

O'REILLY®

SRE with Java Microservices

Patterns for Reliable Microservices
in the Enterprise



Jonathan Schneider
Foreword by Josh Long

SRE with Java Microservices

In a microservices architecture, the whole is indeed greater than the sum of its parts. But in practice, individual microservices can inadvertently impact others and alter the end user experience. Effective microservices architectures require standardization on an organizational level with the help of a platform engineering team.

This practical book provides a series of progressive steps that platform engineers can apply technically and organizationally to achieve highly resilient Java applications. Author Jonathan Schneider covers many effective SRE practices from companies leading the way in microservices adoption. You'll examine several patterns discovered through much trial and error in recent years, complete with Java code examples.

Chapters are organized according to specific patterns, including:

- **Application metrics:** Monitoring for availability with Micrometer
- **Debugging with observability:** Logging and distributed tracing; failure injection testing
- **Charting and alerting:** Building effective charts; KPIs for Java microservices
- **Safe multicloud delivery:** Spinnaker, deployment strategies, and automated canary analysis
- **Source code observability:** Dependency management, API utilization, and end-to-end asset inventory
- **Traffic management:** Concurrency of systems; platform, gateway, and client-side load balancing

"This book pulls concepts together from leading companies that are challenging yet easy to read. Jonathan comes with real-world experience that is different from other authors—taking theoretical concepts and providing practical experiences to help you along the way."

—Troy Gaines
Director of Information Services,
Shelter Insurance

Jonathan Schneider is CEO and cofounder of Moderne, where he works to modernize applications and infrastructure through automated source code transformation and asset visibility. Previously, he worked for the Spring team on application monitoring and engineering tools at Netflix, assembling engineering tools related to distributed refactoring, dependency insight, and build tools. He also worked on refactoring at Gradle.

PROGRAMMING

US \$59.99

CAN \$79.99

ISBN: 978-1-492-07392-5



9 781492 073925

Twitter: @oreillymedia
facebook.com/oreilly

SRE with Java Microservices

*Patterns for Reliable Microservices
in the Enterprise*

Jonathan Schneider

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

SRE with Java Microservices

by Jonathan Schneider

Copyright © 2020 Jonathan Schneider. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Melissa Duffield

Indexer: WordCo Indexing Services, Inc.

Development Editor: Melissa Potter

Interior Designer: David Futato

Production Editor: Deborah Baker

Cover Designer: Karen Montgomery

Copyeditor: JM Olejarz

Illustrator: O'Reilly Media, Inc.

Proofreader: Amanda Kersey

October 2020: First Edition

Revision History for the First Edition

2020-08-26: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492073925> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *SRE with Java Microservices*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-07392-5

[GP]

Table of Contents

Foreword.....	ix
Preface.....	xiii
1. The Application Platform.....	1
Platform Engineering Culture	2
Monitoring	7
Monitoring for Availability	7
Monitoring as a Debugging Tool	10
Learning to Expect Failure	12
Effective Monitoring Builds Trust	13
Delivery	13
Traffic Management	15
Capabilities Not Covered	15
Testing Automation	15
Chaos Engineering and Continuous Verification	17
Configuration as Code	17
Encapsulating Capabilities	18
Service Mesh	19
Summary	21
2. Application Metrics.....	23
Black Box Versus White Box Monitoring	24
Dimensional Metrics	25
Hierarchical Metrics	26
Micrometer Meter Registries	27
Creating Meters	30
Naming Metrics	31

Common Tags	36
Classes of Meters	38
Gauges	39
Counters	42
Timers	45
“Count” Means “Throughput”	46
“Count” and “Sum” Together Mean “Aggregable Average”	46
Maximum Is a Decaying Signal That Isn’t Aligned to the Push Interval	50
The Sum of Sum Over an Interval	53
The Base Unit of Time	53
Using Timers	55
Common Features of Latency Distributions	59
Percentiles/Quantiles	60
Histograms	65
Service Level Objective Boundaries	69
Distribution Summaries	73
Long Task Timers	74
Choosing the Right Meter Type	77
Controlling Cost	77
Coordinated Omission	80
Load Testing	82
Meter Filters	87
Deny/Accept Meters	88
Transforming Metrics	89
Configuring Distribution Statistics	91
Separating Platform and Application Metrics	92
Partitioning Metrics by Monitoring System	96
Meter Binders	98
Summary	99
3. Debugging with Observability.....	101
The Three Pillars of Observability...or Is It Two?	101
Logs	102
Distributed Tracing	103
Metrics	104
Which Telemetry Is Appropriate?	104
Components of a Distributed Trace	107
Types of Distributed Tracing Instrumentation	109
Manual Tracing	109
Agent Tracing	110
Framework Tracing	110
Service Mesh Tracing	111

Blended Tracing	112
Sampling	114
No Sampling	114
Rate-Limiting Samplers	114
Probabilistic Samplers	115
Boundary Sampling	116
Impact of Sampling on Anomaly Detection	116
Distributed Tracing and Monoliths	117
Correlation of Telemetry	118
Metric to Trace Correlation	119
Using Trace Context for Failure Injection and Experimentation	120
Summary	123
4. Charting and Alerting.....	125
Differences in Monitoring Systems	127
Effective Visualizations of Service Level Indicators	132
Styles for Line Width and Shading	132
Errors Versus Successes	134
“Top k” Visualizations	135
Prometheus Rate Interval Selection	137
Gauges	137
Counters	139
Timers	143
When to Stop Creating Dashboards	147
Service Level Indicators for Every Java Microservice	148
Errors	148
Latency	153
Garbage Collection Pause Times	161
Heap Utilization	164
CPU Utilization	170
File Descriptors	172
Suspicious Traffic	174
Batch Runs or Other Long-Running Tasks	175
Building Alerts Using Forecasting Methods	176
Naive Method	177
Single-Exponential Smoothing	179
Universal Scalability Law	181
Summary	185
5. Safe, Multicloud Continuous Delivery.....	187
Types of Platforms	188
Resource Types	189

Delivery Pipelines	191
Packaging for the Cloud	194
Packaging for IaaS Platforms	196
Packaging for Container Schedulers	198
The Delete + None Deployment	199
The Highlander	200
Blue/Green Deployment	200
Automated Canary Analysis	205
Spinnaker with Kayenta	209
General-Purpose Canary Metrics for Every Microservice	214
Summary	218
6. Source Code Observability.....	221
The Stateful Asset Inventory	223
Release Versioning	226
Maven Repositories	227
Build Tools for Release Versioning	230
Capturing Resolved Dependencies in Metadata	234
Capturing Method-Level Utilization of the Source Code	240
Structured Code Search with OpenRewrite	243
Dependency Management	252
Version Misalignments	252
Dynamic Version Constraints	253
Unused Dependencies	254
Undeclared Explicitly Used Dependencies	255
Summary	256
7. Traffic Management.....	257
Microservices Offer More Potential Failure Points	257
Concurrency of Systems	258
Platform Load Balancing	259
Gateway Load Balancing	259
Join the Shortest Queue	262
Instance-Reported Availability and Utilization	264
Health Checks	267
Choice of Two	269
Instance Probation	269
Knock-On Effects of Smarter Load Balancing	270
Client-Side Load Balancing	270
Hedge Requests	272
Call Resiliency Patterns	273
Retries	274

Rate Limiters	276
Bulkheads	278
Circuit Breakers	280
Adaptive Concurrency Limits	283
Choosing the Right Call Resiliency Pattern	284
Implementation in Service Mesh	285
Implementation in RSocket	287
Summary	288
Index.....	289

Foreword

“To production and beyond!”

—Buzz Lightyear (paraphrasing)

I know Buzz said “to infinity and beyond,” but that whole notion never sat well with me as a child. How could you go beyond infinity? It was only later in life, when I became a software engineer, that it dawned on me—software is never complete. It’s never finished. It’s...infinite. Buzz missed his calling in software!

Software has no end. Software is like the oceans, the stars, and the bugs in your code: endless! Hopefully, that’s not controversial. The last few decades have seen all of us in the software field pivot around this insight: the endless tail of software maintenance is the most expensive part of what we do. A good deal of the significant movements in software figure around that. Testing and continuous integration. Continuous delivery. Cloud computing. Microservices. It’s not hard to get to production the first time, but these practices optimize for the many subsequent trips to production. They optimize for day two. They optimize for cycle time: how quickly can you take an idea and see it delivered into production, from concept to customer? They optimize for “and beyond.”

This insight—that software has no end—introduces a ton of new practices and puts the lie to as many existing practices. It changes the focus from the initial development and MVP to the maintenance and management of that software. The focus is on production.

I *love* production. You should love production. You should go to production, as early and often as possible. Bring the kids. Bring the family. The weather’s amazing.

Great software engineers build for production and for the return trip. They build for the endless journey. It is no longer acceptable to wring our collective hands and throw our code over the proverbial wall: “Well, it works on my machine! Now *you* deploy it!” There’s a whole new frontier out there—production—that demands a different set of skills. Site reliability engineering (SRE) is, according to Ben Treynor,

founder of Google’s site reliability team, “what happens when a software engineer is tasked with what used to be called operations.” SREs share a lot of skills with traditional software engineers, but they target them differently, with an eye on production.

Spring Boot developers can and should develop SRE skills, and few know the ropes better than this book’s author, Jonathan Schneider. Spring Boot succeeds by being laser-focused on production. It is extraordinary because it, along with frameworks like Dropwizard, is built stem to stern for production. It supports easy aggregation of metrics with [Micrometer](#), so-called fat `.jar` deployments, the Actuator management endpoints, application life cycle events, 12 Factor-style configuration, and more. Spring Cloud is a set of Spring Boot extensions designed to support the microservices architecture. And all that is to say nothing of the rich platform support. Spring Boot is container-native. It and platforms like Cloud Foundry or Kubernetes are two sides of the same coin. Spring Boot supports graceful shutdown, health groups, liveness and readiness probes, Docker image generation (leveraging CNCF buildpacks), and so much more.

And I’ll bet you didn’t know about most of those features! But *Jonathan knows*. Jonathan lives and breathes this stuff. He created the Micrometer project, a dimensional metrics framework that supports dozens of different metrics and monitoring platforms, and then helped integrate it into Spring Boot’s Actuator module, and into countless other third-party open source projects. I’ve watched him work on metrics, continuous delivery tools like Spinnaker, and observability tools like Micrometer, and generally pave the path to production for others. He’s got years of experience in leveraging Spring and Spring Boot at a global scale, and while he can sling Spring Data repositories and craft HTTP APIs with the best of them, his genius is in the way he builds for that endless journey.

And now we can learn from him in this book.

[Chapter 1](#) is a manifesto of sorts—read this to get in the right frame of mind for the book. I should’ve read this chapter first!

I should’ve, but I didn’t. I skipped ahead to [Chapter 2](#), which introduces instrumentation and metrics. While I was eager to read the book cover to cover, I was most excited about this chapter. It’s not surprising that the creator of Micrometer could so well articulate the concepts in this chapter. Chapters [2](#) through [4](#) are a few hours very well spent. They’re brilliant. 5/5: would (and did) read again.

[Chapter 5](#) introduces the clouds, core concepts, types of platforms, and patterns unique to these platforms. This chapter was one of the most insightful for me. It starts slowly, and the next thing you know, you’re up to your deployment scripts in a game-changing discussion of continuous delivery, canary analysis, and more. Read this chapter twice.

Chapter 6 gives you a framework and specific solutions to understand your codebase and its dependencies. I've never seen all these concerns presented in a comprehensive framework like this.

Chapter 7 is a fitting closing chapter to an amazing book in that it looks at the interactions of deployed services in production. By this point you'll have learned how to get the software to production and how to observe services and their source code. This chapter is all about the service interactions and the dynamics of load on an architecture.

I learned something new in *every chapter*. The book is full of the wisdom of a true cloud native, and one that I think the community needs. Jonathan is a fantastic guide on the endless journey to production...and beyond.

— *Josh Long (@starbuxman)*
Spring Developer Advocate
Spring team, VMware
San Francisco, CA
July 2020

Preface

This book presents a phased approach to building and deploying more reliable Java microservices. The capabilities presented in each chapter are meant to be followed in order, each building upon the capabilities of earlier chapters. There are five phases in this journey:

1. Measure and monitor your services for availability.
2. Add debuggability signals that allow you to ask questions about periods of unavailability.
3. Improve your software delivery pipeline to limit the chance of introducing more failure.
4. Build the capability to observe the state of deployed assets all the way down to source code.
5. Add just enough traffic management to bring your services up to a level of availability you are satisfied with.

Our goal isn't to build a perfect system, to eliminate all failure. Our goal is to end with a highly reliable system and avoid spending time in the space of diminishing returns.

Avoiding diminishing returns is why we will spend so much time talking about effective measurement and monitoring, and why this discipline precedes all others.

If you are in engineering management, [Chapter 1](#) is your mission statement: to build an application platform renowned for its reliability and the culture of an effective platform engineering team that can deliver these capabilities to a broader engineering organization.

The chapters that follow contain the blueprints for achieving this mission, targeted at engineers. This book is intentionally narrowed in scope to *Java* microservices precisely so that I can offer detailed advice on how to go about this, including specific measurements battle-tested for Java microservices, code samples, and other

idiosyncrasies like dependency management concerns that are unique to the Java virtual machine (JVM). Its focus is on immediate actionability.

My Journey

My professional journey in software engineering forms an arc that led me to write this book:

- A scrappy custom software startup
- A traditional insurance company called Shelter Insurance in Missouri
- Netflix in Silicon Valley
- A Spring team engineer working remotely
- A Gradle engineer

When I left Shelter Insurance, despite my efforts, I didn't understand public cloud. In almost seven years there, I had interacted with the same group of named virtual machines (bare metal actually, originally). I was used to quarterly release cycles and extensive manual testing before releases. I felt like leaders emphasized and reemphasized how "hard" we expected code freezes to be leading up to releases, how after a release a code freeze wasn't as hard as we would have liked, etc. I had never experienced production monitoring of an application—that was the responsibility of a network operations center, a room my badge didn't provide access to because I didn't need to know what happened there. This organization was successful by most measures. It has changed significantly in some ways since then, and little in others. I'm thankful for the opportunity to have learned under some fantastic engineers there.

At Netflix I learned valuable lessons about engineering and culture. I left after a time with a great sense of hope that some of these same practices could be applied to a company like Shelter Insurance, and joined the Spring team. When I founded the open source metrics library Micrometer, it was with a deep appreciation of the fact that organizations are on a journey. Rather than supporting just the best-in-class monitoring systems of today, Micrometer's first five monitoring system implementations contained three legacy monitoring systems that I knew were still in significant use.

A couple of years working with and advising enterprises of various sizes on application monitoring and delivery automation with Spinnaker gave me an idea of both the diversity of organizational dynamics and their commonalities. It is my understanding of the commonalities, those practices and techniques that every enterprise could benefit from, that form the substance of this book. Every enterprise Java organization can apply these techniques, given a bit of time and practice. That includes your organization.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

O'Reilly Online Learning



For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <http://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata and any additional information. You can access this page at https://oreil.ly/SRE_with_Java_Microservices.

Email bookquestions@oreilly.com to comment or ask technical questions about this book.

For news and information about our books and courses, visit <http://oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

Olga Kundzich

What I didn't know before writing this book is how much voices from an author's circle of colleagues find their way into a book. It makes complete sense, of course. We influence each other simply by working together! Olga's insightful views on a wide range of topics have probably had the greatest single influence on my thinking in the last couple of years, and her voice is everywhere in this book (or at least the best approximation of it I can represent). Thoughts you'll find on "the application platform," continuous delivery (no no, not continuous deployment—I kept confusing the two), asset inventory, monitoring, and elements of traffic management are heavily influenced by her. Thank you Olga for investing so much energy into this book.

Troy Gaines

To Troy I owe my initial introduction to dependency management, build automation, continuous integration, unit testing, and so many other essential skills. He was an early and significant influence in my growth as a software developer, as I know he has been to many others. Thank you, old friend, for taking the time to review this work as well.

Tommy Ludwig

Tommy is one of the rare telemetry experts that contributes to both distributed tracing and aggregated metrics technologies. It is so common that contributors in the observability space are hyper-focused on one area of it, and Tommy is one of the few that floats between them. To put it mildly, I dreaded Tommy's review of Chapter 3, but was happy to find that we had more in common on this than I expected. Thanks for pointing out the more nuanced view of distributed tracing tag cardinality that made its way into Chapter 3.

Sam Snyder

I haven't known Sam for long, but it didn't take long for me to understand that Sam is an excellent mentor and patient teacher. Thank you Sam for agreeing to subject yourself to the arduous task of reviewing a technical book, and leaving so much positive and encouraging feedback.

Mike McGarr

I received an email out of the blue from Mike in 2014 that, a short time later, resulted in me packing everything up and moving to California. That email set me on a course that changed everything. I came to know so many experts at Netflix that accelerated me through the learning process because Mike took a chance on me. It radically changed the way I view software development and operations. Mike is also just a fantastic human being—a kind and inquisitive friend and leader. Thanks, Mike.

Josh Long

Once in the book, I quoted a typical Josh Long phrase about there being “no place like” production. I thought I was being cheeky and fun. And then Josh wrote a foreword that features Buzz Lightyear...Josh is an unstoppable ball of energy. Thank you Josh for injecting a bit of that energy into this work.

CHAPTER 1

The Application Platform

Martin Fowler and James Lewis, who initially proposed the term *microservices*, define the architecture in their seminal [blog post](#) as:

...a particular way of designing software applications as suites of independently deployable services. While there is no precise definition of this architectural style, there are certain common characteristics around organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.

Adopting microservices promises to accelerate software development by separating applications into independently developed and deployed components produced by independent teams. It reduces the need to coordinate and plan large-scale software releases. Each microservice is built by an independent team to meet a specific business need (for internal or external customers). Microservices are deployed in a redundant, horizontally scaled way across different cloud resources and communicate with each other over the network using different protocols.

A number of challenges arise due to this architecture that haven't been seen previously in monolithic applications. Monolithic applications used to be primarily deployed on the same server and infrequently released as a carefully choreographed event. The software release process was the main source of change and instability in the system. In microservices, communications and data transfer costs introduce additional latencies and potential to degrade end-user experience. A chain of tens or hundreds of microservices now work together to create that experience. Microservices are released independently of each other, but each one can inadvertently impact other microservices and therefore the end-user experience, too.

Managing these types of distributed systems requires new practices, tools, and engineering culture. Accelerating software releases doesn't need to come at the cost of stability and safety. In fact, these go hand in hand. This chapter introduces the culture

of an effective platform engineering team and describes the basic building blocks of reliable systems.

Platform Engineering Culture

To manage microservices, an organization needs to standardize specific communication protocols and supporting frameworks. A lot of inefficiencies arise if each team needs to maintain its own full stack development, as does friction when communicating with other parts of a distributed application. In practice, standardization leads to a platform team that is focused on providing these services to the rest of the teams, who are in turn focused on developing software to meet business needs.

We want to provide guardrails, not gates.

—Dianne Marsh, *director of engineering tools at Netflix*

Instead of building gates, allow teams to build solutions that work for them first, learn from them, and generalize to the rest of the organization.

Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.

—Conway's Law

Figure 1-1 shows an engineering organization built around specialties. One group specializes in user interface and experience design, another building backend services, another managing the database, another working on business process automation, and another managing network resources.

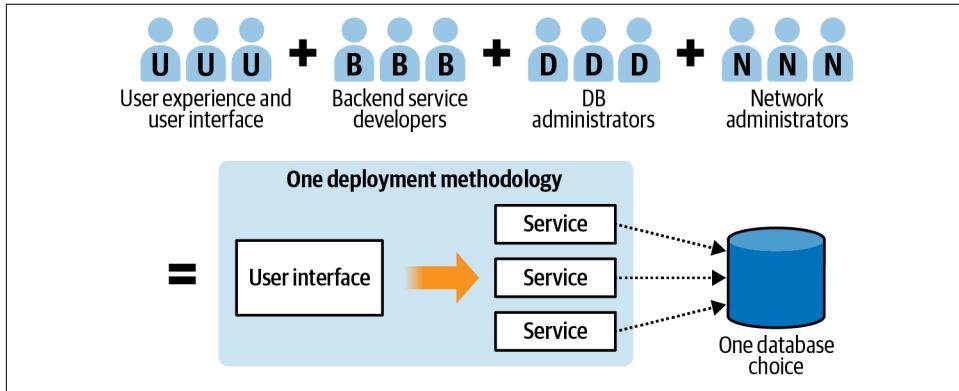


Figure 1-1. Organization built around technical silos

The lesson often taken from Conway's Law is that cross-functional teams, as in [Figure 1-2](#), can iterate faster. After all, when team structure is aligned to technical specialization, any new business requirement will require coordination across all of these specializations.

