged pull requests + commits)
  - *MTTR*—Problem Fixed Time – Problem Identification Time
  - *Continuous Release Quality Rating (CHD)*—(25 - ABS((int)(cloc/50)-1) * 5) + (25-ABS(Estimate Health) * 7) + (Recidivism = 25 – ((bN / (fN + bN)) *50)) + (25 – (Escaped Defects * 10))
- Start off by taking time to explore your data to understand how it represents your team and how they work.
  - Use the data you have already to define what in your development cycle is good and worth repeating.
- Once you have a good understanding of your data, start combining data points to create metrics to track what matters most to your development cycle.
- Mind mapping is great way to get to the root of what to measure.
- Using the result of your mind map, create the formula to generate custom metrics.
- Adding live data into your formula gives you metrics that you can track in your development cycle.
- When changing your process, looking for new ways to measure your team is a key to success.
- Different teams within the same company can use similar ratings made up of different components as long as they're measuring the same conceptual things.

8 Measuring the technical quality of your software

This chapter covers

- Measuring software quality
- Using non-functional requirements, also known as the code "ilities," to measure code quality

In previous chapters we've looked at a lot of data that you can collect throughout your development cycle to gain insight into how your team is performing. In this chapter we'll transition from measuring the process to measuring the product by using that data to determine how good your software products are.

The question we're asking in this chapter is, "Is your software good?" Before you can answer, you must ask yourself, "What is good software?" Once you know that, you can compare what you have to the ideal picture to determine where your software products stand.

At a high level there are only two dimensions to good software:

- *If it does what it's supposed to do*—These are the functional requirements.
- *If it's built well*—These are the non-functional requirements.

Functional requirements are what differentiate one software product from another. These all relate to what you're building and how your consumers are interacting with what you're producing. In chapter 6 we talked about how to track metrics that are product-specific. Metrics around your functional requirements tell you if your software is working and satisfying the consumer.

Non-functional requirements relate to how well your application is built. A well-built application will be easier to update and deploy predictably, which brings us back to early and continuous delivery through good architectures, designs, and technical excellence.

The previous chapters give you the tools you need to measure functional and non-functional requirements. This chapter uses the tools and data we've explored up to this point to measure the technical quality of your software products.

If you drill into the agile principles from the Agile Manifesto, you'll see a strong focus on delivering frequently in the face of change:

- Your highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale.

All of these are telling you to change things quickly and frequently. Then there are a few that go even deeper:

- Continuous attention to technical excellence and good design enhances agility.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- Working software is the primary measure of progress.

Anyone with an IDE and access to the internet can create working software; the really important thing to dig into is how to create a software product that you can iterate on quickly and is stable enough to handle your consumer's expectations. At a glance, measuring a codebase across these principles looks a bit nebulous, so to do that we'll dive into non-functional requirements, or the code "ilities."

Measuring your team against all the agile tenets

If you'd like to know how to measure all of the agile tenets, head over to chapter 10 where I've broken them down into what you should measure and how to do it.

8.1 Preparing for analysis: setting up to measure your code

In earlier chapters we looked into the tools and practices you should be following to get the data you need. In chapter 5 we talked about using static analysis in your CI systems, and in chapter 6 we walked through using APM tools to gather data on how well your application is functioning and if it's doing what it's supposed to do. The data generated from components in your CI and your APM systems will get you what you need.

As a reference, here's a list of the tools we've used in previous chapters and that we'll be referencing in this chapter.

Tool	Measures	Metrics
New Relic	Application monitoring	Page response time, uptime, response time, error rate
HyperSpin	Availability	Uptime, response time
Splunk	Reliability	Error rate, mean time between failures
OWASP ZAP	Security	Dynamic analysis issues
SonarQube	Maintainability	CLOC, code coverage, issues, complexity
Checkmarx	Security	Static analysis issues

8.2 Measuring the NFRs through the code "ilities"

The code "ilities," or non-functional requirements[1] (NFRs), are a set of properties that describe how well software is built. These aren't anything new; any software engineer should be familiar with them and how they indicate the quality of software. Here are some examples of non-functional requirements that indicate how well-built software is:

- *Maintainability/extensibility*—How easy is it to add features or fix issues and deploy your product?
- *Reliability/availability*—Can your consumers get what they need from your application consistently?
- *Security*—Is your consumer's information safe when they use your application?
- *Usability*—Is your application intuitive and easy to use?

Figure 8.1 shows the code "ilities" and where they fall in the development lifecycle.

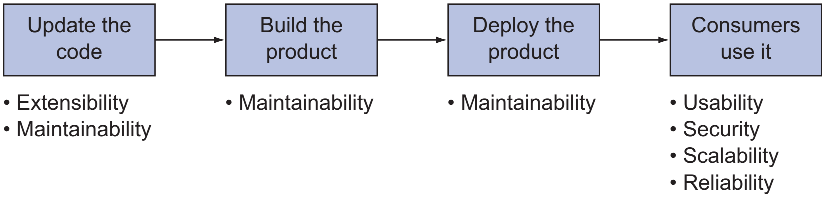


Figure 8.1 The code "ilities" illustrated through the life of a software product

[1] For much more detail, see en.wikipedia.org/wiki/Non-functional_requirement.

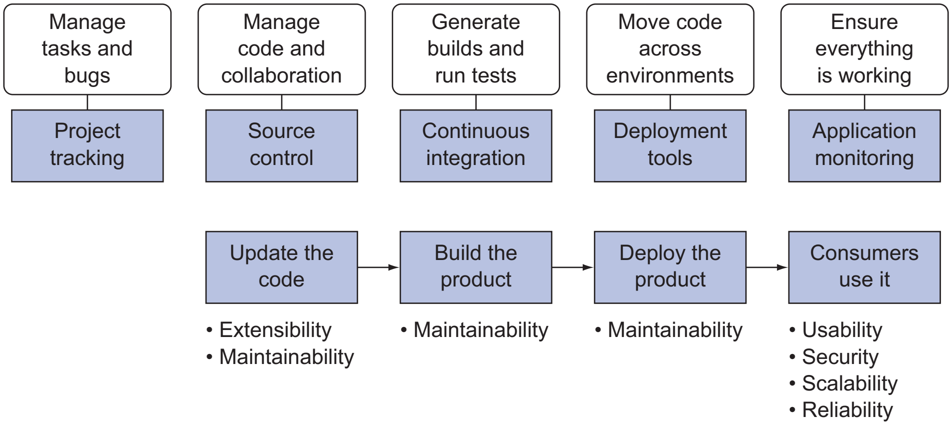


Figure 8.2 The code "ilities" in relation to the components of the build lifecycle and the tools used to measure them

If you look at this through the lens of the systems you're using to measure the product and the process, you'll see the chart shown in figure 8.2.

Maintainability encompasses getting changes out to the consumer and all the pieces of the build lifecycle that contribute to that.

Usability tells you if the consumer gets what they need out of the system with the least amount of friction possible. Wrapped up in that is whether the consumer has a secure, reliable experience that can scale to meet demand.

If you wrap extensibility with maintainability and you group security, scalability, and reliability under usability, you can redraw figure 8.2 as figure 8.3.

With these in mind we'll focus on the two biggest code "ilities" as we look into how to measure good software:

- Maintainability to represent how easily you can get changes out to the consumer
- Usability to represent customer satisfaction

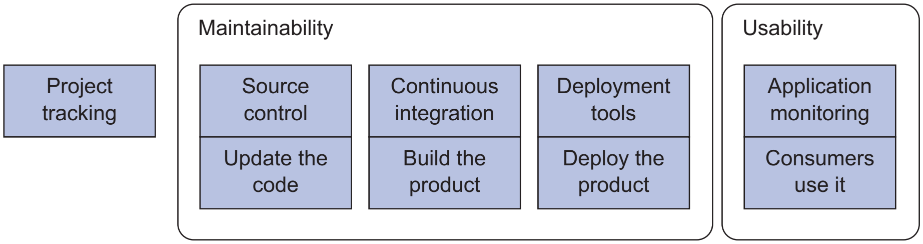


Figure 8.3 Grouping systems used in the development lifecycle together by the two parent code "ilities"

Maintainability and extensibility

In the context of agile delivery, the ability to easily change and deploy your code is paramount. Often maintainability and extensibility are treated as separate entities. Maintainability refers to the effort needed to keep your software alive and running. Extensibility refers to the level of effort required to add new features or extend an application. In today's agile world where continuous methods are becoming more common, there's no longer a clear line between maintenance and extension. For that reason I like to combine these two properties in the context of agile projects and certainly in the context of projects that practice CD.

8.3 Measuring maintainability

In an agile and CD context, maintainability means more than updating code; it also means everything that goes into delivering changes to your consumers. As you saw in figure 8.3, that includes your build and deploy systems. When you're looking at maintainability you need to look at all of the properties of your codebase that contribute to easier code updates and faster deploy times.

Maintainability encompasses your entire development cycle and as a result is best measured as the aggregate of data from several different systems. The main components of maintainability are outlined here:

- *Mean time to repair (MTTR)*—The measure of time from when you realize something is wrong in production, the issue is triaged, and a fix is determined and deployed.
- *Lead time*—The measure of time between the definition of a new feature and when it gets to the consumer.
- *Code coverage*—The amount of code measured in LOC that is covered by a unit test.
- *Coding standard rules*—How well your code adheres to standards of the language you're using.
- *How much code must be changed for features or bug fixes*—The CLOC associated with tasks that go all the way through the development cycle.
- *Bug rates*—The number of bugs that are generated as new features are being delivered.

If maintainability is measured by the ability to make frequent changes, then the two most important metrics in this list are MTTR and lead time because they both measure the amount of time it takes to get changes to your consumers.

8.3.1 MTTR and lead time

As we dig into MTTR and lead time, you'll see other important metrics that affect them. In figure 8.4 you start to see those other metrics emerging.

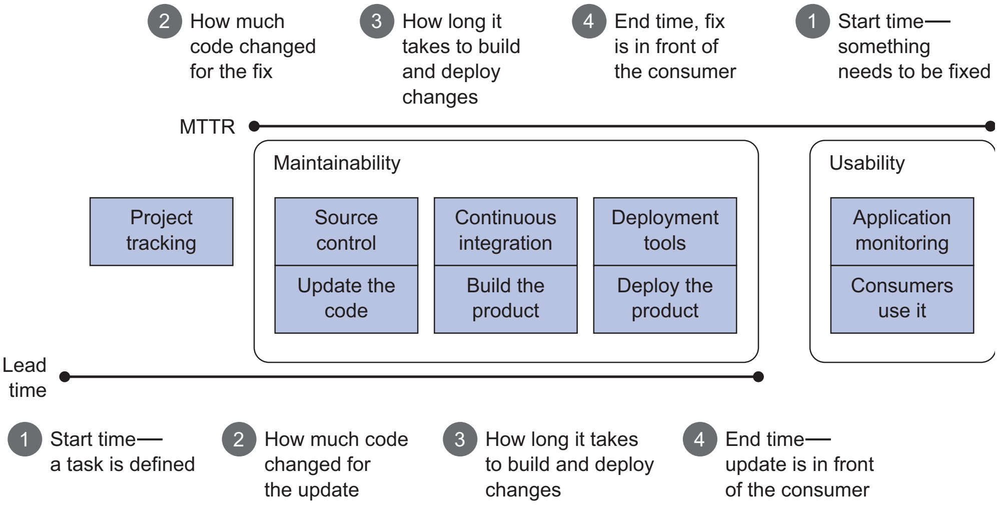

2 How much code changed for the fix
3 How long it takes to build and deploy changes
4 End time, fix is in front of the consumer
1 Start time—something needs to be fixed
MTTR
Maintainability
Usability
Project tracking
Source control
Update the code
Continuous integration
Build the product
Deployment tools
Deploy the product
Application monitoring
Consumers use it
Lead time
1 Start time—a task is defined
2 How much code changed for the update
3 How long it takes to build and deploy changes
4 End time—update is in front of the consumer

Figure 8.4 Components of lead time and MTTR as they move through the delivery process

The two key metrics to measure how fast you're getting changes to consumers are:

- *MTTR*—How long it takes to get a small code change out to the consumer
- *Lead time*—How long it takes to get a new feature out to the consumer

In figure 8.4 we started breaking out the individual steps of these two metrics. For another point of view, figure 8.5 mind maps what goes into them both.

As illustrated in figure 8.5, the key difference between the two is that MTTR measures triage of the existing system; lead time measures the addition of new pieces to

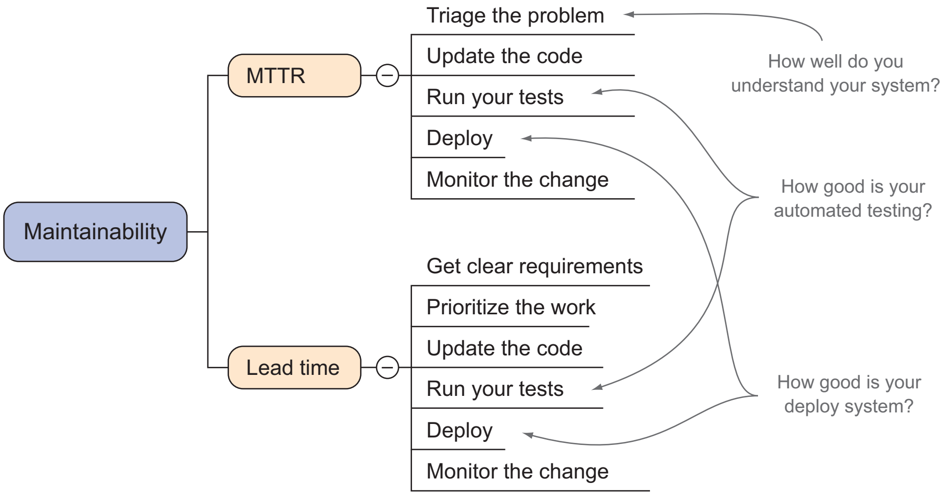


Figure 8.5 A mind map breakdown of what you need to measure maintainability

the system. Even though the systems measure similar things, finding these values comes from different places. Lead time can be found from your project tracking system (PTS) alone, because that measures the time between when something was defined and when something was completed, all of which is tracked with tickets in your PTS. You can get lead time with the following formula:

```
Lead Time = PTS: Task Complete - PTS: Task Start
```

The start time in MTTR is when a problem arises for your consumers. These problems are detected in your APM systems, the code changes are made in your SCM system, and the update makes it through your build system and gets deployed. To keep it really simple you can calculate MTTR with the following:

```
MTTR = APM: Anomaly End - APM: Anomaly Start
```

This formula is great for giving you the big picture, but it glosses over a lot of the details that go into these time ranges. If you end up with an MTTR of 16 hours, the next question will inevitably be, "How can we get faster at fixing our code?" You can start by breaking down the phases of your delivery into the phases of your delivery cycle. If you go back to the steps defined in the mind map in figure 8.5, you can start visualizing where you need to focus to continually improve your delivery times.

NOTE MTTR and lead time are measured so similarly that for the rest of this section I'll focus on MTTR. In the case study at the end of this chapter you'll find an example of breaking down lead time.

Let's start with an MTTR of 35 hours over four releases. If you break that out to see the amount of time it takes for each release, you may end up with something like figure 8.6.

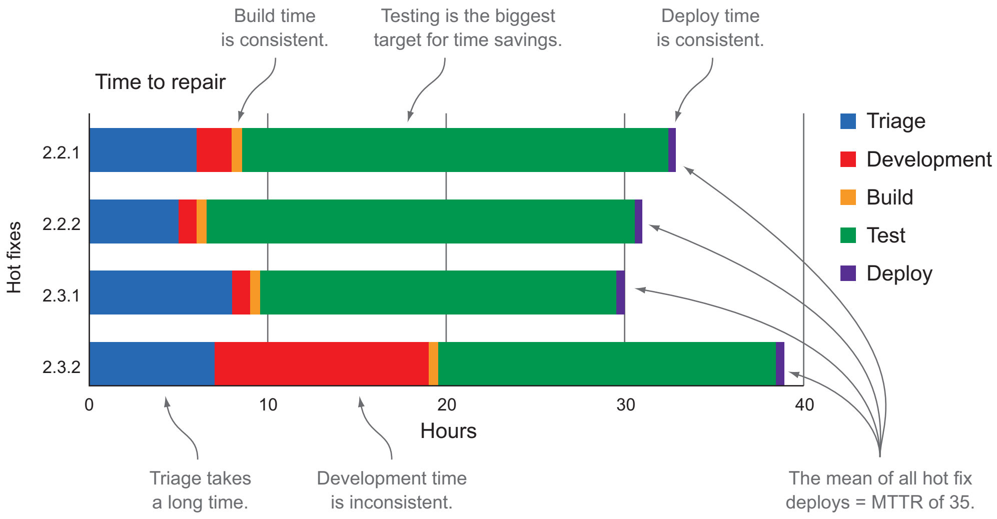


Figure 8.6 An example breakdown of MTTR over four releases; testing takes much more time than anything else in the fix cycle.

In this example the MTTR is over 30 hours, most of which is testing. It also takes a consistently long time to triage problems, and developing fixes doesn't seem predictable. The best parts of the process to address are the ones that are usually long; that way you can get a consistently measurable benefit from addressing them.

The example in figure 8.6 is a good example of code with poor maintainability:

- The system appears to be so complex it takes a very long time to triage. In my experience if a problem takes longer than an hour to triage, your system is way too complex.
- Development time of fixes is unpredictable. Release 2.3.2 took a much longer time to fix than the other releases; it would be worth drilling in to find out the cause.
- Test cycles that rely heavily on manual testing or a focus on complete regression for small changes are a symptom of a lack of understanding of production behavior.

In figure 8.6 it's hard to tell where to start. If you want to get fixes out faster, it looks like you should:

- Try to improve the test cycle.
- Figure out why it takes so long in triage to understand what's wrong when the system breaks.

Figure 8.7 shows a very different scenario that depicts predictability through a breakdown of MTTR.

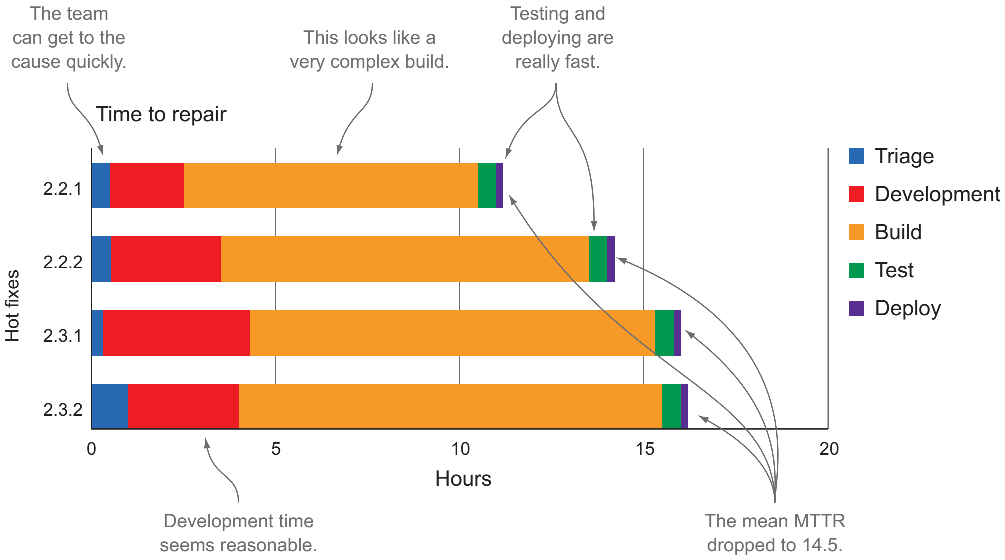


Figure 8.7 Another example breaking down MTTR over four releases: with testing fully automated, MTTR goes way down. The next focus for improvement should be the build process.

In figure 8.7 the team can get to the root of problems and update the code quickly, indicating the codebase is much more maintainable than that shown in figure 8.6. But the build takes a really long time, so reducing build time is an actionable item the team can take away from these measurements. In this case it may also be a good idea to cross-reference the shorter test cycle with metrics to ensure that your team has adequate test coverage and they're not just skipping tests for the sake of speed.

For the teams on the ground adding features and fixing problems, you can break into even finer details like how much code has to change in these time ranges and what's taking time inside the build process. For this detail you need to add more data.

8.3.2 Adding SCM and build data

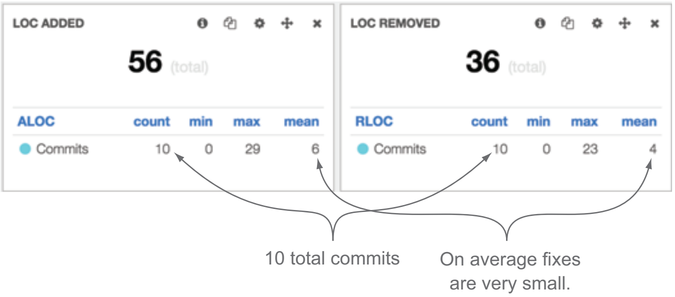
What gets built and how often is the next key indicator of a maintainable codebase. Commits with a high CLOC (changed lines of code) are a sign that it takes a lot of work to change things. If you start to break down the elements of a change to your software products, the number of lines of code it takes to fix things or deliver new features is one of the primary indicators of maintainability. *A good target to have is a low CLOC, a high number of releases, and a low number of fixes*. This ultimately shows that your code is maintainable enough to easily make small tweaks and get in front of the consumer quickly.

Let's look at an example. A team is practicing CD and releasing updates to their consumers three times a day. They keep track of their CLOC for each release, as shown in figure 8.8, which includes a month's worth of hot-fix release data. Based on their target, is their code base maintainable?

Figure 8.8 shows the following statistics over the course of a month with 60 releases made by the CD system:

- Ten commits were made and different hotfixes were deployed.
- Of those ten commits, on average only six lines of code were added and four were removed.
- Therefore 16% (10 in 60) of the total releases were hot fixes.

If you assume that the CD system makes 3 releases in an 8-hour day with about 3 hours between releases, then the following are true:



LOC ADDED

56 (total)

ALOC	count	min	max	mean
Commits	10	0	29	6

LOC REMOVED

36 (total)

RLOC	count	min	max	mean
Commits	10	0	23	4

10 total commits

On average fixes are very small.

Figure 8.8 The amount of code that changes for all hot fixes in a month

- For the 10 releases that needed to be fixed, for about 30 hours in the month (out of 730 total hours in a month) the code was in a state that needed to be fixed.
- That's just 4% of the entire month when the released version of the product had a problem.

Pretty good.

As a counter example, let's take a team that releases their software every two weeks with 1 day of regression testing and a 4-hour release at the end of their development cycle. To complete 10 releases would take 10 full days of regression testing and an additional 5 days of release activity, and that's not even measuring the time it would take to find problems and fix code. That would mean that over half of their time was used fixing bugs no matter what the CLOC was on the fixes.

Realistically there are potential outcomes in these scenarios:

- Teams don't fix bugs because they know they won't be able to complete any more features if they spend all of their time testing and releasing code.
- Teams jam as many fixes as possible into a release, and thus you'll see a very high CLOC on hot fixes. This results in longer test cycles and bigger releases, both of which are indicators of code with poor maintainability.

To make sure CLOC and number of fixes are relevant, ensure that you're also counting total deployments so you can get a percentage of fix to release (FRP). To summarize, the formula for that is

```
Fix Release Percentage (FRP) = Total fixes / releases
```

You want your fix release percentage to be as low as possible. A 0 is perfect and anything over .5 is usually pretty bad. By combining MTTR and FRP you can calculate a maintainable release rating with this formula:

```
Maintainable Release = MTTR(in minutes) * (Total Fixes / Releases)
```

In this case the closer to 0 you are the better. Continuing the initial example, if you have 60 releases and 10 fixes, you have a 16% FRP. If the same team has a 4-hour MTTR, that team then has a maintainable release rating of 40. To compare this to a bad number, if you have a team that has a hot fix with every release and an MTTR of 12 hours, that would give you a maintainability release rating of 720. Due to the broad range of potential values, this metric becomes a good benchmark that your team can focus on improving over time rather than targeting a specific value out of the gate. Table 8.1 shows example inputs and outputs for the maintainable release rating (MRR).

Table 8.1 Example inputs and outputs for a maintainable release rating

Inputs	MRR	Notes
MTTR = 240 minutes Total fixes = 3 Total releases = 30	24	Given a 4-hour release time, this team needs to fix only 1 in every 3 releases, so the rating isn't bad.

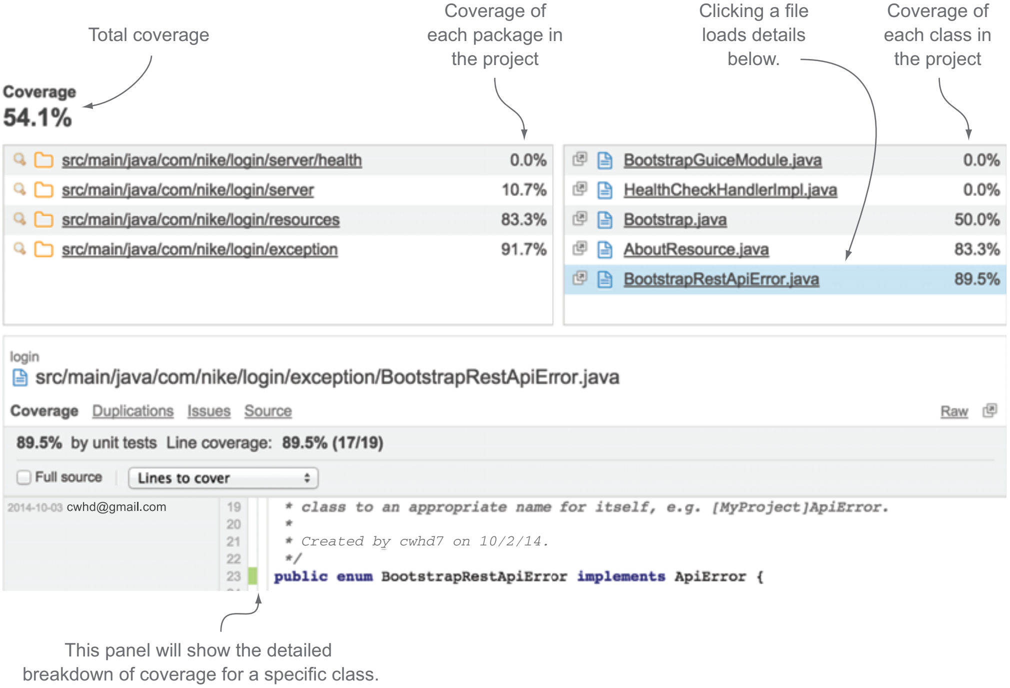
Table 8.1 Example inputs and outputs for a maintainable release rating ***(continued)***

Inputs	MRR	Notes
MTTR = 240 minutes Total fixes = 10 Total releases = 5	480	Even though this team has a 4-hour release time, they have 2 fixes per release on average, which is really bad.
MTTR = 480 minutes Total fixes = 1 Total releases = 100	4.8	This team has a higher MTTR, but they rarely have to fix things (once in 100 releases), so they're doing great.

8.3.3 Code coverage

Code coverage is the percentage of your codebase that's covered by automated tests. You can measure code coverage during the build process with a number of tools that run during the build. Some examples are Cobertura, JaCoCo, Clover, NCover, and Gcov. I'll take advantage of the comprehensive dashboards in SonarQube to show you how to turn code coverage reports into actionable data. An example coverage report is shown in figure 8.9.

In theory, if you have great test coverage, then your project is more maintainable because developers can find out if their changes affected the rest of the system by simply running their unit tests. The tricky thing about code coverage is that it tells you how



Total coverage

Coverage of each package in the project

Clicking a file loads details below.