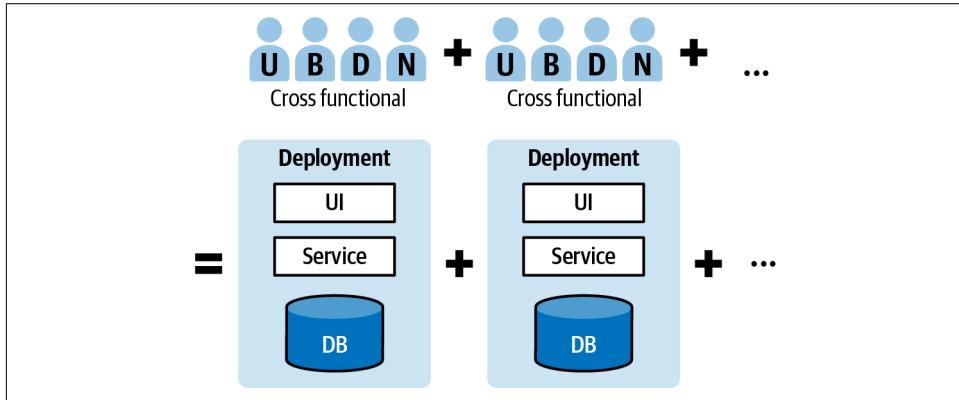


Figure 1-2. Cross-functional teams

There is obviously waste in this system as well though, specifically that specialists on each team are developing capabilities independently of one another. Netflix did not have dedicated site reliability engineers per team, as Google promotes in [*Site Reliability Engineering*](#) edited by Betsy Beyer et al. (O'Reilly). Perhaps because of a greater degree of homogeneity to the type of software being written by product teams (mostly Java, mostly stateless horizontally scaled microservices), the centralization of product engineering functions was more efficient. Does your organization more resemble Google, working on very different types of products from automated cars to search to mobile hardware to browsers? Or does it more resemble Netflix, composed of a series of business applications written in a handful of languages running on a limited variety of platforms?

Cross-functional teams and completely siloed teams are just on the opposite ends of a spectrum. Effective platform engineering can reduce the need for a specialist per team for some set of problems. An organization with dedicated platform engineering is more of a hybrid, like in [Figure 1-3](#). A central platform engineering team is strongest when it views product teams as customers that need to be constantly won over and exercises little to no control over the behavior of its customers.

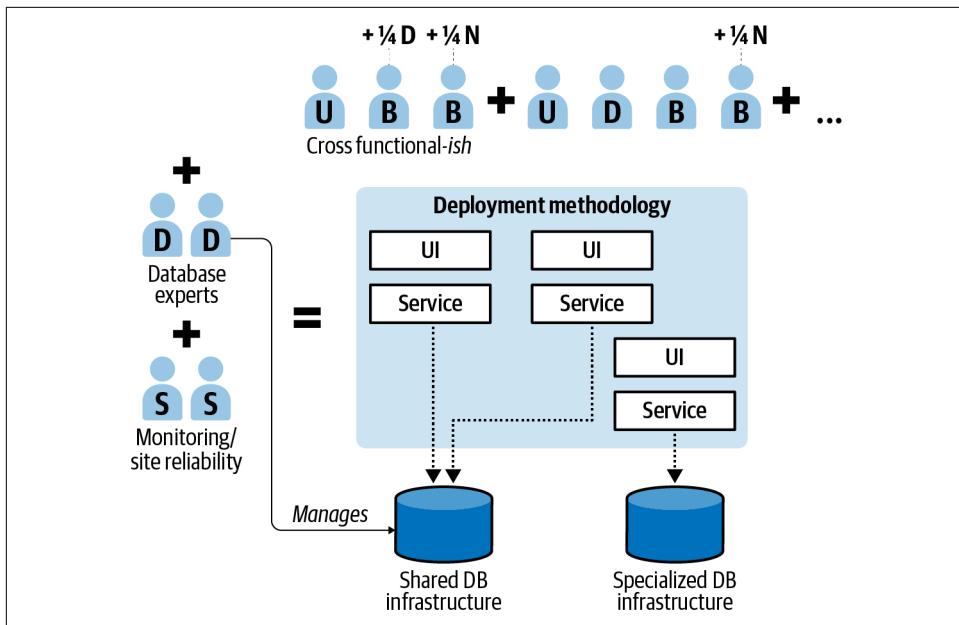


Figure 1-3. Product teams with dedicated platform engineering

For example, when monitoring instrumentation is distributed throughout the organization as a common library included in each microservice, it shares the hard-won knowledge of availability indicators known to be broadly applicable. Each product team can spend just a little time adding availability indicators that are unique to its business domain. It can communicate with the central monitoring team for information and advice on how to build effective signals as necessary.

At Netflix, the strongest cultural current was “freedom and responsibility,” defined in a somewhat famous [culture deck](#) from 2001. I was a member of the engineering tools team but we could not require that everyone else adopt a particular build tool. A small team of engineers managed Cassandra clusters on behalf of many product teams. There is an efficiency to this concentration of build tool or Cassandra skill, a natural communication hub through which undifferentiated problems with these products flowed and lessons were transferred to product-focused teams.

The build tools team at Netflix, at its smallest point, was just two engineers serving the interests of roughly 700 other engineers while transitioning between recommended build tools (Ant to Gradle) and performing two major Java upgrades (Java 6 to 7 and then Java 7 to 8), among other daily routines. Each product team completely owned its build. Because of “freedom and responsibility,” we could not set a hard date for when we would completely retire Ant-based build tooling. We could not set a hard date for when every team had to upgrade its version of Java (except to the extent that a new Oracle licensing model did this for us). The cultural imperative drove us to

focus so heavily on developer experience that product teams *wanted* to migrate with us. It required a level of effort and empathy that could only be guaranteed by absolutely preventing us from setting hard requirements.

When a platform engineer like myself serves the interests of so many diverse product teams in a focused technical speciality like build tooling, inevitably patterns emerge. My team saw the same script playing out over and over again with binary dependency problems, plug-in versioning, release workflow problems, etc. We worked initially to automate the discovery of these patterns and emit warnings in build output. Without the freedom-and-responsibility culture, perhaps we would have skipped warnings and just failed the build, requiring product teams to fix issues. This would have been satisfying to the build tools team—we wouldn't be responsible for answering questions related to failures that we tried to warn teams about. But from the product team perspective, every “lesson” the build tools team learned would be disruptive to them at random points in time, and especially disruptive when they had more pressing (if temporary) priorities.

The softer, non-failing warning approach was shockingly ineffective. Teams rarely paid any attention to successful build logs, regardless of how many warnings were emitted. And even if they did see the warnings, attempting to fix them incurred risk: a working build with warnings is better than a misbehaving one without warnings. As a result, carefully crafted deprecation warnings could go ignored for months or years.

The “guardrails not gates” approach required our build tools team to think about how we could share our knowledge with product teams in a way that was visible to them, required little time and effort to act on, and reduced the risk of coming along with us on the paved path. The tooling that emerged from this was almost over the top in its focus on developer experience.

First, we wrote tooling that could rewrite the Groovy code of Gradle builds to autoremediate common patterns. This was much more difficult than just emitting warnings in the log. It required making indentation-preserving abstract syntax tree modifications to imperative build logic, an impossible problem to solve in general, but surprisingly effective in specific cases. Autoremediation was opt-in though, through the use of a simple command that product teams could run to accept recommendations.

Next, we wrote monitoring instrumentation that reported patterns that were potentially remediable but for which product teams did not accept the recommendation. We could monitor each harmful pattern in the organization over time, watch as it declined in impact as teams accepted remediations. When we reached the long tail of a small number of teams that just wouldn't opt in, we knew who they were, so we could walk over to their desks and work with them one on one to hear their concerns and help them move forward. (I did this enough that I started carrying my own mouse around. There was a suspicious correlation between Netflix engineers who

used trackballs and Netflix engineers who were on the long tail of accepting remediations.) Ultimately, this proactive communication established a bond of trust that made future recommendations from us seem less risky.

We went to fairly extreme lengths to improve the visibility of recommendations without resorting to breaking builds to get developers' attention. Build output was carefully colorized and stylized, sometimes with visual indicators like Unicode check marks and X marks that were hard to miss. Recommendations always appeared at the end of the build because we knew that they were the last thing emitted on the terminal and our CI tools by default scrolled to the end of the log output when engineers examined build output. We taught Jenkins how to masquerade as a TTY terminal to colorize build output but ignore cursor movement escape sequences to still serialize build task progress.

Crafting this kind of experience was technically costly, but compare it with the two options:

Freedom and responsibility culture

Led us to build self-help autoremediation with monitoring that helped us understand and communicate with the teams that struggled.

Centralized control culture

We probably would have been led to break builds eagerly because we “owned” the build experience. Teams would have been distracted from their other priorities to accommodate our desire for a consistent build experience. Every change, because it lacked autoremediation, would have generated far more questions to us as the build tools team. The total amount of toil for every change would have been far greater.

An effective platform engineering team cares deeply about developer experience, a singular focus that is at least as keen as the focus product teams place on customer experience. This should be no surprise: in a well-calibrated platform engineering organization, developers *are* the customer! The presence of a healthy product management discipline, expert user experience designers, and UI engineers and designers that care deeply about their craft should all be indicators of a platform engineering team that is aligned for the benefit of their customer developers.

More detail on team structure is out of the scope of this book, but refer to *Team Topologies* by Matthew Skelton and Manuel Pais (IT Revolution Press) for a thorough treatment of the topic.

Once the team is culturally calibrated, the question becomes how to prioritize capabilities that a platform engineering team can deliver to its customer base. The remainder of this book is a call to action, delivered in capabilities ordered from (in my view) most essential to less essential.

Monitoring

Monitoring your application infrastructure requires the least organizational commitment of all the stages on the journey to more resilient systems. As we'll show in the subsequent chapters, framework-level monitoring instrumentation has matured to such an extent that you really just need to turn it on and start taking advantage. The cost-benefit ratio has been skewed so heavily toward benefit that if you do nothing else in this book, start monitoring your production applications now. [Chapter 2](#) will discuss metrics building blocks, and [Chapter 4](#) will provide the specific charts and alerts you can employ, mostly based on instrumentation that Java frameworks provide without you having to do any additional work.

Metrics, logs, and distributed tracing are three forms of observability that enable the measure of service availability and aid in debugging complex distributed systems problems. Before going further in the workings of any of these, it is useful to understand what capabilities each enables.

Monitoring for Availability

Availability signals measure the overall state of the system and whether that system is functioning as intended in the large. It is quantified by *service level indicators* (SLIs). These indicators include signals for the health of the system (e.g., resource consumption) and business metrics like number of sandwiches sold or streaming video starts per second. SLIs are tested against a threshold called a *service level objective* (SLO) that sets an upper or lower bound on the range of an SLI. SLOs in turn are a somewhat more restrictive or conservative estimate than a threshold you agree upon with your business partners about a level of service you are expected to provide, or what's known as a *service level agreement* (SLA). The idea is that an SLO should provide some amount of advance warning of an impending violation of an SLA so that you don't actually get to the point where you violate that SLA.

Metrics are the primary observability tool for measuring availability. They are a measure of SLIs. Metrics are the most common availability signal because they represent an aggregation of all activity happening in the system. They are cheap enough to not require sampling (discarding some portion of the data to limit overhead), which risks discarding important indicators of unavailability.

Metrics are numerical values arranged in a time series representing a sample at a particular time or an aggregate of individual events that have occurred in an interval:

Metrics

Metrics *should* have a fixed cost irrespective of throughput. For example, a metric that counts executions of a particular block of code should only ship the number of executions seen in an interval regardless of how many there are. By this I mean that a metric should ship “N requests were observed” at publish time, not “I saw a request N distinct times” throughout the publishing interval.

Metrics data

Metrics data cannot be used to reason about the performance or function of any individual request. Metrics telemetry trades off reasoning about an individual request for the application’s behavior across all requests in an interval.

To effectively monitor the availability of a Java microservice, a variety of availability signals need to be monitored. Common signals are given in [Chapter 4](#), but in general they fall into four categories, known together as the L-USE method:¹

Latency

This is a measure of how much time was spent executing a block of code. For the common REST-based microservice, REST endpoint latency is a useful measure of the availability of the application, particularly max latency. This will be discussed in greater detail in [“Latency” on page 153](#).

Utilization

A measure of how much of a finite resource is consumed. Processor utilization is a common utilization indicator. See [“CPU Utilization” on page 170](#).

Saturation

Saturation is a measurement of extra work that can’t be serviced. [“Garbage Collection Pause Times” on page 161](#) shows how to measure the Java heap, which during times of excessive memory pressure leads to a buildup of work that cannot be completed. It’s also common to monitor pools like database connection pools, request pools, etc.

Errors

In addition to looking at purely performance-related concerns, it is essential to find a way to quantify the error ratio relative to total throughput. Measurements of error include unanticipated exceptions yielding unsuccessful HTTP responses on a service endpoint (see [“Errors” on page 148](#)), but also more indirect measures like the ratio of requests attempted against an open circuit breaker (see [“Circuit Breakers” on page 280](#)).

¹ I first learned of the USE criteria from Brendan Gregg’s description of his [method](#) for monitoring Unix systems. In that context, latency measurement isn’t as granular, thus the missing *L*.

Utilization and saturation may seem similar at first, and internalizing the difference will have an impact on how you think about charting and alerting on resources that can be measured both ways. A great example is JVM memory. You can measure JVM memory as a utilization metric by reporting on the amount of bytes consumed in each memory space. You can also measure JVM memory in terms of the proportion of time spent garbage collecting it relative to doing anything else, which is a measure of saturation. In most cases, when both utilization and saturation measurements are possible, the saturation metric leads to better-defined alert thresholds. It's hard to alert when memory utilization exceeds 95% of a space (because garbage collection will bring that utilization rate back below this threshold), but if memory utilization routinely and frequently exceeds 95%, the garbage collector will kick in more frequently, more time will be spent proportionally doing garbage collection than anything else, and the saturation measurement will thus be higher.

Some common availability signals are listed in [Table 1-1](#).

Table 1-1. Examples of availability signals

SLI	SLO	L-USE criteria
Process CPU usage	<80%	Saturation
Heap utilization	<80% of available heap space	Saturation
Error ratio for a REST endpoint	<1% of total requests to the endpoint	Errors
Max latency for a REST endpoint	<100 ms	Latency

Google has a much more prescriptive view on how to use SLOs.

Google's approach to SLOs

[*Site Reliability Engineering*](#) by Betsy Beyer et al. (O'Reilly) presents service availability as a tension between competing organizational imperatives: to deliver new features and to run the existing feature set reliably. It proposes that product teams and dedicated site reliability engineers agree on an error budget that provides a measurable objective for how unreliable a service is allowed to be within a given window of time. Exceeding this objective should refocus the team on reliability over feature development until the objective is met.

The Google view on SLOs is explained in great detail in the “Alerting on SLOs” chapter in [*The Site Reliability Workbook*](#) edited by Betsy Beyer et al. (O'Reilly). Basically, Google engineers alert on the probability that an error budget is going to be depleted in any given time frame, and they react in an organizational way by shifting engineering resources from feature development to reliability as necessary. The word “error” in this case means exceeding any SLO. This might mean exceeding an acceptable ratio of server failed outcomes in a RESTful microservice, but could also mean exceeding an acceptable latency threshold, getting too close to overwhelming file descriptors on

the underlying operating system, or any other combination of measurements. With this definition, the time that a service is unreliable in a prescribed window is the proportion when one or more SLOs were not being met.

Your organization doesn't need to have separate functions for product engineer and site reliability engineer for error budgeting to be a useful concept. Even a single engineer working on a product completely alone and wholly responsible for its operation can benefit from thinking about where to pause feature development in favor of improving reliability and vice versa.

I think the overhead of the Google error budget scheme is overkill for a lot of organizations. Start measuring, discover how alerting functions fit into your unique organization, and once practiced at measuring, consider whether you want to go all in on Google's process or not.

Collecting, visualizing, and alerting on application metrics is an exercise in continuously testing the availability of your services. Sometimes an alert itself will contain enough contextual data that you know how to fix a problem. In other cases, you'll want to isolate a failing instance in production (e.g., by moving it out of the load balancer) and apply further debugging techniques to discover the problem. Other forms of telemetry are used for this purpose.

A less formal approach to SLOs

A less formal system worked well for Netflix, where individual engineering teams were responsible for their services' availability, there was no SRE/product engineer separation of responsibility on individual product teams, and there wasn't such a formalized reaction to error budgets, at least not cross-organizationally. Neither system is right or wrong; find a system that works well for you.

For the purpose of this book, we'll talk about how to measure for availability in simpler terms: as tests against an error rate or error ratio, latencies, saturation, and utilization indicators. We won't present violations of these tests as particular "errors" of reliability that are deducted from an error budget over a window of time. If you want to then take those measurements and apply the error-budgeting and organizational dynamics of Google's SRE culture to your organization, you can do that by following the guidance given in Google's writings on the topic.

Monitoring as a Debugging Tool

Logs and distributed traces, covered in detail in [Chapter 3](#), are used mainly for troubleshooting, once you have become aware of a period of unavailability. Profiling tools are also debuggability signals.

It is very common (and easy, given a confusing market) for organizations to center their entire performance management investment around debuggability tools.

Application performance management (APM) vendors can sometimes sell themselves as an all-in-one solution, but with a core technology built entirely on tracing or logging and providing availability signals by aggregating these debugging signals.

In order to not single out any particular vendor, consider [YourKit](#), a valuable profiling (debuggability) tool that does this task well without selling itself as more. YourKit excels at highlighting computation- and memory-intensive hotspots in Java code, and looks like [Figure 1-4](#). Some popular commercial APM solutions have a similar focus, which, while useful, is not a substitute for a focused availability signal.

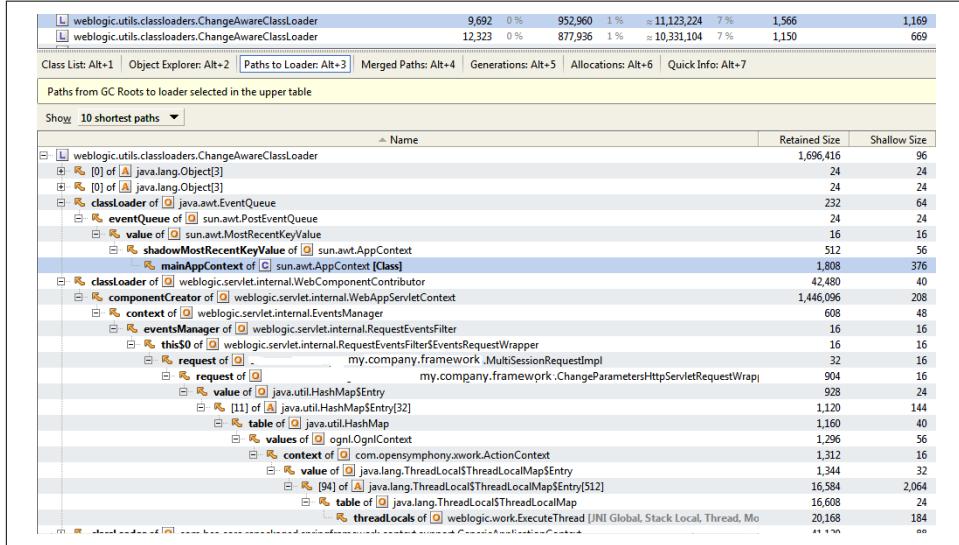


Figure 1-4. YourKit excels at profiling

These solutions are more granular, recording in different ways the specifics of what occurred during a particular interaction with the system. With this increased granularity comes cost, and this cost is frequently mitigated with downsampling or even turning off these signals entirely until they are needed.

Attempts to measure availability from log or tracing signals generally force you to trade off accuracy for cost, and neither can be optimized. This trade-off exists for traces because they are generally sampled. The storage footprint for traces is higher than for metrics.

Learning to Expect Failure

If you aren't already monitoring applications in a user-facing way, as soon as you start, you're likely to be confronted with the sobering reality of your software as it exists today. Your impulse is going to be to look away. Reality is likely to be ugly.

At a midsize property-casualty insurance company, we added monitoring to the main business application that the company's insurance agents use to conduct their normal business. Despite strict release processes and a reasonably healthy testing culture, the application manifested over 5 failures per minute for roughly 1,000 requests per minute. From one perspective, this is only a 0.5% error ratio (maybe acceptable and maybe not), but the failure rate was still a shock to a company that thought its service was well tested.

The realization that the system is not going to be perfect switches the focus from trying to be perfect to monitoring, alerting, and quickly resolving issues that the system experiences. No amount of process control around the rate of change will yield perfect outcomes.

Before evolving the delivery and release process further, the first step on the path to resilient software is adding monitoring to your software as it is released now.

With the move to microservices and changing application practices and infrastructure, monitoring has become even more important. Many components are not directly under an organization's control. For example, latency and errors can be caused by failures in the networking layer, infrastructure, and third-party components and services. Each team producing a microservice has the potential to negatively impact other parts of the system not under its direct control.

End users of software also do not expect perfection, but do want their service provider to be able to effectively resolve issues. This is what is known as the *service recovery paradox*, when a user of the service will trust a service more after a failure than they did before the failure.

Businesses need to understand and capture the user experience they want to provide to the end users—what type of system behavior will cause issues to the business and what type of behavior is acceptable to users. *Site Reliability Engineering* and *The Site Reliability Workbook* have more on how to pick these for your business.

Once identified and measured, you can adopt Google style, as seen in “[Google’s approach to SLOs](#)” on page 9, or Netflix’s more informal “context and guardrails” style, or anywhere in between to help you reason about your software or the next steps. See the first chapter on Netflix in *Seeking SRE* by David N. Blank-Edelman (O’Reilly) to learn more about context and guardrails. Whether you follow the Google practice or a simpler one is up to your organization, the type of software you develop, and the engineering culture you want to promote.

With the goal of never failing replaced with the goal of being able to meet SLAs, engineering can start building multiple layers of resiliency into systems, minimizing the effects of failures on end-user experience.

Effective Monitoring Builds Trust

In certain enterprises, engineering can still be seen as a service organization rather than a core business competency. At the insurance company with a five-failures-per-minute error rate, this is the prevailing attitude. In many cases where the engineering organization served the company's insurance agents in the field, the primary interaction between them happened through reporting and tracking software issues through a call center.

Engineering routinely prioritized bug resolution, based on defects learned from the call center, against new feature requests and did a little of both for each software release. I wondered how many times the field agents simply didn't report issues, either because a growing bug backlog suggested that it wasn't an effective use of their time or because the issue had a good-enough workaround. The problem with becoming aware of issues primarily through the call center is that it made the relationship entirely one way. Business partners report and engineering responds (eventually).

A user-centric monitoring culture makes this relationship more two-way. An alert may provide enough contextual information to recognize that rating for a particular class of vehicle is failing for agents in some region today. Engineering has the opportunity to reach out to the agents proactively with enough contextual information to explain to the agent that the issue is already known.

Delivery

Improving the software delivery pipeline lessens the chance that you introduce more failure into an existing system (or at least helps you recognize and roll back such changes quickly). It turns out that good monitoring is a nonobvious prerequisite to evolving safe and effective delivery practices.

The division between continuous integration (CI) and continuous delivery (CD) tends to be blurred by the fact that teams frequently script deployment automation and run these scripts as part of continuous integration builds. It is easy to repurpose a CI system as a flexible general-purpose workflow automation tool. To make a clear conceptual delineation between the two, regardless of where the automation runs, we'll say that continuous integration ends at the publication of a microservice artifact to an artifact repository, and delivery begins at that point. In [Figure 1-5](#), the software delivery life cycle is drawn as a sequence of events from code commit to deployment.

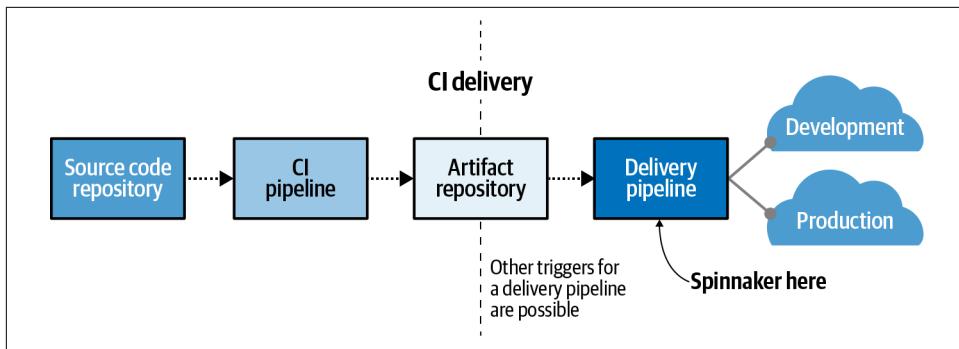


Figure 1-5. The boundary between continuous integration and delivery

The individual steps are subject to different frequencies and organizational needs for control measures. They also have fundamentally different goals. The goal of continuous integration is to accelerate developer feedback, fail fast through automated testing, and encourage eager merging to prevent **promiscuous integration**. The goal of delivery automation is to accelerate the release cycle, ensure security and compliance measures are met, provide safe and scalable deployment practices, and contribute to an understanding of the deployed landscape for the monitoring of deployed assets.

The best delivery platforms also act as an inventory of currently deployed assets, further magnifying the effect of good monitoring: they help turn monitoring into action. In [Chapter 6](#), we'll talk about how you can build an end-to-end asset inventory, ending with a deployed asset inventory, that allows you to reason about the smallest details of your code all the way up to your deployed assets (i.e., containers, virtual machines, and functions).



Continuous Delivery Doesn't Necessarily Mean Continuous Deployment

Truly continuous deployment (every commit passing automated checks goes all the way to production automatically) may or may not be a goal for your organization. All things being equal, a tighter feedback loop is preferable to a longer feedback loop, but it comes with technical, operational, and cultural costs. Any delivery topics discussed in this book apply to continuous delivery in general, as well as continuous deployment in particular.

Once effective monitoring is in place and less failure is being introduced into the system by further changes to the code, we can focus on adding more reliability to the running system by evolving traffic management practices.

Traffic Management

So much of a distributed system's resiliency is based on the expectation of and compensation for failure. Availability monitoring reveals these actual points of failure, debuggability monitoring helps understand them, and delivery automation helps prevent you from introducing too many more of them in any incremental release. Traffic management patterns will help live instances cope with the ever-present reality of failure.

In [Chapter 7](#), we'll introduce particular mitigation strategies involving load balancing (platform, gateway, and client-side) and call resilience patterns (retrying, rate limiters, bulkheads, and circuit breakers) that provide a safety net for running systems.

This is covered last because it requires the highest degree of manual coding effort on a per-project basis, and because the investment you make in doing the work can be guided by what you learn from the earlier steps.

Capabilities Not Covered

Certain capabilities that are common focuses of platform engineering teams are not included in this book. I'd like to call out a couple of them, testing and configuration management, and explain why.

Testing Automation

My view on testing is that testing automation available in open source takes you a certain way. Any investment beyond that is likely to suffer from diminishing returns. Following are some problems that are well solved already:

- Unit testing
- Mocking/stubbing
- Basic integration testing, including test containers
- Contract testing
- Build tooling that helps separate computationally expensive and inexpensive test suites

There are a couple other problems that I think are worth avoiding unless you really have a lot of resources (both computationally and in engineering time) to expend. Contract testing is an example of a technique that covers some of what both of these test, but in a far cheaper way:

- Downstream testing (i.e., whenever a commit happens to a library, build all other projects that depend on this library both directly or indirectly to determine whether the change will cause failure downstream)
- End-to-end integration testing of whole suites of microservices

I'm very much for automated tests of various sorts and very suspicious of the whole enterprise. At times, feeling the social pressure of testing enthusiasts around me, I may have gone along with the testing fad of the day for a little while: 100% test coverage, behavior-driven development, efforts to involve nonengineer business partners in test specification, Spock, etc. Some of the cleverest engineering work in the open source Java ecosystem has taken place in this space: consider Spock's creative use of bytecode manipulation to achieve data tables and the like.

Traditionally, working with monolithic applications, software releases were viewed as the primary source of change in the system and therefore potential for failure. Emphasis was placed on making sure the software release process didn't fail. Much effort was expended to ensure that lower-level environments mirrored production to verify that pending software releases were stable. Once deployed and stable, the system was assumed to remain stable.

Realistically, this has never been the case. Engineering teams adopt and double down on automated testing practices as a cure for failure, only to have failure stubbornly persist. Management is skeptical of testing in the first place. When tests fail to capture problems, what little faith they had is gone. Production environments have a stubborn habit of diverging from test environments in subtle and seemingly always catastrophic ways. At this point, if you forced me to choose between having close to 100% test coverage and an evolved production monitoring system, I'd eagerly choose the monitoring system. This isn't because I think less of tests, but because even in reasonably well-defined traditional businesses whose practices don't change quickly, 100% test coverage is mythical. The production environment will simply behave differently. As Josh Long likes to say: "There is no place like it."

Effective monitoring warns us when a system isn't working correctly due to conditions we can anticipate (i.e., hardware failures or downstream service unavailability). It also continually adds to our knowledge of the system, which can actually lead to tests covering cases we didn't previously imagine.

Layers of testing practice can limit the occurrence of failure, but will never eliminate it, even in industries with the tightest quality control practices. Actively measuring outcomes in production lowers time to discovery and ultimately remediation of failures. Testing and monitoring together are then complementary practices reducing how much failure end users experience. At their best, testing prevents whole classes of regressions, and monitoring quickly identifies those that inevitably remain.

Our automated test suites prove (to the extent they don't contain logical errors themselves) what we know about the system. Production monitoring shows us what happens. An acceptance that automated tests won't cover everything should be a tremendous relief.

Because application code will always contain flaws stemming from unanticipated interactions, environmental factors like resource constraints, and imperfect tests, effective monitoring might be considered even more of a requirement than testing for any production application. A test proves what we think will happen. Monitoring shows what *is* happening.

Chaos Engineering and Continuous Verification

There is a whole discipline around continuously verifying that your software behaves as you expect by introducing controlled failures (chaos experiments) and verifying. Because distributed systems are complex, we cannot anticipate all of their myriad interactions, and this form of testing helps surface unexpected emergent properties of complex systems.

The overall discipline of chaos engineering is broad, and as it is covered in detail in *Chaos Engineering* by Casey Rosenthal and Nora Jones (O'Reilly), I won't go into it in this book.

Configuration as Code

The 12-Factor App teaches that configuration ought to be **separated from code**. The basic form of this concept, configuration stored as an environment variable or fetched at startup from a centralized configuration server like Spring Cloud Config Server, I think is straightforward enough to not require any explanation here.

The more complicated case involving *dynamic* configuration—whereby changes to a central configuration source propagates to running instances, influencing their behavior—is in practice exceedingly dangerous and must be handled with care. Pairing with the open source Netflix *Archaius* configuration client (which is present in Spring Cloud Netflix dependencies and elsewhere) was a proprietary Archaius server which served this purpose. Unintended consequences resulting from dynamic configuration propagation to running instances caused a number of production incidents of such magnitude that the delivery engineers wrote a whole canary analysis process around scoping and incrementally rolling out dynamic configuration changes, using the lessons they had learned from automated canary analysis for different versions of code. This is beyond the scope of this book, since many organizations will never receive substantial enough benefit from automated canary analysis of code changes to make that effort worthwhile.

Declarative delivery is an entirely different form of configuration as code, popularized again by the rise of Kubernetes and its YAML manifests. My early career left me with a permanent suspicion of the completeness of declarative-only solutions. I worked on a policy administration system for an insurance company that consisted of a backend API returning XML responses and a frontend of XSLT transformations of these API responses into static HTML/JavaScript to be rendered in the browser.

It was a bizarre sort of templating scheme. Its proponents argued that the XSLT lent the rendering of each page a declarative nature. And yet, it turns out that XSLT itself is Turing complete with a convincing [existence proof](#). The typical point in favor of declarative definition is simplicity leading to an amenability to automation like static analysis and remediation. But as in the XSLT case, these technologies have a seemingly unavoidable way of evolving toward Turing completeness. The same forces are in play with JSON ([Jsonnet](#)) and Kubernetes ([Kustomize](#)). These technologies are undoubtedly useful, but I can't be another voice in the chorus calling for purely declarative configuration. Short of making that point, I don't think there is much this book can add.

Encapsulating Capabilities

As under fire as object-oriented programming (OOP) may be today, one of its fundamental concepts is *encapsulation*. In OOP, encapsulation is about bundling state and behavior within some unit, e.g., a class in Java. A key idea is to hide the state of an object from the outside, called *information hiding*. In some ways, the task of the platform engineering team is to perform a similar encapsulation task for resiliency best practices for its customer developer teams, hiding information not out of control, but to unburden them from the responsibility of dealing with it. Maybe the highest praise a central team can receive from a product engineer is “I don’t have to care about what you do.”

The subsequent chapters are going to introduce a series of best practices as I understand them. The challenge to you as a platform engineer is to deliver them to your organization in a minimally intrusive way, to build “guardrails not gates.” As you read, think about how you can encapsulate hard-won knowledge that’s applicable to every business application and how you can deliver it to your organization.

If the plan involves getting approval from a sufficiently powerful executive and sending an email to the whole organization requiring adoption by a certain date, it’s a gate. You still want buy-in from your leadership, but you need to deliver common functionality in a way that feels more like a guardrail:

Explicit runtime dependencies

If you have a core library that every microservice includes as a runtime dependency, this is almost certainly your delivery mechanism. Turn on key metrics, add common telemetry tagging, configure tracing, add traffic management patterns, etc. If you have heavy Spring usage, use autoconfiguration classes. You can similarly conditionalize configuration with CDI if you are using Java EE.

Service clients as dependencies

For traffic management patterns especially (fallbacks, retry logic, etc.), consider making it the responsibility of the team producing the service to also produce a service *client* that interacts with the service. After all, the team producing and operating it has more knowledge than anybody about where its weaknesses and potential failure points are. Those engineers are likely the best ones to formalize this knowledge in a client dependency such that each consumer of their service uses it in the most reliable way.

Injecting a runtime dependency

If the deployment process is relatively standardized, you have an opportunity to *inject* runtime dependencies in the deployed environment. This was the approach employed by the Cloud Foundry buildpack team to inject a platform metrics implementation into Spring Boot applications running on Cloud Foundry. You can do something similar.

Before encapsulating too eagerly, find a handful of teams and practice this discipline explicitly in code in a handful of applications. Generalize what you learn.

Service Mesh

As a last resort, encapsulate common platform functionality in sidecar processes (or containers) alongside the application, which when paired with a control plane managing them is called a *service mesh*.

The service mesh is an infrastructure layer outside of application code that manages interaction between microservices. One of the most recognizable implementations today is [Istio](#). These sidecars perform functions like traffic management, service discovery, and monitoring on behalf of the application process so that the application does not need to be aware of these concerns. At its best, this simplifies application development, trading off increased complexity and cost in deploying and running the service.

Over a long enough time horizon, trends in software engineering are often cyclic. In the case of site reliability, the pendulum swings from increased application and developer responsibility (e.g., Netflix OSS, DevOps) to centralized operations team responsibility. The rise of interest in service mesh represents a shift back to centralized operations team responsibility.

Istio promotes the concept of managing and propagating policy across a suite of microservices from its centralized control plane, at the behest of an organizationally centralized team that specializes in understanding the ramifications of these policies.

The venerable Netflix OSS suite (the important pieces of which have alternative incarnations like Resilience4j for traffic management, HashiCorp Consul for discovery, Micrometer for metrics instrumentation, etc.) made these application concerns. Largely, though, the application code impact was just the addition of one or more binary dependencies, at which point some form of autoconfiguration took over and decorated otherwise untouched application logic. The obvious downside of this approach is language support, with support for each site reliability pattern requiring library implementations in every language/framework that the organization uses.

Figure 1-6 shows an optimistic view of the effect on this engineering cycle on derived value. With any luck, at each transition from decentralization to centralization and back, we learn from and fully encapsulate the benefits of the prior cycle. For example, Istio could conceivably fully encapsulate the benefits of the Netflix OSS stack, only for the next decentralization push to unlock potential that was unrealizable in Istio's implementation. This is already underway in Resilience4j, for example, with discussion about adaptive forms of patterns like bulkheads that are responsive to application-specific indicators.

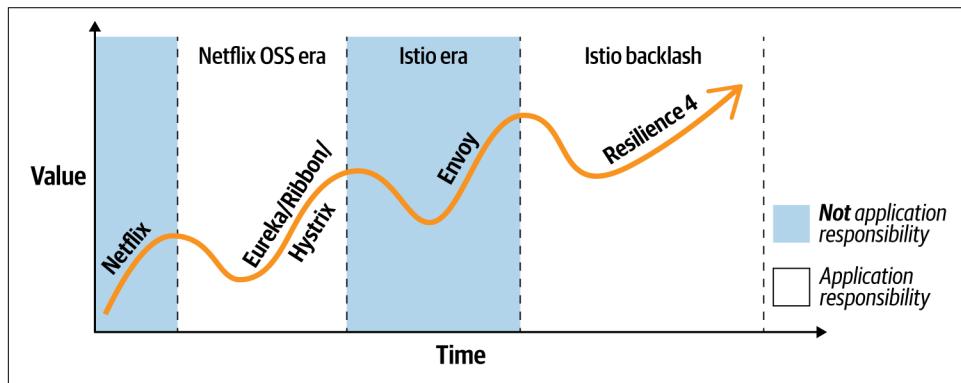


Figure 1-6. The cyclic nature of software engineering, applied to traffic management

Sizing of sidecars is also tricky, given this lack of domain-specific knowledge. How does a sidecar know that an application process is going to consume 10,000 requests per second, or only 1? Zooming out, how do we size the sidecar control plane up front not knowing how many sidecars will eventually exist?



Sidecars Are Limited to Lowest-Common-Denominator Knowledge

A sidecar proxy will always be weakest where domain-specific knowledge of the application is the key to the next step in resiliency. By definition, being separate from the application, sidecars cannot encode any knowledge this domain specific to the application without requiring coordination between the application and sidecar. That is likely at least as hard as implementing the sidecar-provided functionality in a language-specific library includable by the application.

I believe testing automation available in open source takes you a certain way. Any investment beyond that is likely to suffer from diminishing returns, as discussed in “[Service Mesh Tracing](#)” on page 111, and against using sidecars for traffic management, as in “[Implementation in Service Mesh](#)” on page 285, unpopular as these opinions might be. These implementations are lossy compared to what you can achieve via a binary dependency either explicitly included or injected into the runtime, both of which add a far greater degree of functionality that only becomes cost-prohibitive if you have a significant number of distinct languages to support (and even then, I’m not convinced).

Summary

In this chapter we defined platform engineering as at least a placeholder phrase for the functions of reliability engineering that we will discuss through the remainder of this book. The platform engineering team is most effective when it has a customer-oriented focus (where the customer is other developers in the organization) rather than one of control. Test tools, the adoption path for those tools, and any processes you develop against the “guardrails not gates” rule.

Ultimately, designing your platform is in part designing your organization. What do you want to be known for?

CHAPTER 2

Application Metrics

The complexity of distributed systems comprised of many communicating microservices means it is especially important to be able to observe the state of the system. The rate of change is high, including new code releases, independent scaling events with changing load, changes to infrastructure (cloud provider changes), and dynamic configuration changes propagating through the system. In this chapter, we will focus on how to measure and alert on the performance of the distributed system and some industry best practices to adopt.

An organization must commit at a minimum to one or more monitoring solutions. There are a wide range of choices including open source, commercial on-premises, and SaaS offerings with a broad spectrum of capabilities. The market is mature enough that an organization of any size and complexity can find a solution that fits its requirements.

The choice of monitoring system is important to preserve the fixed-cost characteristic of metrics data. The StatsD protocol, for example, requires an emission to a StatsD agent from an application on a per-event basis. Even if this agent is running as a sidecar process on the same host, the application still suffers the allocation cost of creating the payload on a per-event basis, so this protocol breaks at least this advantage of metrics telemetry. This isn't always (or even commonly) catastrophic, but be aware of this cost.

Black Box Versus White Box Monitoring

Approaches to metrics collection can be categorized according to what the method is able to observe:

Black box

The collector can observe inputs and outputs (e.g., HTTP requests into a system and responses out of it), but the mechanism of the operation is not known to the collector. Black box collectors somehow intercept or wrap the observed process to measure it.

White box

The collector can observe inputs and outputs and also the internal mechanisms of the operation. White box collectors do this in application code.

Many monitoring system vendors provide agents that can be attached to application processes and that provide black box monitoring. Sometimes these agent collectors reach so deep into well-known application frameworks that they start to resemble white box collectors in some ways. Still, black box monitoring in whatever form is limited to what the writer of the agent can generalize about all applications that might apply the agent. For example, an agent might be able to intercept and time Spring Boot's mechanism for database transactions. An agent will never be able to reason that a `java.util.Map` field in some class represents a form of near-cache and instrument it as such.

Service-mesh-based instrumentation is also black box and is generally less capable than an agent. While agents can observe and decorate individual method invocations, a service mesh's finest-grained observation is at the RPC level.

On the other side, white box collection sounds like a lot of work. Some useful metrics are truly generalizable across applications (e.g., HTTP request timings, CPU utilization) and are well instrumented by black box approaches. A white box instrumentation library with some of these generalizations encapsulated when paired with an application autoconfiguration mechanism resembles a black box approach. White box instrumentation autoconfigured requires the same level of developer effort as black box instrumentation: specifically *none!*

Good white box metrics collectors should capture everything that a black box collector does but also support capturing more internal details that black box collectors by definition cannot. The difference between the two for your engineering practices are minimal. For a black box agent, you must alter your delivery practice to package and configure the agent (or couple yourself to a runtime platform integration that does this for you). For autoconfigured white box metrics collection that captures the same set of detail, you must include a binary dependency at build time.

Vendor-specific instrumentation libraries don't tend to have this black box feel with a white box approach because framework and library authors aren't inclined to add a wide range of proprietary instrumentation clients even as optional dependencies and instrument their code N different times. A vendor-neutral instrumentation facade like Micrometer has the advantage of the "write once, publish anywhere" experience for framework and library authors.

Black box and white box collectors can of course be complementary, even when there is some overlap between them. There is no across-the-boards requirement to choose one over the other.

Dimensional Metrics

Most modern monitoring systems employ a dimensional naming scheme that consists of a metric name and a series of key-value tags.