Coverage of each class in the project

Coverage
54.1%

src/main/java/com/nike/login/server/health	0.0%
src/main/java/com/nike/login/server	10.7%
src/main/java/com/nike/login/resources	83.3%
src/main/java/com/nike/login/exception	91.7%

BootstrapGuiceModule.java	0.0%
HealthCheckHandlerImpl.java	0.0%
Bootstrap.java	50.0%
AboutResource.java	83.3%
BootstrapRestApiError.java	89.5%

login
src/main/java/com/nike/login/exception/BootstrapRestApiError.java

89.5% by unit tests Line coverage: 89.5% (17/19)

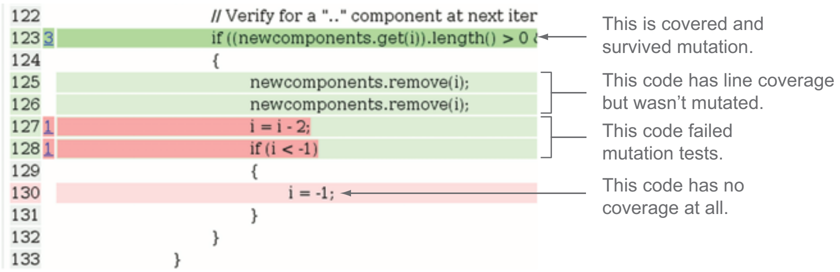
□ Full source Lines to cover

2014-10-03 cwhd@gmail.com

```
 * class to an appropriate name for itself, e.g. [MyProject]ApiError.
 *
 * Created by cwhd7 on 10/2/14.
 */
public enum BootstrapRestApiError implements ApiError {
```

This panel will show the detailed breakdown of coverage for a specific class.

Figure 8.9 An example breakdown of code coverage in SonarQube



```
                // Verify for a ".." component at next iter
                if ((newcomponents.get(i)).length() > 0 &
                {
                    newcomponents.remove(i);
                    newcomponents.remove(i);
                    i = i - 2;
                    if (i < -1)
                    {
                        i = -1;
                    }
                }
            }
```

This is covered and survived mutation.

This code has line coverage but wasn't mutated.

This code failed mutation tests.

This code has no coverage at all.

Figure 8.10 An example mutation test report using Pitest

much of your codebase is exercised by tests, not how good your tests are. If you write a test that runs a method but doesn't assert anything to see if the result is the expected one, then that method is considered covered, even though the test doesn't check anything. There are two things you can do to your code coverage metric to add more value:

- *Mutation testing*—Comparing test results with test results after the underlying code has changed or mutated
- *Adding data points to show the value of coverage*—Using data from PTS, SCM, and CI

Mutation testing is an automated test for your tests. Essentially it messes with your code before your unit tests run. This way, if your tests pass after inserting blatant errors, you know that even though a test is executing your code, it's not really testing your code. A great tool for this is Pitest (pitest.org/). An example screenshot of a Pitest analysis is shown in figure 8.10.

Another strategy for vetting your code coverage numbers is to track it alongside other metrics. When your codebase has a great suite of automated tests, that usually translates into a fix release percentage and a maintainable release rating. If you have great coverage but your releases aren't going well, that's a good sign that either you're not testing the right things or your tests are bad.

Often code coverage gets lumped in with static code analysis because the two reports are typically generated at the same time. The two are complementary but different measurements of a maintainable codebase.

8.3.4 Adding static code analysis

Static code analysis can check your code for best practices against common rule sets for whatever language you're using. A number of tools can do this for you. SonarQube is a great option.

As I mentioned in chapter 5, *SonarQube in Action*, published by Manning, is a great book with a lot of details, but I'll give you a quick rundown of maintainability highlights that you can start using today.

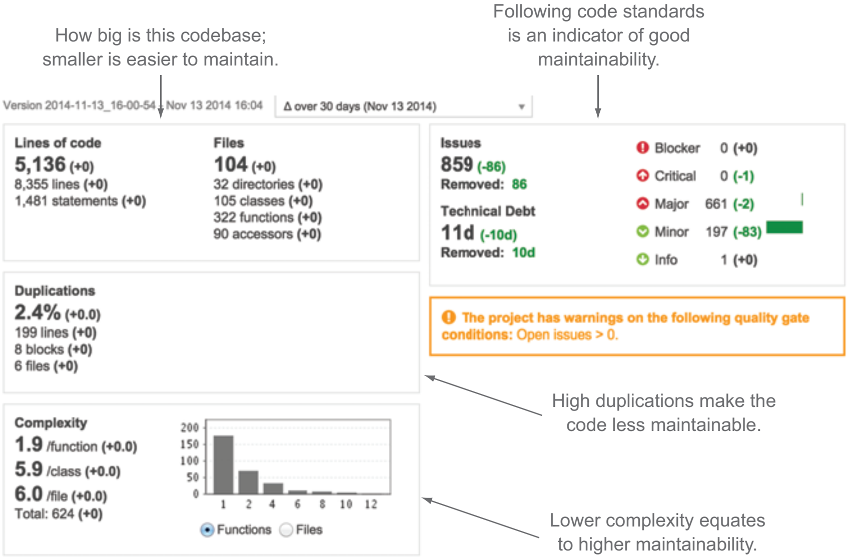
The easiest things to look at are these:

- *Lines of code*—A large codebase is usually more complicated to build and deploy. This usually also indicates that your codebase does a lot of things, which usually sets your team up for conflicting changes when different developers or teams are working on different features. For better maintainability it's good to keep your modules small and focused.

- *Duplications*—The classic problem with duplicate code is that if you change it in one place and not another, you end up with a bug where you didn't change the code. Duplications also directly conflict with coding practices of modularity and reusability (a few more "ilities" that fit under maintainability). Ideally you shouldn't have duplicate code.
- *Issues*—These will measure your code against the coding standards for whatever language you're using. The better developers are at writing code aligned with standards, the easier it is to make changes and the easier it is for new developers or other teams to jump in and make changes. Static analysis tools all have classifications of issues from really important to not so important; at a minimum you should ensure that the really important issues are kept at zero.
- *Complexity*—Also known as *cyclomatic complexity* or the amount of nested code you have. If you have an `if` statement nested inside a loop and two other `if` statements, you'll have a high cyclomatic complexity. This code is hard to read, debug, and test. Ideally you want to keep complexity as low as possible.

Figure 8.11 shows a screenshot of SonarQube and some default key statistics to be aware of.

All of the metrics are really easy to read, and you can see how they change over time. Figure 8.11 has an interesting anomaly; over the last 30 days no code has



How big is this codebase; smaller is easier to maintain.

Following code standards is an indicator of good maintainability.

Version 2014-11-13_16-00-54 | Nov 13 2014 16:04 | Δ over 30 days (Nov 13 2014)

Lines of code
5,136 (+0)
8,355 lines (+0)
1,481 statements (+0)

Files
104 (+0)
32 directories (+0)
105 classes (+0)
322 functions (+0)
90 accessors (+0)

Issues
859 (-86)
Removed: 86

Technical Debt
11d (-10d)
Removed: 10d

Blocker	0	(+0)
Critical	0	(-1)
Major	661	(-2)
Minor	197	(-83)
Info	1	(+0)

Duplications
2.4% (+0.0)
199 lines (+0)
8 blocks (+0)
6 files (+0)

The project has warnings on the following quality gate conditions: Open issues > 0.

Complexity
1.9 /function (+0.0)
5.9 /class (+0.0)
6.0 /file (+0.0)
Total: 624 (+0)

High duplications make the code less maintainable.

Lower complexity equates to higher maintainability.

Figure 8.11 Key metrics from static analysis that indicate maintainability

changed but the number of issues and estimated tech debt have decreased. That's probably because this team ran their analysis and then changed the rule set by which the issues were measured. Updating code standards is a great task for any development team because it forces the team to look at the rules, discuss them, and decide what makes the most sense for the way they write code.

A metric that I'm not diving into is estimated technical debt. SonarQube has built-in algorithms that will calculate how long it should take your team to fix all the issues identified by static analysis. Even though this is a really cool feature of SonarQube, it doesn't take into account bigger pictures like potential problems with your build system or architectural issues related to integration between code modules.

Balancing tech debt and delivery

In chapter 3 our case study addressed identifying tech debt trends using PTS data. It's important to use this data to find when poor maintainability is starting to creep into your software products because it's easy to tie to the bottom line. If you're tracking how the speed of delivery slows down as a result of tech debt, it's usually not that tough to get buy-in from project sponsors to clean it up to ensure you can keep delivering at your best rate instead of constantly slowing down. You can add the data-tracking maintainability into your tech debt impact analysis to break it down into small enough chunks for your development team to take action on, and you can use the PTS data as an indicator to show your sponsors how it affects their project. You'll see more on this in chapter 9.

Static analysis is awesome and every development team should use it. But like everything else, static code analysis alone can't give you the complete picture. Some problems with using static code analysis alone are these:

- Even if the code is easy to update and has great tests, that doesn't make it easy to deploy.
- There could be integration issues with the larger systems that you can't see from static analysis.
- This doesn't check the behavior of the application.

As with the rest of your maintainability metrics, static code analysis is a great tool to use to give you actionable insight into your codebase but doesn't stand alone as the de facto measurement for maintainability.

What static analysis alone doesn't give you is the correlation between the textbook quality of your code and its actual performance, or how it relates to the productivity of your team.

8.3.5 Adding more PTS data

Adding PTS data will help round out the picture to show how all of these metrics affect how your team is delivering. We covered how to collect these metrics and their importance in previous chapters, so here I'll highlight how they tie back into indicating how maintainable your codebase is:

- *Bug rates*—If you see lots of bugs being generated as you're delivering good code, that's an indication your code isn't maintainable. You'll usually see bug rates parallel poor code coverage, high issues from static analysis, and high MTTR and lead times.
- *Recidivism*—Recidivism directly impacts MTTR and lead time because as things go backward in the workflow, you're effectively doing the same things more than once. If a developer says something is done but it ends up getting rejected by QA, that development work needs to be done again, impacting your delivery times. High recidivism typically parallels high bug rates and can be affected by all the other metrics we've discussed relating to maintainability.
- *Velocity*—All of the metrics discussed in this chapter potentially affect velocity. It's possible to have a steady cadence of delivery on a terrible codebase. In that case you'll have a team that's not reaching their potential output. If you see a consistent velocity with terrible maintainability metrics, then you should think about how you can make your codebase more maintainable to get more done.

If your application is very maintainable, you should be able to deliver to your consumers consistently and with high quality. Delivery is the first half of the equation; now you have to make sure that what you're delivering is what your consumers want and something they can use.

8.4 Measuring usability

There are several different "ilities" that demonstrate how usable your application is or how good an experience you're giving to your consumers. The big ones we'll focus on are outlined in figure 8.12.

Most of these metrics measure how well your application is performing. Performance metrics are important because they give you real data on how healthy your

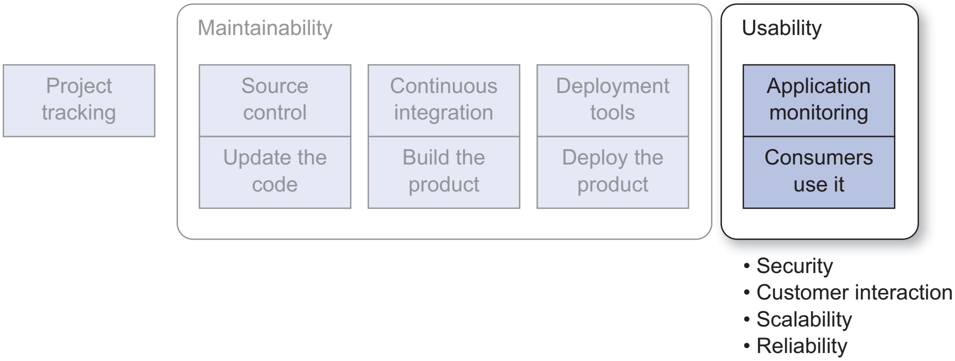


Figure 8.12 Once something has been delivered, you need to measure it to make sure it's good.

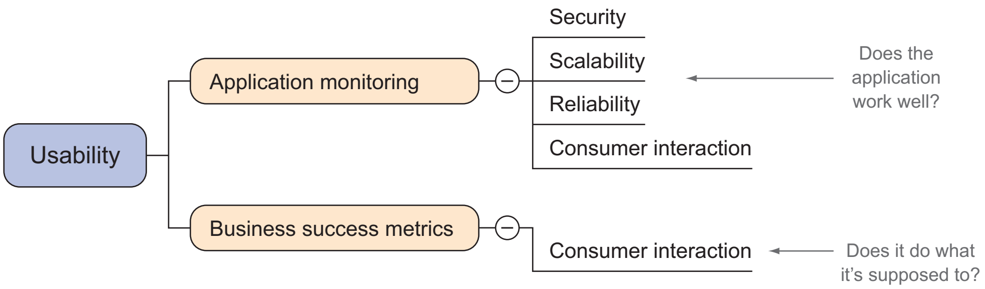


Figure 8.13 Mind mapping the parent-child relationships of usability and data

application is, and combined with business success metrics and customer satisfaction they show you where you can improve your product to make your consumers happy.

To show how to break maintainability into measureable and actionable items, I started with the high-level indicators—MTTR and lead time—and then dove into the elements that affect those metrics. For usability the parent indicators of success are the business success metrics as defined for your application and how consumers interact with your site, as shown in figure 8.13.

In chapter 6 I talked about using arbitrary metrics in your logs or from instrumented code to measure the value your consumers are getting from your application. These metrics reflect your functional requirements, or the specific data points that tell you if your application is providing value to your consumers. Because an entire chapter is devoted to high-level metrics to track, I'll jump right into the elements that affect consumer satisfaction and business success criteria and where to get them.

8.4.1 Reliability and availability

Reliability and availability are often grouped together when the code "ilities" are discussed. I used to think they were the same thing, but when I measure them they definitely point to different but closely related factors.

Availability is the measure of how much time your application is functioning relative to how frequently your consumers want to use it. If you have a web application for a global user base, then your availability needs to be as close to 100% as possible because people from different time zones will be logging in and using it as the globe spins. In addition, you may have contracts with partners that specify service level agreements (SLAs) that bind you to supporting availability times of 99.999%, or about 4 hours a month. Conversely, if your application is used in a retail location that is open from 7 a.m. to 9 p.m. seven days a week, it probably doesn't matter if it's not available from 9:01 p.m. to 6:59 a.m. every day. Availability can be measured by the following metrics:

- *Uptime*—What percentage of the time your application is functioning.
- *Page response time*—If your application is so slow that pages aren't loading in the time your consumer expects, you can consider it unavailable.

Reliability means how consistently your application does what it's supposed to do. If you have intermittent issues with your application, it's not reliable. For example, if you have an e-commerce application where you can't add items to the shopping cart when the site is under heavy load, it's not very reliable. You can measure reliability with the following metrics:

- *Mean time between failures*—How frequently your application breaks for your consumers.
- *Response time*—If your response time isn't consistent, then your application isn't reliable.
- *Error rate*—By monitoring your logs, you can see how many errors you have over time.

You could have an application that is up all the time (highly available) but doesn't function correctly 50% of the time (low reliability). You could also have an application that does what it's supposed to do all the time (highly reliable) but is down for maintenance for an hour a day (low availability).

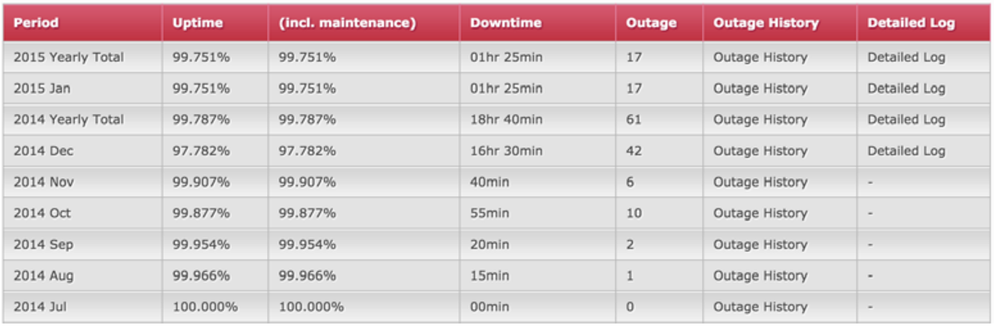
Even though the two are different, there's overlap in how they're measured. Two products that help measure uptime are Hyperspin (www.hyperspin.com/en/) and New Relic (www.newrelic.com/). Example reports from Hyperspin in figure 8.14 show the data you need to measure availability.

As you can see, you have uptime, downtime, and the number of outages, and you can click into the report to get details on outage history.

New Relic has a similar feature that creates an availability report. The New Relic version is shown in figure 8.15.

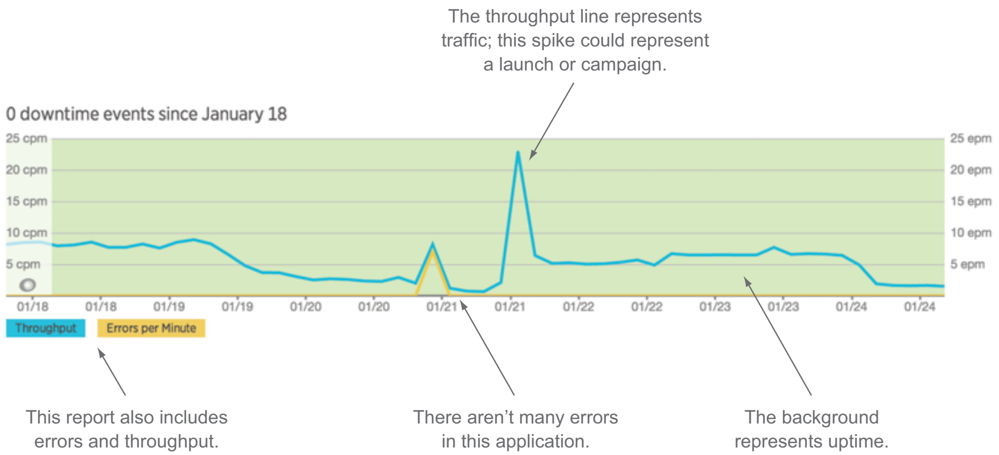
As you can see, availability is straightforward to measure and there are plenty of tools that can monitor it for you. Figure 8.15 also shows that New Relic can give you error rates for your application.

Another way to get error rate and mean time between failures is through log analysis. Figure 8.16 shows a simple query in Splunk (www.splunk.com/) that can get the error rate as output by the logs.



Period	Uptime	(incl. maintenance)	Downtime	Outage	Outage History	Detailed Log
2015 Yearly Total	99.751%	99.751%	01hr 25min	17	Outage History	Detailed Log
2015 Jan	99.751%	99.751%	01hr 25min	17	Outage History	Detailed Log
2014 Yearly Total	99.787%	99.787%	18hr 40min	61	Outage History	Detailed Log
2014 Dec	97.782%	97.782%	16hr 30min	42	Outage History	Detailed Log
2014 Nov	99.907%	99.907%	40min	6	Outage History	-
2014 Oct	99.877%	99.877%	55min	10	Outage History	-
2014 Sep	99.954%	99.954%	20min	2	Outage History	-
2014 Aug	99.966%	99.966%	15min	1	Outage History	-
2014 Jul	100.000%	100.000%	00min	0	Outage History	-

Figure 8.14 Example basic report from Hyperspin



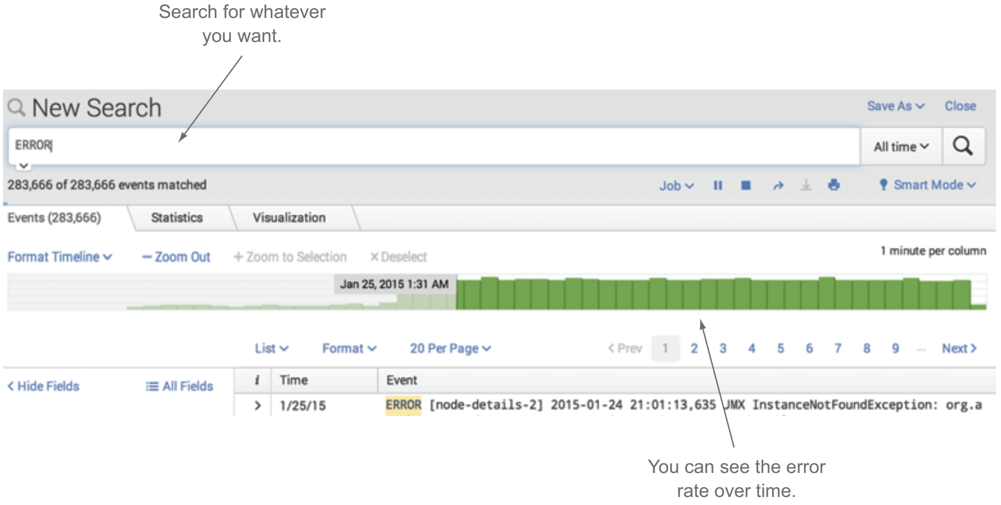
The throughput line represents traffic; this spike could represent a launch or campaign.

This report also includes errors and throughput.

There aren't many errors in this application.

The background represents uptime.

Figure 8.15 New Relic's version of visualizing and reporting availability



ERROR

283,666 of 283,666 events matched

i	Time	Event
>	1/25/15	ERROR [node-details-2] 2015-01-24 21:01:13,635 JMX InstanceNotFoundException: org.a

Search for whatever you want.

You can see the error rate over time.

Figure 8.16 Visualizing error rate using Splunk

Splunk and New Relic have rich APIs that offer reliability and availability data to visualize along with the rest of your data. Regardless of which tools you use, measuring the health of your application by availability and reliability produces core metrics that tell you how well you software is working for your consumer.

8.4.2 Security

Your consumers expect to be in a secure place where their personal information isn't in jeopardy of being stolen. I list security under usability because if you don't have a secure site, then no one will want to use it. Keeping your customers secure and their data safe is one of the most critical things to keep in mind if you're in the business of building consumer-facing software products.

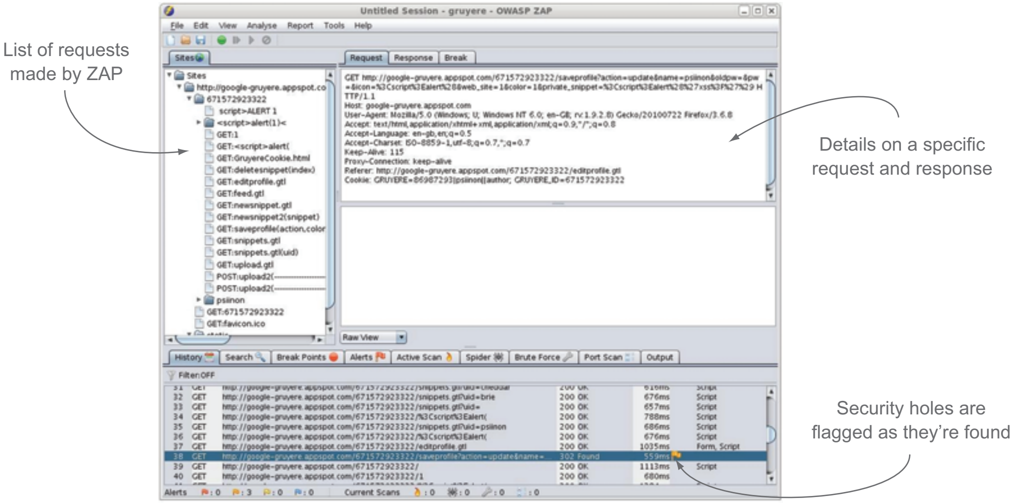
"Hack yourself first" is a good mantra to ensure you're testing your site for security holes. The following tools are popular options for ensuring your site is secure:

- *Static code analysis*—SonarQube has some rules on ensuring your application is secure, but there are other static analysis tools that specialize in this. A few popular ones are Checkmarx (www.checkmarkx.com/), Coverity (www.coverity.com/), and Fortify.[2]
- *Dynamic code analysis*—Examples are Veracode[3] and WhiteHat (www.whitehatsec.com/). A great open source option is OWASP Zed Attack Proxy[4] (ZAP).

Security is a big enough topic for a completely separate book, but if you're not sure where to start, here are some tips.

Memorize the Open Web Application Security Project (OWASP) top 10.[5] Note that the details on this page sometimes gets out of date but the list remains valid. There's a list for mobile applications as well as one for web applications. These lists give you the 10 most important things to keep in mind as you're designing a secure software product.

OWASP ZAP is a great way get started doing security scanning on your application. ZAP will use a spider to crawl through your site and use common hacking techniques to attack it and report back on vulnerabilities. An example of ZAP is shown in figure 8.17.


List of requests made by ZAP
Details on a specific request and response
Security holes are flagged as they're found
Untitled Session - gruyere - OWASP ZAP
File
Edit
View
Analyse
Report
Tools
Help
Sites
Request
Response
Break
Raw View
History
Search
Break Points
Alerts
Active Scan
Spider
Brute Force
Port Scan
Output
Filter:OFF

```
Sites
  http://google-gruyere.appspot.co
    671572923322
      script>ALERT 1
      <script>alert(1)<
      GET:1
      GET:<script>alert(
      GET:GruyereCookie.html
      GET:deletesnippet(index)
      GET:editprofile.gtl
      GET:feed.gtl
      GET:newsnippet.gtl
      GET:newsnippet2(snippet)
      GET:saveprofile(action,color
      GET:snippets.gtl
      GET:snippets.gtl(uid)
      GET:upload.gtl
      POST:upload2(-----------------
      POST:upload2(-----------------
      psiinon
    GET:671572923322
    GET:favicon.ico
```

```
GET http://google-gruyere.appspot.com/671572923322/saveprofile?action=update&name=psiinon&oldpw=&pw
=&icon=%3Cscript%3Ealert%28&web_site=1&color=1&private_snippet=%3Cscript%3Ealert%28%27xss%3F%27%29 H
TTP/1.1
Host: google-gruyere.appspot.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-GB; rv:1.9.2.8) Gecko/20100722 Firefox/3.6.8
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-gb,en;q=0.5
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Proxy-Connection: keep-alive
Referer: http://google-gruyere.appspot.com/671572923322/editprofile.gtl
Cookie: GRUYERE=86987293|psiinon||author; GRUYERE_ID=671572923322
```

```
32  GET  http://google-gruyere.appspot.com/671572923322/snippets.gtl?uid=brie        200 OK     676ms     Script
33  GET  http://google-gruyere.appspot.com/671572923322/snippets.gtl?uid=            200 OK     657ms     Script
34  GET  http://google-gruyere.appspot.com/671572923322/%3Cscript%3Ealert(           200 OK     788ms     Script
35  GET  http://google-gruyere.appspot.com/671572923322/snippets.gtl?uid=psiinon     200 OK     686ms     Script
36  GET  http://google-gruyere.appspot.com/671572923322/%3Cscript%3Ealert(           200 OK     676ms     Script
37  GET  http://google-gruyere.appspot.com/671572923322/editprofile.gtl              200 OK     1035ms    Form, Script
38  GET  http://google-gruyere.appspot.com/671572923322/saveprofile?action=update&name=...  302 Found  559ms
39  GET  http://google-gruyere.appspot.com/671572923322/                             200 OK     1113ms    Script
40  GET  http://google-gruyere.appspot.com/671572923322/1                            200 OK     680ms
Alerts  : 0  : 3  : 0  : 0    Current Scans  : 0  : 0  : 0  : 0
```

Figure 8.17 An example screenshot of OWASP ZAP

[2] Fortify static code analyzer, www8.hp.com/us/en/software-solutions/static-code-analysis-sast/index.html?.

[3] Executing data in real time, www.veracode.com/products/dynamic-analysis-dast/dynamic-analysis.

[4] Integrated penetration testing tool for finding vulnerabilities in web applications, www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.

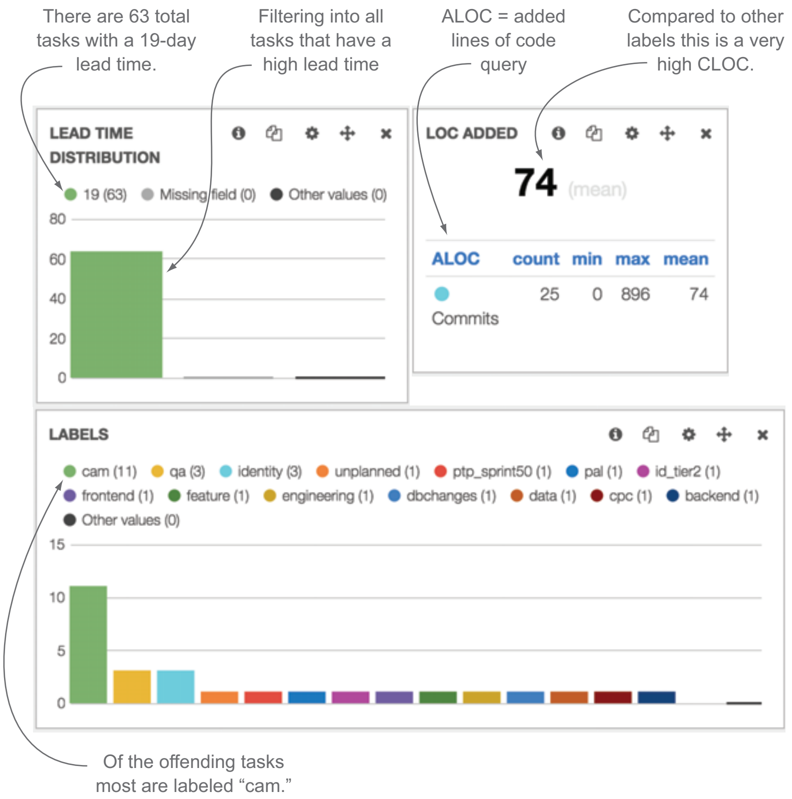
[5] Tips for designing secure software, www.owasp.org/index.php/OWASP_Top_Ten_Cheat_Sheet.

If your site fails security analysis, you should stop whatever you're doing and fix the problems. Suffice it to say that a site with poor security has very low usability.

8.5 Case study: finding anomalies in lead time

In this case study we'll look at a team that's practicing Kanban for their task management. They had a steady stream of work, and they paid close attention to lead time to make sure that everything was flowing through their development cycle as consistently as possible. They started tagging their tasks so they could see not only overall lead time but also lead time per tagged group of tasks. Once they started tracking tagged groups of tasks, they started to realize that lead time for everything tagged "cam" was much higher than everything else and was skewing their average. Figure 8.18 shows what they saw when they filtered by the "cam" tag.

Once the team began digging, they also noticed that everything tagged "cam" also had a much higher CLOC. They broke out their lead time to see where in the process the extra time was coming in. As shown in figure 8.19, they could see that the build and deploy process was fine, but development was taking longer than usual on these tasks.



There are 63 total tasks with a 19-day lead time.

Filtering into all tasks that have a high lead time

ALOC = added lines of code query

Compared to other labels this is a very high CLOC.

LEAD TIME DISTRIBUTION

● 19 (63) ● Missing field (0) ● Other values (0)
80
60
40
20
0

LOC ADDED

74 (mean)

ALOC	count	min	max	mean
● Commits	25	0	896	74

LABELS

● cam (11) ● qa (3) ● identity (3) ● unplanned (1) ● ptp_sprint50 (1) ● pal (1) ● id_tier2 (1)
● frontend (1) ● feature (1) ● engineering (1) ● dbchanges (1) ● data (1) ● cpc (1) ● backend (1)
● Other values (0)
15
10
5
0

Of the offending tasks most are labeled "cam."

Figure 8.18 The result of breaking down lead time by tags

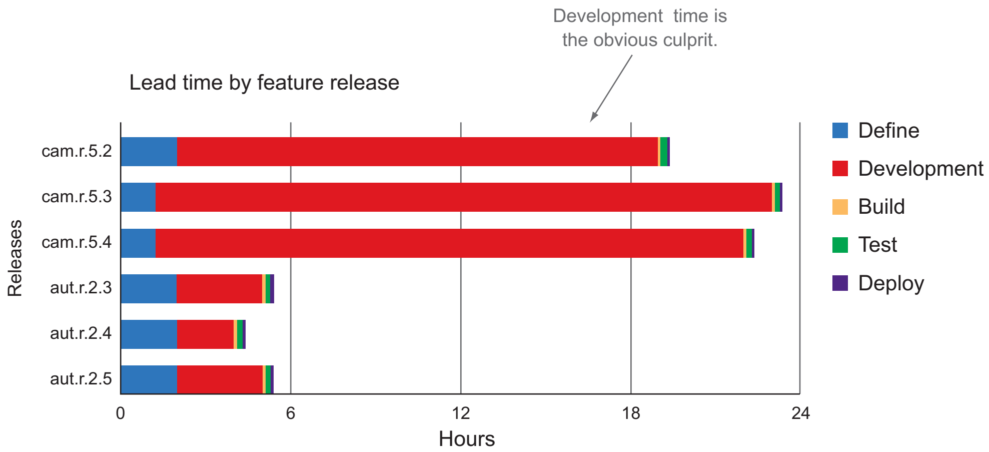
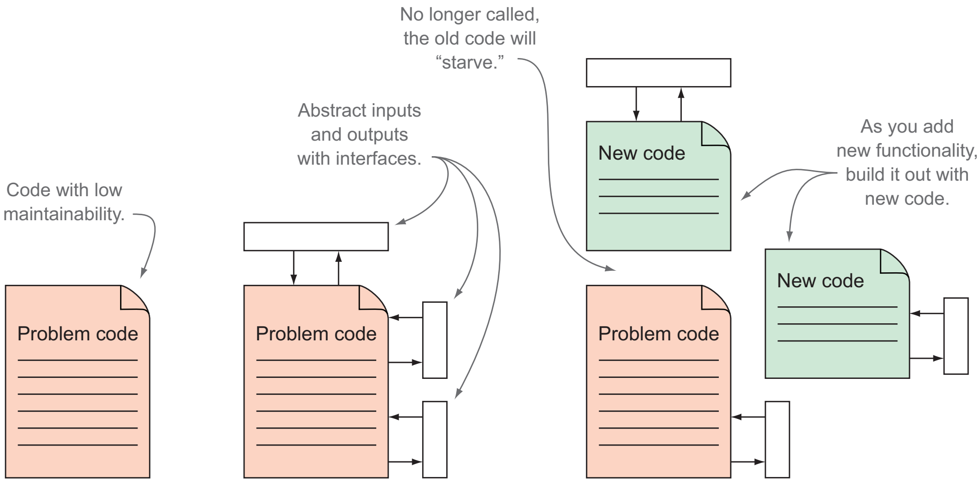

Development time is the obvious culprit.

Figure 8.19 Breaking down lead time for tagged tasks to find where the time sink is happening


Code with low maintainability.

Abstract inputs and outputs with interfaces.

No longer called, the old code will "starve."

As you add new functionality, build it out with new code.

Figure 8.20 The surround and starve refactoring pattern

They decided to refactor this part of the code base, abstracting the inputs and outputs with interfaces and then writing new code that handled requests, as shown in figure 8.20.

This allowed them to write new code with good test coverage in smaller modules and eventually stop calling the problem code in favor of the new code.

Refactoring patterns

There are a number of patterns you can use to refactor your codebase; using the right one will depend on the context of your changes. I've seen the pattern illustrated in figure 8.20, called "surround and starve," work well for moving from large monolithic codebases to smaller, more modular, and easier-to-maintain codebases, as alluded to in this case study. There are entire books dedicated to refactoring. Two good ones are *Refactoring: Improving the Design of Existing Code* by Martin Fowler (Addison-Wesley Professional, 1999) and *Refactoring to Patterns* by Joshua Kerievsky (Addison-Wesley Professional, 2004).

After breaking down their development plan, the team realized that they could refactor as they went along without adding much additional time to their current updates. By focusing on new code that was decoupled from the problematic code, they could ensure that they had good code coverage and clean code from the start. They created a new project in SonarQube so they could monitor their new projects and ensure they had good code coverage and were following standards.

Every time they got a new task that would be tagged "cam," they started developing with their new approach. At first, development time for new tasks went up by a few days, but after only five releases they started to see the benefits of this approach, as shown in figure 8.21.

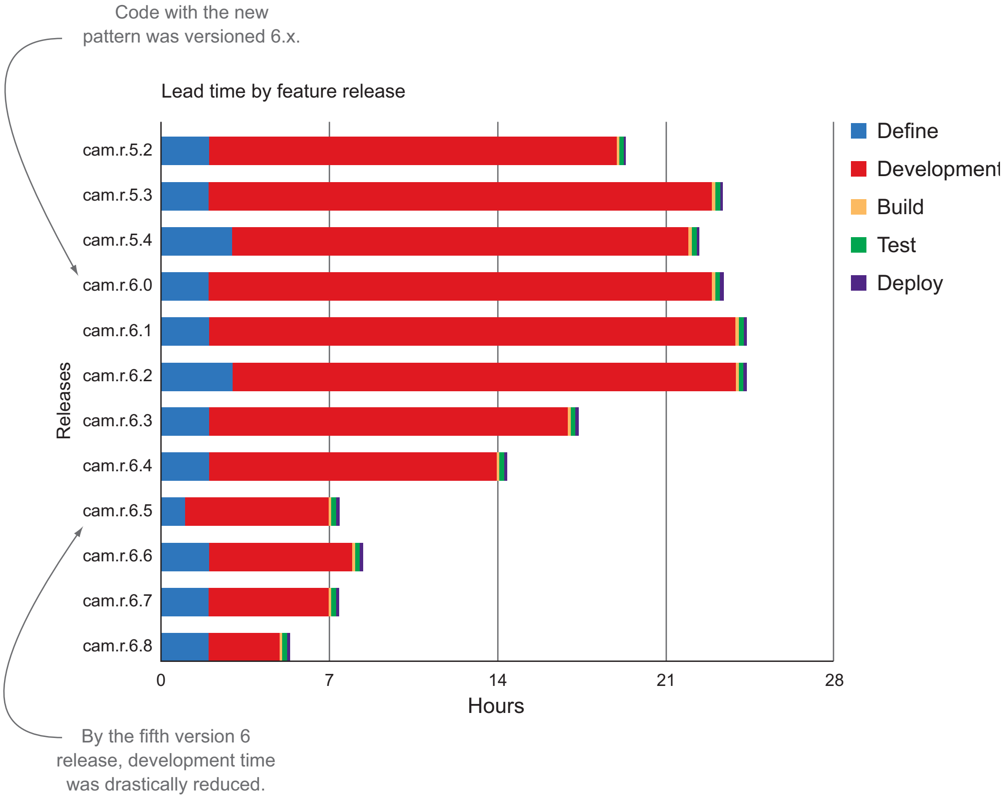


Figure 8.21 After five releases development time was reduced by over 60%.

Without question, this strategy helped this team improve their delivery and their software products by making them more maintainable.

8.6 Summary

Good software is measured across several dimensions. Using the code "ilities" and several freely available tools, you can measure these dimensions to understand the quality of your code. In this chapter you learned the following:

- To measure good software in an agile context you can go back to some agile principles that recommend the following:
  - Continuous and frequent delivery
  - Technical excellence
  - Good architectures
  - Working software
- Use the code "ilities" or non-functional requirements to measure how well your software is built.
- Maintainability and usability are top-level measures.
- Maintainability tells you how fast you can iterate, and it can be measured with these metrics:
  - MTTR tells you how fast you can fix issues for your consumers.
  - Lead time tells you how fast you can get new features in your products.
  - CLOC ensures code can be changed without huge efforts.
  - Code coverage shows that your code is well covered by automated tests.
- Usability tells you how well your application works for your consumers and is measured with these metrics:
  - Availability tells you how often your consumers can use your application.
  - Reliability tells you how consistently your application works for your consumers.
  - Security tells you that your application is safe for your consumers.
- The following tools will help you get better insight into measures of maintainability:
  - Sonar is a great tool to visually analyze code coverage and rules compliance.
  - Other code coverage analysis tools include Cobertura, JaCoCo, Clover, NCover, and Gcov.
  - Pitest is used for mutation testing, which helps test your tests.
- The following tools will help give you better insight into measures of usability:
  - New Relic for availability and reliability
  - Splunk for reliability
  - Coverity, Checkmarx, Fortify, or OWASP Zed Attack Proxy (ZAP) for security scanning
- You can measure how maintainable your releases are by combining key metrics:
  - *MTTR(in minutes) * (Total Fixes / Total Releases)*

9 Publishing metrics

This chapter covers

- How to successfully publish metrics across the organization
- Different methods of publishing metrics and dashboards
- What types of metrics are important to which parts of your organization

As you've learned about the different metrics you can build and use throughout this book, you've been publishing them along the way. If you create a gigantic dashboard with everything we've talked about, you'll end up with lots of pretty charts and graphs that are impossible for anyone except you to understand. Organizing this data for the right audiences is an important way to convey the metrics about your team effectively.

My cardinal rule of designing metrics is to keep them actionable, and the same thing applies to publishing metrics. Publish the data to people who can affect the outcome of your metrics in a positive way. Giving people metrics they can't improve will lead to either unnecessary stress or a disregard of the data you're showing them.

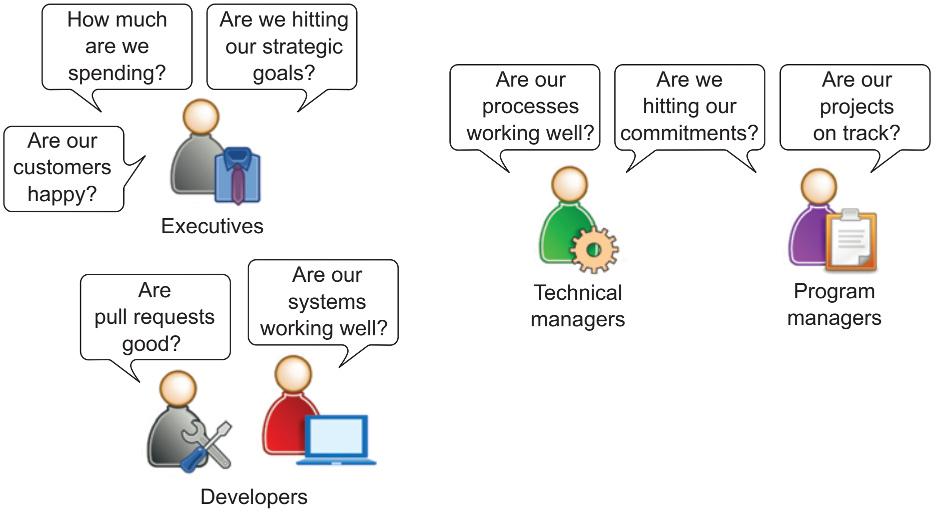

How much are we spending?
Are we hitting our strategic goals?
Are our customers happy?
Executives
Are pull requests good?
Are our systems working well?
Developers
Are our processes working well?
Are we hitting our commitments?
Are our projects on track?
Technical managers
Program managers

Figure 9.1 Different levels/roles in the organization and the questions they ask

9.1 The right data for the right audience

People at different levels of your organizations will need different types of input in order to answer the questions pertaining to their roles. Figure 9.1 shows examples of such questions.

Data should be distributed across the organization in such a way that everyone can get the data they care about at a glance. Here are a few traps that teams fall into when they start publishing metrics:

- *Sending all the data to all stakeholders*—When you're collecting a lot of great metrics, it can be easy to forget that not everyone cares about everything. You may be really excited that your CLOC per hot fix is down to 10, but that may not make sense to your product owners. You don't want to hide information, but you also don't want to overreport data.
- *Emailing reports that don't make sense out of context*—Some people spend their entire day on a single project; others spend a small amount of their time across several projects. If a stakeholder receives an email with information about two days of progress toward goals from your last retrospective, they may not understand what you're talking about or how it affects the bottom line. If you really feel like you need to send certain data out to the entire company, ensure that you tie it back to metrics or information your audience can relate to.
- *Creating web-based dashboards that have too much data for the intended audience*—Along the same lines of sending all the data to every stakeholder, creating a dashboard that doesn't have data a viewer can make sense of and potentially take action on immediately will confuse and frustrate people. We'll talk about designing dashboards at length later in this chapter.

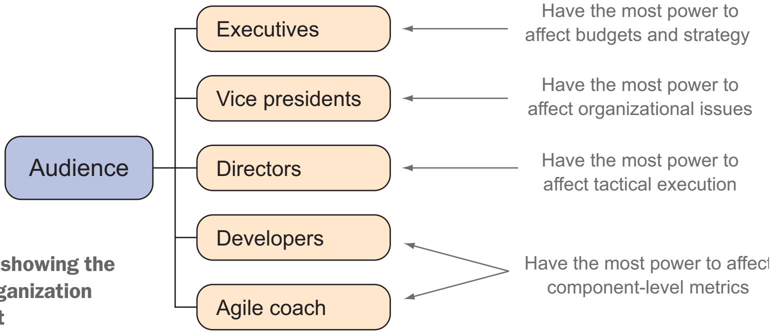

Audience
Executives
Vice presidents
Directors
Developers
Agile coach
Have the most power to affect budgets and strategy
Have the most power to affect organizational issues
Have the most power to affect tactical execution
Have the most power to affect component-level metrics

Figure 9.2 A mind map showing the different levels of the organization and what they can affect

- *Updating published data without letting anyone know*—You may update your data once a sprint, once a day, or once a minute. The interval at which you're publishing data is important for your audience to know and is ideally visualized on your dashboard. If you're publishing data every sprint, you may note the sprint number on your dashboard; if you're publishing data every minute, you may have a running histogram showing the data changing over time.

All of these bullets point to the same problem: giving people the wrong information or too much of it.

To ensure that you provide data to people who can affect the processes reflected by it, start off by looking at what groups or individuals inside your organization can affect directly. An example of this is shown in figure 9.2.

By giving people data they can act on, you're helping keep everyone focused within the boundaries of their responsibilities.

If you take figure 9.2 and align the questions with the data we've been collecting throughout this book, you'll start to see that another way to visualize the right type of data for the right audience would be to put everything on a grid, as shown in figure 9.3.

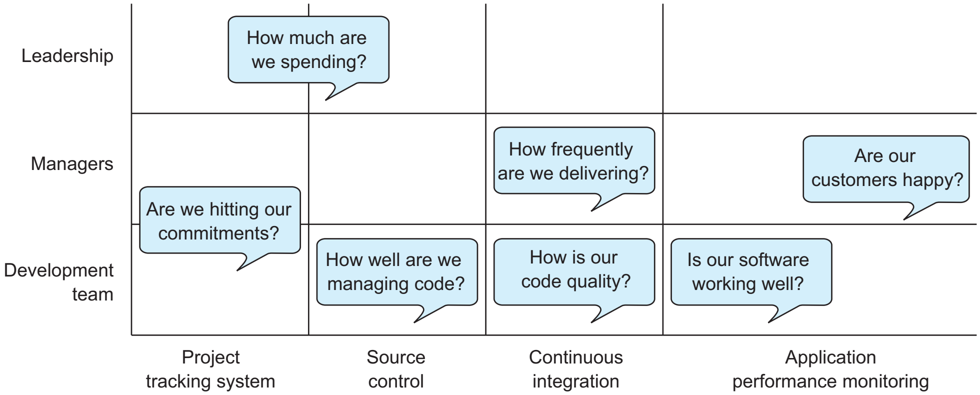

Leadership
Managers
Development team
How much are we spending?
How frequently are we delivering?
Are our customers happy?
Are we hitting our commitments?
How well are we managing code?
How is our code quality?
Is our software working well?
Project tracking system
Source control
Continuous integration
Application performance monitoring

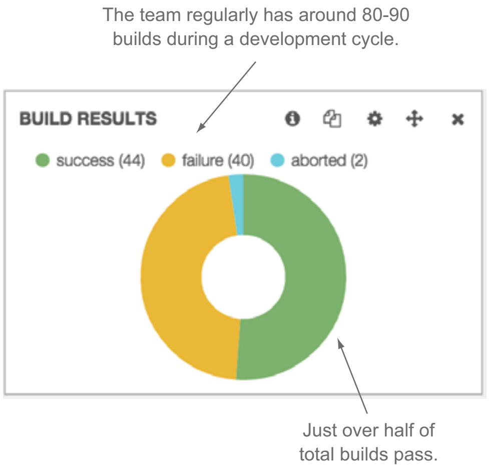
Figure 9.3 The intersection of the levels of a typical organization, the questions they ask, and the data sources where you can find the answers

Even though everyone on the team cares about all of these things, the questions are visually aligned with what each role across the organization can immediately affect.

9.1.1 What to use on your team

The team on the ground delivering changes to your software is generating the building blocks of your metrics and will care about the details because they reflect the day-to-day work. Realistically, it's tough to focus on all the data you're generating all the time. I like to pick key metrics from each system we're pulling data from and publish those as defaults so we're getting a good picture of the entire software lifecycle at a glance. A good strategy for your team is to publish all your data but organize your reporting so your team is paying attention to the metrics you agree on in your planning sessions.

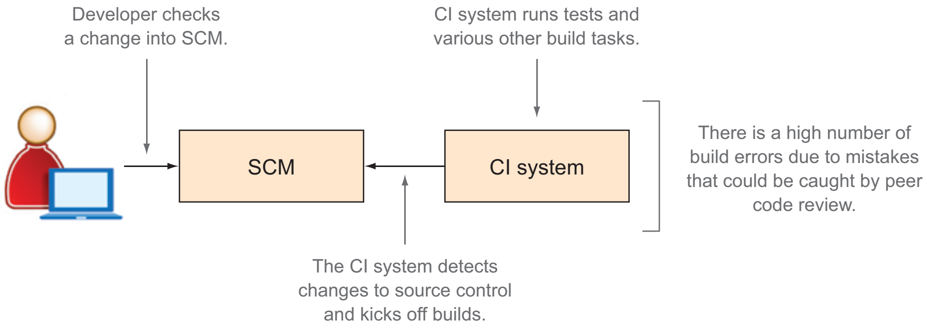
Let's take this scenario as an example: A team found that they had a high percentage of failed builds from their CI system. They were tracking the ratio of good/bad builds, as shown in figure 9.4.


The team regularly has around 80-90 builds during a development cycle.

Just over half of total builds pass.

Figure 9.4 A team's ratio of good and bad builds. They regularly had approximately half of their builds pass.

When they started to investigate the problem, they found that many of the issues were small mistakes that could have been caught if there were another pair of eyes on the code changes. The team had recently moved from using SVN as their SCM system to Git and was simply committing code to the master branch, as shown in figure 9.5.


Developer checks a change into SCM.

CI system runs tests and various other build tasks.

There is a high number of build errors due to mistakes that could be caught by peer code review.

The CI system detects changes to source control and kicks off builds.

Figure 9.5 How the team put code into their SCM.

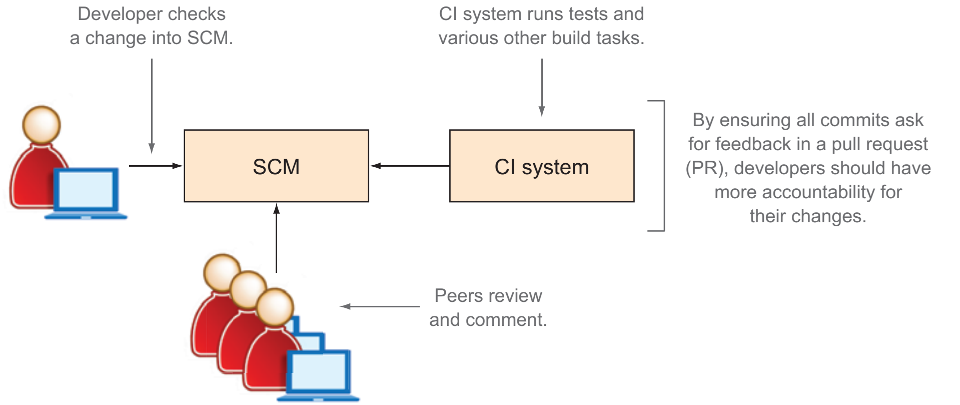

Developer checks a change into SCM.
CI system runs tests and various other build tasks.
SCM
CI system
By ensuring all commits ask for feedback in a pull request (PR), developers should have more accountability for their changes.
Peers review and comment.

Figure 9.6 The proposed updated workflow: moving to pull requests to reduce the number of build failures

They decided to implement the pull request (PR) workflow to add peer code reviews into their development cycle to help improve the quality of their code. When a developer was ready to commit code, instead of checking it directly into the master branch, they would open a pull request and ask their peers for feedback, as shown in figure 9.6.

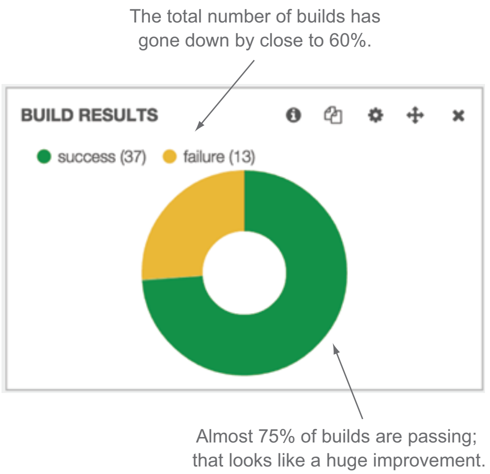


Figure 9.7 Good/bad build ratio had improved significantly, but total number of builds had gone down significantly.

In their next retrospective they noticed that their build ratio had improved significantly, as shown in figure 9.7.

When they looked into their build results they found that they had a lot fewer builds, the success percentage was much higher, but their total number of builds had gone down.

As they discussed the new process in their retrospective, PRs came up as a problem repeatedly. Upon further discussion and analysis, they realized that developers were using a PR as a way to get feedback instead of as a quality check.

A good PR should include code complete with tests and should be ready to ship to consumers. The purpose of the PR would be for peer developers who had deep understanding of the software to ensure that the changes being made were appropriate for the end goal and that they didn't conflict with anything else under change. A good PR would have few or no comments and should get merged fairly quickly.

Instead, developers were checking in half-baked code to start design discussions through a PR. This was causing the team members to get distracted from their own changes and have lengthy discussions in the comments of the PRs. Developers would commit multiple times to the same PR, which would stay open for days while they exchanged comments.