While the storage mechanism varies substantially from one monitoring system to another, in general every unique combination of name and tags is represented as a distinct entry or row in storage. The total cost in storage terms of a metric then is the product of the cardinality of its tag set (meaning the total number of unique key-value tag pairs).

For example, an application-wide counter metric named `http.server.requests` that contains a tag for an HTTP method of which only GET and POST are ever observed, an HTTP status code where the service returns one of three status codes, and a URI of which there are two in the application results in up to $2 * 3 * 2 = 12$ distinct time series sent to and stored in the monitoring system. This metric could be represented in storage roughly like in [Table 2-1](#). Coordination between tags, like the fact that only endpoint `/a1` will ever have a GET method and only `/a2` will ever have a POST method can limit the total number of unique time series below the theoretical maximum, to only six rows in this example. In many dimensional time series databases, for each row representing a unique set of name and tags, there will be a value ring buffer that holds the samples for this metric over a defined period of time. When the system contains a bounded ring buffer like this, the total cost of your metrics is fixed to the product of the number of permutations of unique metric names/tags and the size of the ring buffer.

Table 2-1. The storage of a dimensional metric

Metric name and tags	Values
http.server.requests{method=GET,status=200,uri=/a1}	[10,11,10,10]
http.server.requests{method=GET,status=400,uri=/a1}	[1,0,0,0]
http.server.requests{method=GET,status=500,uri=/a1}	[0,0,0,4]
http.server.requests{method=POST,status=200,uri=/a2}	[10,11,10,10]
http.server.requests{method=POST,status=400,uri=/a2}	[0,0,0,1]
http.server.requests{method=POST,status=500,uri=/a2}	[1,1,1,1]

In some cases, metrics are periodically moved to long-term storage. At this point, there is an opportunity to squash or drop tags to reduce storage cost at the expense of some dimensional granularity.

Hierarchical Metrics

Before dimensional metrics systems became popular, many monitoring systems employed a hierarchical scheme. In these systems, metrics were defined only by name, with no key-value tag pairs. Tags are so useful that a convention emerged to append tag-like data to metric names with something like dot separators. So a dimensional metric like `httpServerRequests`, which has a `method` tag of `GET` in a dimensional system, might be represented as `httpServerRequests.method.GET` in a hierarchical system. Out of this arose query features like wildcard operators to allow simple aggregation across “tags,” as in [Table 2-2](#).

Table 2-2. Aggregation of hierarchical metrics with wildcards

Metric query	Value
httpServerRequests.method.GET	10
httpServerRequests.method.POST	20
httpServerRequests.method.*	30

Still, tags are not a first-class citizen in hierarchical systems, and wildcarding like this breaks down. In particular, when an organization decides that a metric like `httpServerRequests` that is common to many applications across the stack should receive a new tag, it has the potential to break existing queries. In [Table 2-3](#), the true number of requests independent of method is 40, but since some application in the stack has introduced a new status tag in the metric name, it is no longer included in the aggregation. Even assuming we can agree as a whole organization to standardize on this new tag, our wildcarding queries (and therefore any dashboards or alerts built off of them) misrepresent the state of the system from the time the tag is introduced in the

first application until it is fully propagated through the codebase and redeployed everywhere.

Table 2-3. Failures of aggregation of hierarchical metrics with wildcards

Metric query	Value
httpServerRequests.method.GET	10
httpServerRequests.method.POST	20
httpServerRequests.status.200.method.GET	10
httpServerRequests.method.*	30 (!!)

Effectively, the hierarchical approach has forced an ordering of tags when they are really independent key-value pairs.

If you are starting with real-time application monitoring now, you should be using a dimensional monitoring system. This means you will also have to use a dimensional metrics instrumentation library in order to record metrics in a way that fully takes advantage of the name/tag combination that makes these systems so powerful. If you already have some instrumentation using a hierarchical collector, the most popular being Dropwizard Metrics, you are going to have to ultimately rewrite this instrumentation. It's possible to flatten dimensional metrics into hierarchical metrics by developing a naming convention that in some way iterates over all the tags and combines them with the metric name. Going the other direction is difficult to generalize, because the lack of consistency in naming schemes makes it difficult to split a hierarchical name into dimensional metrics.

From this point on, we'll be examining dimensional metrics instrumentation alone.

Micrometer Meter Registries

The remainder of this chapter will use [Micrometer](#), a dimensional metrics instrumentation library for Java that supports many of the most popular monitoring systems on the market. There are only two main alternatives to Micrometer available:

Monitoring system vendors often provide Java API clients

While these work for white box instrumentation at the application level, there is little to no chance that the remainder of the Java ecosystem, especially of third-party open source libraries, will adopt a particular vendor's instrumentation client for its metrics collection. Probably the closest we have come to this is some spotty adoption in open source libraries of the Prometheus client.

OpenTelemetry

OpenTelemetry is a hybrid metrics and tracing library. At the time of this writing, OpenTelemetry does not have a 1.0 release, and its focus has certainly been more on tracing than metrics, so metrics support is much more basic.

While there is some variation in capabilities from one dimensional metrics instrumentation library to another, most of the key concepts described apply to each of them, or at least you should develop an idea of how alternatives should be expected to mature.

In Micrometer, a `Meter` is the interface for collecting a set of measurements (which we individually call metrics) about your application.

Meters are created from and held in a `MeterRegistry`. Each supported monitoring system has an implementation of `MeterRegistry`. How a registry is created varies for each implementation.

Each `MeterRegistry` implementation that is supported by the Micrometer project has a library published to Maven Central and JCenter (e.g., `io.micrometer:micrometer-registry-prometheus`, `io.micrometer:micrometer-registry-atlas`):

```
MeterRegistry registry = new PrometheusMeterRegistry(PrometheusConfig.DEFAULT);
```

`MeterRegistry` implementations with more options contain a fluent builder as well, for example the InfluxDB registry shown in [Example 2-1](#).

Example 2-1. Influx fluent builder

```
MeterRegistry registry = InfluxMeterRegistry.builder(InfluxConfig.DEFAULT)
    .httpClient(myCustomizedHttpClient)
    .build();
```

Metrics can be published to multiple monitoring systems simultaneously with `CompositeMeterRegistry`.

In [Example 2-2](#), a composite registry is created that ships metrics to both Prometheus and Atlas. Meters should be created with the composite.

Example 2-2. Composite meter registry that ships to Prometheus and Atlas

```
MeterRegistry prometheusMeterRegistry = new PrometheusMeterRegistry(
    PrometheusConfig.DEFAULT);
MeterRegistry atlasMeterRegistry = new AtlasMeterRegistry(AtlasConfig.DEFAULT);

MeterRegistry registry = new CompositeMeterRegistry();
registry.add(prometheusMeterRegistry);
registry.add(atlasMeterRegistry);
```

```
// Create meters like counters against the composite,
// not the individual registries that make up the composite
registry.counter("my.counter");
```

Micrometer packs with a global static `CompositeMeterRegistry` that can be used in a similar way that we use an SLF4J `LoggerFactory`. The purpose of this static registry is to allow for instrumentation in components that cannot leak Micrometer as an API dependency by offering a way to dependency-inject a `MeterRegistry`. [Example 2-3](#) shows the similarity between the use of the global static registry and what we are used to from logging libraries like SLF4J.

Example 2-3. Using the static global registry

```
class MyComponent {
    Timer timer = Timer.builder("time.something")
        .description("time some operation")
        .register(Metrics.globalRegistry);

    Logger logger = LoggerFactory.getLogger(MyComponent.class);

    public void something() {
        timer.record(() -> {
            // Do something
            logger.info("I did something");
        });
    }
}
```

By adding any `MeterRegistry` implementations that you wire in your application to the global static registry, any low-level libraries using the global registry like this wind up registering metrics to your implementations. Composite registries can be added to other composite registries. In [Figure 2-1](#), we've created a composite registry in our application that publishes metrics to both Prometheus and Stackdriver (i.e., we've called `CompositeMeterRegistry#add(MeterRegistry)` for both the Prometheus and Stackdriver registries). Then we've added *that* composite to the global static composite. The composite registry you created can be dependency-injected by something like Spring, CDI, or Guice throughout your application for your components to register metrics against. But other libraries are often outside of this dependency-injection context, and since they don't want Micrometer to leak through their API signatures, they register with the static global registry. In the end, metrics registration flows down this hierarchy of registries. So library metrics flow down from the global composite to your application composite to the individual registries. Application metrics flow down from the application composite to the individual Prometheus and Stackdriver registries.

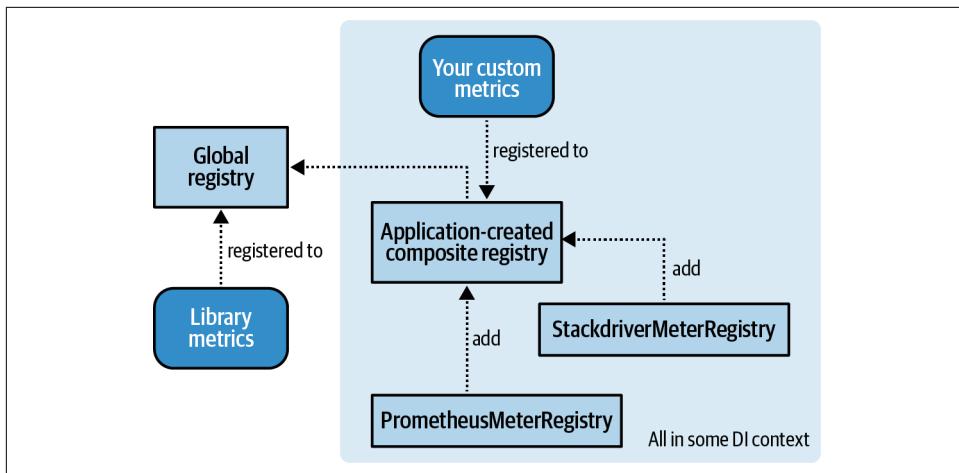


Figure 2-1. Relationship between global static registry and your application's registries



Spring Boot Autoconfiguration of MeterRegistry

Spring Boot autoconfigures a composite registry and adds a registry for each supported implementation that it finds on the classpath. A dependency on `micrometer-registry-{system}` in your runtime classpath along with any required configuration for that system causes Spring Boot to configure the registry. Spring Boot also adds any `MeterRegistry` found as a `@Bean` to the global static composite. In this way, any libraries that you add to your application that provide Micrometer instrumentation automatically ship their metrics to your monitoring system! This is how the black-box-like experience is achieved through white box instrumentation. As the developer, you don't need to explicitly register these metrics; just their presence in your application makes it work.

Creating Meters

Micrometer provides two styles to register metrics for each supported `Meter` type, depending on how many options you need. The fluent builder, as shown in [Example 2-4](#), provides the most options. Generally, core libraries should use the fluent builder because the extra verbosity required to provide robust description and base unit detail adds value to all of their users. In instrumentation for a particular microservice with a small set of engineers, opting for more compact code and less detail is fine. Some monitoring systems support attaching description text and base units to metrics, and for those, Micrometer will publish this data. Furthermore, some monitoring systems will use base unit information on a metric to automatically scale and label the *y*-axis of charts in a way that is human readable. So if you publish a

metric with a base unit of “bytes,” a sophisticated monitoring system will recognize this and scale the *y*-axis to megabytes or gigabytes, or whatever is the most human-readable value for the range of this metric. It’s much easier to read “2 GB” than “2147483648 bytes.” Even for those monitoring systems that don’t fundamentally support base units, charting user interfaces like [Grafana](#) allow you to manually specify the units of a chart, and Grafana will do this kind of intelligent human-readable scaling for you.

Example 2-4. Meter fluent builder

```
Counter counter = Counter.builder("requests") // Name
    .tag("status", "200")
    .tags("method", "GET", "outcome", "SUCCESS") // Multiple tags
    .description("http requests")
    .baseUnit("requests")
    .register(registry);
```

`MeterRegistry` contains convenience methods to construct `Meter` instances with a shorter form, as in [Example 2-5](#).

Example 2-5. Meter construction convenience methods

```
Counter counter = registry.counter("requests",
    "status", "200", "method", "GET", "outcome", "SUCCESS");
```

Regardless of which of the two methods you use to construct a meter, you will have to decide on its name and which tags to apply.

Naming Metrics

To get the most out of metrics, they need to be structured in such a way that selecting just the name and aggregating over all tags yields a meaningful (if not always useful) value. For example, if a metric is named `http.server.requests`, then tags may identify application, region (in the public cloud sense), API endpoint, HTTP method, response status code, etc. An aggregate measure like throughput for this metric of all unique combinations of tags yields a measure of throughput for every interaction with many applications across your application stack. The ability to pivot on this name into various tags makes this useful. We could explode this metric dimensionally by region and observe a regional outage or drill down on a particular application—for example, for successful responses to a particular API endpoint to reason about throughput through that one key endpoint.

Assuming many applications are instrumented with some metric such as `http.server.requests`, when building a visualization on `http.server.requests` the monitoring system will display an aggregate of the performance of all of the

`http.server.requests` across all applications, regions, etc. until you decide to dimensionally drill down on something.

Not everything should be a tag, however. Suppose we are trying to measure the number of HTTP requests and the number of database calls separately.

Micrometer employs a naming convention that separates lowercase words with a . (dot) character. The naming shown in [Example 2-6](#) provides enough context so that if just the name is selected the value is at least potentially meaningful. For example, if we select `database.queries` we can see the total number of calls to all databases. Then we can group by or select by database to drill down further or perform comparative analysis on the contribution of calls to each database.

Example 2-6. Recommended approach

```
registry.counter("database.queries", "db", "users")
registry.counter("http.requests", "uri", "/api/users")
```

With the approach shown in [Example 2-7](#), if we select the metric `calls` we will get a value that is an aggregate of the number of calls to the database and HTTP requests. This value is not useful without dimensionally drilling down further.

Example 2-7. Bad practice: using a type tag where the meter name should be different instead

```
registry.counter("calls",
  "type", "database",
  "db", "users");

registry.counter("calls",
  "type", "http",
  "uri", "/api/users");
```

[Figure 2-2](#) shows the effect of this bad naming. Suppose for every HTTP request you make 10 database calls. If you just chart `calls`, you get the top-line rate of approximately 11,000 calls. But 11,000 is an awkward sum of two types of calls which are always an order of magnitude off in frequency. To get any utility out of this, we need to break down by type dimensionally, at which point we discover the 10x relationship of database calls to HTTP requests. Having to drill down right away to build an intelligible chart is a sign that something isn't right about the metric naming scheme.

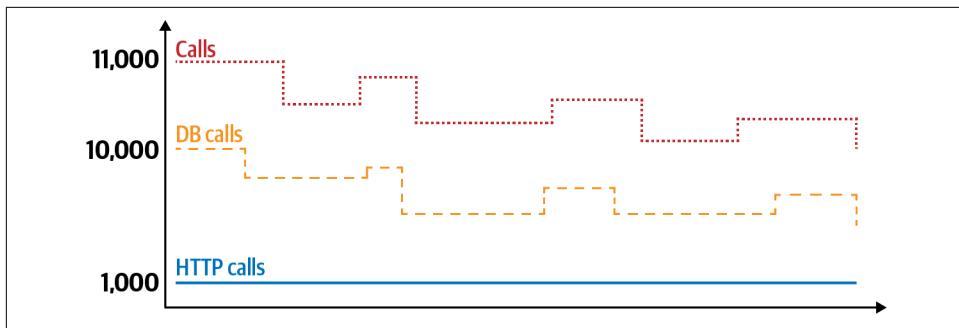


Figure 2-2. The effect of bad naming on chart usability

It is a good practice to group related data together, such as by prefixing metric names with namespaces like “jvm” or “db.” For example, a suite of metrics related to JVM garbage collection could be prefixed with `jvm.gc`:

```
jvm.gc.live.data.size
jvm.gc.memory.promoted
jvm.gc.memory.allocated
```

This namespacing not only helps group related metrics alphabetically in many monitoring system UIs and dashboarding utilities, but also can be used to affect a group of metrics in one pass with a `MeterFilter`. For example, to disable all `jvm.gc` metrics, we can apply a `deny MeterFilter` on this name:

```
MeterRegistry registry = ...;
registry.config().meterFilter(MeterFilter.denyNameStartsWith("jvm.gc"));
```

Different monitoring systems have different recommendations regarding naming convention, and some naming conventions may be incompatible for one system and not another. Recall that Micrometer employs a naming convention that separates lowercase words with a . (dot) character. Each Micrometer implementation for a monitoring system comes with a naming convention that transforms lowercase dot notation names into the monitoring system’s recommended naming convention.

Additionally, this naming convention sanitizes metric names and tags of special characters that are disallowed by the monitoring system. The convention turns out to be more than just a play at appearing more idiomatic. If shipped in this form without any naming convention normalization two metrics, `http.server.requests` and `http.client.requests`, would break Elasticsearch indexing, which treats dots as a form of hierarchy for the purpose of indexing. If these metrics were *not* shipped with dots to SignalFx, we wouldn’t be able to take advantage of UI presentation in SignalFx that hierarchizes metrics with dot separators in them. These two differing opinions on the dot character are mutually exclusive. With naming convention normalization, these metrics are shipped as `httpServerRequests` and `httpClientRequests` to Elasticsearch and with the dot notation to SignalFx. So the application code maintains

maximum portability without having to change instrumentation. When using Micrometer, meter names and tag keys should follow these guidelines:

- Always use dots to separate parts of the name.
- Avoid adding unit names or words like `total` to meter names.

So choose `jvm.gc.memory.promoted` instead of `jvmGcMemoryPromoted` or `jvm_gc_memory_promoted`. If you prefer one of the latter (or if your monitoring system requires it), configure the naming convention on the registry to do this conversion. But using dot separators in metric names up and down the whole software stack yields consistent outcomes for a variety of monitoring systems.

For some monitoring systems, the presence of unit names and the like are part of the idiomatic naming scheme. Again, naming conventions can add these bits where appropriate. For example, Prometheus's naming convention adds `_total` to the suffix of counters and `_seconds` to the end of timers. Also, the base unit of time varies based on the monitoring system. With a Micrometer `Timer`, you record in whatever granularity you'd like and the time values are scaled at publishing time. Even if you always record in a certain granularity, including the unit name in the meter name is inaccurate. For example, [Example 2-8](#) shows up as `requests_millis_seconds` in Prometheus, which is awkward.

Example 2-8. Bad practice: adding a unit to the meter name

```
registry.timer("requests.millis")
    .record(responseTime, TimeUnit.MILLISECONDS);
```

The default naming convention of a `MeterRegistry` can be overridden with a custom one, which can build upon some basic building blocks provided in the `NamingConvention` interface, as shown in [Example 2-9](#).

Example 2-9. A custom naming convention that adds base units as a suffix

```
registry.config()
    .namingConvention(new NamingConvention() { ❶
        @Override
        public String name(String name, Meter.Type type, String baseUnit) {
            String camelCased = NamingConvention.snakeCase.name(name, type, baseUnit);
            return baseUnit == null ? camelCased :
                camelCased + "_" + baseUnit;
        }
    });
}
```

- ❶ `NamingConvention` is a functional interface, so this can be simplified to a lambda, but for the sake of clarity here we leave the anonymous class.

As mentioned in “[Dimensional Metrics](#)” on page 25, the total storage cost of a metric is the product of the cardinality of the value set of each of its tags. Choose tag names that help identify the failure mode of a piece of software. For example, if monitoring an auto insurance rating application, tagging policy metrics with vehicle class is more useful than tagging with a unique vehicle identification number. As a result of a bug or downstream service outage, a class of vehicles like classic trucks may start failing. Responding to an alert on a policy rating error ratio that exceeds a predetermined threshold, an engineer may quickly identify that the handling of classic trucks is problematic based on the top three rating failures by vehicle class.



Limit Total Unique Tag Values to Control Storage Cost

Beware of the potential for tag values coming from user-supplied sources to blow up the cardinality of a metric. You should always carefully normalize and bound user-supplied input. Sometimes the cause is sneaky. Consider the URI tag for recording HTTP requests on service endpoints. If we don’t constrain 404s to a value like “NOT_FOUND,” the dimensionality of the metric would grow with each resource that can’t be found. Even more tricky, an application that redirects all nonauthenticated requests to a login endpoint could return a 403 for a resource that ultimately will not be found once authenticated, and so a reasonable URI tag for a 403 might be “REDIRECTION.” Allowing a tag value set to grow without bound can result in an overrun of storage in your monitoring system, increasing cost and potentially destabilizing a core part of your observability stack.

In general, avoid recording tags on unique values like user ID unless it is known that the population is small.

Tag values must be nonnull and ideally nonblank. Even though Micrometer technically supports blank tag values in limited situations, like for the Datadog registry implementation, blank tag values are not portable to other monitoring systems that don’t support them.



Limit Total Unique Tag Values to Control Query Cost

In addition to increases to storage costs with an increasing number of unique tag values, query costs (in both time and resources) also increase as more and more time series need to be aggregated in query results.