Collaborating through pull requests was apparently ensuring that few small mistakes were being made, but by implementing the PR workflow they ended up getting side effects they didn't expect. If they could go back to the practice of submitting complete code and get the benefit of finding problems before they went into the build pipeline, they might be able to get better quality and much more frequent delivery. They decided to track the metrics that pointed to negative side effects:

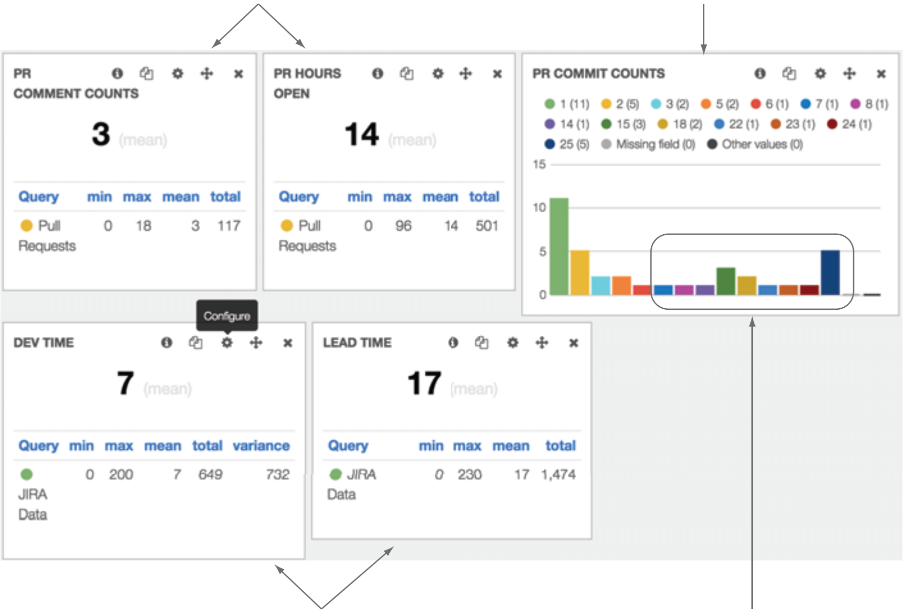
- *High number of comments*—They saw that if a PR had more than six comments it was an indication that it was getting tossed around too much. If they saw comments going up, they should investigate to find out why a solution was so controversial.
- *PR duration*—If a PR was good, it should get merged within the day. If it needed a slight tweak, then maybe it would get fixed and merged the day after it was originally submitted. PRs open for multiple days indicated there was a problem somewhere; either the team didn't agree on the technical implementation and they were spinning on the best solution or for some reason no one was reviewing, approving, or merging the PRs.
- *Number of commits per PR*—A PR starts off as a few commits. If there are comments that need to be addressed or the PR gets rejected, then the original developer will add more code through commits to address the issues. The more commits on a PR, the worse the original PR was, so this number should be low.

High numbers of PR comments and commits along with long PR durations were contributing to longer lead times and longer development times. In their effort to improve they put those three metrics at the top of their operations dashboard, which the team members looked at and reviewed every day. PR duration and number of commits per PR should stay low. The goal was to keep PRs open for less than a day; by tracking that in hours they wanted to make sure it didn't creep higher than 24. Commits per PR should also stay low; the team noticed that by exploring their data when commits for a PR started to go over 6, there was usually a problem.

The dashboard they configured is shown in figure 9.8.

Note this is a focused list of metrics that the team decided to concentrate on. They can still pay attention to the other metrics on their dashboard such as lead time and development time, but this targeted list of metrics will help them improve their process where they've already identified a problem.

Just as in the previous example, during your team's retrospective you should identify things that could be done better and metrics that measure those items. Update your dashboard each development period to reflect what the team agreed to be



The top row of their dashboard had the key stats for monitoring pull requests.

By showing the distribution of PR commit counts they could find anomalies quickly and investigate.

PR COMMENT COUNTS

3 (mean)

Query	min	max	mean	total
● Pull Requests	0	18	3	117

PR HOURS OPEN

14 (mean)

Query	min	max	mean	total
● Pull Requests	0	96	14	501

PR COMMIT COUNTS

DEV TIME

7 (mean)

Query	min	max	mean	total	variance
● JIRA Data	0	200	7	649	732

LEAD TIME

17 (mean)

Query	min	max	mean	total
● JIRA Data	0	230	17	1,474

The main metrics they wanted to affect were lead time and development (dev) time.

A PR with a commit count higher than 6 usually indicated a problem.

Figure 9.8 An example operational dashboard focusing on metrics the team identified as measures of their process that need work: (clockwise from top left) comment count per PR, PR hours open, PR commits, lead time, and development time. Based on the team's history, low PR comments, commits, and hours open also led to faster lead and development times.

immediately aware of, but keep the old metrics on it to get a truly holistic picture of how things are working.

If you're just getting started and aren't sure what to focus on, the following metrics are a good starting point for your team. They will inevitably raise questions that cause you to drill down farther and create and pay attention to a set of measurements you use every day.

- *Tags and labels*—These will show you what's trending day to day. The trends you see should reflect the goals and the immediate tasks the team is working on. If you start to see something that doesn't feel right, you can address it and react quickly.
- *Pull requests and commits*—These show that code is changing and the team is working together. I like to set goals for the team to have at least one pull request submitted and one pull request merged per developer per day. If your team can manage that goal, it usually means that tasks are small enough to make steady

progress and your team is pushing changes through the pipeline at a fairly consistent rate.

- *Task flow and recidivism*—As with pull requests and commits, I like to see cards consistently moving, preferably forward. This also shows that your team has a clear understanding of their tasks and is moving them through the process. Because recidivism points to churn in your workflow, it's a good metric to look at every day. If you start to see that number increase, it's best to dig into the problematic tasks and get to the bottom of the issue.
- *Good/bad build ratio*—Knowing that your team is moving through the workflow is important, but you want to make sure they're moving good code through the workflow. Looking at the good/bad build ratio is a good indicator that your team is publishing changes within the boundaries of your automated test criteria and your code quality rule sets. I firmly believe that when a build breaks, everyone should stop what they're doing and figure out how to fix it. The build rules are set up to make sure everything you've determined necessary is working. If for some reason it's not working, it should be the team's priority to figure out why and fix the problem. If you're following this methodology, then this number becomes a very important indicator of your team's ability to develop within the rules you've defined.
- *Consumer-facing quality rating*—I've seen many teams that separate their production support from their development teams. When this happens, oftentimes developers don't really know how their work is impacting the consumer. That's bad, especially if your team is trying to achieve a continuous delivery workflow. In this context it's great to have an aggregated metric of quality that combines your most important statistics from your consumer-facing system to show the team how well things are working.

An example dashboard with all of these metrics is shown in figure 9.9.

This base set of metrics is a great way to keep a daily pulse on the consistent performance of your team. By combining this base data with specific metrics that your team has committed to improve during retrospectives, you have the basics along with specific hot metrics that you need to focus on to bring about positive change.

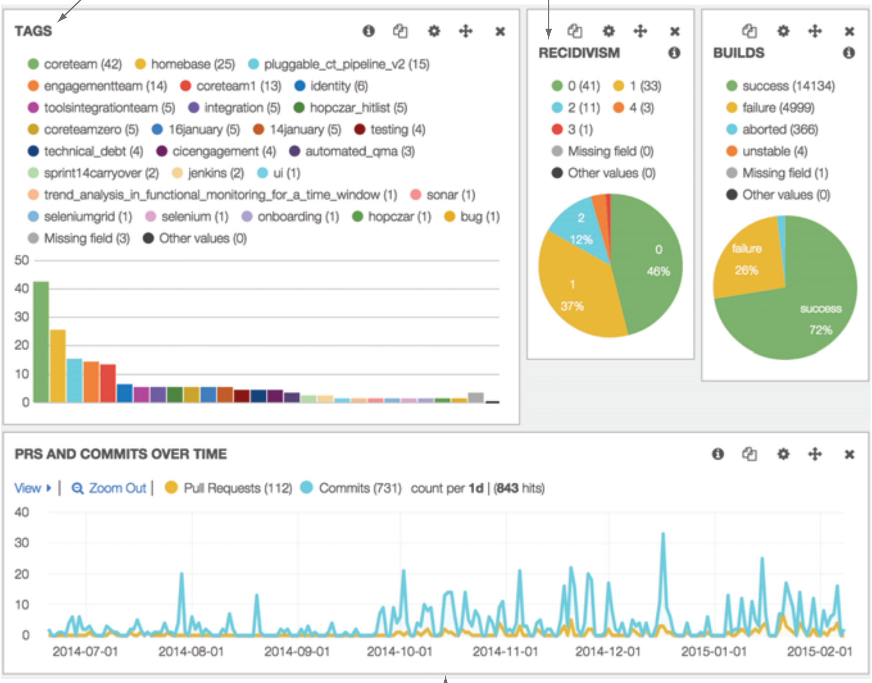
9.1.2 What managers want to see

Managers can affect how a team interfaces with the rest of the organization, how it operates, and who plays what role on the team. Managers are usually also the conduit between the development team and the senior leadership team, so they not only need to understand the details of how the team is performing, but they also need to know how to roll it up in a way that they can show off to the next level of the organizational chart.

Managers should care about the team's daily breakdown of metrics. If you have a strong team that has true ownership over their products, then managers should be able to let the team handle their day-to-day metrics on their own. Instead of micromanaging the details, managers should look at the data over time to see how the team

Tags show you what is currently trending in the dev cycle.

Recidivism shows you how your workflow is going.


TAGS
● coreteam (42) ● homebase (25) ● pluggable_ct_pipeline_v2 (15)
● engagementteam (14) ● coreteam1 (13) ● identity (6)
● toolsintegrationteam (5) ● integration (5) ● hopczar_hitlist (5)
● coreteamzero (5) ● 16january (5) ● 14january (5) ● testing (4)
● technical_debt (4) ● cicengagement (4) ● automated_qma (3)
● sprint14carryover (2) ● jenkins (2) ● ui (1)
● trend_analysis_in_functional_monitoring_for_a_time_window (1) ● sonar (1)
● seleniumgrid (1) ● selenium (1) ● onboarding (1) ● hopczar (1) ● bug (1)
● Missing field (3) ● Other values (0)
50
40
30
20
10
0
RECIDIVISM
● 0 (41) ● 1 (33)
● 2 (11) ● 4 (3)
● 3 (1)
● Missing field (0)
● Other values (0)
2
12%
0
46%
1
37%
BUILDS
● success (14134)
● failure (4999)
● aborted (366)
● unstable (4)
● Missing field (1)
● Other values (0)
failure
26%
success
72%
PRS AND COMMITS OVER TIME
View ▸ | Zoom Out | ● Pull Requests (112) ● Commits (731) count per 1d | (843 hits)
40
30
20
10
0
2014-07-01
2014-08-01
2014-09-01
2014-10-01
2014-11-01
2014-12-01
2015-01-01
2015-02-01

You can see the volume of work by showing PR and commit data over time.

Figure 9.9 A good example default dashboard to get you started with using metrics in your development cycle

is improving. Managers should also be able to compare data across teams to be able to find similar trends and determine when something is working well for one team and if they should consider trying to implement a similar thing on another.

To be able to accomplish these things managers must have a different view into the data of the organization. Typical elements that I've found work well from a management perspective are:

- *Lead time*—The time between defining and completing a task gives you the high-level health of how well you can get work through your team.
- *Velocity/volume*—Velocity is great to show you how consistently your team is working; volume tells you how many tasks are getting done. Because velocity shows your estimates over time and volume shows your total tasks over time, the two together can give you a good idea of the real amount of work your team gets done.

- *Estimate health*—Your team's ability to estimate accurately tells you how well they understand their work and how predictable they are.
- *Committers and pull requestors*—Managers are usually also concerned about the careers of their team members. They're going to care who the top performers on the team are and who is doing what work. Understanding who is committing code and who is reviewing code can show you trends in contribution across the team that can be used to help coach developers if you see things that shouldn't be happening.
- *Tags and labels*—These help managers slice and dice their reports by what the team is working on and how they identify their work. By giving teams the freedom to label as they see fit, managers can get deep insight into how their team thinks, works, and feels.
- *Pull requests and commits over time*—As with the development dashboard, managers should want to keep a pulse on the productivity of the team.
- *Consumer-facing quality rating over time*—This is also an important statistic for managers to care about over time because the trends they see here indicate the success of the team and in what direction it's trending. If a team's consumer quality is going up over a long period of time, then the manager should dig in to find what's good and how to replicate that success across other teams.

One thing to note is that it's great not only to show stats but also to show terms so you can click into the data that isn't ideal. Figure 9.10 shows the distribution of lead time and estimate health along with the statistical overview of them side by side.

In this case an estimate health of 6 means that tasks are taking on average 6 days longer than developers think they will; that's bad. By showing the distribution you can look at the data in more detail to see how big this problem is. In this case the data is skewed by a few tasks that took a lot longer than estimated; specifically there are 4 values of 197, 124, 91, and 57 days that are skewing the mean. For this team let's say that anything in the queue for longer than a month is a task that was deprioritized and forgotten. With that in mind you can remove all tickets that have an estimate health of greater than 30, thus removing the outliers. After this query modification you'll see a much different picture, as shown in figure 9.11.

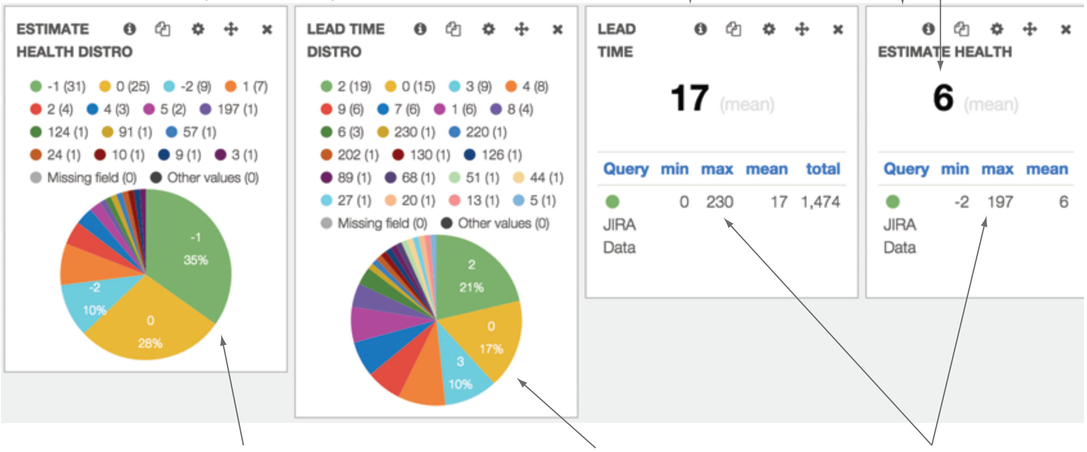
As you can see, those values were skewing lead time and estimate health significantly. By digging into the details and honing their queries to remove anomalies when it makes sense, they could see the true picture and dive into the problem tasks to find out why they sat incomplete for two to four months. The potential next step for this team would be to repeat this exercise for lead time to make sure the statistics represent the true work of the team.

Using this technique with the data shown, managers can get a good idea of the big picture and drill into trends when they become apparent. Managers care about individuals as well as data across teams. Ultimately managers also care about how their team's performance affects strategic goals because that's what they're responsible for showing to the higher levels of leadership in their organization.

By showing the distribution ("distro") as terms, you can click into areas you want to investigate.

Showing the stats for these gives you a high level idea of the health of the statistic.

An estimate health of 6 means the team is taking on average 6 days longer than estimated to finish tasks—that's bad.



Query	min	max	mean	total
● JIRA Data	0	230	17	1,474

Query	min	max	mean
● JIRA Data	-2	197	6

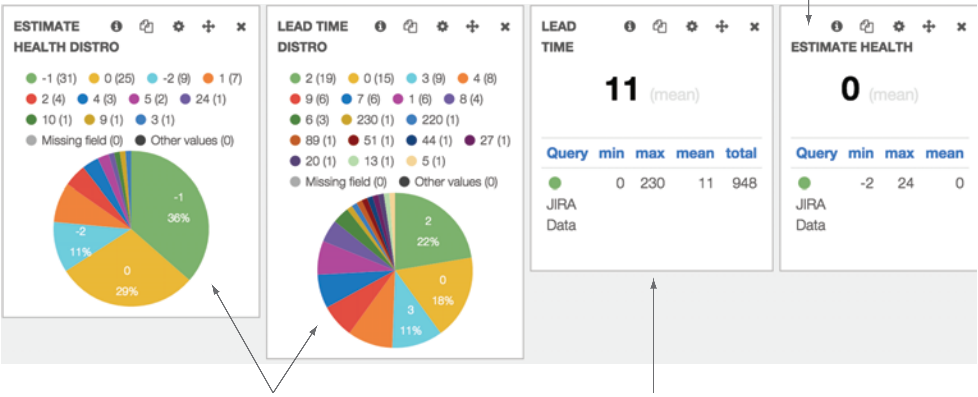
0 means estimates are spot on. Negative numbers indicate overestimating. Positive numbers indicate underestimating. By checking the distribution we can see almost 75% of estimates are correct within a few days despite the mean of 6.

Even though the mean lead time is 17 almost 50% of tasks are done within 3 days. Looking at the distribution shows us the outliers are killing the mean.

The very large max is probably skewing the mean, so it's important to look at the distribution also.

Figure 9.10 Mean lead time and estimate health don't properly represent the team because they're skewed by high maximum values. By breaking out the distribution next to the statistical overviews, you can get a better picture of the outliers and a better representation of the data.

By removing the biggest anomaly mean estimate health is perfect.



Query	min	max	mean	total
● JIRA Data	0	230	11	948

Query	min	max	mean
● JIRA Data	-2	24	0

The distributions are largely unchanged since we only removed a few values from our data source query.

After removing the highest outliers from estimate health, lead time has also improved. This query could still be tweaked to remove the maximum lead time value.

Figure 9.11 After removing the four largest outliers from the estimate health and lead time, the statistics look much closer to the team's targets. The query to get data for lead time could be tweaked even further.

9.1.3 What executives care about

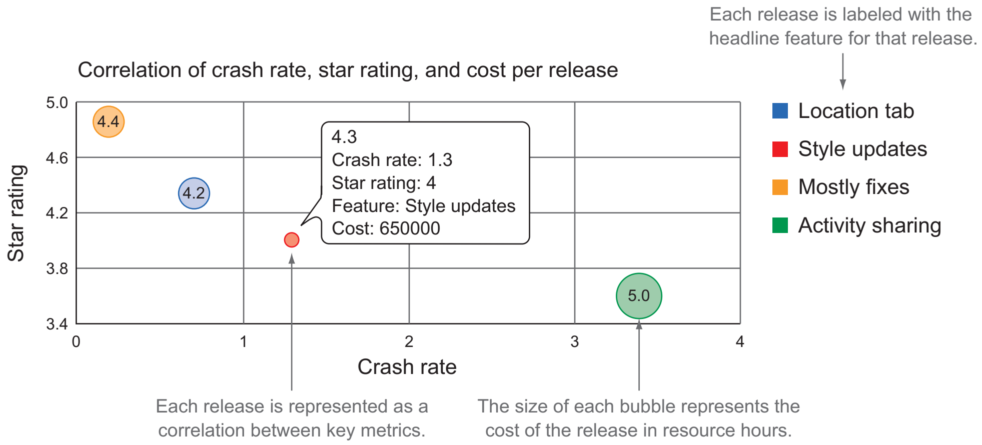
The next few levels up the organization chart can change team composition or organization structure, budgets, and strategies. Leadership teams care how things are working currently, but they're usually more interested in the big picture and how that affects the well-being of the company as a whole. If teams are hitting their goals and the organization is successful, you may not even have to worry about rolling reports up to this level. When I have to address senior leadership, I make sure that I can demonstrate the information they need as quickly and efficiently as possible. Interestingly enough, communication to leadership often happens through presentations instead of through dashboards and ad hoc metrics. The size of your organization and how hands-on your leadership team is will determine the best way to communicate data to them. Normally they'll be most interested in data around strategic objectives. If your leadership team sets a strategic direction to better engage customers by releasing code more frequently through a DevOps model, you should publish the release frequency over time so they can see the progress you're making on the strategy.

The business metrics we talked about in chapter 6 are great candidates for data that executives would care about. If you created a metric that determined business success criteria of a product, then that metric over time is definitely going to be something the people sponsoring your team will care about. If there are several development teams in your organization and each has its own set of business success criteria metrics, the aggregation of all these metrics would produce a great example of a dashboard or report that leadership would care about.

If you're not sure where to start but want to put an executive dashboard together, here are some metrics that I've found leadership teams typically care about regardless of strategy:

- *Number of releases/features per release*—The frequency and volume with which you get changes out that your consumers respond well to show how engaged you are with them through your software products.
- *Consumer-facing quality rating over time*—This is the one metric that everyone in the organization should care about. From an executive level this tells you if everything is moving in the right direction.
- *Development cost*—Knowing how much you're spending on development is something anyone who controls a budget cares about. When you couple this with consumer engagement and satisfaction, it can help everyone understand the value they're getting from their investment.

Let's take an example of a native mobile app. The consumer-facing quality rating comes from the app store—iTunes for iOS apps and Google Play for Android apps. The rating is pretty simple; consumers give the app zero to five stars based on how much they like it, and the rating is published through the store. A quality check on this number could be the crash percentage for the version of the app released.



Each release is labeled with the headline feature for that release.

Each release is represented as a correlation between key metrics.

The size of each bubble represents the cost of the release in resource hours.

Figure 9.12 An example executive dashboard showing releases sized by cost and measured against crash rate and star rating. Minor releases are much less expensive and tend to be of higher quality.

The development cost is most easily calculated by adding up the hours of the team that worked on the project. Devices, equipment, and support are usually included in overhead costs.

For mobile apps it's usually easiest to focus on a single feature at a time. In our example each release is tied to a single headline feature that's published with the metrics.

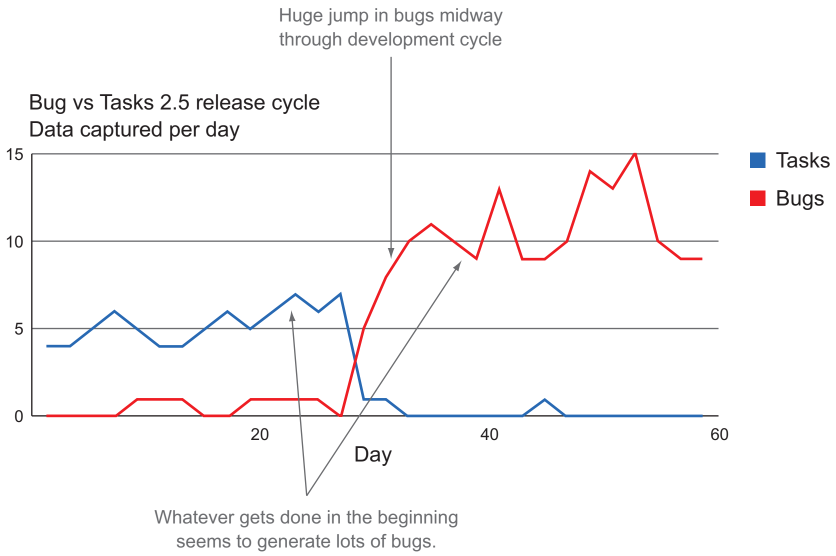
For the executive team it's best to get all the data in a single place so they can see the whole picture at a glance. In the example graph shown in figure 9.12 we use a bubble series chart to show all of these metrics at once. There's a bubble for each release on an axis, sized by the cost in total hours for the release. The x-axis represents crash rate and the y-axis represents the star rating.

As you can see in figure 9.12, there's a clear correlation between crash rate and star rating. When the crash rate is low, releases tend to get better ratings. You can also see that the minor releases (4.2, 4.3, 4.4) cost a lot less than the major release (5.0). From an executive point of view it looks like the team is more efficient releasing minor pieces to the app than full-fledged features.

If your team isn't doing so well, then more frequent reports to leadership are usually in order. In those updates you should use the key data points you're tracking to get your team back on track to show that you know the problem, know how to fix it, and are making progress. Demonstrating how changes you're making affect your metrics and that you show progress as they're improving are exactly what leadership wants to see; if there's a problem, at least you understand it, are correcting it, and are tracking it effectively.

9.1.4 Using metrics to prove a point or effect change

Because you've made it this far through this book, you know that metrics are awesome, but you can't get a clear picture without several of them that show different facets of



Huge jump in bugs midway through development cycle

Whatever gets done in the beginning seems to generate lots of bugs.

Figure 9.13 The trends for a team that spent one month on development and one month on hardening

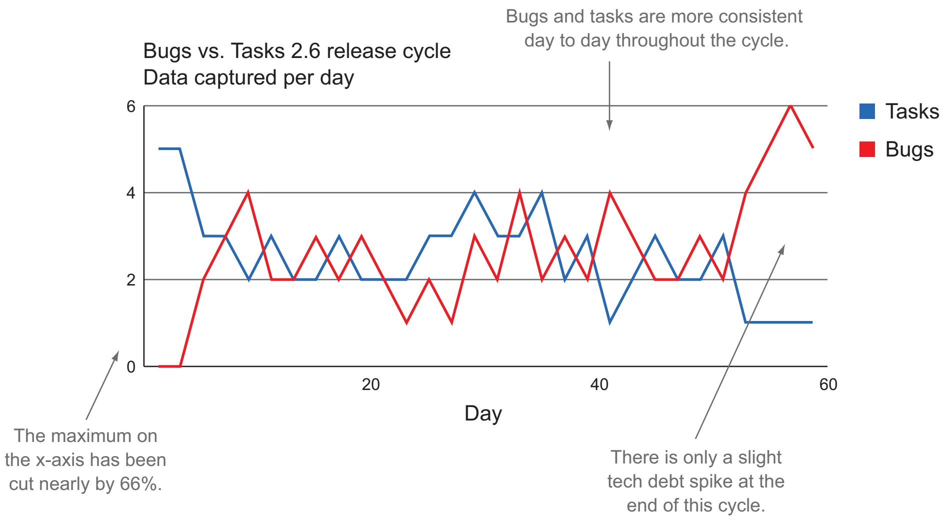
what you're trying to measure. Sometimes someone will want to make a point and will pull a specific combination of data to prove their perspective. The problem that can stem from pulling only part of the whole data set is that they'll pull only the data they need to prove a point, which can lead to uninformed and usually poor decisions.

If you want to use data to effect change, you should find a metric that matters to the people you're trying to convince, but always keep an open mind; your assumptions may be incorrect. Data can be misleading when used in a superficial fashion; that is, a single metric doesn't tell the whole story. When you're working to prove or disprove a hypothesis, be sure to have different data points that check each other.

Take, for example, a team that's developing features for a mobile app that has a two-month development cycle for each feature, the second half of which is purely dedicated to hardening the codebase and cleaning up tech debt. If you look at some data for this team, you could see trends like those shown in figure 9.13.

At a glance it seemed horrible that there was a one-month hardening period. Some of the developers on the project wanted to move to a more continuous delivery type model where they didn't build up any tech debt as they went along to potentially shave time off the release cycle.

The team was able to use the data from their release cycle to convince the stakeholders to change how the team was delivering code with the promise of more frequent releases. After a release they started seeing the benefits. They finished tasks at a slower pace, but they were able to work out the kinks and get the release out the door on time. The updated release cycle is shown in figure 9.14.



Bugs and tasks are more consistent day to day throughout the cycle.

The maximum on the x-axis has been cut nearly by 66%.

There is only a slight tech debt spike at the end of this cycle.

Figure 9.14 The updated velocity and volume for the team after they changed their development practices

At first everyone was really excited that they were able to deliver faster. Counterintuitively, when the release went out they noticed their star ratings on the app store started to decline. They dug into the results and noticed consumers nitpicking the new features they had released. After meeting and discussing how this could have happened, they pointed back to the shortened development cycle, which included a much-abridged beta. It turned out that in their previous way of developing software they were able to get a buggy version into their beta users' hands very quickly. This led to an extended hardening period not only for code but also for the features themselves. By focusing on just the data from their project management system and ignoring the data they were collecting from their beta and how it affected the project, they ended up hurting their consumer when all they wanted was to deliver more efficiently.

In this example the team narrowed their focus to where they thought the problem was instead of looking at the big picture. They ignored the beta feedback as a data point in their lifecycle and looked only at data from their immediate team. To avoid this problem the team could have acknowledged that the beta cycle was a critical part of delivery and figured out how to measure its value before cutting it.

As you dig into the data, make sure you do your analysis with an open mind instead of manipulating the data to fit your hypothesis.

9.2 Different publishing methods

Every organization has different quirks around their methods of communication. Wherever you work there are probably different emails, dashboards, and reports that different people rely on for information. The best way to communicate across an organization is to communicate within the boundaries of that organization. If everyone expects to see a status report every Tuesday, then maybe piggy-backing on that report is a great idea. If everyone sends those emails straight to their deleted folder, then maybe using a dashboard where people can pull data when they need it will work better. Following are some tips on using dashboards, emails, and reports; you know what works best in your organization so you should take what works best and apply it.

9.2.1 Building dashboards

We've been using dashboards throughout this book. Web-based dashboards are a great way to publish results so anyone in your company can check out the data when they want. The following sections provide some dashboarding tips.

DON'T RESTRICT ACCESS INSIDE THE COMPANY, BUT KEEP IT INTERNAL

Sometimes people like to hide data to protect themselves or others. This is usually a bad idea within your own company or across your own team. If data is used in a collaborative and open way across your company, then you should allow anyone to access the data. If you work in an environment where culture encourages the use of data in a negative way (for political gain, for example), then it may be best to just keep the data accessible inside your team to avoid distraction from outside.

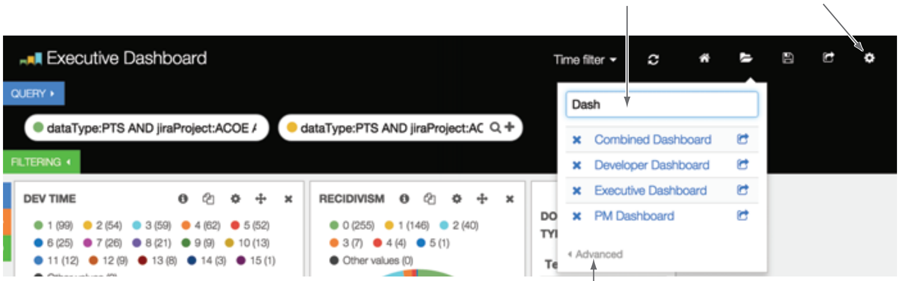
On the flip side, dashboards contain a lot of information about how you work and in some cases what you're working on. It's always best to think about keeping your websites and data as secure as possible at all times. Even though you should keep it open inside the office, you should keep it from prying eyes outside your walls.

MAKE IT CUSTOMIZABLE

Managers, developers, executives, and coaches will all want to see data differently. You should give them the flexibility to see metrics the way they want and when they want. If you're using the tools we've been using throughout this book, you can use Kibana's great UI to create your own dashboard and save it. An example of the elements used to save dashboards and update widgets is shown in figure 9.15.

Name the dashboard whatever you want.

Each component has customizable settings.



You can save different dashboard configurations in Elasticsearch.

Figure 9.15 The Kibana header and some of its customizable elements

One of the worst things you can do is give someone a dashboard where the data they need isn't right where they expect it to be. When this happens, people will be less likely to look at it.

ENSURE PEOPLE KNOW THAT DATA IS THERE AS A TOOL, NOT A WEAPON

The biggest fear people have around metrics is that someone will use them to prove they're not good or not good enough. Never use metrics as a weapon. Always communicate that the data is there to help everyone understand how the team is working and to help track improvements to their process.

Look for opportunities inside your environment to evangelize these techniques in a positive way. I've found that starting off retrospectives with data ends up leading to honest and open conversations that help everyone understand how their work affects metrics and why they're important to track.

Ensure that the people generating the data have a say in which metrics they think are most important. For example, if you think you should be measuring CLOC, ask your development team what they think of the idea. If they don't like that metric, ask why; this will usually lead to them helping you determine which metrics they actually care about. Inclusion in the conversation and collaboration from the team will lead to better success and broader buy-in.

USE PAGE TRACKING TO UNDERSTAND HOW YOUR DASHBOARDS ARE USED

Your dashboard is a product for a consumer, so treat it that way. I've found that often internal tools aren't tracked the same way consumer-facing tools are, which ends up leading to things people don't use or aren't happy with. By using page tracking on your dashboards, you'll see what people click on the most, what they're drilling in on, and how frequently they use them. You can then use this information to continue to hone which metrics mean the most to your team and your company. Go back to chapter 6 and apply the same techniques you would to track success on your metrics dashboards. Some metrics that I've found useful in this context are page hits and metric clicks:

- *Page hits*—How many people are using your dashboard, which views are they frequenting, and how long are they spending on those pages?
- *Metric clicks*—What metrics are people clicking on? This shows you what everyone cares about the most and will help you optimize default dashboards.

9.2.2 Using email

Email can be a great tool but it can also be a curse. When this book was written, Kleiner Perkins Caufield & Byers (KPBC)[1] Internet Trends,[2] a standard in reporting consumer technology trends, estimated that Americans spend nearly an hour a day checking email or messaging on their phones or tablets, and of total email traffic more than 70% of it is spam. If you combine these two facts you'll see proof of something you probably already know; people spend a lot of time deleting emails. If you spend that much time

[1] Kleiner Perkins Caufield & Byers is a venture capital firm in Menlo Park in Silicon Valley, en.wikipedia.org/wiki/Kleiner_Perkins_Caufield_%26_Byers.

[2] A briefing on the Internet Trends 2014—Code Conference, www.kpcb.com/internet-trends.

filtering junk you don't care about, you should definitely make sure that you get information out that your audience cares about when they're ready for it.

Following are a few email tips.

ALLOW PEOPLE TO OPT IN TO YOUR EMAIL REPORTS

Most people are pretty email savvy, and when they start getting something they don't care about, they'll send it to their spam folder. Even worse, if they're using a spam filter, your nice reports could get caught up in it. If you can give your audience the ability to opt in to an email from your dashboard, then they've shown they care enough about the data that they want to have updates periodically.

On the flip side, you should allow people to opt out as well. Make sure to pay attention to how many people are receiving your report as a success metric of whether the email communicates what people need.

GIVE THE LEAST AMOUNT OF DATA NEEDED AND REFERENCE A DYNAMIC DASHBOARD

When you send someone an email, it should contain just the information they need now. Going back to the principle of showing people only what they can affect, keep your emails to a few key metrics that recipients can take direct action on if they choose to. At the same time, perhaps they'd like to dig deeper or see more data. In this case, instead of continually adding more data to email reports, add it to the dashboard and reference the dashboard in your email.

ESTABLISH THE RIGHT CADENCE

Don't send emails so frequently that your coworkers get annoyed or, even worse, create an email rule that sends your reports into an abyss. Developers and Scrum masters might want daily reports in their inbox, but higher-level managers may need them only weekly or biweekly. If you have good page-tracking setup in your dashboard, then you should be able to correlate the user activity with your email schedules to see how effective your emails are at driving people to the data you're publishing.

9.3 Case study: driving visibility toward a strategic goal

In this case study we'll look at a company that has brick-and-mortar retail and has recently established an e-commerce site to increase sales. Once the site was up and running, they started looking at differences in buying trends between their e-commerce site and their physical stores. One significant trend they noticed was that when consumers purchased items off their site they typically bought one item. Customers in their retail locations would typically buy three to four items at a time. Their retail strategy had been to group related items together so as consumers were shopping for what they came in for, they also found related items.

As the leadership team discussed this trend, they decided they wanted to sponsor a development initiative to increase sales of related items on their e-commerce site. According to their calculations, if they could achieve a similar trend on their e-commerce site as they were seeing in their retail locations, e-commerce revenue would increase by over 70%. The goal they set for their development team was to increase related product sales by 100% by the end of their fiscal year.

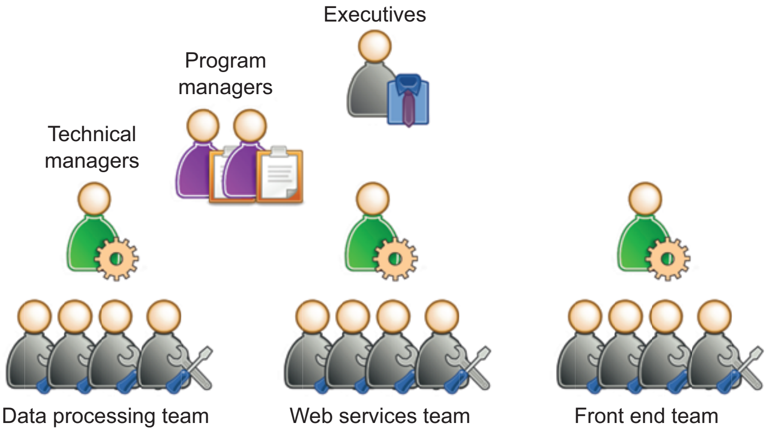


Figure 9.16 An overview of the organization and responsibilities in delivering the feature

The program managers broke this goal down into why consumers weren't buying related products. Following the same model as the retail locations, they decided that they needed some way to show related items to consumers as they were shopping. The feature they decided on was a product recommendation system that would tell a consumer what would complement the products the consumer was looking at. They also realized that in an e-commerce site they could continually improve their site recommendations as they analyzed the patterns of what people bought together.

The delivery team came up with the technical designs and worked with program management to flesh out the delivery plan. From the start they decided to create dashboards for everyone involved in the project so all levels of the organization could see the progress as they moved forward. There were a few different delivery teams that had to coordinate efforts to move this feature forward; they're shown in figure 9.16.

They started with the business success metrics they wanted to get out of the feature they were delivering. In this case the business success metrics they came up with were as follows:

- *Items sold*—They were already tracking this as a success metric of the site in general.
- *Number of items per order*—If this feature was successful, they would see this number going up. This was the key metric that related to the strategic goal laid out by the leadership team.
- *Recommended items per order*—This was the key metric that showed the development team how effective their feature was. If this number was going up, then the product recommender was a success. If they noticed this number increasing at the same rate as the number of items per order, then they would know that this was the driving feature for the executive goal.

Like the team in chapter 6, they used StatsD to instrument their code and send these metrics back to their monitoring system, as shown in the following listing.

Listing 9.1 Using StatsD to add business-specific metrics to the code

```
private static final StatsDClient statsd = new
     NonBlockingStatsDClient("the.prefix", "statsd-host", 8125);

...
    statsd.incrementCounter("orders");
    statsd.recordGaugeValue("itemsPerOrder", x);
    statsd.recordGaugeValue("recommendedItems", y);
```

Setting up the StatsD client

Incrementing the counter for total orders

Setting item count per order for this order

How many items in this order were from recommendations

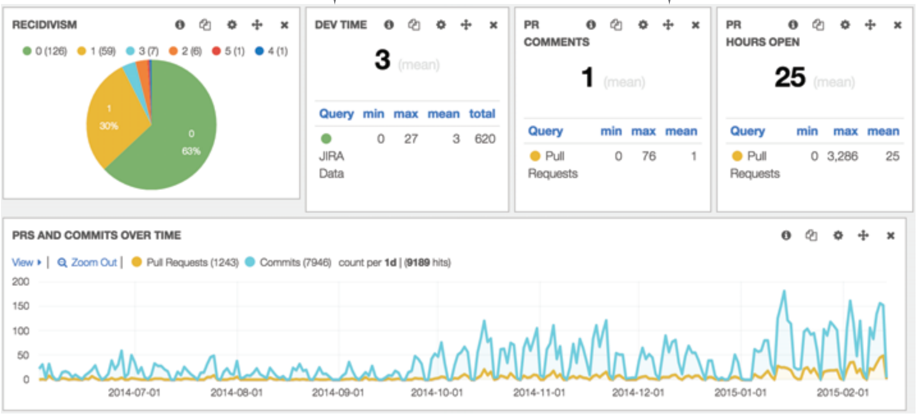
Different teams had different release metrics. The web services team was working on improving their pull request workflow, so they focused on PR comments and pull requests/commits to make sure that was healthy. This is shown in figure 9.17.

The team building a data processor for recommendations knew they had to iterate quickly, so they focused on estimate health, lead time, and deploy frequency to ensure they could react to changes predictably. That is shown in figure 9.18.

The program management across the entire project wanted to keep tabs on all of the teams. They wanted to ensure that the workflow was predictable and consistent, so they cared most about velocity, recidivism (how tasks moved through the workflow), lead time, and estimate health. They wanted to see stats across all projects but also to be able to drill into individual projects. Their dashboard is shown in figure 9.19.

They used development time and recidivism as primary metrics, so they kept these up top along with their focus metrics.

PR-specific metrics helped the team keep tabs on their immediate concerns.



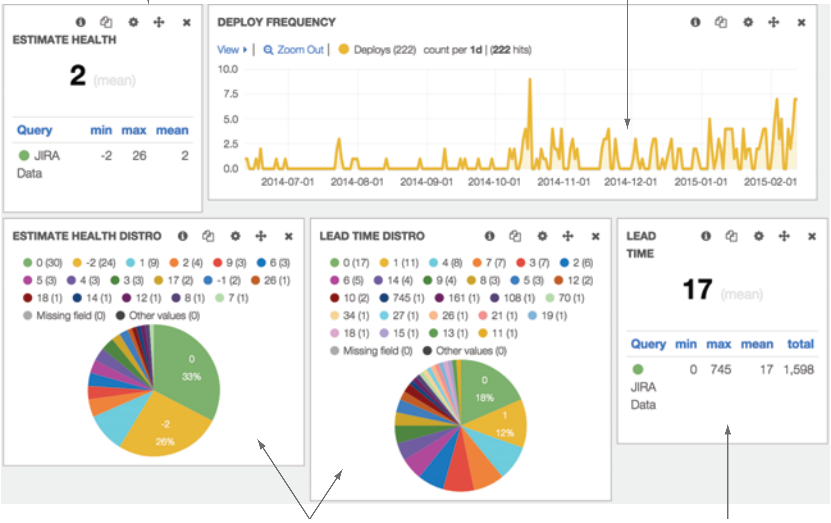
Query	min	max	mean	total
● JIRA Data	0	27	3	620

Query	min	max	mean
● Pull Requests	0	76	1

Query	min	max	mean
● Pull Requests	0	3,286	25

They watched PRs and commits over time to ensure code was being updated consistently.

Figure 9.17 The web services team's dashboard. Development time, PR comments, and PR hours open are right on target. The pulse at the bottom shows that SCM activity over time is healthy.



They want to ensure they understand the work, so they track that by checking their estimate health.

Because this team is practicing CD they want to deploy multiple times a day.

ESTIMATE HEALTH

2 (mean)

Query	min	max	mean
● JIRA Data	-2	26	2

LEAD TIME

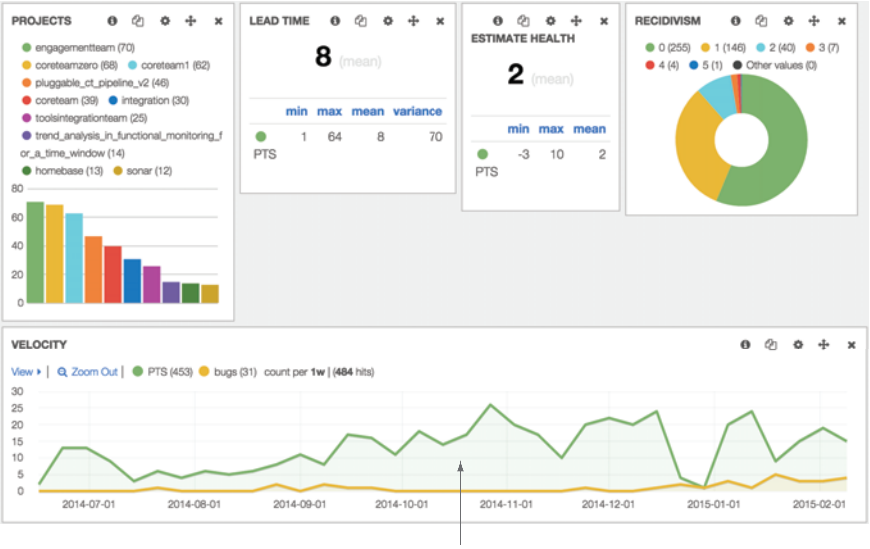
17 (mean)

Query	min	max	mean	total
● JIRA Data	0	745	17	1,598

To ensure the mean doesn't give them a false sense of security they break out the distribution of lead time and estimate health.

This team strives for a short lead time to check the relationship between requirements and task completion.

Figure 9.18 The dashboard for the development team working on the data processing



The clickable list of projects allows the viewer to sort this data by project.

Lead time and estimate health tell the viewer how predictable these teams can be.

Recidivism shows how healthy the workflows are across teams.

LEAD TIME

8 (mean)

	min	max	mean	variance
● PTS	1	64	8	70

ESTIMATE HEALTH

2 (mean)

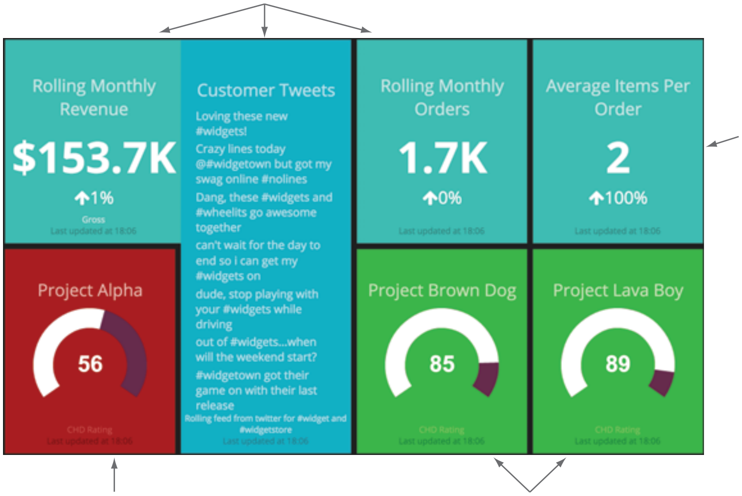
	min	max	mean
● PTS	-3	10	2

PROJECTS
● engagementteam (70)
● coreteamzero (68) ● coreteam1 (62)
● pluggable_ct_pipeline_v2 (46)
● coreteam (39) ● integration (30)
● toolsintegrationteam (25)
● trend_analysis_in_functional_monitoring_for_a_time_window (14)
● homebase (13) ● sonar (12)
80
60
40
20
0
RECIDIVISM
● 0 (255) ● 1 (146) ● 2 (40) ● 3 (7)
● 4 (4) ● 5 (1) ● Other values (0)
VELOCITY
View ▸ | Zoom Out | ● PTS (453) ● bugs (31) count per 1w | (484 hits)
30
25
20
15
10
5
0
2014-07-01
2014-08-01
2014-09-01
2014-10-01
2014-11-01
2014-12-01
2015-01-01
2015-02-01

Velocity over bugs shows how much the team is getting done along with how many issues they're creating.

Figure 9.19 The dashboard used by the program and project management team

Top line info for the leadership team included total monthly rolling revenue and orders and the latest customer tweets.



Loving these new #widgets!

Crazy lines today @#widgetown but got my swag online #nolines

Dang, these #widgets and #wheelits go awesome together

can't wait for the day to end so i can get my #widgets on

dude, stop playing with your #widgets while driving

out of #widgets...when will the weekend start?

#widgetown got their game on with their last release

Rolling feed from twitter for #widget and #widgetstore

Last updated at 18:06

The current initiative was supposed to increase items per order, so that got top billing on the dashboard.

This team is using the CHD rating introduced in chapter 7. When it dips below 80 the team health shows red.

These projects (code named by the teams themselves) are on track based on the CHD rating.

Figure 9.20 The dashboard used by the leadership team to show the success of the initiative and the overall health of the contributing projects. Leadership is always paying attention to the consumer voice through their Twitter hashtags and the total revenue. They've added the average items per order to track the success of the latest development initiative and show overall ratings for each development team on the project.

The leadership team trusted their managers to run the factory and wanted data that showed them the bottom line. They wanted to know how sales were going, how projects were progressing, and if their strategies were improving their online business. Their standard metrics were the monthly rolling revenue, customer feedback, and rolling monthly orders. For this specific project they added a widget to their dashboard to track average items per order. In addition, they added one widget per project to represent the health of the project; if a project was off track it would change to a shade of red, and if it was on track it would be green. Their dashboard is shown in figure 9.20.

By showing the key measurement of the current initiative, the leadership team could track the success to potentially check and adjust the approach of the teams if necessary. They didn't care to get involved in the daily activity of the teams, so they didn't need the detail on pull requests, estimates, and completed tasks. They did care if a project was not healthy, and so to track that they used the code health determination (CHD) rating introduced in chapter 7, which is a combination of workflow, code quality, and continuous delivery release efficiency.

As consumers started using the feature, the team watched the business metrics start to be affected as well. Now they had all the data in place that they needed; the

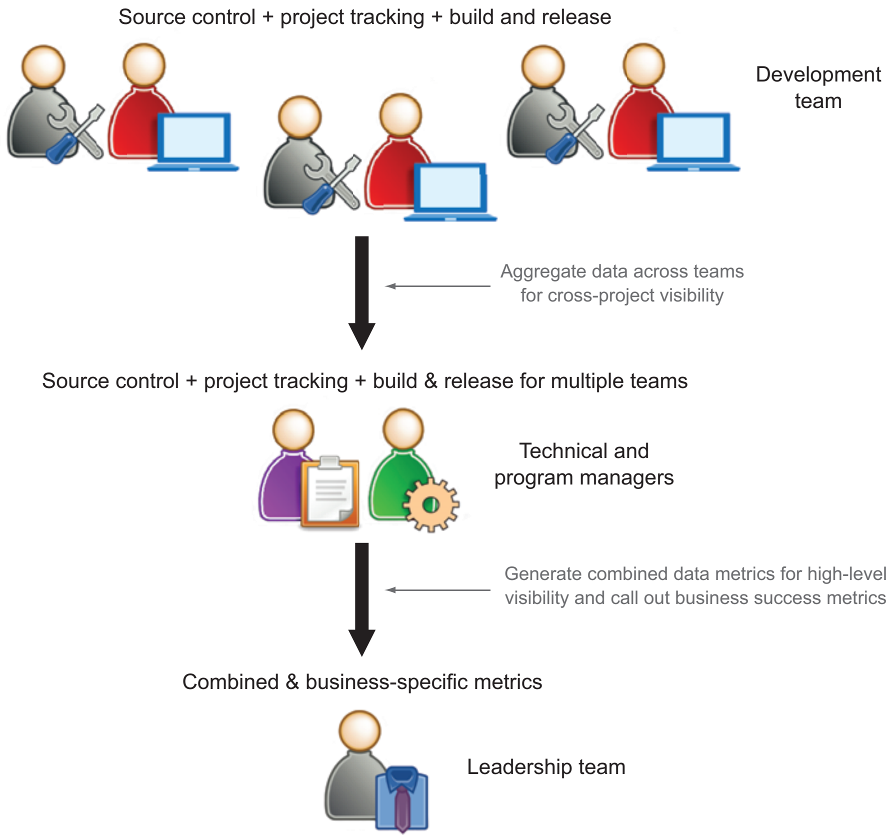

Source control + project tracking + build and release
Development team
Aggregate data across teams for cross-project visibility
Source control + project tracking + build & release for multiple teams
Technical and program managers
Generate combined data metrics for high-level visibility and call out business success metrics
Combined & business-specific metrics
Leadership team

Figure 9.21 A review of the organization and the data being used for each team

development teams were tracking the details they cared about, the management layer was tracking cross-team data, and they were able to collect data to show the strategic objectives were being met for the leadership team. Figure 9.21 shows the different members of the teams and the data they were using.

Everyone was happy, Yay!

9.4 Summary

Different levels of the organization need different information from your development teams. Publishing the right information to the relevant audience is a key factor in implementing successful metrics reporting across the organization. In this chapter you learned the following:

- Publish metrics to audiences that can act on them.
- Metrics at the team level are the detailed metrics that show you how consistently your team is working. Key things to track on your team are these:
  - Tags and labels

  - Pull requests and commits
  - Task flow and recidivism
  - Good/bad build ratio
  - Consumer-facing quality rating
- When identifying improvement areas during retrospectives and reflection periods, be sure to also identify corresponding metrics to ensure you're making daily progress.
- Metrics at the manager level should show how teams are doing over time and how individuals are doing on a team. Key managerial metrics to focus on include the following:
  - Lead time
  - Velocity/volume
  - Estimate health
  - Committers and pull requestors
  - Tags and labels
  - Pull requests and commits over time
  - Consumer-facing quality rating over time
- Metrics at the executive level should show how the progress of the team affects strategic goals. A few default metrics executives should see are these:
  - Number of releases/features per release
  - Consumer-facing quality rating over time
  - Development cost
- Dashboards are a great way to communicate metrics across the organization. Effective dashboards have these characteristics:
  - They don't restrict access inside the company but keep it internal.
  - They make it customizable.
  - They ensure people know that data is there as a tool, not a weapon.
  - They use page tracking to show how your dashboards are used.
- Emails are a good communication method under the following conditions:
  - They allow people to opt into your email reports.
  - They give the least amount of data needed and reference a dynamic dashboard.
  - They establish the right cadence.
- Showing distributions as well as stats allows you identify anomalies at a glance and remove them when appropriate.

10

Measuring your team against the agile principles

This chapter covers

- Breaking down the agile principles into measurable pieces
- Applying the techniques in this book to the agile principles
- Associating metrics with the agile principles
- Measuring your team's adherence to the agile principles

If you ask the CIOs of most Fortune 500 companies if their teams are practicing agile development, they'll probably all say yes. If you sit on any of the development teams in those companies, you'll notice that they operate differently to varying degrees. That's okay; in fact, one of the great things about agile development frameworks is that they allow development teams to move quickly in their own context.

Often as teams evolve their agile development processes they'll start to question their agility. One way to combat this is to measure your team against the agile principles themselves.

Because we're talking about continuous improvement in the scope of agile software projects, it seems fitting to complete this book by talking about measuring your project and team against the agile principles. Using the concepts we've talked about in this book, we'll break down the agile principles and show you what to use to measure your team against each one.

10.1 Breaking the agile principles into measurable components

As a reference point for this chapter let's start off by reviewing the agile principles as they're written in the Agile Manifesto.[1] Just in case you don't remember them off the top of your head, here they are in order.

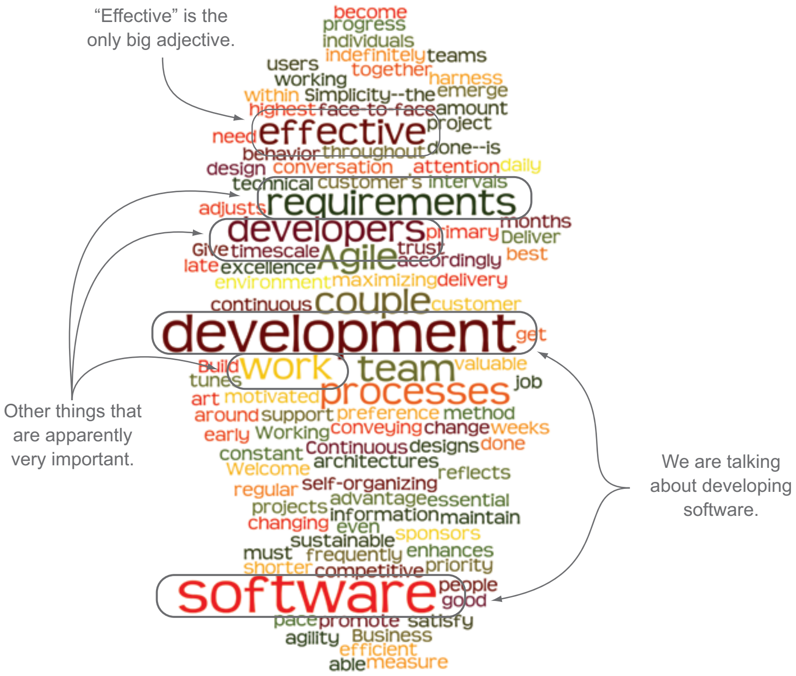
- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

To get started, let's have a bit of fun with the agile principles. If you put them into a word cloud, as shown in figure 10.1, you can get a different perspective on what the agile principles are talking about by seeing which words are used the most.