Common Tags

When lower-level libraries provide common instrumentation (Micrometer provides meter binders out of the box—see “[Meter Binders](#)” on page 98), they cannot know the context of the application this instrumentation will be collected in. Is the app running in a private datacenter on one of a small set of named VMs whose names never change? On infrastructure-as-a-service public cloud resources? In Kubernetes? In virtually every case, there is more than one running copy of a particular application that might ship metrics, even if there is just one copy in production and one in a lower-level testing environment. It’s useful if we can partition in some way the metrics streaming out of these various instances of an application by some dimensions that allow us to attribute a behavior back to a particular instance.

Meter filters (covered in more detail in “[Meter Filters](#)” on page 87) allow you to add common tags to accomplish exactly this, enriching every metric published from an application with additional tags. Pick common tags that help turn your metrics data into action. Following are common tags that are always useful:

Application name

Consider that some metrics, like HTTP request metrics instrumented by the framework, will have the same name across various applications, e.g., `http.server.requests` for HTTP endpoints served by the application, and `http.client.requests` for outbound requests to other services. By tagging with application name, you could then, for example, reason about all outbound requests to some particular service endpoint across multiple callers.

Cluster and server group name

In “[Delivery Pipelines](#)” on page 191 we talk more about the formal definition of a cluster and server group. If you have such a topology, tagging with cluster and server group is always useful. Some organizations don’t have this level of complexity, and that’s OK too.

Instance name

This may be the machine’s host name in some situations, but not always (and this helps explain why Micrometer doesn’t preemptively tag with things like host name, because it really does depend on the deployed environment). In public cloud environments, host name may not be the right tag. AWS EC2, for example, has local and external host names that are distinct from the instance ID. Instance ID is actually the easiest of these three to locate a particular instance with in the AWS console, and it does uniquely identify the instance. So in this context, instance ID is a better tag than host name. In Kubernetes, the pod ID is the right instance-level tag.

Stack

“Stack” in this context means development versus production. You may have multiple levels of nonproduction environments. Shelter Insurance at one point had “devl,” “test,” “func,” and “stage” nonproduction environments, each of which served its own purpose (and I might be forgetting one or two). It’s nice to practice monitoring even on unstable lower-level environments to baseline your expectations about the performance and number of errors produced by a piece of code as it progresses along its promotion path on the route to production.

Some other ideas for tags for different deployed environments are included in [Table 2-4](#).

Table 2-4. Common tags by cloud provider

Provider	Common tags
AWS	Instance ID, ASG name, region, zone, AMI ID, account
Kubernetes	Pod ID, namespace, cluster name, Deployment/StatefulSet name, ReplicaSet name
Cloud Foundry	CF app name (which may be distinct from the application name), organization name, space name, instance ordinal, foundation name

This table illustrates why Micrometer doesn’t add these tags by default. The singular concept of “namespace” has three different names across these cloud providers: region, namespace, and organization/space. Where AWS and Kubernetes have a single-value namespace concept, CloudFoundry has two: organization and space!

The application of common tags is a great place to begin thinking as a platform engineering organization about how to encapsulate and standardize for your organization.

[Example 2-10](#) shows how in Spring Boot, you can apply common tags via property-based configuration.

Example 2-10. Adding common tags by property in Spring Boot

```
management.metrics.tags:  
  application: ${spring.application.name}  
  region: us-east-1  
  stack: prod
```

Alternatively, you could apply tags in an autoconfigurable @Configuration class, as in [Example 2-11](#).

Example 2-11. Adding common tags programmatically in Spring Boot

```
@Configuration
public class MetricsConfiguration {
    @Bean
    MeterFilter commonTags(@Value("${spring.application.name}") String appName) {
        return MeterFilter.commonTags(
            "application", appName,
            "region", "us-east-1", ①
            "stack", "prod"
        )
    }
}
```

- ① This should be sourced from the environment as well.

Assuming you have some central dynamic configuration server, like Spring Cloud Config Server, that applications interrogate at startup for properties, the property-based configuration allows you to deliver these common tag opinions across your application stack immediately and with no code changes or dependency requirements for each application.

The programmatic form can be delivered via an explicit runtime binary dependency from each app or by injecting a dependency into the deployed form of the app or a container running it, like Tomcat.

Classes of Meters

Many metrics collectors provide several classes of meters, each of which may emit one or more metrics or statistics. The most common are gauges, counters, and timers/summaries. More specialized meters include long task timers, time gauges, and interval counters. Strive to use the most specialized meter available for the task. Timers, for example, always emit a count to measure throughput. There is little advantage to counting executions of a particular block of code, when timing it would have generated the same count statistic but also richer information about the latency distribution of that block of code.

A decision guide for which built-in meter type to choose follows the introduction of each type in “[Choosing the Right Meter Type](#)” on page 77.

Gauges

Gauges are a measure of an instantaneous value that may increase and decrease over time. A time series plot of a gauge is a collection of samples of instantaneous values at intervals where metrics were published from the application. Because they are sampled instantaneous values, it is possible and even likely that the value would have been higher or lower if it had been sampled at a different point in time.

The speedometer and fuel level on a vehicle are classic examples of gauges. As you drive along the road, you periodically glance at the speedometer (hopefully). Seeing a periodic instantaneous measurement of your speed is enough to keep speed under control, but it is still true that you miss the variations in speed that occurred between looks.

In applications, typical examples for gauges would be the size of a collection or map or the number of threads in a running state. Memory and CPU measurements are also taken using gauges. [Figure 2-3](#) is a gauge time series of a single metric, `jvm.mem.ory.used`, that is tagged with several dimensions, including the memory space. This stack chart shows one way in which making a single concept like memory consumption dimensional provides richness to its representation in a chart.

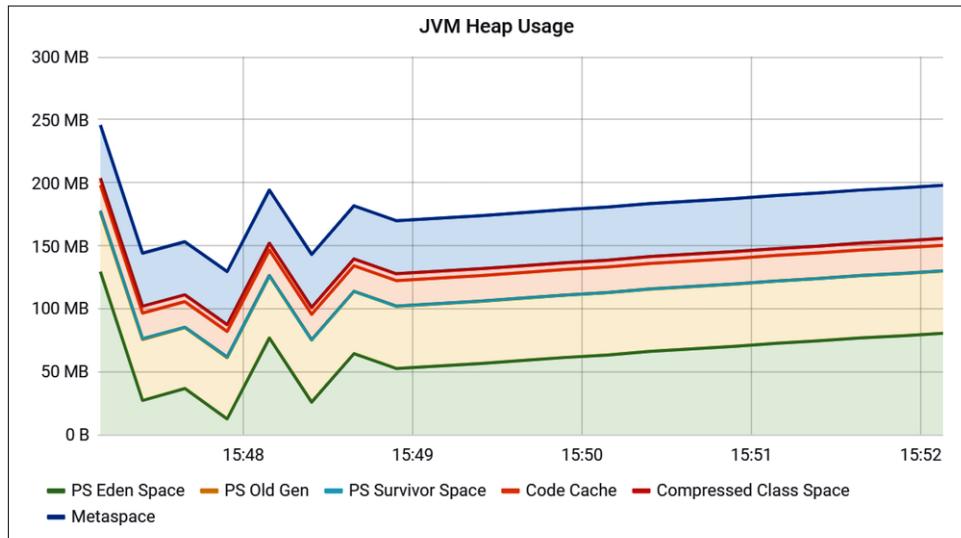


Figure 2-3. Heap used by space

Micrometer takes the stance that gauges should be sampled and not be set, so there is no information about what might have occurred between samples. After all, any intermediate values set on a gauge are lost by the time the gauge value is reported to a

metrics backend anyway, so there seems to be little value in setting those intermediate values in the first place.

Think of a gauge as a meter that only changes when it is observed. Every other class of meter accumulates intermediate counts toward the point where the data is sent to the metrics backend.

The `MeterRegistry` interface contains methods, some of which are shown in [Example 2-12](#), for building gauges to observe numeric values, functions, collections, and maps.

Example 2-12. Creating gauges

```
List<String> list = registry.gauge(
    "listGauge", Collections.emptyList(),
    new ArrayList<>(), List::size);

List<String> list2 = registry.gaugeCollectionSize(
    "listSize2", Tags.empty(),
    new ArrayList<>());

Map<String, Integer> map = registry.gaugeMapSize(
    "mapGauge", Tags.empty(), new HashMap<>());
```

In the first case, a slightly more common form of gauge is one that monitors some nonnumeric object. The last argument establishes the function that is used to determine the value of the gauge when the gauge is observed. Micrometer provides convenience methods for monitoring map and collection size, since these are such common cases.

Most forms of creating a gauge maintain only a *weak reference* to the object being observed, so as not to prevent garbage collection of the object. It is your responsibility to hold a strong reference to the state object that you are measuring with a gauge. Micrometer is careful to not create strong references to objects that would otherwise be garbage collected. Once the object being gauged is de-referenced and is garbage collected, Micrometer will start reporting a NaN or nothing for a gauge, depending on the registry implementation.

Generally the returned `Gauge` instance is not useful except in testing, as the gauge is already set up to track a value automatically upon registration.

In addition to the shortcut methods for creating a gauge directly from a `MeterRegistry`, Micrometer provides a gauge fluent builder (see [Example 2-13](#)) as well, which has more options. Notice the `strongReference` option, which does prevent garbage collection of the monitored object, contrary to the default behavior.

Example 2-13. Fluent builder for gauges

```
Gauge gauge = Gauge
    .builder("gauge", myObj, myObj::gaugeValue)
    .description("a description of what this gauge does") // Optional
    .baseUnit("speed")
    .tags("region", "test") // Optional
    .strongReference(IS_STRONG) // Optional
    .register(registry);
```

Micrometer has built-in metrics that include several gauges. Some examples are given in [Table 2-5](#).

Table 2-5. Examples of gauges in Micrometer built-in instrumentation

Metric name	Description
jvm.threads.live	The current number of live threads, including both daemon and non-daemon threads
jvm.memory.used	The amount of used memory in bytes
db.table.size	The total number of rows in a database table
jetty.requests.active	Number of requests currently active

A special kind of Gauge called a `TimeGauge` is specifically used for gauging values that are measuring time (see [Table 2-6](#)). Like a `Gauge`, there is no need to set a `TimeGauge`, since its value changes when observed. The only difference between them is that the value of a `TimeGauge` will be scaled to the monitoring system's base unit of time as it is published. In other cases, follow the general rule that values should be measured in whatever the natural base unit is (e.g., bytes for storage, connections for connection pool utilization). Monitoring systems only differ in their expectation of what the base unit is when describing time.

Table 2-6. Examples of time gauges in Micrometer built-in instrumentation

Metric name	Description
process.uptime	The uptime of the Java virtual machine, as reported by Java's Runtime MXBean
kafka.consumer.fetch.latency.avg	The average time taken for a group sync, as calculated and reported by the Kafka Java client

Kafka consumer fetch latency average is an example of where sometimes Java client libraries only provide coarse statistics like average where, if we could affect the Kafka client code directly, a timer would have been more suitable. In addition to seeing the average, we'd have information about decaying max latency, percentiles, etc.

One last special type of `Gauge` is a `MultiGauge`, which helps manage the gauging of a growing or shrinking list of criteria. Often this feature is used when we want to select a well-bounded but slightly changing set of criteria from something like a SQL query

and report some metric for each row as a Gauge. It doesn't have to be data fetched from a database, of course. The gauge could be built off a map-like structure in memory, or any other structure with rows containing a number of columns with at least one numeric column to use for the gauge value. [Example 2-14](#) shows how a Multi Gauge is created.

Example 2-14. Creating a multi-gauge

```
// SELECT count(*), city from customers group by city WHERE country = 'US'  
MultiGauge statuses = MultiGauge.builder("customers")  
    .tag("country", "US")  
    .description("The number of customers by city")  
    .baseUnit("customers")  
    .register(registry);  
  
...  
  
// Run this periodically whenever you rerun your query  
statuses.register(  
    resultSet.stream().map(result ->  
        Row.of(  
            Tags.of("city", result.getString("city")),  
            result.getInt("count")  
        )  
    )  
);
```

Before trying to build a gauge to report a rate at which something in your application is happening, consider a counter, which is better suited for this purpose.



Should I Use a Gauge or a Counter?

Never gauge something you can count.

Counters

Counters report a single metric, a count. The Counter interface allows you to increment by a fixed amount, which must be positive.

It is possible, though rare, to increment a counter by a fractional amount. For example, you could be counting sums of a base unit like dollars, which naturally have fractional amounts (although it would probably be more useful to count sales with a different meter type, as shown in “[Distribution Summaries](#)” on page 73).

The `MeterRegistry` interface contains convenience methods for creating counters, as shown in [Example 2-15](#).

Example 2-15. Creating counters

```
// No tags
Counter counter = registry.counter("bean.counter");

// Adding tags in key-value pairs with varargs
Counter counter = registry.counter("bean.counter", "region", "us-east-1");

// Explicit tag list creation ①
Counter counter = registry.counter("bean.counter", Tags.of("region", "us-east-1"));

// Adding tags to some other precreated set of tags.
Iterable<Tag> predeterminedTags = Tags.of("region", "us-east-1");
Counter counter = registry.counter("bean.counter",
    Tags.concat(
        predeterminedTags,
        "stack", "prod"
    )
);
```

- ① This is not specific to counters—there are similar APIs we will see on other meter types.

The `Counter` fluent builder, as seen in [Example 2-16](#), contains more options.

Example 2-16. Fluent builder for counters

```
Counter counter = Counter
    .builder("bean.counter")
    .description("a description of what this counter does") // Optional
    .baseUnit("beans")
    .tags("region", "us-east-1") // Optional
    .register(registry);
```

Micrometer has built-in metrics that include several counters. Some examples are given in [Table 2-7](#).

Table 2-7. Examples of counters in Micrometer built-in instrumentation

Metric name	Description
<code>jetty.async.requests</code>	Total number of async requests
<code>postgres.transactions</code>	Total number of transactions executed (commits + rollbacks)
<code>jvm.classes.loaded</code>	The number of classes that are currently loaded in the Java virtual machine
<code>jvm.gc.memory.promoted</code>	Count of positive increases in the size of the old generation memory pool before GC to after GC

When building graphs and alerts off of counters, generally you should be most interested in measuring the rate at which some event is occurring over a given time interval. Consider a simple queue. Counters could be used to measure things like the rate at which items are being inserted and removed.



When a Counter Measures Occurrences, It Measures Throughput

When we talk about the rate of occurrences of things happening, conceptually we are talking about *throughput*. When displaying counters as a rate on a chart, we are displaying a measure of throughput, the rapidity with which this counter is being incremented. When you have the opportunity to instrument an operation with metrics for each individual occurrence, you should almost always use timers (see “[Timers” on page 45](#)) instead, which provide a measure of throughput in addition to other useful statistics.

You might want at first to conceive of visualizing absolute counts rather than a rate, but the absolute count is usually both a function of the rapidity with which something is used and the longevity of the application instance under instrumentation. Building dashboards and alerts of the rate of a counter per some interval of time disregards the longevity of the app, letting you see aberrant behavior long after the application has started.

In many cases, when we’ve dug into why engineers attempt to visualize absolute counts, it is to show some true business-related number (number of sales, revenue, etc.). Remember that metrics instrumentation is optimized for signaling availability, so its implementation is naturally going to trade off durability for instrumentation performance. Any given metrics publishing interval can fail due to things like (physical or virtual) machine failure or network issues between the application and the metrics backend, and it won’t be retried because the assumption is you’ll just catch up on the next interval. Even with cumulative counters, if the final value before shutdown doesn’t make it to the backend, it is lost. Mission-critical counts, such as those required for legal reporting, should use some other durable storage instead of being shipped as a metric (or maybe *in addition* to being shipped as a metric).

For some monitoring systems, like Atlas, counters are published as a rate from Micrometer. Querying for and plotting a counter in Atlas, as seen in [Example 2-17](#), displays a chart whose *y*-axis is a rate.

Example 2-17. Atlas counter rate

```
name,queue.insert,:eq
```

Still, some monitoring systems expect counters to be published as a cumulative statistic. It is only at query time that the counter is converted to a rate for display, as in [Example 2-18](#). In this case, we have to use the specific `rate` function to convert the cumulative statistic into a rate. In almost every monitoring scenario, you'll use this `rate` function when accessing counters.

Example 2-18. Prometheus counter rate

```
rate(queue_insert_sum[2m])
```

It is tempting to count something like errors emitted from a particular method, or the total number of successful invocations of a method, but it is even better to record these events with a timer, because they include a count and other useful information about the latency of the operation.



Should I Use a Counter or a Timer (or a Distribution Summary)?

Never count something you can time. And if the base unit is not a unit of time, then the corollary is, awkwardly: never count something you can record with a distribution summary.

Timers

Timers are for measuring short-duration latencies and the frequency of such events. All implementations of `Timer` report at least a few individual statistics:

Count

A measure of the number of individual recordings of this timer. For a `Timer` measuring an API endpoint, this count is the number of requests to the API. `Count` is a measure of throughput.

Sum

The sum of the time it took to satisfy all requests. So if there are three requests to an API endpoint that took 5 ms, 10 ms, and 15 ms, then the sum is $5 + 10 + 15 = 30\text{ms}$. The sum may be shipped literally as 30 ms, or as a rate, depending on the monitoring system. We'll discuss shortly how to interpret this.

Maximum

The individual timing that took the longest, but decaying over an interval. Micrometer maintains a series of overlapping intervals in a ring buffer and keeps track of maximum in each of these intervals. The maximum value is then a little sticky in a sense. The important thing to keep in mind is that this isn't a maximum of all samples seen from the beginning of the app (this wouldn't be very

useful), but the maximum value seen “recently.” It is possible to configure the recency to make this value decay faster or slower.

In addition, timers can *optionally* ship other statistics:

Service level objective (SLO) boundaries

The count (i.e., total number) of requests observed that are less than or equal to a particular boundary value—for example, how many requests to an API endpoint took less than 100 ms. Since this is a count, it can be divided by the overall count to arrive at an idea of what percentage of requests met your service level objective, and is very cheap to calculate provided you know in advance what objectives you want to set.

Percentiles

Precomputed percentiles which cannot be combined with percentiles from other tags (e.g., percentiles for several instances in a cluster cannot be combined).

Histograms

Similar to SLO boundaries, histograms are comprised of a series of counts for a set of buckets. Histograms can be summed together across dimensions (e.g., sum the counts for like buckets across a series of instances) and be used to create percentile approximations by some monitoring systems. We’ll discuss histograms in more detail in [“Histograms” on page 65](#).

Let’s go through a couple of examples of how these statistics might be used and the relationship between them.

“Count” Means “Throughput”

The count statistic of a timer is individually useful. It is a measure of *throughput*, the rate at which the timed operation is happening. When timing an API endpoint, it’s the number of requests to that endpoint. When measuring messages on a queue, it’s the number of messages being placed on the queue.

The count statistic should be used exactly as described in [“Counters” on page 42](#), as a rate. Depending on the monitoring system, this statistic will be either a cumulative count or a rate when shipped from Micrometer.

“Count” and “Sum” Together Mean “Aggregable Average”

With the one exception we’ll discuss shortly, sum is not really meaningful on its own. Without being considered in relation to the rate at which operations are occurring, a sum has no purpose. A sum of 1 second may be bad for an individual request to a user-facing API endpoint, but 1,000 requests of 1 ms each yielding a sum of 1 second sounds quite good!

Sum and count together can be used to create an *aggregateable* average. If we instead published the average of a timer directly, it couldn't be combined with the average data from other dimensions (such as other instances) to reason about the overall average.

Consider the scenario described in [Figure 2-4](#), where a load balancer has distributed seven requests to four application instances. Three of these application instances are in Region 1 and one instance is in Region 2.

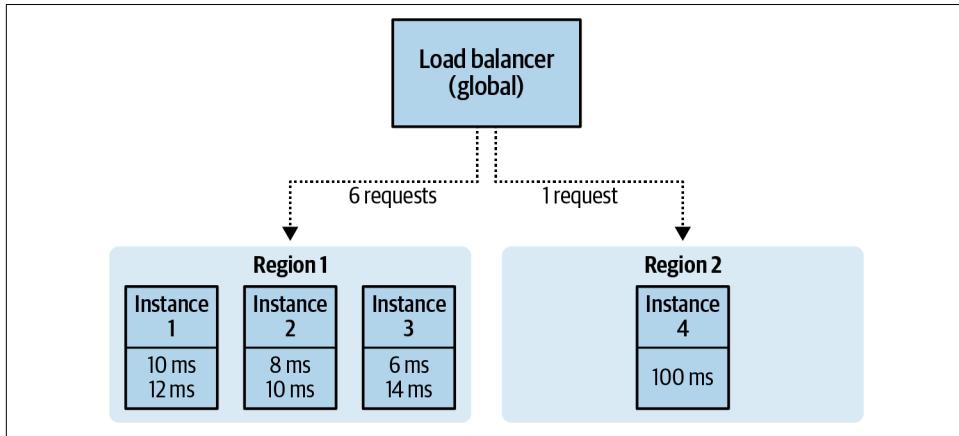


Figure 2-4. Timings for requests going to a hypothetical application

Suppose we've tagged the timer metrics for each of these instances with the instance ID and region. The monitoring system then sees time series for timers with four different combinations of tags:

- Instance=1, Region=1
- Instance=2, Region=1
- Instance=3, Region=1
- Instance=4, Region=2

There will be time series for count and sum for each of these timers. In [Table 2-8](#), after these seven requests have occurred, a cumulative monitoring system will have values for sum and count for their corresponding tags. Also included is the average for that instance individually.

Table 2-8. Cumulative sum and count for each timer

Instance	Region	Count (operations)	Sum (seconds)	Average (seconds/operation)
1	1	2	0.022	0.011
2	1	2	0.018	0.009
3	1	2	0.020	0.010
4	2	1	0.100	0.100

To find the average latency for this timer across all instances in both regions of this application, we add the sums and divide that by the sum of the counts (see [Equation 2-1](#)).

Equation 2-1. Computing the cluster average

$$\frac{0.022 + 0.018 + 0.020 + 0.100}{2 + 2 + 2 + 1} = 0.017 \text{ seconds/op} = 17 \text{ milliseconds/op}$$

If instead Micrometer only shipped averages from each instance, we wouldn't have an easy way of calculating this same value. Averaging the averages, as shown in [Equation 2-2](#), is not correct. The “average” here is too high. Several more requests went to Region 1 than Region 2, and Region 1 was serving responses much more quickly.

Equation 2-2. An INCORRECT calculation of cluster average

$$\frac{0.011 + 0.009 + 0.010 + 0.100}{4 \text{ instances}} = 0.032 \text{ seconds/request} = 32 \text{ milliseconds/request}$$

The demonstration for how average is calculated across the cluster here assumes that Micrometer was shipping a cumulative value for sum and count. How does it work out if Micrometer was shipping a rate instead? [Table 2-9](#) shows rate-normalized values such as those that would be shipped to Atlas. For the sake of this table, assume that the seven requests shown in [Figure 2-4](#) happened over a one-minute interval and that this interval is aligned with the interval in which we push metrics to Atlas.

Table 2-9. Rate-normalized sum and count for each timer

Instance	Region	Count (requests/second)	Sum (unitless)	Average (seconds/request)
1	1	0.033	0.00037	0.011
2	1	0.033	0.00030	0.009
3	1	0.033	0.00033	0.010
4	2	0.017	0.00167	0.100

The count column now has a unit of “requests/second” rather than just “requests.” It will be requests/second no matter what the publishing interval is. In this case, we’re publishing every minute; so since we saw two requests to Instance 1, we conclude that the requests/second rate of requests to this instance is [Equation 2-3](#).

Equation 2-3. The rate of throughput to Instance 1

$$2 \text{ requests}/\text{minute} = 2 \text{ requests}/60 \text{ seconds} = 0.033 \text{ requests}/\text{second}$$

The sum column now is unitless rather than being in seconds. It’s unitless because the numerator of the rate is seconds as well as the denominator, and these units cancel out. So for Instance 1, the sum is [Equation 2-4](#). This unitless nature of sum in a rate-normalized system serves to underscore its meaninglessness independent of being combined with count (or some other dimensioned value).

Equation 2-4. The rate-normalized sum of Instance 1

$$22 \text{ milliseconds}/\text{minute} = 0.022 \text{ seconds}/60 \text{ seconds} = 0.00037$$

The average per instance is the same as in the cumulative table because of the effect of the units canceling out.

For the purpose of average, it doesn’t matter what the interval is. If the interval were two minutes rather than one minute, our idea of throughput changes (i.e., it’s exactly half), but the extra minute cancels out in the average calculation. In the case of Instance 1, the count in requests/seconds is [Equation 2-5](#).

Equation 2-5. The rate of throughput to Instance 1 over a two-minute interval

$$2 \text{ requests}/2 \text{ minutes} = 2 \text{ requests}/120 \text{ seconds} = 0.01667 \text{ requests}/\text{second}$$

The sum is [Equation 2-6](#). But when we divide these, the average is still the same. In this division, you can essentially factor out the 2 from both numerator and denominator.

Equation 2-6. The rate-normalized sum of Instance 1 over a two-minute interval

$$22 \text{ milliseconds}/2 \text{ minutes} = 0.022 \text{ seconds}/120 \text{ seconds} = 0.00018$$

Average is not ideal for monitoring availability. In general, you are better off accepting a slightly worse average and better worst-case (e.g., greater than 99th percentile) performance since the worst case tends to happen much more frequently than our intuition leads us to believe. Still, the combination of sum divided by count is easy to compute and possible on almost all monitoring systems, even those without more

sophisticated math operations. So average is at least some baseline to have across a range of wildly different monitoring systems. If at all possible, don't use average at all.

Average: a random number that falls somewhere between the maximum and 1/2 the median. Most often used to ignore reality.

—Gil Tene

Instead, for availability it is more useful to look at maximum or a high-percentile statistic, as we'll see in detail in “[Timers](#)” on page 143.

Maximum Is a Decaying Signal That Isn't Aligned to the Push Interval

Micrometer decays the maximum rather than aligning it to the publishing interval like it does for sum and count. If we perfectly aligned the view of maximum time to the push interval, then a dropped metrics payload means we potentially miss out on seeing a particularly high maximum value (because in the next interval we'd only consider samples that occurred in that interval).

For other statistics like count, missing a publishing interval is generally not problematic, because the counter continues to accumulate during a period where a metrics payload is dropped, and the next successful payload will surface it.

Practically, there are many reasons why a high maximum latency and a dropped metrics payload would be correlated. For example, if the application is under heavy resource pressure (like a saturated network interface), a response to the user for an API endpoint that is being timed (and for which a maximum value is being tracked) may be exceedingly high at the same time that a metrics post request to the monitoring system fails with a read timeout. But such conditions can be (and many times are) temporary.

Perhaps you have a client-side load-balancing strategy that recognizes that (from the client's perspective) API latency has gone up sharply for this instance that is under resource pressure and begins preferring other instances. By relieving pressure on this instance it recovers.

In some subsequent interval, after the instance has recovered, it's nice to be able to push a maximum latency seen during this time of trouble that would otherwise have been skipped. In fact, it's precisely these times of duress that we care about the most, not the maximum latency under fair-weather conditions!

The effect of this decaying though is that a maximum value will “linger” for a period of time after it actually occurred. In [Figure 2-5](#), we see a maximum value of approximately 30 ms for a timed operation. This 30 ms operation occurred sometime in the metrics publishing interval immediately before when the line first spikes up from 0 (around 19:15).

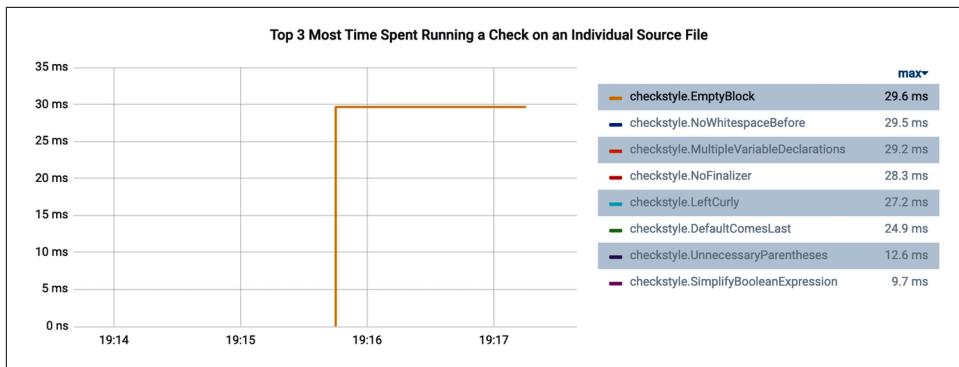


Figure 2-5. The decaying maximum lingers on a chart for some time

This timer was configured to decay maximum values over a period of two minutes. So it lingers until about 19:17. Since this timer didn't see any operations after this initial interval in which the 30 ms timing was seen, the time series disappears after the maximum value decays.

Micrometer keeps track of maximums in a **ring buffer** to achieve this decaying behavior. The ring buffer has configuration options `distributionStatisticsBufferLength` and `distributionStatisticExpiry` on `Timer.Builder` that you can use to decay for longer, as shown in [Example 2-20](#). By default, Micrometer builds timers with a ring buffer of length 3, and the pointer will be advanced every 2 minutes.

[Figure 2-6](#) is an illustration of a ring buffer of three elements. This ring buffer is just an array with a pointer to a particular element from which the maximum value will be polled whenever we publish metrics. Every `distributionStatisticExpiry`, the pointer is advanced to the next element in the ring buffer. The zeroth index element in this buffer has no samples in it. The first and second indexed elements are storing the state of the largest sample they have seen since their last reset, 10 ms. The darkened ring around the first index indicates that this index is the element that is currently being polled from.

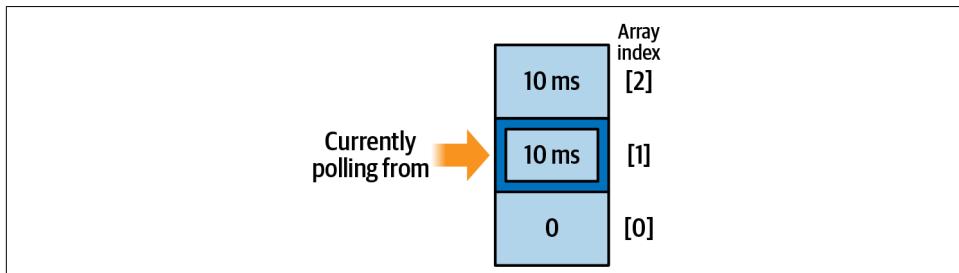


Figure 2-6. A ring buffer of three elements

Figure 2-7 shows a timer max ring buffer with three elements and a two-minute expiry as it evolves over an eight-minute period of time. Below the figure is a minute-by-minute description of how the values are changing where t is the wall time in minutes.

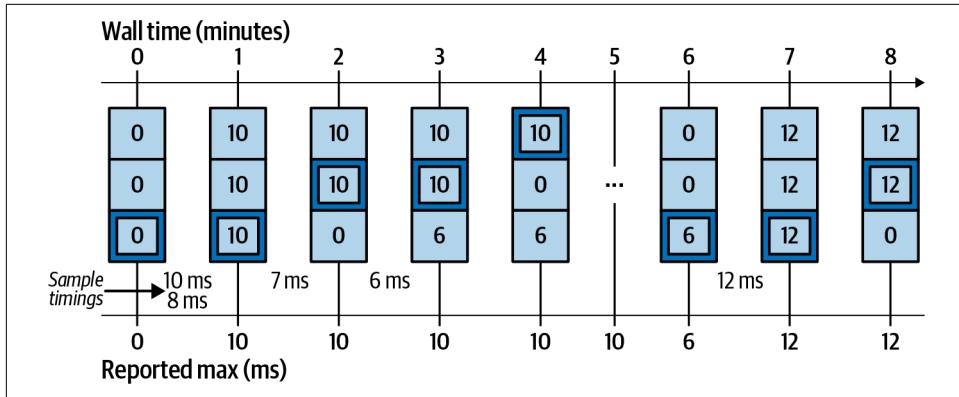


Figure 2-7. A timer max ring buffer of length 3 and two-minute expiry

$t=0$

This is the initial state. Each ring buffer element is empty. No timer recordings have been observed. Between $t=0$ and $t=1$, two timer recordings are observed: one at 10 ms and one at 8 ms.

$t=1$

Since 10 ms is the greater of the two recordings seen and every ring buffer element was previously empty, all of them now are tracking 10 ms as the maximum. If we were to publish metrics at $t=1$, the max would be 10 ms. Between $t=1$ and $t=2$ we observe a 7 ms timing, but it is not greater than the samples in any of the ring buffer elements.

$t=2$

The zeroth ring buffer element is reset because the expiry has been reached and the pointer is moved to index 1. Between $t=2$ and $t=3$ we see a timer recording of 6 ms. Since the zeroth element has been cleared, it now is tracking 6 ms as its view of the max. The polled max is 10 ms.

$t=3$

The oldest two ring buffer elements are still seeing 10 ms as the max, and the zeroth element is tracking 6 ms. The polled max is still 10 ms since the pointer is on index 1.

t=4

Index 1 is reset and the pointer is advanced to index 2. Index 2 is still tracking 10 ms, so the polled max is 10 ms.

t=5

Nothing changes. The polled max is 10 ms. Note that the timer would have reported a count and sum of 0 at this time and a max of 10 ms! This is what is meant by max not being aligned to the publishing interval in the same way that count and sum are.

t=6

Index 2 is reset and the pointer circles back to index 0, which is still hanging on to the 6 ms sample it observed between $t=2$ and $t=3$ as the max. The polled max is 6 ms. Between $t=6$ and $t=7$, a 12 ms sample is observed, which becomes the max across the ring buffer.

t=7

The polled max is 12 ms, observed shortly before $t=7$.

t=8

The zeroth ring buffer element is reset and the pointer moves to index 1. The polled max is 12 ms.

The Sum of Sum Over an Interval

There are few cases where the sum of sums is useful. In fact, I've only ever encountered one case of it, which we'll see later in [“Proportion of time spent in garbage collection” on page 162](#), where the sum of time spent in garbage collection (GC) is divided by the total amount of time in the interval in which garbage collection was happening (e.g., how much time was spent in GC altogether). Even in this case, we probably could develop a better alert signal for garbage collection if the JVM provided us with discrete timings for every garbage collection event as it occurred. If it did, we might look at high-percentile GC times or max GC time by cause.

The Base Unit of Time

The appropriate base unit for timers varies by monitoring system. For example, Prometheus expects floating point second-precision data because conceptually seconds are a base unit of time. Atlas expects nanosecond-precision data because it can then accept and store integral values. Since it isn't possible to measure a subdivision of a nanosecond (and in many cases it isn't even possible to measure in true nanosecond precision), the backend takes advantage of this optimization. Regardless, Micrometer automatically scales timings to the expected base unit for each monitoring system.

Neither one of these base units is right or wrong, and neither less precise. It is simply a matter of convention in each. The base unit of time doesn't affect the *precision* of charts. Even though Micrometer ships times in seconds to Prometheus, for example, the chart will often still be displayed, like in [Figure 2-8](#), in milliseconds for common timers like those used to monitor user-facing API endpoints.

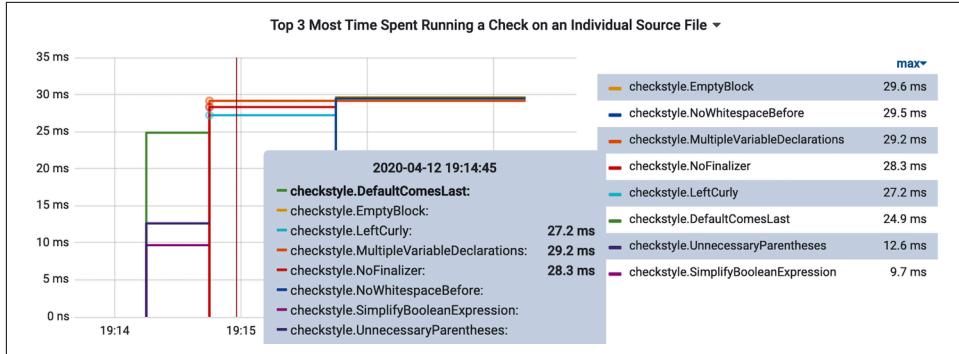


Figure 2-8. A timer shipped with seconds base units displayed in milliseconds

We'll cover charting more in [Chapter 4](#), but for now, just know that scaling in this way is often accomplished automatically by the charting interface. We just need to tell it how to interpret the statistic, i.e., to say that the time series being displayed is dimensioned in seconds, as in [Figure 2-9](#).

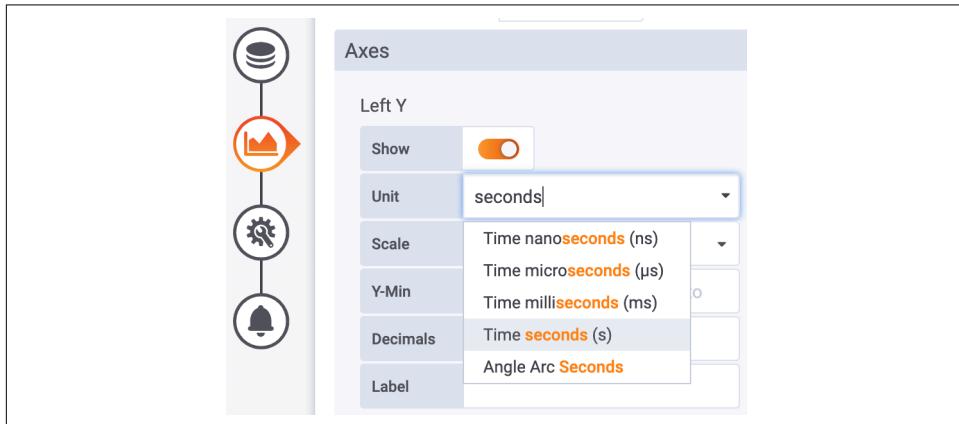


Figure 2-9. Informing the charting library how to interpret timer base units

In this case, Grafana is smart enough to know that it is easier for humans to read in milliseconds than small fractions of a second, so it scales the seconds data down to milliseconds. Similarly, if it is canonical to represent timings in nanoseconds to a particular monitoring system, the charting library would perform the opposite math to scale the values up to milliseconds.



Common Base Units Don't Limit How You View the Data

For common units like time and data size (e.g., bytes), you shouldn't need to concern yourself with how you intend to *view* the data to decide what scale you *record* at. It's generally preferable to be consistent with your base units everywhere and allow the charting library later to do this sort of automatic scaling to a human-readable format. Rather than recording the payload size of a response body in bytes (because it will generally be small) and the size of the heap in megabytes (because it will generally be large), record both in bytes. They will both be scaled to reasonable values later when you go to view them.

For monitoring systems that do not have a clear preference for base unit of time, Micrometer chooses one; and it is not generally configurable because consistency across all applications is more important than changing the base unit of time when precision is not sacrificed either way.

Using Timers

The `MeterRegistry` interface contains convenience methods for creating timers, as shown in [Example 2-19](#).

Example 2-19. Creating timers

```
// No tags
Timer timer = registry.timer("execution.time");

// Adding tags in key-value pairs with varargs
Timer timer = registry.timer("execution.time", "region", "us-east-1");

// Explicit tag list creation
Timer timer = registry.timer("execution.time", Tags.of("region", "us-east-1"));
```

The `Timer` fluent builder (see [Example 2-20](#)) contains more options. Most of the time you won't use all of these options.

Example 2-20. Fluent builder for timers

```
Timer timer = Timer
    .builder("execution.time")
    .description("a description of what this timer does")
    .distributionStatisticExpiry(Duration.ofMinutes(2))
    .distributionStatisticBufferLength(3)
    .serviceLevelObjectives(Duration.ofMillis(100), Duration.ofSeconds(1))
    .publishPercentiles(0.95, 0.99)
    .publishPercentileHistogram()
```

```
.tags("region", "us-east-1")
.register(registry);
```

The `Timer` interface exposes several convenience overloads for recording timings inline, such as in [Example 2-21](#). Additionally, a `Runnable` or `Callable` can be wrapped with instrumentation and returned for use later.

Example 2-21. Recording execution with a timer

```
timer.record(() -> dontCareAboutReturnValue());
timer.recordCallable(() -> returnValue());

Runnable r = timer.wrap(() -> dontCareAboutReturnValue());
Callable c = timer.wrap(() -> returnValue());
```



Timers Versus Distribution Summaries

Timers are really just a specialized form of distribution summaries (see “[Distribution Summaries](#)” on page 73) that are aware of how to scale durations to the base unit of time of each monitoring system and that have an automatically determined base unit. In almost every case where you want to measure time, you should use a `Timer` rather than a `DistributionSummary`. The rare exception to this is when recording many long-duration events in a short interval, such that the nanosecond-precision `Timer` would overflow ~290 years (since Java’s `long` can effectively store a maximum of `9.22e9` seconds) inside of a single interval.

You may also store start state in a sample instance that can be stopped later. The sample records a start time based on the registry’s clock. After starting a sample, execute the code to be timed, and finish the operation by calling `stop(Timer)` on the sample.

Notice in [Example 2-22](#) how the timer that the sample is accumulating to is not determined until it is time to stop the sample. This allows some tags to be determined dynamically from the end state of the operation we are timing. The use of `Timer.Sample` is especially common when we are dealing with some event-driven interface with a listener pattern. This example is a simplified form of Micrometer’s JOOQ execution listener.

Example 2-22. Use of timer samples for event-driven patterns

```
class JooqExecuteListener extends DefaultExecuteListener {
    private final Map<ExecuteContext, Timer.Sample> sampleByExecuteContext =
        new ConcurrentHashMap<>();

    @Override
```

```

public void start(ExecuteContext ctx) {
    Timer.Sample sample = Timer.start(registry);
    sampleByExecuteContext.put(ctx, sample);
}

@Override
public void end(ExecuteContext ctx) {
    Timer.Sample sample = sampleByExecuteContext.remove(sample);

    sample.stop(registry.timer("jooq.query", ...)); ①
}
}

```

- ① We would typically add some tags based on the result of the execution with data elements found in ExecuteContext.