It's not surprising that the two most frequent words are *development* and *software* because these principles are about developing software. A few other interesting qualities of this viewpoint stand out:

- The only adjective used more than once is *effective*.
- *Requirements*, *developers*, *work*, *team*, and *process* are all focal points.

[1] "Principles behind the Agile Manifesto," agilemanifesto.org/principles.html.



"Effective" is the only big adjective.

Other things that are apparently very important.

We are talking about developing software.

Figure 10.1 The agile principles as a word cloud

If you start to think about how to measure agile teams, you should definitely think about how you're measuring the main focal points. If you start breaking these down into questions you can apply measurements to, you'll get something like the following:

- Are your *development teams* effective?
- Are your *processes* effective?
- Are your *requirements* effective?
- Is your *software* effective?

The end goal is great software. You can put these into a simple equation like the one shown in figure 10.2.

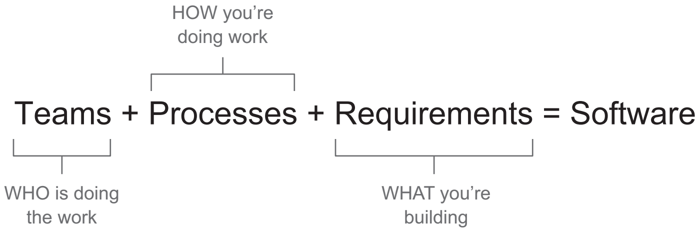


Figure 10.2 The core elements of the agile principles

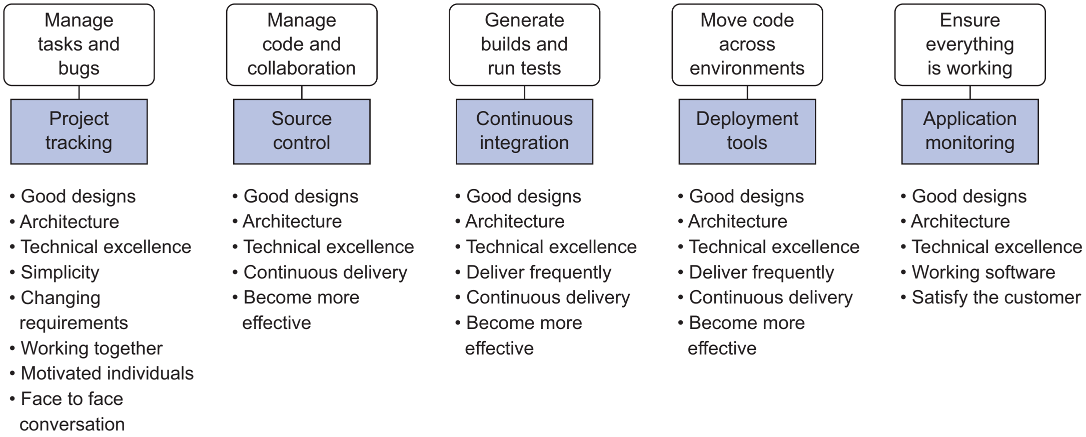


Figure 10.3 More agile tenets on the delivery lifecycle

The next step is to use the data you've been collecting throughout the book to answer your questions. Using the high-level questions fleshed out previously, you can start breaking the principles into categories in order to measure them. If you look at each agile principle in detail, you'll start to see that of the twelve principles three imply measuring software, four implicate teamwork, four represent process, and one references requirements. Let's start off with aligning the principles with the delivery lifecycle.

10.1.1 Aligning the principles with the delivery lifecycle

If you take keywords from each of the principles and transpose them over the delivery lifecycle that you've been looking at in previous chapters, you can see where to get the data you need to measure them, as shown in figure 10.3.

Another way to look at these associations is to put all of the key measurements in a matrix against the systems you can get data from, as shown in table 10.1.

Table 10.1 Highlights of the agile principles and where to get data to measure them

	Project tracking systems	Source control management	CI and deployment tools	Application monitoring
Good designs		X	X	X
Good architectures		X	X	X
Technical excellence		X	X	X
Changing requirements	X	X	X	
Working together	X			

Table 10.1 Highlights of the agile principles and where to get data to measure them

	Project tracking systems	Source control management	CI and deployment tools	Application monitoring
Motivated individuals	X			
Face-to-face communication	X			
Continuous delivery		X	X	
Becoming more effective	X	X	X	
Delivering frequently	X		X	
Working software				X
Satisfied customers				X
Simplicity	X	X		

With the agile principles mapped across the delivery lifecycle and broken down across four key questions, you can start using metrics from previous chapters to start getting specific on mapping metrics to agility.

10.2 Three principles for effective software

The following three principles all have keywords that make you ask, "Is our software effective?" The keywords are italicized in the following list of principles:

- *Working software* is the primary measure of progress.
- Our highest priority is to *satisfy the customer* through *early and continuous delivery* of valuable software.
- Continuous attention to *technical excellence and good design* enhances agility.

Working software is perhaps the most obvious of the software-related measures but many times the hardest to measure. Chapter 9 is dedicated to measuring how well your software is working.

Satisfying the customer is measured using techniques from chapter 6: using telemetry and business-specific metrics to measure how well your software does what it's supposed to do.

Early and continuous delivery is covered in chapter 4 in our discussion of CI and deployment systems. This is enabled by build systems that output digestible reports and automate enough of your build and deployment cycle to get comprehensive data around build and deployment times. You can also measure this through your PTSs covered in chapter 3 by tracking lead time or development time.

You can argue that technical excellence and good design are measured by how fast you can iterate on your code or how maintainable it is, and how well it satisfies the consumer or how usable it is. Both of these are detailed in chapter 9.

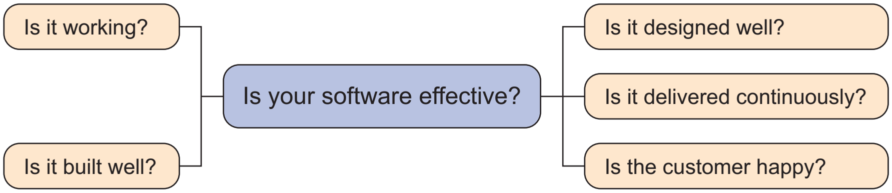


Figure 10.4 Mind mapping the big question "Is your software effective?" into smaller bits that represent the principles

10.2.1 Measuring effective software

Let's start with a mind map to align the principles with the big question you're trying to answer, as shown in figure 10.4.

If you single out just these questions over the delivery lifecycle, you'll see that these questions end up spanning most of it, as shown in figure 10.5.

We talked about measuring CD in chapter 5, and there are concrete metrics you can use to ensure that it is working well like the following:

- Successful versus failed builds:
  - How well is your code review process working?
  - How good is your local development environment?
- Is your team thinking about quality software?
- How frequently do you get updates in front of your consumers?

Ensuring that your software products are working well and your customers are satisfied are the topics we covered in chapter 6 when we talked about production monitoring and arbitrary metrics. Here are some metrics that can help you keep an eye on these:

- *Business/application-specific metrics*—These are defined based on your application and tell you how consumers are using your site.

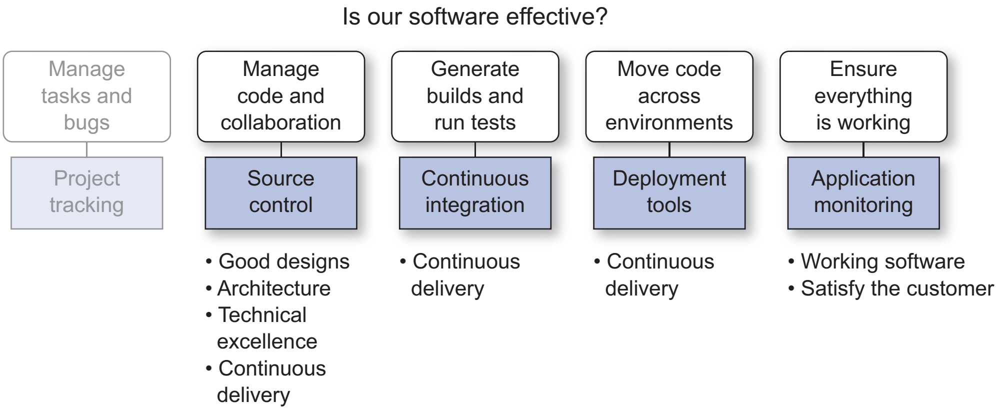


Figure 10.5 The systems you can get metrics out of to see if your software is effective

- *Common site health statistics*—These tell you how your application is performing and how healthy it is. Some key stats to keep an eye on for this would be:
  - Error count
  - CPU/memory utilization
  - Response times
  - Transactions
  - Disk space
  - Garbage collection
  - Thread counts
- *Semantic logging*—This can also help you monitor application-specific metrics.

We also covered some of these metrics in chapter 8 when discussing what makes usable software with the following metrics:

- Usability
- Uptime
- MTTF

You can use the metrics we covered in chapter 8 to look at what makes well-built software:

- Maintainability
- *MTTR*—How fast can you get fixes out to consumers?
- *Lead time*—How fast can you get features out to consumers?

Using this comprehensive list of metrics you can get a clear picture of how effective your software is, measured across the agile principles. Building software is *what* you're doing; your process is *how* you do it. The following four principles give you an outline on how to measure your process.

10.3 Four principles for effective process

The next group of four principles answers the question "Is your process effective?"

- *Simplicity*—The art of maximizing the amount of work not done is essential.
- *Deliver working software frequently*, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- *Face-to-face conversation*—The most efficient and effective method of conveying information to and within a development team.
- *Maintain a constant pace* indefinitely—Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

As a software engineer, my initial reaction to simplicity was that it related to software measurement. After all, you should use simple designs and standard patterns to maximize efficiency on your development team and increase the maintainability of your software. Although maximizing the amount of work done relates to maintainable code, simplicity is most easily measured by looking at data from your PTS, such as task volume or the number of tasks completed. Maximizing the amount of work done implies an effective process: understanding the minimum of what needs to get done in order to improve your consumer's experience.

Delivering working software frequently is easily measured by the amount of time it takes to get tasks through the lifecycle and out to consumers. One caution for measuring the delivery of working software is to measure the time it takes to get working software to the consumer, not to the end of a development cycle if that doesn't include deployments. I've seen teams celebrate the fact that their development time was very short, even though they did gigantic monthly releases to their consumers. Be honest with your delivery time.

Face-to-face conversation is great, but I've been in plenty of meetings or planning sessions where what is said ends up getting interpreted differently by different people in the room. The real key to this principle is ensuring that the team is directly communicating with each other for the highest possible clarity and understanding of what you're all working on.

Maintaining a constant pace is most commonly measured by that old agile steadfast metric, velocity. Your ability to maintain a constant pace is affected by a host of factors including these:

- *Maintainability of your code*—If you have code that's easy or at least predictable to change and deploy, you should be able to keep a constant pace.
- *The consistency of your estimates*—By showing that you understand changes and your team is breaking tasks down into manageable chunks, you avoid hiccups in your pace.

10.3.1 Measuring effective processes

Figure 10.6 shows a mind map that helps you take a closer look at measuring your process.

Again we'll map the highlights across the delivery lifecycle to see what kind of data you should care the most about in the context of your process. See figure 10.7.

Starting with simplicity, there are a few different metrics you can look at:

- Use CLOC from your SCM system to see how much code is changing.
- Estimates and volume from your PTSs show you the amount of effort and number of tasks completed.

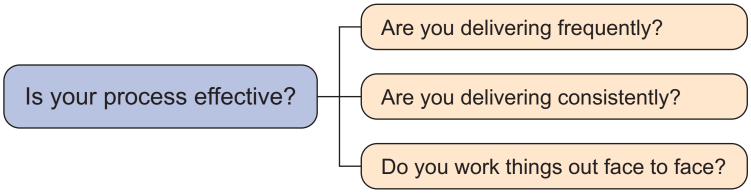

Is your process effective?
Are you delivering frequently?
Are you delivering consistently?
Do you work things out face to face?

Figure 10.6 Mind mapping "Is your process effective?"

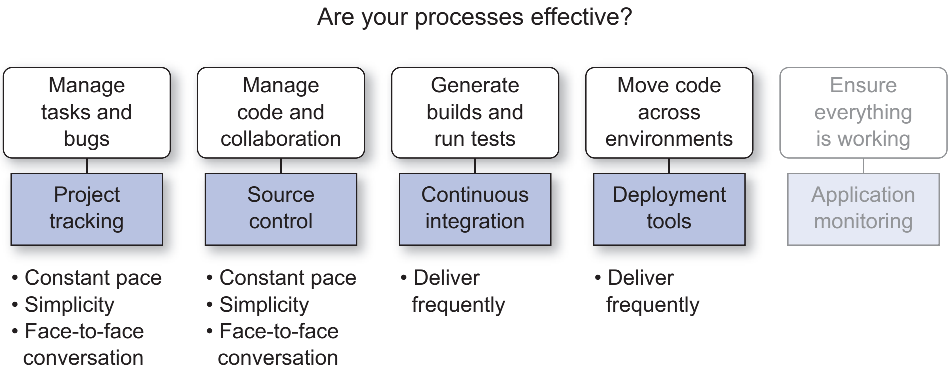


Figure 10.7 Are your processes effective?

- Use estimate health and estimates together to check how predictable your estimates really are.
- Lead time, development time, and volume give you the high-level picture of how fast you can deliver and at what frequency.

Frequent delivery is measured not only by how often you ship code but also the pace at which you move. You can measure both frequency and pace with the following metrics from chapters 3 and 5:

- The old agile metric of velocity is great at tracking consistency.
- Use the number of successful deployments from your build and/or deploy systems.
- Lead time will tell you how long it takes to get a task all the way through the lifecycle from concept to delivery.
- MTTR tells you how quickly you can react and tweak your system when necessary.
- Bug counts are a good way to check whether you're delivering value or you're churning on issues due to poor code quality.
- As a check to bug count you can add code coverage and static analysis to the mix to make sure you're delivering maintainable code.

Face-to-face conversation isn't so easy to directly measure with agile tools used today. But you can measure the amount of communication through comments captured in your systems at various points in the delivery flow and cross-reference those with arbitrary tags and labels in your PTS to figure out what level of measureable communication works best for your team:

- PTS and SCM comment counts, as noted previously, imply how well collaboration is going.
- Cross-reference these comment counts with labels and tags on tasks that your team uses to indicate if they think communication is working well.

Your team is building the software through agile processes; measuring how well your team works together is the next set of agile principles we'll take a look at.

10.4 Four principles for an effective team

The next group of four answers the question "Is your development team effective?"

- The best architectures, requirements, and designs emerge from self-organizing teams.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- At regular intervals, the team reflects on how to become more effective and then tunes and adjusts its behavior accordingly.

These principles are all people issues: how well your team works together, how motivated they are, your level of autonomy, and how well you can check and adjust. How well your team is working together affects many of your other metrics. How quickly tasks go through your workflow, how much code is written per task, how many bugs turn up, and how tasks move through your workflow are all indicators of how well your team is working together.

The first principle in this list ties team autonomy, or self-organization, directly back to the quality of the software.

You could argue that the fourth item in this list, reflecting at regular intervals, is a process question because checking and adjusting are part of the agile development process. But I've seen dysfunctional teams take the time to reflect, only to chase the wrong goals because they can't be honest with each other or they don't engage in the improvement process. It's better to measure the ability to check and adjust as part of measuring the effectiveness of your team because it shows your team's ability to face up to their weaknesses and take action to improve them.

The key indicators of good teamwork will be how well things move through your workflow and how your team feels about their work.

10.4.1 Measuring an effective development team

We'll start off again by creating a mind map for these questions, as shown in figure 10.8.

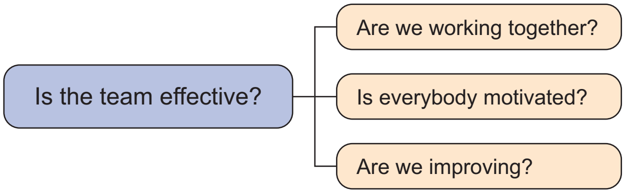


Figure 10.8 Mind mapping "Is the team effective?"

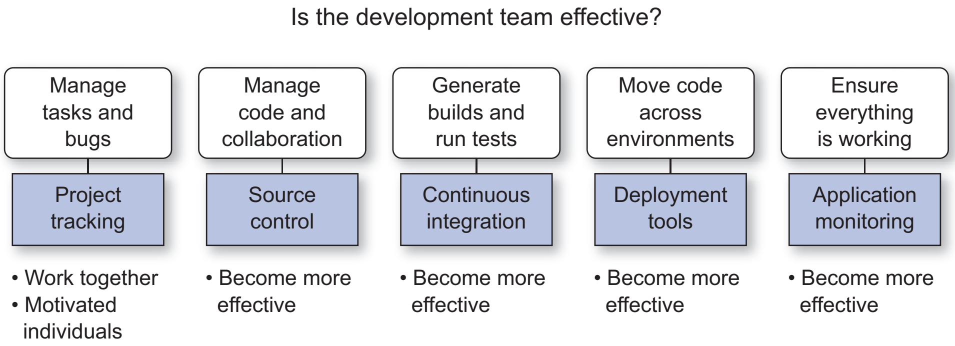


Figure 10.9 Transposing the principles regarding the development team over the development lifecycle

Transposing this over the delivery lifecycle, you'll see that these questions end up spanning the entire lifecycle, as shown in figure 10.9.

In chapters 3 and 4 we went through the PTS and looked at data that helps analyze how your team is working together and how motivated they are through the following points:

- By tagging or labeling tasks in your PTS with how well developers think the task was executed, you can measure how motivated individuals are. We covered this in section 3.2.5 with the use of happy and unhappy tags and used the same technique as a mutiny indicator inside the front cover.
- Teamwork can be measured by comment counts in PTS and SCM data. Keep in mind that what is good or bad will vary from team to team, so it's important to calibrate the numbers based on how well your team is working.
- Recidivism will also give you a good idea of how well your team is working together. If tasks are moving through your workflow, that's usually a good indicator that the team is working well together. You can check this by adding in consumer-facing defects; low recidivism and a high number of consumer-facing defects mean that your team is passing tasks through without properly vetting them—that's bad. Conversely, high recidivism with low consumer defects means that your team is churning, but at least it's turning out code of good quality.

In chapter 2 we talked through how you can use tags and labels as a potential replacement of the Nico-nico calendar, an agile way to measure the motivation of your team. If you encourage your team to label tasks with how they're feeling as they finish tasks, then you'll start to build up data around the motivation of your team. What's better is that you'll have tasks that relate to the happiness of your team.

If you look at measuring improvement of effectiveness of your development team in this context, you can look at frequency of delivery. You can measure that with the following metrics:

- *Lead time*—The amount of time it takes from inception to delivery of tasks

- *Development time*—The time from when a task enters the development flow to its completion
- *Deploy frequency*—How often you get changes out to consumers
- *Good/failed builds*—How frequently you're delivering working software

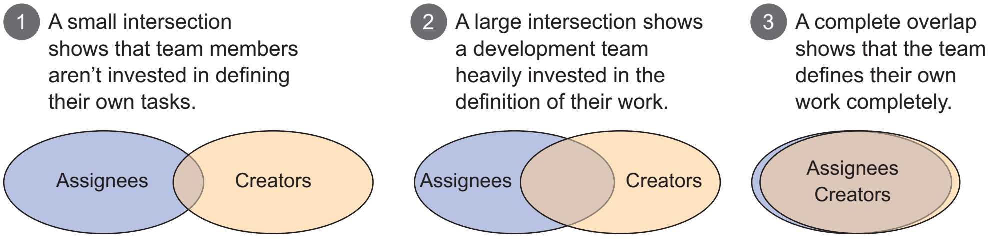
Measuring autonomy with the data you have isn't very straightforward. I've had success with this by checking the counts of people entering and commenting tasks in the system before they enter the work stream along with the counts of people who are assigned the tasks. In very autonomous teams you'll see that assignees have enough ownership to collaborate toward the definition of a task before it gets assigned to the team. In teams that simply receive marching orders, you typically see members outside the immediate team doing the bulk of the creation and definition of tasks. Another pattern to look out for is nonteam members moving tasks through the workflow. A pictorial representation is shown in figure 10.10.

Of the three different ratios, you should stay away from either extreme (the examples on the left and right in figure 10.10). In the first example, a small intersection between assignees and creators typically can lead to members of the team who specialize in creating requirements and filling out tasks and other team members who just work the specs they're given. You'll see symptoms of this when you start to hear people complaining about requirements or blaming requirements for poor or incorrect implementations.

In the example on the right in figure 10.10, on teams where developers are responsible for creating their own tasks, you can start to see people entering the bare minimum in the system to indicate they have a task to work on. This extreme also tends to lead to poor and incomplete requirements, and this becomes a problem when people forget what their minimal cards meant and try to estimate them, or if work ends up getting divided among other team members.

The second example (the one in the middle) is usually a good mix of members inside the team and outside stakeholders entering tasks into the work stream. This mix typically gives the development team enough investment in their work but also has enough outside influence to keep requirements complete.

An effective team creates effective software using effective processes. The final element is the requirements these teams use to build their software. The last agile



1 A small intersection shows that team members aren't invested in defining their own tasks.

2 A large intersection shows a development team heavily invested in the definition of their work.

3 A complete overlap shows that the team defines their own work completely.

Figure 10.10 A pictorial representation of the overlap between task assignees and creators and how it relates to the level of team autonomy

principle belongs in a category by itself; although it's only a single line, it speaks to how your team defines what they're doing before they do it and can be measured from a few dimensions.

10.5 One principle for effective requirements

The final question you need to answer is "Are your requirements effective?"

- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Changing requirements is one of the most frustrating things that a team needs to take in stride. I've been in many sprint retrospectives where teams complain about missing goals due to changing requirements, I've seen products make it out the door with a fraction of intended functionality because of indecision around requirements throughout the development of the product, and I've been a part of several development cycles where our work was scrapped because architectures, features, or experiences changed after we started.

When it comes to figuring out how well your team can handle changing requirements, you need to keep two things in perspective:

- How do you know if requirements are changing?
- Can your team maintain a consistent pace when requirements change?

For this you'd have to know your current metrics for consistency (for example, lead time, recidivism, and velocity) and you'd need to know when requirements are changing. With both of those in hand you can measure how consistently your team performs in the face of change.

10.5.1 Measuring effective requirements

I've mind mapped the final principle in figure 10.11.

The interesting thing here is that you need to compare your consistency when requirements are static against when they change. To do that you need to measure your team's consistency and you need to track when requirements are in flux.

Velocity alone is a terrible metric to measure how well a team can handle change and as a result is often a bone of contention when requirements are changing. If a team makes a commitment and starts a sprint, and then midway through the process the end goal moves, the team is set up to miss their commitment. You can reestimate or refactor your sprint goal, but in the grand scheme of things it will look like your team went through a patch of low productivity when perhaps they were more productive than ever.

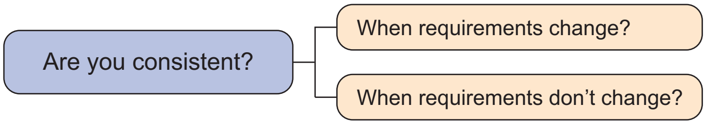


Figure 10.11 Mapping out "Are you consistent?"

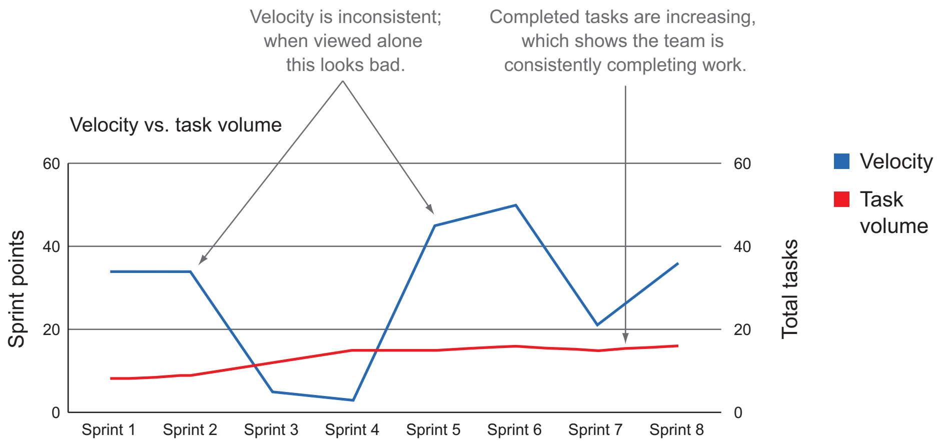


Figure 10.12 Velocity for this team is inconsistent, but total completed tasks are actually increasing. This is a typical indicator that the team completes tasks consistently despite changing commitments and/or requirements.

Better metrics to use to track consistency through changing requirements are task volume and average estimates. The number of tasks that your team is able to accomplish in the face of change tempered by average estimates will give you a good picture of how consistently your team is completing its work. If your estimates consistently correlate to a few days of work and your task volume stays constant, you should be in good shape. If you see these trends along with an inconsistent velocity, it shows you that your team is doing great in the face of change.

Other metrics that help check team consistency while velocity is in flux are recidivism and lead time. Stable lead time and recidivism trends during periods of inconsistent velocity show that through changing commitments the time it takes to get tasks out the door and the health of your workflow remain solid. This is demonstrated in figure 10.12.

Effective requirements can most easily be measured at the beginning and the end of the development cycle, as shown in figure 10.13.

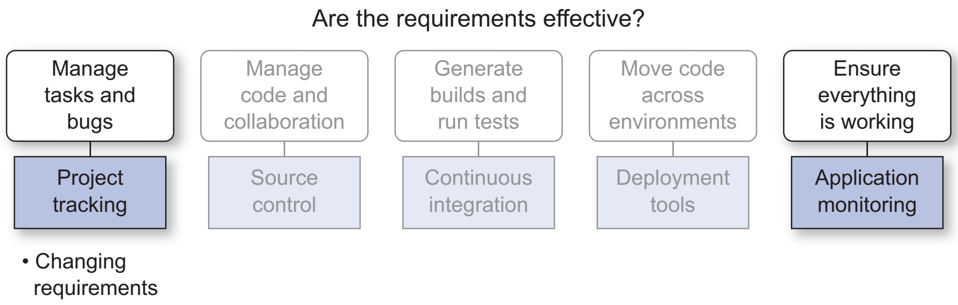

Are the requirements effective?
Manage tasks and bugs
Manage code and collaboration
Generate builds and run tests
Move code across environments
Ensure everything is working
Project tracking
Source control
Continuous integration
Deployment tools
Application monitoring
• Changing requirements

Figure 10.13 Where to find data to measure effective requirements

I've identified only a single agile principle that directly addresses requirements, but you can also argue that some of the principles I've categorized under an effective team also apply here because requirements are often a contract between the business, developer, and quality roles on the team. You can look at requirements from a few angles. First, does your development team understand them and can they work with them? Second, do your customers actually get what they want? You can measure those with the following metrics:

- *Recidivism*—When recidivism is high, that means someone in the workflow didn't have the same standard as someone downstream. This is usually an indicator of incomplete or inconsistent requirements.
- *Lead time, velocity, and development time*—These all measure how long it takes for your team to get tasks across the finish line.