There is also an AutoCloseable form of timer sample that is useful when timing blocks of code that contain checked exception handling, as shown in [Example 2-23](#). The pattern does require nested try statements, which are a bit unusual. If you are uncomfortable with this pattern, you can absolutely stick to a simple `Timer.Sample`.

Example 2-23. Use of timer samples

```

try (Timer.ResourceSample sample = Timer.resource(registry, "requests")
        .tag("method", request.getMethod()) ①
        .description("This is an operation")
        .publishPercentileHistogram()) {
    try { ②
        if (outcome.equals("error")) {
            throw new IllegalArgumentException("boom");
        }
        sample.tag("outcome", "success"); ③
    } catch (Throwable t) {
        sample.tag("outcome", "error");
    }
}

```

- ① This tag will apply to both outcomes. Description text and percentile histograms will also apply to both outcomes.
- ② This nested try statement makes it possible to access the `Timer.ResourceSample` in the catch block for adding error tags.
- ③ We can add tags at each branching point in the try/catch block to record information about the outcome.

Micrometer has built-in metrics that include several timers. Some examples are given in [Table 2-10](#).

Table 2-10. Examples of timers in Micrometer built-in instrumentation

Metric name	Description
http.server.requests	Spring Boot records timings for executions of WebMVC and WebFlux request handlers.
jvm.gc.pause	Time spent in GC pause.
mongodb.driver.commands	Time spent in MongoDB operations.

Timers are the metrics corollary to distributed tracing (discussed in depth in [Chapter 3](#)) in the sense that trace spans and timers can instrument the same code, as in [Example 2-24](#).



The Intersection of Tracing and Metrics

The overlap between distributed tracing and metrics is strictly limited to timing.

The sample code uses Zipkin's Brave instrumentation, which we'll see again later.

Example 2-24. Tracing and timing the same block of code

```
// Start a new trace
ScopedSpan span = tracer.startScopedSpan("encode"); ①
try (Timer.ResourceSample sample = Timer.resource(registry, "encode")) {
    try {
        encoder.encode();
        sample.stop(registry.timer("encode", "result", "success"));
    } catch (RuntimeException | Error e) {
        span.error(e);
        sample.stop(registry.timer("encode", "result", "failure"));
        throw e;
    } finally {
        span.finish();
    }
}
```

- ① Brave doesn't have an AutoCloseable construct like Micrometer, so the instrumentation looks a little asymmetric.

It's natural to assume that a particular block of code, given similar inputs, would execute in roughly the same amount of time. Our intuition about this can be misleading.

Common Features of Latency Distributions

It is important to understand some common characteristics of timings in Java applications. The same block of code will not execute in the same amount of time on each execution due to variance in the input parameters, downstream systems, the state of the heap, and many other variables. Nevertheless, many requests with similar inputs will commonly be satisfied in a similar amount of time.

Intuition may lead to a belief that timings are roughly normally distributed, i.e., that there is a central hump around the average with lower-probability tails for both faster and slower response times, like in [Figure 2-10](#).

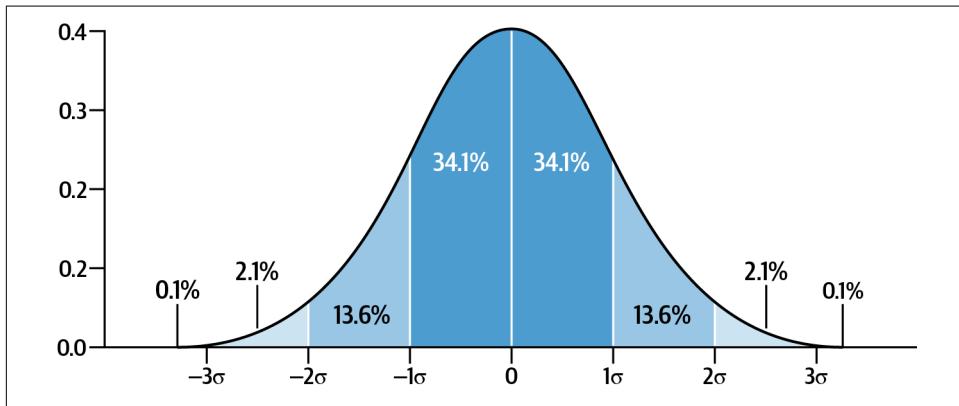


Figure 2-10. The normal distribution

In real-world cases, timings are almost always multimodal, meaning there is more than one “hump,” or grouping of timings, along the latency spectrum. Most commonly, Java timings are bimodal (two humps, as shown in [Figure 2-11](#)), with the smaller, rightmost hump representing events like garbage collection and VM pauses. That second hump also includes a ripple effect of the multimodality in downstream services.

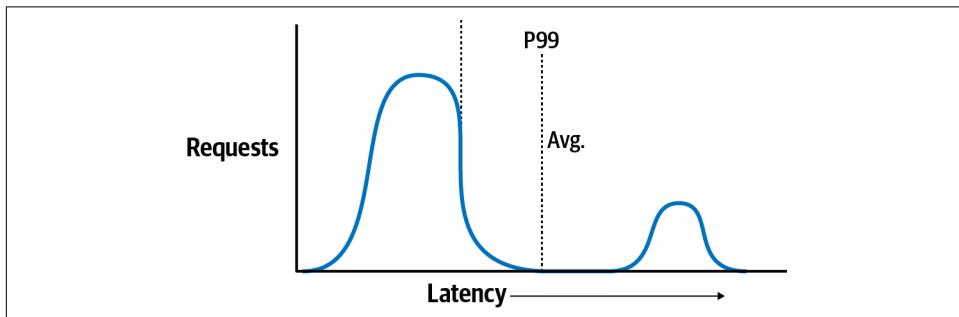


Figure 2-11. A typical bimodal distribution of Java latencies

Bizarrely, the bigger, leftmost hump is often (though not always, of course) quite narrow and contains more than 99% of the timings. As a result, the 99th percentile (see “[Percentiles/Quantiles](#)” on page 60) will in many cases be below the average, which is skewed higher by the second hump.



Standard Deviation

Some metrics instrumentation libraries and systems ship and display standard deviation. This metric only makes sense in the context of a normal distribution, which we've seen is essentially never the case. Standard deviation is not a meaningful statistic for real-world timings of Java executions. Ignore it! Also, ignore average.

Percentiles/Quantiles

Did I mention not to use average latency? At this point, you can probably see that I won't miss an opportunity to pounce on average latency.

Average: a random number that falls somewhere between the maximum and 1/2 the median. Most often used to ignore reality.

—Gil Tene

Average latency is a poor metric to monitor to assess latency behavior, and max is a useful alert threshold but can be spiky. For comparative performance, we can look at high-percentile values, which are less spiky. High max spikes will certainly be more prevalent in less-performant code, but in any case *when* they occur is not under your control, making side-by-side comparisons even of identical blocks of code running on two different instances difficult.

Depending on the monitoring system, the term *percentile*, or *quantile*, is used to describe a point in a set of samples that relates to the rank order of its values. So the middle quantile, which is also called the median or the 50th percentile, is the middle value in a set of samples ranked from least to greatest.



Median Versus Average

Average is rarely useful for monitoring timings. Statistically, the average is the sum of all samples divided by the total number of samples. The average is just a different measure of centrality than the median, not better or worse in general at representing centrality. Even if it were a “perfect” measure of centrality, for the purpose of proving our system is reliable we care more about the *worst* half of the distribution, not the best half.

Percentiles are a special type of quantile, described relative to 100%. In a list of 100 samples ordered from least to greatest, the 99th percentile (P99) is the 99th sample in order. In a list of 1,000 samples, the 99.9th percentile is the 999th sample.

By this definition, percentiles, particularly high percentiles, are useful for determining what *most* users are experiencing (i.e., P99 is the worst latency that 99 out of 100 users experienced). For timings, percentiles usefully cut out the spiky behavior of VM or garbage collection pauses while still preserving majority user experience.

It is tempting to monitor a high-percentile value like the 99th and feel at ease that your users are experiencing good response times. Unfortunately, our intuition leads us astray with these statistics. The top 1% typically hides latencies that are one or two orders of magnitude larger than P99.

Any single request will avoid the top 1% exactly 99% of the time. When considering N requests, the chance that at least one of these requests is in the top 1% is $(1 - 0.99^N) * 100\%$ (assuming these probabilities are independent, of course). It takes surprisingly few requests for there to be a greater than majority chance that one request will hit the top 1%. For 100 individual requests, the chance is $(1 - 0.99^{100}) * 100\% = 63.3\%$!

Consider the fact that a user interaction with your system likely involves many resource interactions (UI, API gateway, multiple microservice calls, some database interactions, etc.). The chance that any individual end-to-end *user interaction* experiences a top 1% latency on some resource in the chain of events satisfying their request is actually much higher than 1%. We can approximate this chance as $(1 - 0.99^N) * 100\%$. If a single request in a chain of microservices experiences a top 1% latency, then the whole user experience suffers, especially given the fact that the top 1% tends to be an order of magnitude or more worse in performance than requests under the 99th percentile.



Time (Anti-)Correlation of Samples

The formulas given to determine the chance of experiencing a top 1% latency are only approximations. In reality, time-correlation of high/low latency leads to a lower chance, and anti-correlation would lead to a higher chance. In other words, the probabilities for each request aren't truly independent. We almost never have information about this correlation, so the approximations are useful guides for how you should reason about your system.

Micrometer supports two ways of computing percentiles for timers:

- Precompute the percentile value and ship it directly to the monitoring system.
- Group timings into discrete sets of latency buckets, and ship the sets of buckets together to the monitoring system (see “[Histograms](#)” on page 65). The monitoring system is then responsible for computing the percentile from a histogram.

Precomputing percentile values is the most portable approach since many monitoring systems don’t support histogram-based percentile approximation, but it’s useful only in a narrow set of circumstances (we’ll see why a little later in this section). Pre-computed percentiles can be added to a timer in a few different ways.

The `Timer` fluent builder supports adding percentiles directly as the `Timer` is being constructed, as shown in [Example 2-25](#).

Example 2-25. Adding percentiles to a Timer via the builder

```
Timer requestsTimer = Timer.builder("requests")
    .publishPercentiles(0.99, 0.999)
    .register(registry);
```

[Example 2-26](#) shows how to add percentiles with a `MeterFilter`.

Example 2-26. Adding percentiles to a Timer via a MeterFilter

```
registry.config().meterFilter(new MeterFilter() {
    @Override
    public DistributionStatisticConfig configure(Meter.Id id,
        DistributionStatisticConfig config) {
        if (id.getName().equals("requests")) {
            DistributionStatisticConfig.builder()
                .publishPercentiles(0.99, 0.999)
                .build()
                .merge(config);
        }
        return config;
    }
});
```

...

```
// The filter will apply to this timer as it is created
Timer requestsTimer = registry.timer("requests");
```

Lastly, frameworks like Spring Boot offer property-driven `MeterFilter` equivalents that allow you to add percentiles to `Timers` declaratively. The configuration shown in [Example 2-27](#) adds percentiles to any timer prefixed with the name `requests`.

Example 2-27. Adding percentiles to metrics prefixed with “requests” in Spring Boot

```
management.metrics.distribution.percentiles.requests=0.99,0.999
```

Adding percentiles through a `MeterFilter` allows you to add percentile support to `Timers` that are created not just in your application code, but in other libraries you are including in your application that contain Micrometer instrumentation.



Adding Timers to Library Code

If you are authoring a library and including timing code, do not preconfigure your timers with features like percentiles, histograms, and SLO boundaries. These features all have some performance cost, even if it is minimal. Allow the consumers of your library to determine if the timing is an important enough indicator to warrant the extra expense of these statistics. In particular, end users will want to turn on histograms when they intend to use the timing as part of a comparative measure, like in [“Automated Canary Analysis” on page 205](#). When needed, users can configure these statistics with a `MeterFilter`.

Wherever a timer metric has more than a few total unique combinations of tags, pre-computed percentiles are unusable because they cannot be combined or aggregated. On a cluster of two instances, if the 90th percentile latency for a request endpoint is 100 ms on one application instance and 200 ms on another, we can’t simply average these two values together to arrive at a cluster-wide 90th percentile latency of 150 ms.

[Table 2-11](#) shows why, using the median (50th percentile) as an example. Since the individual samples that went into this percentile calculation were thrown away, there is no way to reconstitute them to derive a cluster-wide percentile at the monitoring system.

Table 2-11. P50 (median) request latency in a cluster of two instances

Instance	Individual latencies (ms)	P50 latency (ms)
1	[100,110,125]	110
2	[125,130,140]	130
Whole cluster	[100,110,125,125,130,140]	125

The best we can do with precomputed percentiles is to simply plot all of the values and look for outliers, as in [Figure 2-12](#), generated from the Prometheus query `requests_second{quantile=0.99}`.

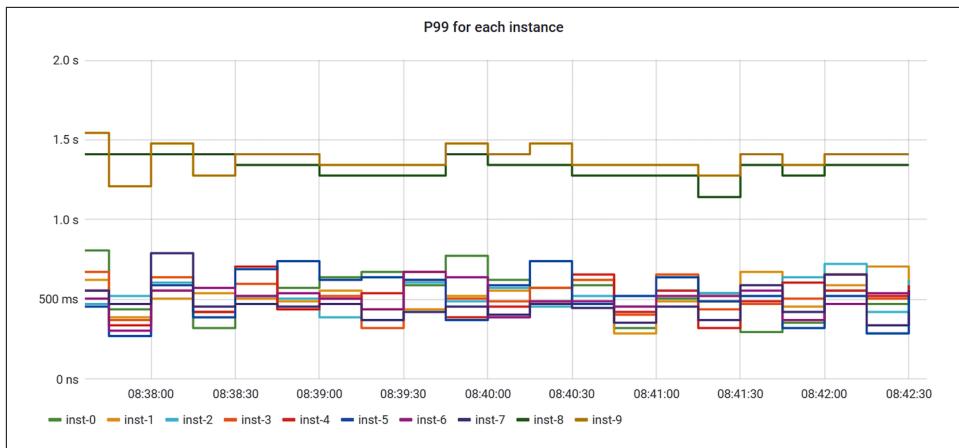


Figure 2-12. 99th percentile of a single timer for individual application instances

This scales to a point; but as the number of instances grows (imagine we had a cluster of 100 instances!), the visualization quickly becomes crowded. Attempts to limit the number of lines displayed to select just the top N worst latencies can result in situations where the legend is still full of individual instance IDs. This is because, as we see in [Figure 2-13](#), where we are selecting the top three worst latencies with the Prometheus query `topk(3, requests_second{quantile=0.99})`, the third-worst instance changes virtually every interval.

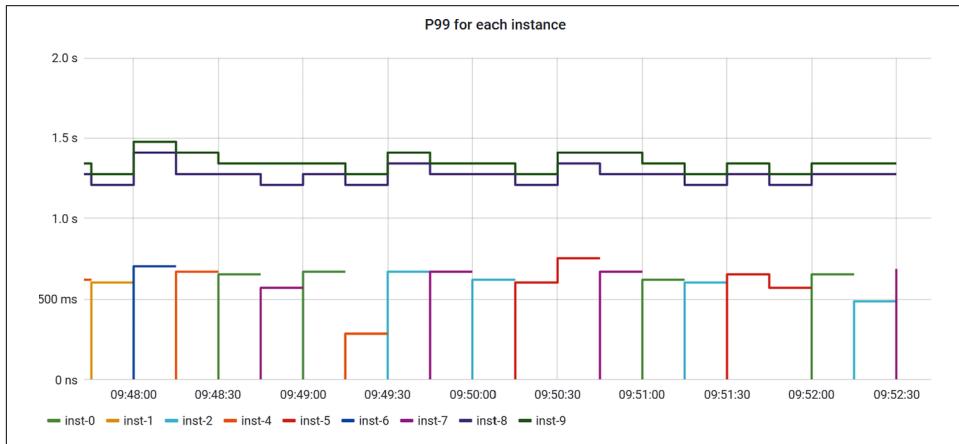


Figure 2-13. Top three worst 99th percentile of a single timer for individual application instances

Because of the limitations of precomputed percentiles, if you are working with a monitoring system that supports histograms, *always* use them instead, as described in the following section.

Histograms

Metrics are always presented in aggregated form to the monitoring system. The individual timings that together we represent as the latency of a block of code are not shipped to the monitoring system. If they were, metrics would no longer have a fixed cost irrespective of throughput.

We can send an approximation of what the individual timings looked like together in a histogram. In a histogram, the range of possible timings is divided into a series of buckets. For each bucket (also known as an *interval* or *bin*), the histogram maintains a count of how many individual timings fell into that bucket. The buckets are consecutive and nonoverlapping. They are not often of equal size, since there is generally some part of the range which we care about at a more fine-grained level than others. For example, for an API endpoint latency histogram, we care more about the distinction between 1, 10, and 100 ms latencies than we do about 40 s and 41 s latencies. The latency buckets will be more granular around the expected value than well outside the expected value.

Importantly, by accumulating all the individual timings into buckets, and controlling the number of buckets, we can retain the shape of the distribution while maintaining a fixed cost.

Histograms are represented in the monitoring system's storage as a series of counters. In the case of Prometheus, as in [Table 2-12](#), these counters have a special tag `le` that indicates that the metric is a count of all samples less than or equal to the tag value (in seconds).

Table 2-12. How histogram buckets are stored in a time series database (Prometheus)

Metric name	Values
<code>http_server_requests_seconds_bucket{status=200,le=0.1}</code>	[10,10,12,15]
<code>http_server_requests_seconds_bucket{status=200,le=0.2}</code>	[20,20,24,26]
<code>http_server_requests_seconds_bucket{status=200,le=0.5}</code>	[30,30,40,67]
<code>http_server_requests_seconds_bucket{status=500,le=0.1}</code>	[1,1,2,5]
<code>http_server_requests_seconds_bucket{status=500,le=0.2}</code>	[1,1,2,6]
<code>http_server_requests_seconds_bucket{status=500,le=0.5}</code>	[1,1,2,6]

Depending on the monitoring system, a histogram may appear as either a normal or a cumulative histogram. A visual distinction between these two types of histograms is shown in [Figure 2-14](#). Cumulative histogram buckets represent the count of all timings less than or equal to their boundary. Note that the timer's count is equal to the sum of all buckets in a normal histogram.

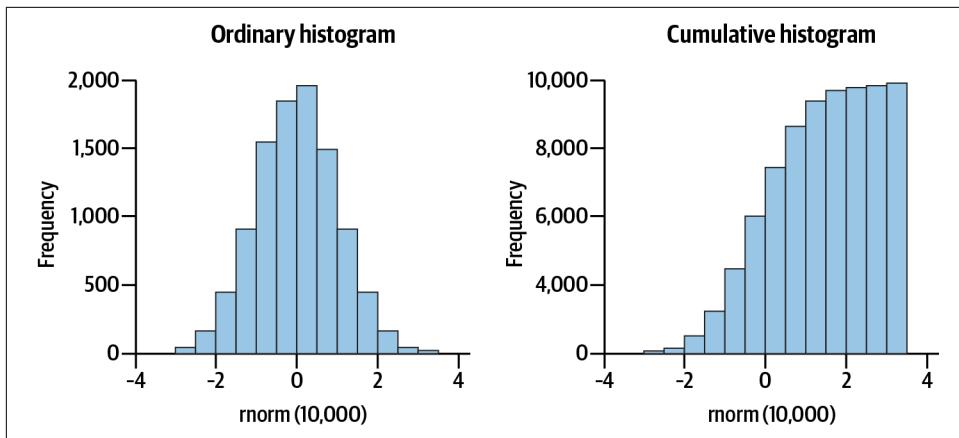


Figure 2-14. Cumulative versus normal histogram

Like adding any other tag, adding histogram data to a timer increases the total required storage by a factor equal to the number of buckets that the range is subdivided into. In this example, since there are three buckets (0.1 s, 0.2 s, and 0.5 s) there will be three times the number of permutations of other tags' time series stored.

This histogram is published each interval to the monitoring system. The histograms for each interval can be assembled into a heatmap. Figure 2-15 shows the heatmap for the latency of an API endpoint. Most requests are served in ~1 ms, but there is a long tail of latencies leading all the way up to >100 ms in each interval.