Your *team* translates *requirements* into *software* using agile *processes*. The agile principles from the Agile Manifesto can be broken down into these four measurable elements that you can track to see how well your team is aligning to the agile principles. Now let's take a look at a team in our case study that puts these measurements into practice.

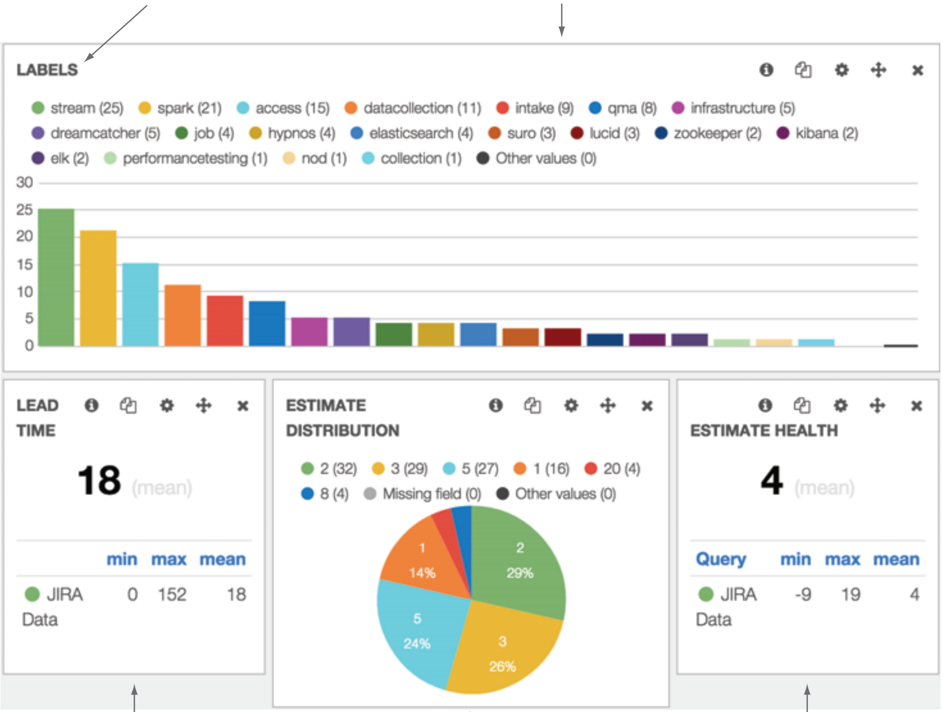
10.6 Case study: a new agile team

Many companies move their development practices to be more agile but don't get all the way there. Sometimes legacy systems can't be turned off; mainframes or large financial reconciliation systems simply can't be rebuilt or refactored in a reasonable amount of time and can't be deployed even as frequently as every few weeks.

This case study takes us to a team that has moved many of their web and mobile development teams to agile practices but has a large financial mainframe that gets updated once a quarter. After seeing the speed at which the agile teams are delivering code, the leadership team sponsors an effort to implement a new system that will take on the functionality of the mainframe one piece at a time. During this transition they'll move developers from the mainframe maintenance teams over to the new agile teams, train them to work differently, and expect them to start practicing CD.

Technical problems are the easy ones; teaching people new processes when they've been doing the same thing for several years is much harder. To help transition the team, the leaders started by agreeing on target metrics to track so the team could see the progress. The strategy was to start at the top, ensure that everyone understood what the high-level goals were, and when they established a baseline, then start adding more detailed metrics to continually improve the process.

Each sprint they created headlines; each headline correlated to a demonstrable piece of functionality that they committed to finishing by the end of the sprint. The headlines were what was determined would *satisfy the customer.* They used the headlines as their indicators of *working software* and made sure that they were breaking their tasks down into small enough chunks that they could deliver pieces *early and continuously* through their sprint. As they broke tasks down, they used labels to associate headlines with tasks so they could track individual metrics for tasks grouped by headlines. To


LABELS
● stream (25) ● spark (21) ● access (15) ● datacollection (11) ● intake (9) ● qma (8) ● infrastructure (5)
● dreamcatcher (5) ● job (4) ● hypnos (4) ● elasticsearch (4) ● suro (3) ● lucid (3) ● zookeeper (2) ● kibana (2)
● elk (2) ● performancetesting (1) ● nod (1) ● collection (1) ● Other values (0)
30
25
20
15
10
5
0
LEAD TIME
18 (mean)
ESTIMATE DISTRIBUTION
● 2 (32) ● 3 (29) ● 5 (27) ● 1 (16) ● 20 (4)
● 8 (4) ● Missing field (0) ● Other values (0)
2
29%
3
26%
5
24%
1
14%
ESTIMATE HEALTH
4 (mean)

	min	max	mean
● JIRA Data	0	152	18

Query	min	max	mean
● JIRA Data	-9	19	4

Each headline that represented major features is tracked with a label.

In Kibana you can click on a term to filter all metrics by your selection.

The key metric, lead time, shows the team how long it takes to get tasks all the way through the development cycle.

The distribution of estimates shows the percentage of tasks grouped by their estimates. This team decided that the bulk of estimates should be low numbers.

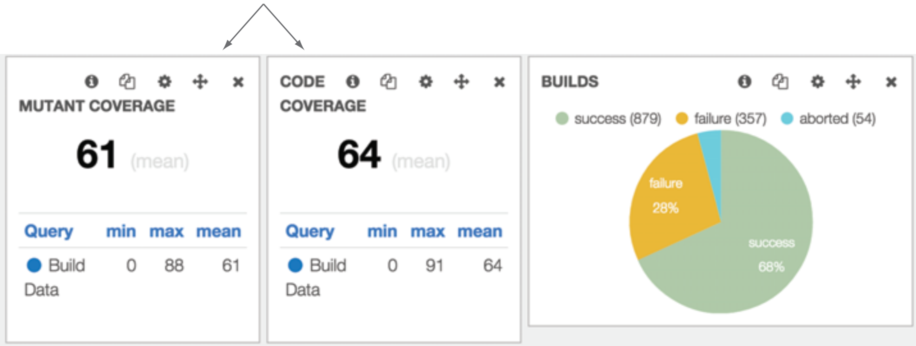
Estimate health shows the team how close their estimates are to actual time. A value of 4 means on average tasks take four days longer then the original estimate.

Figure 10.14 The team's dashboard with labels, estimate distribution, estimate health, and lead time. These metrics helped the team stay aware of how well they were estimating and delivering.

track success in moving to this new process they decided to measure lead time, estimate health, and estimate distribution. The estimate distribution and estimate health would show them how well they were breaking down tasks, and lead time would show how long it actually took to deliver tasks end to end. The dashboard they started using is shown in figure 10.14.

As they were developing, they wanted to ensure they were paying continuous attention to the technical excellence of their software. Before they started developing, the team put all the build and monitoring systems in place to enable easier delivery and better quality. They were able to adopt tools from other teams in the organization that were already practicing agile and CD such as Sonar, their internal delivery pipeline for CI, and APM tools for production monitoring. By using these tools, they would be able

Code coverage shows how well the team is writing tests. Mutant coverage shows how good the tests are. In this case a slight discrepancy in the two isn't bad but still worth investigating.



MUTANT COVERAGE

61 (mean)

Query	min	max	mean
● Build Data	0	88	61

CODE COVERAGE

64 (mean)

Query	min	max	mean
● Build Data	0	91	64

BUILDS

● success (879) ● failure (357) ● aborted (54)

Figure 10.15 The next set of charts used by the development team included good versus bad builds, unit test coverage, and mutant coverage.

to better understand the quality of their products as they were developing them instead of through long QA cycles after weeks of development. Because this was a new concept for the development team, they needed some time to become familiar with the tools and get used to using them in their development process. But once they were up and running, they knew from the success of other teams in the company that they could become a more effective team.

To track their success with CI they tracked unit test and mutant coverage to make sure the team was writing decent tests and tracked good versus bad build ratio to make sure they were iterating on a working product. The dashboard they used for this is shown in figure 10.15.

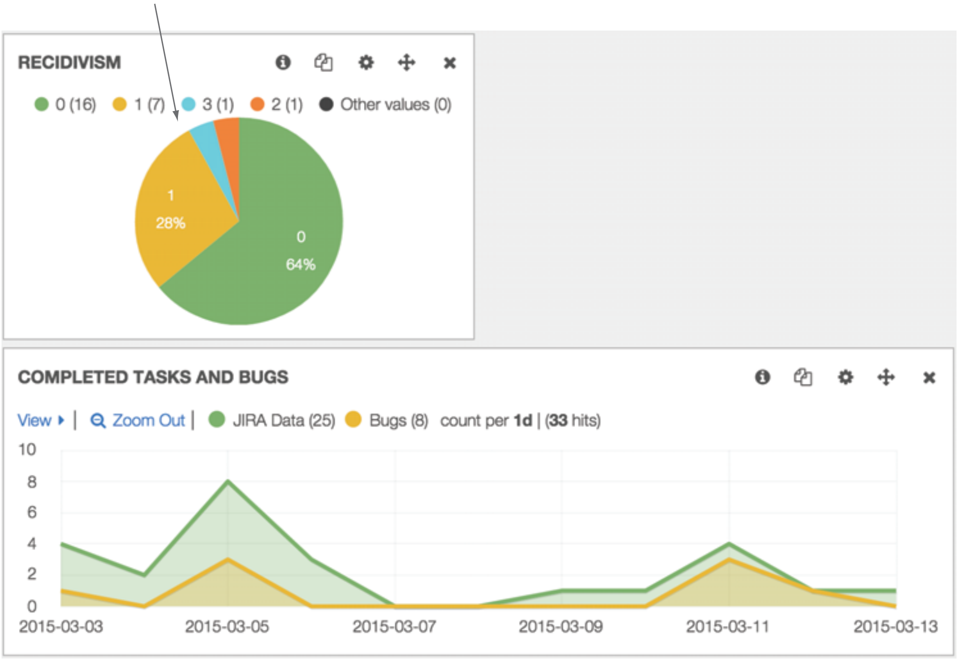
An effective team produces great software, but an effective team is made up of motivated individuals who work together across functions and reflect at regular intervals to become more effective. To measure this, the team tried a few different tactics that other teams across the organization were already doing.

Because they were following Scrum, they had a built-in mechanism to reflect every two weeks in their retrospective. To ensure this turned into an effective and productive session, they drove the sessions with data. When they discussed what they could improve, they also discussed what metrics they could use to track their improvement. The key metric they used to measure everything was lead time. The thinking was that by making the whole team pay attention to the time it took to deliver a feature end-to-end, they would bring the business and development teams together toward the common goal of delivery. After tracking their lead time until they delivered something, they could then all sit down and reflect on what they could do to improve. Of course, along the way they collected all the data they needed to look into the details of the team when they were ready to do so.

After a single sprint the team had delivered a partially functional web service. After another sprint they were able to get a functional UI on top of it that met the criteria for their first set of features. They were operating in two-week sprints and at the end had a lead time of 28 days.

In their retrospective they started breaking down lead time to find efficiencies in their process. They found that there was still a lot of back and forth between the stakeholders, developers, and QA. They correlated that to a lack of good face-to-face conversations and realized that in their case to track improvement they could use recidivism to find the overall churn between the different roles on the team and use the number of bugs to track how well the development and QA teams were communicating, as shown in the dashboard in figure 10.16.

36% of all tasks move backward in the workflow at least once. Of those that move backward nearly 25% move backward more than once. Clearly there is room for improvement with recidivism.



By tracking bugs completed along with tasks completed it's apparent that many of the completed tasks are bugs, and it looks like the percentage of bugs is increasing over time.

Figure 10.16 The dashboard tracking recidivism and bugs versus total tasks to help the team track how well they're working together across disciplines.

After another sprint of analyzing data from their PTS, they started to get the hang of finding problems in retrospectives, using data to track corrections, and coming up with strategies to improve. The development team was not only new to agile, it was also new to working so quickly, and they had a lot to learn about the best way to manage source control and deployments in this new paradigm. They started paying attention to pull requests versus commits and to build and deploy times from their CI systems and added those metrics to their dashboards.

After several sprints the team was using several metrics based on the agile principles to continually improve their development and delivery processes. They felt empowered to deliver changes frequently, and their morale was high after successfully moving toward agile methodologies.

10.7 Summary

At this point you should have a plethora of tools at your disposal to start measuring your team, incorporating metrics into your development cycle, and communicating them across your team and up the chain.

In this chapter you learned the following:

- To measure a team against the agile principles you need to answer four big questions:
  - Are the requirements effective?
  - Is the development team effective?
  - Are your processes effective?
  - Is your software effective?
- You can measure requirements with metrics covered in previous chapters:
  - Recidivism
  - Lead time
  - Development time
  - Velocity
- You can measure the development team with metrics covered in previous chapters:
  - Lead time
  - Development time
  - Deploy frequency
  - Good/failed builds
- You can measure your process with metrics covered in previous chapters:
  - Velocity
  - PTS and SCM comments
  - Successful deployments

- You can measure your software with metrics covered in chapter 8:
  - Successful versus failed builds
  - Business metrics
  - Performance health data

appendix A
DIY analytics using ELK

This appendix covers

- Reviewing the overall architecture of an agile metrics collection and analytics system
- Setting up an ELK server
- Building a data-collection application using Grails
- Installing the data collector on the ELK server

In this appendix I'll walk you through setting up a powerful analytics system using the ELK stack (EC, Logstash, and Kibana). You can download the basic stack from the EC website. You can also get the setup I've used for this book on GitHub at github.com/cwhd/measurementor.

Let's start by looking at the high-level component diagram in figure A.1 to get an idea of the pieces you need to build. You'll need to write some scripts to collect data, set up a database to store it, and put together an indexing, search, and visualization engine. Once you have all of these components, you'll be ready to start building out connectors to external applications used on the development lifecycle to get data and analyze it. You'll use the following technologies for these components:

- *Data collection*—Grails (grails.org/)
- *Database*—MongoDB (www.mongodb.org/)
- *Data indexing and search*—EC (www.elasticsearch.org/)
- *Data visualization*—Kibana (www.elasticsearch.org/overview/kibana/)

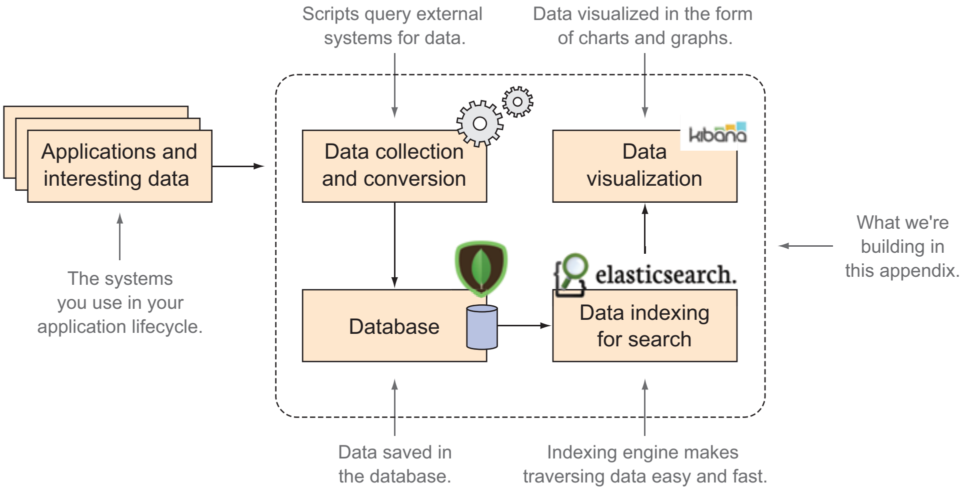

Scripts query external systems for data.
Data visualized in the form of charts and graphs.
Applications and interesting data
Data collection and conversion
kibana
Data visualization
The systems you use in your application lifecycle.
elasticsearch.
Database
Data indexing for search
What we're building in this appendix.
Data saved in the database.
Indexing engine makes traversing data easy and fast.

Figure A.1 The high-level component diagram of the analytics system

Figure A.2 shows how data flows through the system.

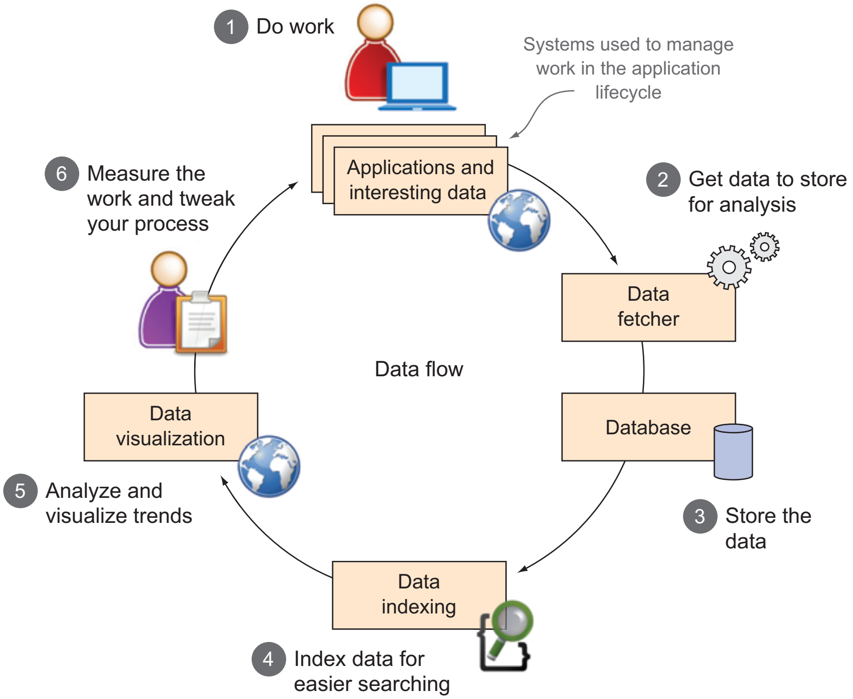

1 Do work
Systems used to manage work in the application lifecycle
Applications and interesting data
6 Measure the work and tweak your process
2 Get data to store for analysis
Data fetcher
Data flow
Data visualization
Database
5 Analyze and visualize trends
3 Store the data
Data indexing
4 Index data for easier searching

Figure A.2 A closer look at the flow of data through the analytics system

Figure A.2 illustrates the data flowing through the analytics system during the application lifecycle. Combining this with the diagram in figure A.1 shows that you're going to do the following:

- Get MongoDB up and running as your database.
- Get EC and Kibana up and running for data indexing, search, and analysis.
- Create the application that gets data and saves it to the database.
- Generate charts and graphic visualization.

I already have this running for you, so go to GitHub, get the latest version, and run the Puppet scripts to set it all up.

If you want to use something other than Grails, it's okay. I have working code, but if you follow along with the examples in appendix B where I go into more detail on the data-gathering application, you can easily translate the system into another language.

Using the examples with Puppet and Vagrant

As you build the system in this appendix you'll be installing tools and a database, and I'm going to assume you have Java installed on your development machine. But wouldn't it be cool if you didn't have to worry about any of that? You could open the system and start developing it without having to worry about installing anything, the version of tools that may already be installed on your computer, or differences between OSs. Versioning environments as you version code have become popular along with the recent rise of DevOps (en.wikipedia.org/wiki/DevOps), the concept of combining development and operations to be able to release software faster and more efficiently. Puppet (puppetlabs.com/) is a technology that allows you to write code that automates the installation and versioning of the software on your environment. Automating the installation and versioning of software is great, but it's even better if you can switch between development environments if you're working on different projects or experimenting with different tools and technologies. Using virtual machines (VMs) allows you to keep multiple environments at your fingertips for just that, switching between OSs or environment configurations to work across projects with different dependencies. Vagrant (www.vagrantup.com) is a development tool that abstracts the details of VMs away from the developer and integrates seamlessly with Puppet so that you can automate the installation of software and jump into your development environment with a simple command from your terminal of choice.

As you implement this system, you'll do it in a way that can be run on your local machine, but the code examples from the website will be set up using Vagrant and Puppet. Vagrant makes developing locally easier by wrapping everything in a VM. Puppet will configure the Vagrant box for you; that way you don't have to worry about whatever changes you're making locally and it's easier to install the solution on different systems.

Before you start building the system, let's take a more in-depth look at its design and specification.

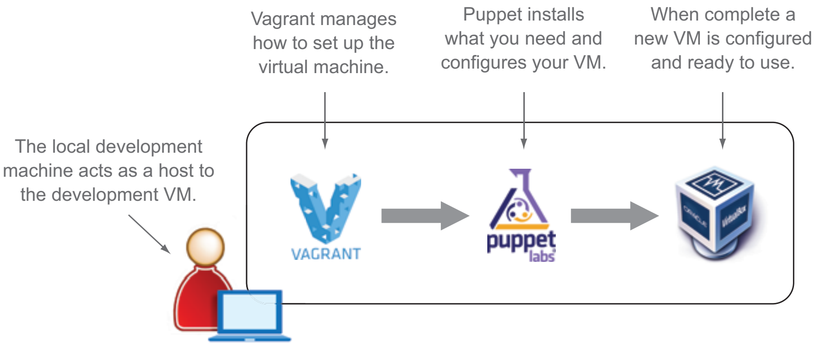

Vagrant manages how to set up the virtual machine.
Puppet installs what you need and configures your VM.
When complete a new VM is configured and ready to use.
The local development machine acts as a host to the development VM.
VAGRANT
puppet labs

Figure A.3 How Vagrant, Puppet, and VirtualBox interact with your development environment

A.1 Setting up your system

Now that we've looked at the high-level design, let's set up the system. Because the system depends on open source components instead of finding, downloading, and installing every component, I've boiled all the pieces into a Puppet script, which you'll use to install everything you need in a VM using Vagrant. Using Puppet you can set up this system anywhere you can create a VM. For the purposes of this book I'll focus on getting you up and running on your localhost.

The interaction of Vagrant, Puppet, and VirtualBox on a local development machine is shown in figure A.3.

First, get the latest version of Vagrant from www.vagrantup.com/downloads.html. If you're using Windows, you'll also need to install `Cygwin` with the `openssh` package so you can connect to your Vagrant machine when it boots up.

Second, download the code for this book from github.com/cwhd/measurementor. The codebase contains the Vagrant and Puppet configuration to install all the components you need. The Vagrant file contains a line of code that sets up a shared directory between your local machine and the Vagrant box to make tweaking the system easier. In the Vagrant file around line 49, change `PATH_TO_DOWNLOADED_PROJECT_HERE` to the path where you downloaded the code. For example, if you downloaded the code to /Users/cwhd/Development/measurementor, replace `PATH_TO_DOWNLOADED_PROJECT_HERE` with `/Users/cwhd/Development/measurementor`. For Windows users, be sure to escape slashes. For example, if you download the code to C:\Agile Metrics\measurementor on a Windows machine, change `PATH_TO_DOWNLOADED_PROJECT_HERE` to `C:\\Agile Metrics\\measurementor`.

> **NOTE** As the versions of these components update, measurementor will evolve on GitHub after this book is printed. For the latest setup instructions please check the readme file for the project.

Finally, navigate to the directory where you downloaded the code and run the following commands.

Listing A.1 Running Vagrant

```
$> vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Checking if box 'hashicorp/precise64' is up to date...
==> default: Resuming suspended VM...
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
    default: Warning: Connection refused. Retrying...
==> default: Machine booted and ready!
$> vagrant ssh
Welcome to Ubuntu 12.04 LTS (GNU/Linux 3.2.0-23-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
New release '14.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


Welcome to your Vagrant-built virtual machine.
Last login: Mon Nov 24 21:24:23 2014 from 10.0.2.2
vagrant@precise64:~$
```

Runs your vagrant file

Vagrant output as it boots up your local VM

Command to tunnel into the VM

Vagrant output as you log in.

You're ready to go!

Now you have a local VM with all the components you need installed on it. In the Vagrant file I've set up ports to share with your localhost, so you can check on EC and Kibana from your web browser at the following URLs:

- EC: http://localhost:9200
- Kibana: http://localhost:5601

If those URLs come up, then you've successfully gotten your local environment up and running.

A few more useful Vagrant commands

Vagrant makes it very convenient to develop in a sandbox. It's so easy to set up a new system that often during development you may want to tear the system down and start again. To completely wipe away your environment use the `vagrant destroy` command, you can always create a new one using `vagrant up again`.

One drawback to using a VM for development is that it will use up a lot of your system's resources, making other programs slow while your VM is booted and running. If you want to halt development and go back to other work, you can pause your Vagrant box using the `vagrant suspend` command. That will effectively pause the state of your VM and return the memory and CPU to your other applications. To resume using it, simply use `vagrant up again`.

For a complete reference visit the Vagrant website.

A.1.1 Checking the database

The previous Puppet script installed Mongo for you. If you want to check directly from the database, you can do so from the command line in your Vagrant host with the commands in the following listing.

Listing A.2 Checking data In MongoDB

```
vagrant@precise64:~$ mongo
MongoDB shell version: 2.6.5
connecting to: test#B
> show collections
jiraData
jiraData.next_id
jobHistory
jobHistory.next_id
> db.jiraData.find()
{ "_id" : NumberLong(8805), "assignees" : [ ], "created" : ISODate("2014-
     11-21T19:19:05Z"), "createdBy" : "james.lee3_nike.com", "dataType" : "PTS",
     "issuetype" : "Bug", "key" : "ACOE-885", "leadTime" : NumberLong(0),
     "movedBackward" : 0, "movedForward" : 0, "storyPoints" : 0, "tags" : [ ],
     "version" : 0 }
> db.jiraData.remove({})
> exit
```

Starts the Mongo console

General informational outputs

Shows all the collections in the database

Inspects the jiraData collection

Deletes the entire collection

Exits MongoDB

Note that if you haven't yet run the data collector, you won't see any data in your database.

For complete documentation on MongoDB, visit www.mongodb.org/.

A.1.2 Configuring your data collector

The application that does everything between the database and the web display (a.k.a. the middle tier) is written with Grails and Groovy. If you want to go in a different direction and write something yourself, you can achieve the same results using other languages. The key is to use whatever you're most comfortable with, because that will make the process as painless and seamless as possible. I personally like Groovy because it's a popular language, easy to use, highly supported, gets better with every release, does a lot of the programming tedium for you, and can run on the Java VM (JVM), which makes it highly portable and scalable. In short, it's a Swiss army knife that lets you build stuff fast. If there are examples in certain languages you really want to see, feel free to request help on the Author Online forum,[1] and I can help post examples in whatever language you'd like to use.

In appendix B I'll give more information about the details of the Grails app if you want to tweak it. But it should work out of the box with a few configuration tweaks if you don't want to get under the hood.

I've included a shell script with the code that will walk you through the configuration of the system. It will ask you for the URLs and credentials for all the supported sys-

[1] www.manning-sandbox.com/forum.jspa?forumID=924&start=0

tems you want to connect to, so you should have those ready. You have to enter credentials as base64-encoded strings that can be used for HTTP basic auth. For example, if your username is UserOne and your password is h0lm3s, then your basic auth string would be UserOne:h0lm3s. You can use a local utility to base64 encode your strings, or you can go to a site like www.base64encode.org/.

The shell script will copy the file measurementor.properties to a new file called application.properties. Then it will take the parameters you pass into the shell script and update the settings for the application. If you don't run the shell script or want to change anything, you can do the steps manually by copying measurementor.properties to a new file called application.properties and adding the URLs to the systems you want to connect to. An example for JIRA connectivity should look like the following listing.

Listing A.3 Setting up your config file

```
jira.credentials=dXNlcjpwYXNzd29yZA==
jira.url= https\://jira.whatever.com
```

The base64 encoded basic auth

URLs to the root of the systems

Because you're focusing on getting this running locally, you can now run the Grails application from inside your Vagrant box. It will do the following:

- Reach out to the systems you've specified.
- Get data out of them.
- Write the data into MongoDB.
- Index the data in EC.

SSH into your box and run the app with the commands in this listing.

Listing A.4 Running the Grails app

```
>$ vagrant up
vagrant@precise64:~$ cd /measurementor
vagrant@precise64:~$ grails run-app
```

Logs into your Vagrant box

Navigates to the directory where the app lives

Runs it

Now everything is good to go! The app is configured to run once a day to get data out of your systems and index it.

Be careful if you've hosted JIRA

If you're using the hosted version of JIRA, be careful how frequently and at what times you run your data collection. I've worked on a fairly large team with a lot of data and I've noticed that under load JIRA completely dies. If you have a small team, then you should be okay, but if you have a good-size team, you should limit your data collection to times when normal usage is low.

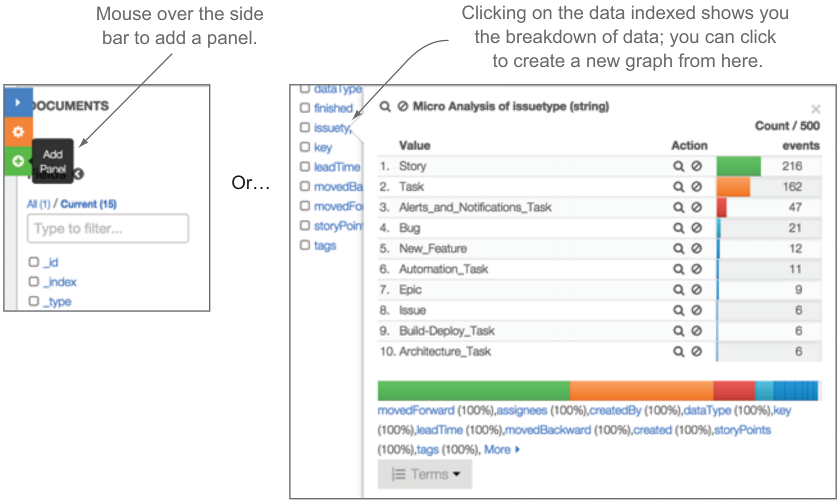
Pay attention to your source system logs

Another lesson learned from doing large-scale data collection is to be wary of the log level of your source systems. I've seen some systems run out of disk space and crash because no one was paying attention to how the system was logging.

A.2 Creating the dashboard

The front end is the open source graphing system Kibana. Once you have data in EC, Kibana is already wired up for you to start creating nice charts and graphs. Figure A.4 shows a few ways to create graphs with Kibana.

At the time this book was published Kibana 4 had just been released. Thus, many of the figures have charts from Kibana 3, though the system has been upgraded to use Kibana 4. For the latest information on how to set up a Kibana dashboard check out my blog at www.cwhd.org/, or read through the Kibana documentation on the EC website.



Mouse over the side bar to add a panel.

Or…

Clicking on the data indexed shows you the breakdown of data; you can click to create a new graph from here.

Micro Analysis of issuetype (string)

	Value	Action	Count / 500 events
1.	Story	🔍 ⊘	216
2.	Task	🔍 ⊘	162
3.	Alerts_and_Notifications_Task	🔍 ⊘	47
4.	Bug	🔍 ⊘	21
5.	New_Feature	🔍 ⊘	12
6.	Automation_Task	🔍 ⊘	11
7.	Epic	🔍 ⊘	9
8.	Issue	🔍 ⊘	6
9.	Build-Deploy_Task	🔍 ⊘	6
10.	Architecture_Task	🔍 ⊘	6

movedForward (100%),assignees (100%),createdBy (100%),dataType (100%),key (100%),leadTime (100%),movedBackward (100%),created (100%),storyPoints (100%),tags (100%), More ▸

Figure A.4 A few simple ways to create new graphs in Kibana

A.3 Summary

The ELK stack is a popular open source set of tools that you can easily use for analyzing your team. Using the code for this book, you should be able to get it running in no time. In this appendix you learned the following:

- Where to get an open source system to do the measurement we talked about in the book
- About the architecture and operation of an analytics system
- How to set up an analytics system that you can use to collect and analyze data
- How to create a dashboard interface for displaying data

appendix B
Collecting data from source systems with Grails

This appendix covers

- The architecture of the Grails component in measurementor
- Structure of the domain objects
- Using Quartz as a job scheduler

This appendix goes into the architecture and code of the Grails data collector in the measurementor project. This will be a good chapter to read if you want to fork, contribute to, extend, or customize the project to better fit into your environment.

This appendix picks up where appendix A left off. In appendix A we talked about using Elasticsearch (EC) to index your data and Kibana to generate graphs and do ad hoc analysis of your data. Combined, these platforms are very powerful and provide a lot of out-of-the-box functionality. But there are two things missing from EC and Kibana alone:

- *You need to get data to index.* You could set up Logstash for that, but it would be a lot of work and you'd need access to all of your source system's installations.
- *EC is for searching data.* If you want to calculate metrics based on combinations of data, you'll have to do that somewhere else.

To accommodate for the shortcomings there's a small Grails-based component to measurementor that reaches out to various systems via their REST-based APIs, gets the data we've been talking about in this book, and sends it to EC for indexing. This is custom code, so you can tweak it do whatever you want, including setting up additional dashboards with metrics you calculate yourself.

The measurementor project

I've open sourced the measurementor project and it's freely available on GitHub: github.com/cwhd/measurementor. Keep in mind that because it's a living project, some of the code examples from the print version of this book may be out of date, but the concepts all hold true.

Throughout this appendix we'll be using the JIRA API in our examples.

Using JIRA in the examples

This app has custom code to get data from each source system it needs to connect to. To walk through the architecture I'll use the code that gets data from JIRA, which is a very common and popular agile tracking system. The concepts and the structure of the code we'll be looking at can be used with any system that has a REST-based API; if you crack open the code, you'll see connectors to other systems. By following the examples in this appendix, you can create your own connectors if you want to get data from other sources.

B.1 Architectural overview

First, let's look at the high-level architecture outlined in appendix A to highlight the area of focus in this appendix; see figure B.1.

Note that we've added a couple of specific technologies to the diagram: Grails and MongoDB. You'll use Grails and MongoDB for their ease of use and flexibility. The data collection and conversion component in figure B.1 breaks out into figure B.2.

Overall the architecture is pretty simple. You'll use jobs to schedule the collection of data, services to parse individual data sources, and domain objects to represent each type of data you want to index. Grails has the following out-of-the-box capabilities to help you set up these components:

- Create jobs with just a few lines of code.
- Create simple objects usually called plain old Groovy objects (POGOs) that use Grails Object Relational Mapping (GORM) to handle interaction with the database.
- Create services that you can autowire into other classes with a single line of code.

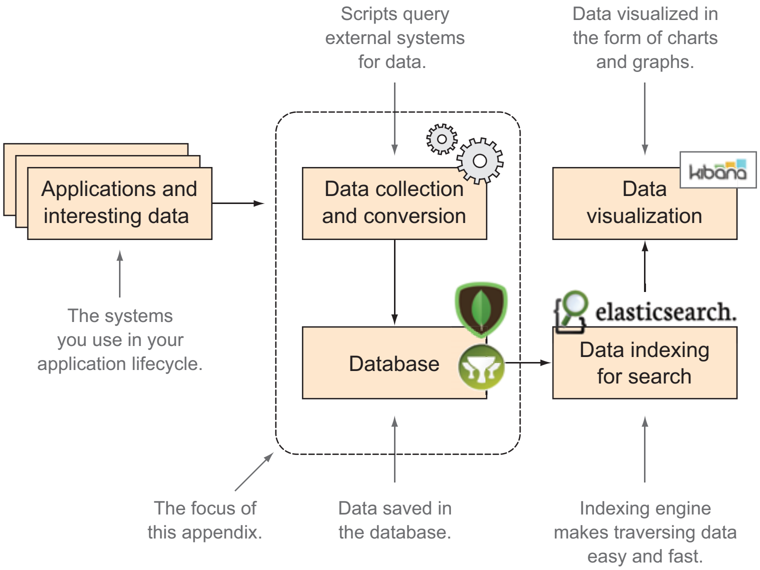


Figure B.1 The focus of this appendix, the Grails portion of measurementor

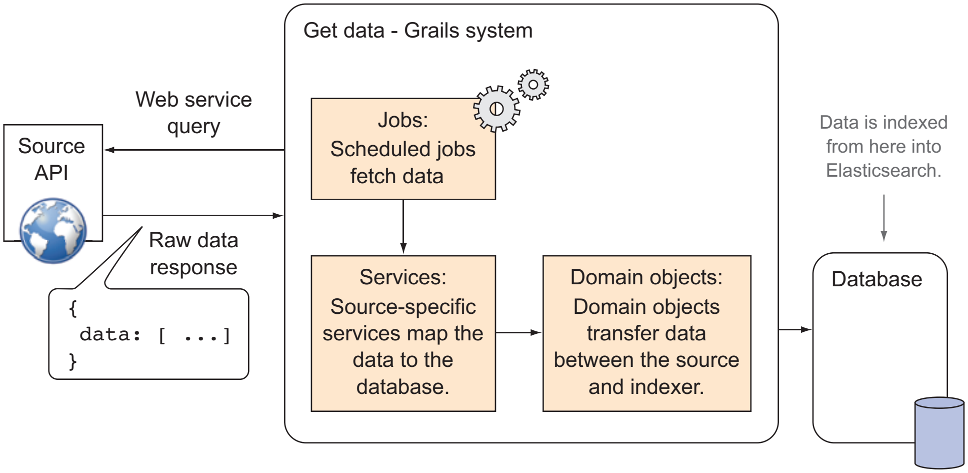


Figure B.2 A closer look into the Grails/Mongo-based data collection system

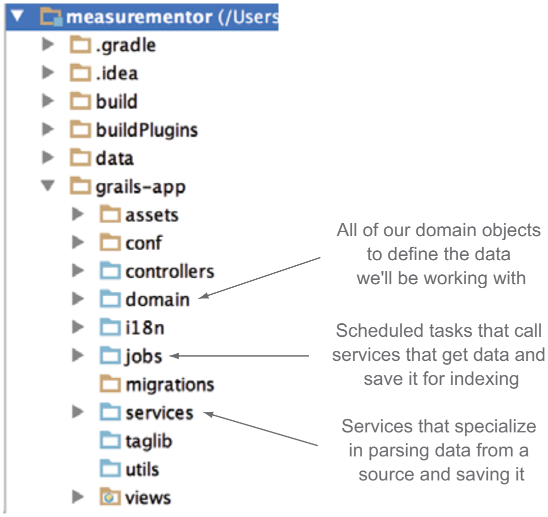
Because the Grails framework takes care of the tedious plumbing that would go into making the previous things work in other languages, you're free to focus on getting the data you want from the source systems without having to focus on only making things work.

A note on using other languages and frameworks

Over the last several years I've built a few different versions of this application. The first rendition used Python instead of Grails. I ended up switching to Grails because I prefer working in it. It runs on the JVM, which made it easy to get up and running on the infrastructure I had available at the time, and it was easier to get something running from scratch. If you hate Grails and can't imagine why I'm using it, then feel free to use the patterns outlined in this appendix to write something in your framework and language of choice. If you do, let me know; I'd love to plug your work on my blog!

If you check out the project and open it in your favorite IDE, you can navigate to all the interesting parts, as shown in figure B.3.

Feel free to poke around the rest of the app, but most of the interesting stuff is in the three sections noted in figure B.3. We'll look at each of these sections so you'll get an idea of what's going on and how to extend the app further.



All of our domain objects to define the data we'll be working with

Scheduled tasks that call services that get data and save it for indexing

Services that specialize in parsing data from a source and saving it

Figure B.3 Where to find the three main components of the app: domain objects, services, and jobs

B.1.1 Domain objects

The APIs from our source systems are so rich that indexing all the fields that come back will make searching for patterns and trends very difficult. It's much easier to get started by focusing on the fields you know you'll need.

The nature of this application is to get data from multiple systems and save it in a central place for indexing. The domain objects will be your contract between the source system and your indexer. The domain objects make it easier to transfer data because they allow you to take advantage of the database plugins and object relational mapping (ORM) provided by the Grails framework. There's a domain object for each system you're getting data from. You can look at one of our domain objects in the following listing to see how simple they are.

Listing B.1 The JIRA domain object

```
package org.cwhd.measure

class JiraData {
  static mapWith = "mongo"
```

Maps this object to MongoDB via ORM.

```
  static searchable = true

  String key
  Date created
  String createdBy
  String issuetype
  int movedForward
  int movedBackward
  int storyPoints
  String[] assignees
  String[] tags
  String dataType
  Date finished
  long leadTime
  long devTime
  int commentCount
  String jiraProject
  int estimateHealth
  long rawEstimateHealth

  static constraints = {
    finished nullable: true
    created nullable: true
    createdBy nullable: true
    issuetype nullable: true
    storyPoints nullable: true
    assignees nullable: true
    tags nullable: true
    leadTime nullable: true
    devTime nullable: true
    commentCount nullable: true
    jiraProject nullable: true
    estimateHealth nullable: true
    rawEstimateHealth nullable: true
  }
}
```

Indexes this object once it's been saved.

Properties that you'll pull out of JIRA's API.

Nullable properties are easier to manage.

As you can see, it's very simple; the domain objects define the data that you're going to move from the source systems to the indexer. The services you're using for data collection are where most of the work of the application is done, but first let's look at the data you need to parse.

B.1.2 The data you're working with

Look at the data you can get back from JIRA's API in the next listing. In earlier chapters we talked about the *who*, *what*, and *when* you can get from source systems. They're noted in figure B.2.

Listing B.2 Excerpts from the raw data in a typical API response

```
{
  "expand": "names,schema",
  "startAt": 0,
  "maxResults": 50,
  "total": 1,
  "issues": [
    {
      "expand": "editmeta,renderedFields,transitions,changelog,operations",
      "id": "58496",
      "self": "https://jira.blastamo.com/rest/api/2/issue/58496",
      "key": "MSP-3888",
      "fields": {
        "customfield_17140": "55728000",
        "created": "2012-01-19T14:50:03.000+0000",
        "project": {
          "key": "MOP",
          "name": "Multi-Operational Platform",
        }
        creator:
        {
          name: "jsmit1",
          emailAddress: "Joseph.Smith@blastamo.com",
          displayName: "Smith, Joseph",
        },
        aggregatetimeoriginalestimate: null,
        assignee: {
        name: "jsmit1",
        emailAddress: "Joseph.Smith@nike.com",
        displayName: "Smith, Joseph",
      },
      issuetype: {
        name: “Task”,
        subtask: false
      },
      status: {
      name: "Done",
      statusCategory: {
      key: "done",
      name: "Complete"
    }
  },
},
```

The when

The what

The who

To make it easier to visualize, we turned this JSON response into a group of domain objects in figure B.4.

Figure B.4 paints a rather complex picture of the domain structure that gets sent back. The Fields object in the response represents all the data in a task that has a value. Fields can be as simple as a name-value pair in most cases, but a field can also be an object that has its own set of name-value pairs. The best example is the User object used to show the creator and the current assignee of the task; each user has a name, email, display name, and other data associated with it. The ChangeLog is another aggregation of objects that represent the history of how the issue has changed over time.

To get data from the source systems, you have a service for each source system that maps the data from the source API back to your domain model. That brings us to the services you'll use to parse the data.

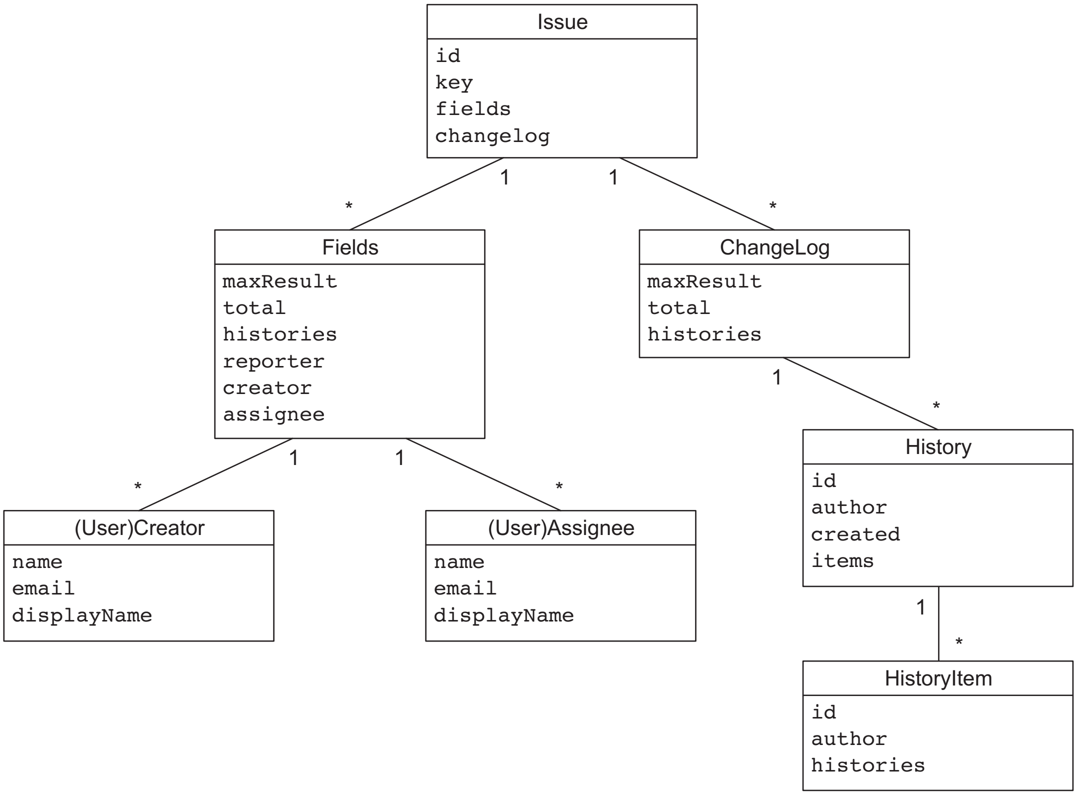


Figure B.4 Representing the response as a grouping of domain objects

B.1.3 Data collection services

The data collection services will get data from the specific data schemas in your source systems and map them to your domain objects. In some cases this is a simple mapping, but others will include functions that combine or calculate data into fields that aren't in the source system. Sticking with JIRA as an example, the next listing shows excerpts of the data collection service for JIRA.

Listing B.3 Collecting data from JIRA

```
def getData(startAt, maxResults, project, fromDate) {
  def url = grailsApplication.config.jira.url
  def path = "/rest/api/2/search"
  def jiraQuery = "project=$project$fromDate"
  def query = [jql: jiraQuery, expand:"changelog",startAt: startAt,
  ➥ maxResults: maxResults, fields:"*all"]

  def json = httpRequestService.callRestfulUrl(url, path, query, true)
```

Configures URL in application.properties.

Key parameters.

The HTTPRequest calls other services.

```
def keepGoing = false

if(json.issues.size() > 0) {
  keepGoing = true
}

//NOTE we need to set the map so we know what direction things are
➥ moving in; this relates to the moveForward & moveBackward stuff

def taskStatusMap = ["In Definition": 1, "Dev Ready":2, "Dev":3,
➥ "QA Ready":4, "QA":5, "Deploy Ready":6, "Done":7]

for(def i : json.issues) {
  def moveForward = 0
  def moveBackward = 0
  def assignees = []
  def tags = []
  def movedToDev
  def commentCount = 0
  def movedToDevList = []

  if (i.changelog) {
    for (def h : i.changelog.histories) {
      for (def t : h.items) {
        if(t.field == "status") {
          if(taskStatusMap[t.fromString] > taskStatusMap[t.toString]){
            moveBackward++
          } else {
            moveForward++
            movedToDevList.add(UtilitiesService.cleanJiraDate
            ➥ (h.created))
            }
          } else if(t.field == "assignee"){
            if(t.toString) {
              assignees.add(UtilitiesService.makeNonTokenFriendly
              ➥ (t.toString))
            }
          }
        }
      }
      movedToDev = movedToDevList.min()
    } else {
      logger.debug("changelog is null!")
    }

    commentCount = i.fields.comment?.total

    tags = i.fields.labels

    def storyPoints = 0
    if(i.fields.customfield_10013)
      storyPoints = i.fields.customfield_10013.toInteger()
    }

    def createdDate = UtilitiesService.cleanJiraDate(i.fields.created)
    def fin = UtilitiesService.cleanJiraDate(i.fields.resolutiondate)
```

Handles paged requests.

This map helps set recidivism rate.

Key variables.

See changelog for historical information.

Checks movement history to calculate recidivism.

All users assigned to this ticket.

storyPoints are a custom field.

```
      def leadTime = 0
      def devTime = 0
      if(createdDate && fin) {
        long duration = fin.getTime() - createdDate.getTime()
        leadTime = TimeUnit.MILLISECONDS.toDays(duration)
      }

      if(movedToDev && fin) {
        long duration = fin.getTime() - movedToDev.getTime()
        devTime = TimeUnit.MILLISECONDS.toDays(duration)
      }
      else if(movedToDev && !fin) {
        long duration =  new Date().getTime() - movedToDev.getTime()
        devTime = TimeUnit.MILLISECONDS.toDays(duration)
      }

      def estimateHealth = UtilitiesService.estimateHealth
      ➥ (storyPoints, devTime, 13, 9, [1, 2, 3, 5, 8, 13])

      def jiraData = JiraData.findByKey(i.key)
      if(jiraData) {
        …
      } else {
        jiraData = new JiraData(…)
    }

    jiraData.save(flush: true, failOnError: true)
  }

  if(keepGoing) {
    getData(startAt + maxResults, maxResults, project, fromDate)
  }
}
```

Time difference between created and completed.

Time difference between dev start and completed.

Estimate health calculation is in UtilitiesService.

Cut for brevity; save everything.

Recursive call for paging.

One thing you may notice is multiple calls to the UtilitiesService, which does things like clean fields to make them easier to index, convert dates from one system to another, and carry out shared complex functionality. I won't go into the specifics of the UtilitiesService in this appendix, so if you want you can check it out on GitHub.[1]

The patterns that you see in this service can be applied to a service that gets data from any other system. Coupled with a domain object that handles persistence and indexing, you have the pieces you need to get data from the source and analyze it in Kibana. The final piece is a set of jobs that update the data on a schedule.

B.1.4 Scheduling jobs for data collection

Normally I'm a big proponent of event-driven systems rather than timer-based systems. But because you're getting data from several source systems, each of which you may not have control over, it's easier for you to use timers to update and index the data.

The technology you'll use to schedule calling the data-collection services is Quartz. Quartz allows you to schedule jobs to run at any frequency you define. The Quartz

[1] If you use this link, github.com/cwhd/measurementor/blob/master/grails-app/services/org/cwhd/measure/UtilitiesService.groovy, you can get directly to the UtilitiesService class.

plugin for Grails is flexible and allows you to define intervals using Cron (en.wikipedia.org/wiki/Cron), which is a standard UNIX convention, or use simple declarations in which you define the repeat interval in milliseconds.

About Quartz

Quartz is a flexible, lightweight, fault-tolerant job scheduler written in Java that's commonly used in applications that need scheduling capabilities. The code for Quartz is open source, so you can compile it on your own if you like, but it's more commonly used in its packaged form. Quartz is very popular and has been around for a long time, so the code is hardened and works well for a variety of scenarios. For more in-depth information check out the documentation on the Quartz website (quartz-scheduler.org/documentation/).

The next listing shows an excerpt from my code that defines a Quartz job that calls one of our services. Take a look at grails-app/jobs/org/cwhd/measure/PopulatorJob.groovy[2] for the full code.

Listing B.4 Basic job to run our services

```
class DataFetchingJob {

def jiraDataService
static triggers = {
  simple name: 'jobTrig', startDelay: 60000, repeatInterval: 100000
}

def execute() {
  def result = "unknown"
  try {
    def startDateTime = new Date()
    jiraDataService.getData(0, 100, "ACOE")
    stashDataService.getAll()
    def doneDateTime = new Date()
    def difference = doneDateTime.getTime()-startDateTime.getTime()
    def minutesDiff = TimeUnit.MILLISECONDS.toMinutes(difference)
    result = "success in $difference ms"
    println "--------------------------------------------------"
    println "ALL DONE IN ~$minutesDiff minutes"
    println "--------------------------------------------------"
  } catch (Exception ex) {
    result = "FAIL: $ex.message"
  }
  JobHistory history=new JobHistory(jobDate:new Date(),jobResult:result)
  history.save(failOnError: true)
}
```

Declares the service to call from the job.

Sets up the trigger to run.

The method called when the job is run; calls the service from here.

Times the method.

Saves the job history for reference.

[2] github.com/cwhd/measurementor/blob/master/grails-app/jobs/org/cwhd/measure/PopulatorJob.groovy

This class could be only a few lines of code because we only need to define the timer and call the services that get the data. The rest of the code times how long it takes to execute, handles errors, and saves the details of the job into the `JobHistory` domain class.

The `JobHistory` class is used to remember when jobs were executed and the result. If for some reason they failed, then the next time the job runs it can try the same query. If it was successful, then we don't have to get the data a second time.

B.2 Summary

Groovy, Grails, and MongoDB are simple and fun to work with. Using the open APIs available through source systems, getting data into a single place for more complex analysis is a piece of cake (or pie). In this appendix you learned the following:

- You can take advantage of the built-in power of Grails to do the following:
  - Manage persistence
  - Request data from RESTful APIs
  - Interface with Elasticsearch
  - Set up jobs to update data
- The measurementor architecture is simple and extendable.
- If you don't like Grails, you can use these patterns to build the same application in your language of choice.
- Quartz is a flexible and lightweight technology used for scheduling jobs.

index

AGILE DEVELOPMENT

Agile Metrics IN ACTION

Christopher W. H. Davis

The iterative nature of agile development is perfect for experience-based, continuous improvement. Tracking systems, test and build tools, source control, continuous integration, and other built-in parts of a project lifecycle throw off a wealth of data you can use to improve your products, processes, and teams. The question is, how to do it?

Agile Metrics in Action teaches you how. This practical book is a rich resource for an agile team that aims to use metrics to objectively measure performance. You'll learn how to gather the data that really count, along with how to effectively analyze and act upon the results. Along the way, you'll discover techniques all team members can use for better individual accountability and team performance.

What's Inside

- Use the data you generate every day from CI and Scrum
- Improve communication, productivity, transparency, and morale
- Objectively measure performance
- Make metrics a natural byproduct of your development process

Practices in this book will work with any development process or tool stack. For code-based examples, this book uses Groovy, Grails, and MongoDB.

Christopher Davis has been a software engineer and team leader for over 15 years. He has led numerous teams to successful delivery using agile methodologies.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/AgileMetricsinAction

Free eBook
SEE INSERT

"A steadfast companion on your expedition into measurement."

—From the Foreword by Olivier Gaudin, SonarSource

"The perfect starting point for your agile journey."

—Sune Lomholt, Nordea Bank

"You'll be coming up with metrics tailored to your own needs in no time."

—Chris Heneghan, SunGard

"Comprehensive and easy to follow."

—Noreen Dertinger
Dertinger Informatics, Inc.

ISBN 13: 978-1-617292-48-4
ISBN 10: 1-617292-48-6
54499
9 781617 292484

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]