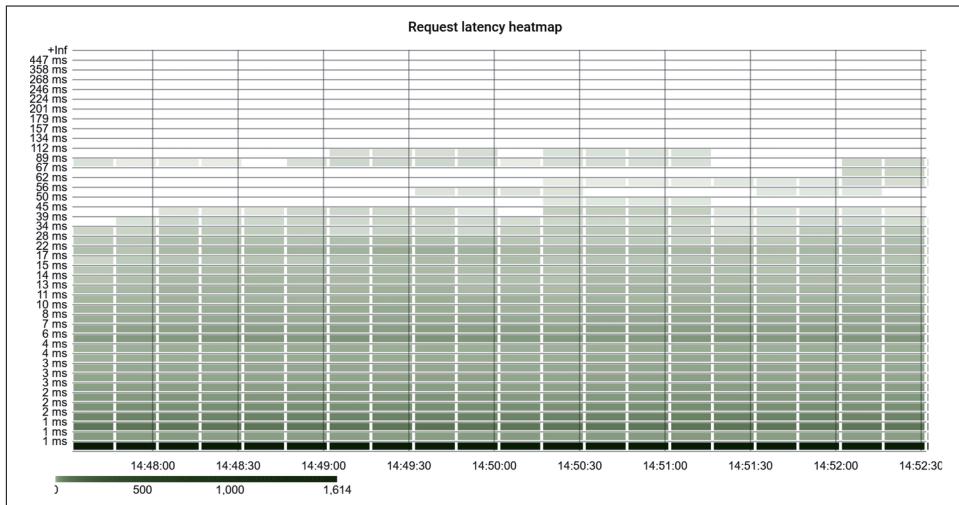


Figure 2-15. Latency heatmap

Histogram data can also be used to perform an approximation for a percentile, and this common use of histogram data is reflected in Micrometer’s option to enable histograms, `publishPercentileHistogram`. High percentiles are especially useful when performing comparative measurements of an application’s performance (such as the relative performance of two versions of an application in “[Automated Canary Analysis](#)” on page 205). Histograms are not enabled by default because of the additional storage cost on the monitoring system and heap usage in your application. Micrometer uses a bucketing function empirically determined at Netflix to generate reasonably low error bounds for these approximations.

Histogram publishing can be enabled for timers in a few ways.

The `Timer` fluent builder supports adding histograms directly as the `Timer` is being constructed, as shown in [Example 2-28](#).

Example 2-28. Adding histograms to a Timer via the builder

```
Timer requestsTimer = Timer.builder("requests")
    .publishPercentileHistogram()
    .register(registry);
```

Histogram support can be added after the fact via a `MeterFilter`, as shown in [Example 2-29](#). This ability is crucial in layering your application with effective monitoring. In addition to their particular business logic, applications almost always contain a rich binary dependency hierarchy as well. It is reasonable for authors of common dependencies like HikariCP for connection pooling or the RabbitMQ Java client to want to include instrumentation involving timers in their code. But it is impossible for the authors of the RabbitMQ Java client to know whether RabbitMQ interactions are significant enough in your application to warrant the extra cost of shipping distribution statistics like percentile histograms (no matter how optimized they may be). Allowing application developers to turn on additional distribution statistics via `MeterFilter` allows the RabbitMQ Java client authors to use a minimal timer in their code.

Example 2-29. Adding histograms to a Timer via a MeterFilter

```
registry.config().meterFilter(new MeterFilter() {
    @Override
    public DistributionStatisticConfig configure(Meter.Id id,
        DistributionStatisticConfig config) {
        if (id.getName().equals("requests")) {
            DistributionStatisticConfig.builder()
                .publishPercentileHistogram()
                .build()
                .merge(config); ①
        }
    }
})
```

```

    return config;
}
});

...
// The filter will apply to this timer as it is created
Timer requestsTimer = registry.timer("requests");

```

- ❶ Combines percentile histogram publishing with whatever other distribution statistics are configured by default (or by other `MeterFilter` configurations).

Lastly, frameworks like Spring Boot offer property-driven `MeterFilter` equivalents that allow you to add histograms to `Timers` declaratively. The configuration shown in [Example 2-30](#) adds histogram support to any timer prefixed with the name `requests`.

Example 2-30. Adding percentile histograms to metrics prefixed with “requests” in Spring Boot

```
management.metrics.distribution.percentiles-histogram.requests=true
```

Histograms are only shipped to monitoring systems that support percentile approximation based on histogram data.

For Atlas, use the `:percentiles` function, as in [Example 2-31](#).

Example 2-31. Atlas percentiles function

```
name,http.server.requests,:eq,
(,99,99.9,),:percentiles
```

For Prometheus, use the `histogram_quantile` function, as in [Example 2-32](#). Recall from [“Percentiles/Quantiles” on page 60](#) that percentiles are just a particular type of quantile. Prometheus histograms contain a special bucket called `Inf` that captures all samples exceeding the greatest bucket that you (or Micrometer) defines. Note that the timer’s count is equal to the count in the `Inf` bucket.

Example 2-32. Prometheus histogram quantile function

```
histogram_quantile(
  0.99,
  rate(http_server_requests_seconds_bucket[2m])
)
```

Service Level Objective Boundaries

Similar to percentiles and histograms, service level objective (SLO) boundaries can be added either through the `Timer` fluent builder or through a `MeterFilter`. You may notice I said “boundaries” and not “boundary” as you might expect. In many circumstances, it’s reasonable to layer your objectives. Gil Tene talks about establishing SLO requirements in his 2013 [talk](#) on monitoring latency, which I’ll paraphrase here because this is such a useful framework for explaining the need to layer SLOs. The SLO requirements interview is captured in [Example 2-33](#).

Example 2-33. The SLO requirements interview

Q: What are your service level objectives?
A: We need an average response of 10 milliseconds ①
Q: What is the worst-case requirement?
A: We don't have one.
Q: So it's OK for some things to take more than 5 hours?
A: No!
Q: So we think that the worst-case requirement is 5 hours.
A: No! Let's make it 100 milliseconds.
Q: Are you sure? Even if the worst case only happens two times a day?
A: OK, make it 2 seconds.
Q: How often is it OK to have a 1-second response?
A: (Annoyed) I thought you said only a couple of times a day!
Q: That was for the worst case. If half the results are better than 10 milliseconds, is it OK for every other request other than max to be just shy of 2 seconds?
A: (More specific requirements...)

- ① Oh no, you said the word “average”...we’re just going to pretend we didn’t hear this.

This interview eventually yields a set of SLOs.

- 90% better than 20 milliseconds
- 99.99% better than 100 milliseconds
- 100% better than 2 seconds

We will configure Micrometer then to publish SLO counts for 20 milliseconds, 100 milliseconds, and 2 seconds. And we can simply compare, for example, the ratio of requests less than 20 milliseconds to the total number of requests; and if this ratio is less than 90%, then alert.

Micrometer will ship a count for each of these boundaries that indicates how many requests did not exceed that boundary. A set of SLO boundaries together form a coarse histogram, as seen in [Figure 2-16](#), where the latency domain is divided into

buckets of zero to the lowest SLO and so on for consecutive SLO boundaries after that.

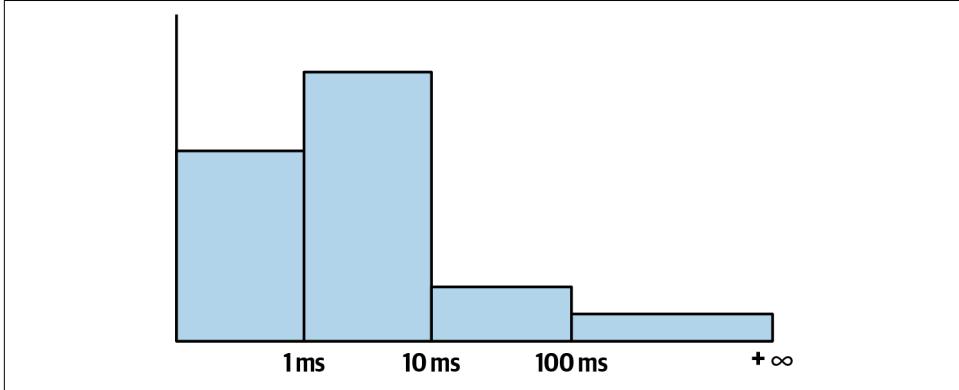


Figure 2-16. Histogram of SLO boundaries

Shipping SLO boundaries does have an effect on total storage in the monitoring system and memory consumption in your application. However, because typically only a small set of boundaries is published, the cost is relatively low compared to percentile histograms.

Percentile histograms and SLOs can be used together. Adding SLO boundaries simply adds more buckets than would be shipped with just a percentile histogram. When SLO boundaries are shipped in addition to percentile histograms, the histogram shipped contains the buckets Micrometer decides are necessary to yield reasonable error bounds on percentile approximations *plus* any SLO boundaries, as shown in Figure 2-17.

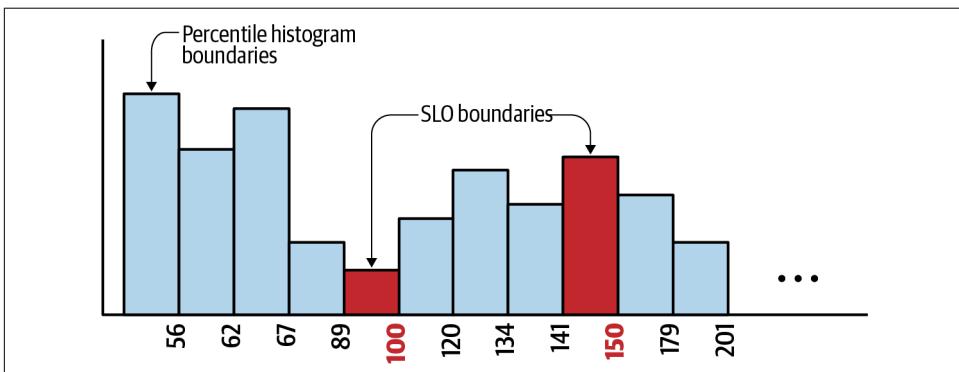


Figure 2-17. Mixed histogram of service level objective boundaries and percentile histogram bucket boundaries

Publishing SLO boundaries is a far cheaper (and accurate) way of testing whether the Nth percentile exceeds a certain value. For example, if you determine that an SLO is that 99% of requests are below 100 ms, then publish a 100 ms SLO boundary. To set an alert on violations of the SLO boundary, simply determine whether the ratio of requests below the boundary to total requests is less than 99%.

This is a little inconvenient when the monitoring system expects normal histograms, like Atlas, because you have to select and sum all the buckets *less than* the SLO boundary. In [Example 2-34](#), we want to test whether 99% of requests are less than 100 ms (0.1 seconds); but since it isn't possible to treat tag values as numerical values, we can't use operators like `:le` to select all time series with a tag less than 0.1 seconds. So we have to resort to performing numerical comparisons with a regular expression operator like `:re`.

Example 2-34. Atlas alert criteria using an SLO boundary

```
name,http.server.requests,:eq,  
:dup,  
slo,0.(0\d+|1),:re,  
:div,  
0.99,:lt,  
uri,_API_ENDPOINT,:eq,:cq
```

Prometheus has the advantage here, given that its histograms are expressed cumulatively. That is, all the samples below 100 ms are accumulated to every boundary less than 100 ms, including this boundary.

The alert criteria is shown in [Example 2-35](#).

Example 2-35. Prometheus alert criteria using an SLO boundary

```
http_server_requests_seconds_bucket{le="0.1", uri="/API-ENDPOINT"}  
/ ❶  
http_server_requests_seconds_count{uri="/API-ENDPOINT"} < 0.99
```

❶ Division symbol

Visually, the effect of these two queries can be seen in [Figure 2-18](#).

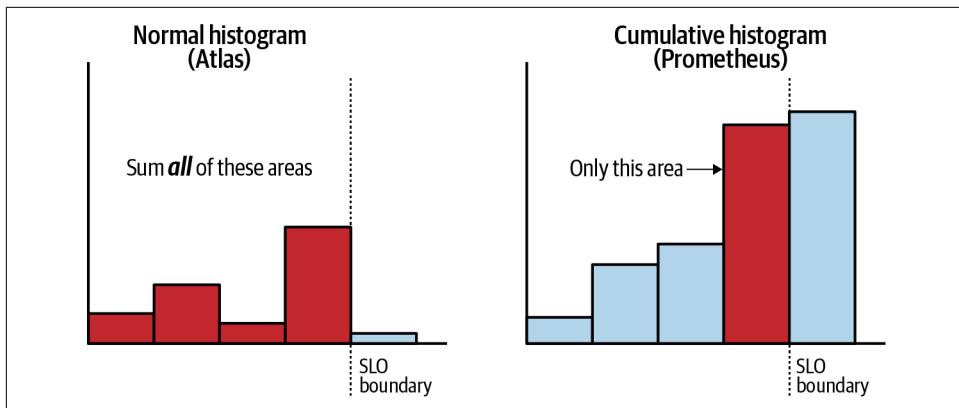


Figure 2-18. SLO boundary alert queries: Atlas versus Prometheus

SLO publishing can be enabled for timers in a few ways.

The `Timer` fluent builder supports adding SLOs directly as the `Timer` is being constructed, as shown in [Example 2-36](#).

Example 2-36. Adding SLO boundaries to a Timer via the builder

```
Timer requestsTimer = Timer.builder("requests")
    .slo(Duration.ofMillis(100), Duration.ofSeconds(1))
    .register(registry);
```

[Example 2-37](#) shows how to add SLO boundaries with a `MeterFilter`.

Example 2-37. Adding SLO boundaries to a Timer via a MeterFilter

```
registry.config().meterFilter(new MeterFilter() {
    @Override
    public DistributionStatisticConfig configure(Meter.Id id,
        DistributionStatisticConfig config) {
        if (id.getName().equals("requests")) {
            DistributionStatisticConfig.builder()
                .slo(Duration.ofMillis(100), Duration.ofSeconds(1))
                .build()
                .merge(config);
        }
        return config;
    }
});
```

...

```
Timer requestsTimer = registry.timer("requests"); ❶
```

- ① The filter will apply to this timer as it is created.

The next meter type is very similar to a timer.

Distribution Summaries

A distribution summary, shown in [Example 2-38](#), is used to track the distribution of events. It is similar to a timer structurally, but it records values that do not represent a unit of time. For example, a distribution summary could be used to measure the payload sizes of requests hitting a server.

Example 2-38. Creating a distribution summary

```
DistributionSummary summary = registry.summary("response.size");
```

Micrometer also provides a fluent builder, shown in [Example 2-39](#) for distribution summaries. For maximum portability, add base units, as they are part of the naming convention for some monitoring systems. Optionally, you may provide a scaling factor that each recorded sample will be multiplied by as it is recorded.

Example 2-39. The distribution summary fluent builder

```
DistributionSummary summary = DistributionSummary
    .builder("response.size")
    .description("a description of what this summary does")
    .baseUnit("bytes")
    .tags("region", "test")
    .scale(100)
    .register(registry);
```

Distribution summaries have all the same percentile, histogram, and SLO options that timers do. Timers are just a specialized distribution summary for measuring time. SLOs are defined as fixed values instead of durations (i.e., 1000 instead of `Duration.ofMillis(1000)`, where 1,000 means something, depending on what base unit is assigned to the distribution summary) and provide convenience methods for timing blocks of code. That is the only difference in the available options.

A common example of a distribution summary is payload size measured in bytes.

Much like with timers, in many real-world cases the distribution is rarely normal. At one point I measured payload size in bytes, which was largely normal except there was a sharp drop-off on the left side of the distribution, because I was including request headers in the payload size. So there were zero occurrences of request payloads less than the size of having a certain set of request headers present.

Because distribution summaries can track any unit of measure, and the distribution of the measured values cannot be generally known as it can be for timers, the best way to alert on a distribution summary has some nuance:

- When the distribution is multimodal, as it is for timers, it is likely best to set alerts on maximum value so that you can keep track of where the “worst case” is.
- In most cases, it still makes sense to use high percentiles like the 99th percentile for comparative analysis (see “[Automated Canary Analysis](#)” on page 205).

Long Task Timers

The long task timer is a special type of timer that lets you measure time while an event being measured is *still running*. A timer does not record the duration until the task is complete. Long task timers ship several statistics:

Active

The number of executions that are currently in progress.

Total duration

The sum of all in-progress execution times of the block of code being measured.

Max

The longest in-progress timing. The max represents the total execution time of the oldest still-running execution time.

Histograms

A set of discretized buckets of in-progress tasks.

Percentiles

Precomputed percentiles of in-progress execution times.

[Figure 2-19](#) shows how a long task timer fundamentally differs from a timer. As soon as an operation is complete, it no longer contributes to the total duration. In contrast, an operation instrumented with a timer isn’t reported *until* it is complete. Notice how at time t=3, total duration increases by 2 rather than just 1. This is because we have two tasks that were executing beginning at t=2, so they each contribute 1 to the total at each interval while they continue to run. At t=4, both tasks stop, and so total duration drops to zero, along with active count.

Long task timer average has a different meaning than a timer average. It is the average of the time active operations that have been running *to this point*. Similarly, maximum represents the longest running task to this point, decayed in a similar way that timer maximum is.

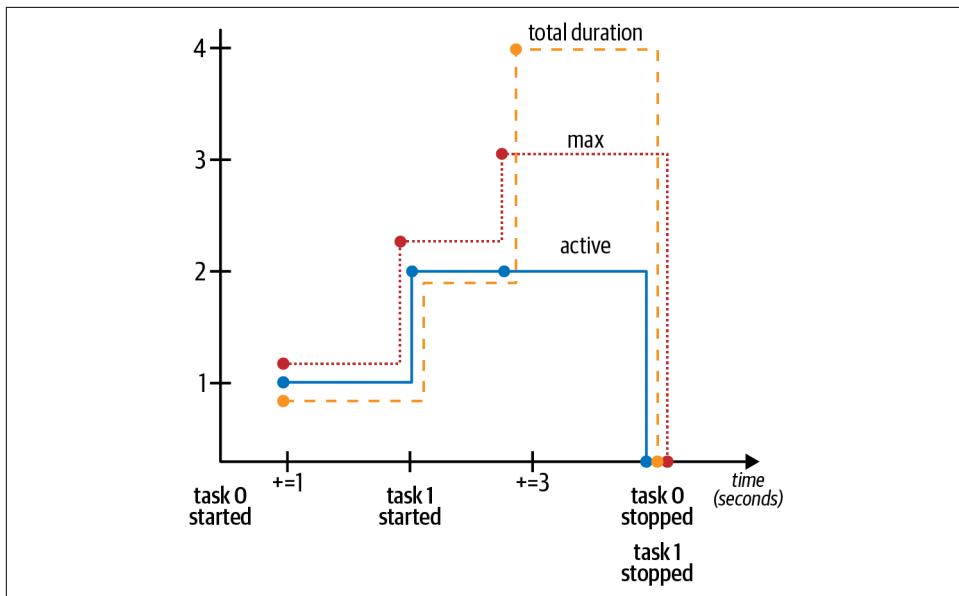


Figure 2-19. Long task timer active and total duration for two tasks

The `MeterRegistry` interface contains convenience methods for creating long task timers, as shown in [Example 2-40](#).

Example 2-40. Creating long task timers

```
// No tags
LongTaskTimer timer = registry.more()
    .longTaskTimer("execution.time");

// Adding tags in key-value pairs with varargs
LongTaskTimer timer = registry.more()
    .longTaskTimer("execution.time", "region", "us-east-1");

// Explicit tag list creation
LongTaskTimer timer = registry.more()
    .longTaskTimer("execution.time", Tags.of("region", "us-east-1"));
```

The `LongTaskTimer` fluent builder contains more options, shown in [Example 2-41](#).

Example 2-41. Fluent builder for long task timers

```
LongTaskTimer timer = LongTaskTimer
    .builder("execution.time")
    .description("a description of what this timer does") // Optional
    .tags("region", "us-east-1") // Optional
    .register(registry);
```

Long task timers have a record method that returns a Sample that can later be stopped and convenience methods for recording a body of code wrapped in a lambda, as shown in [Example 2-42](#).

Example 2-42. Recording execution with a long task timer

```
longTaskTimer.record(() -> dontCareAboutReturnValue());
longTaskTimer.recordCallable(() -> returnValue());

LongTaskTimer.Sample sample = longTaskTimer.start();
// Do something...
sample.stop(longTaskTimer);
```

A good example of a long task timer is in [Edda](#), which caches AWS resources such as instances, volumes, and autoscaling groups. Normally all data can be refreshed in a few minutes. If the AWS services are performing more slowly than usual, it can take much longer. A long task timer can be used to track the overall time for refreshing the metadata.

In application code, it is common for such long-running processes to be implemented with something like Spring Boot's @Scheduled, as shown in [Example 2-43](#).

Example 2-43. An explicitly recorded long task timer for a scheduled operation

```
@Scheduled(fixedDelay = 360000)
void scrapeResources() {
    LongTaskTimer.builder("aws.scrape")
        .description("Time it takes to find instances, volumes, etc.")
        .register(registry)
        .record(() -> {
            // Find instances, volumes, autoscaling groups, etc...
        });
}
```

Some frameworks like Spring Boot also respond to the @Timed annotation to create long task timers when the longTask attribute is set to true, as in [Example 2-44](#).

Example 2-44. An annotation-based long task timer for a scheduled operation

```
@Timed(name = "aws.scrape", longTask = true)
@scheduled(fixedDelay = 360000)
void scrapeResources() {
    // Find instances, volumes, autoscaling groups, etc...
}
```

If we wanted to alert when this process exceeds our threshold, with a long task timer we will receive that alert at the first reporting interval after we have exceeded the

threshold. With a regular timer, we wouldn't receive the alert until the first reporting interval after the process completed, over an hour later!

Choosing the Right Meter Type

In the instrumentation libraries for hierarchical metrics systems (see “[Hierarchical Metrics](#)” on page 26), typically only gauges and counters were supported. This led to a habit of precalculating statistics (most commonly rates like request throughput) and presenting them to the monitoring system as a gauge that could fluctuate up or down. Since Micrometer always exposes counters in a way that lets you derive a rate at query time, there is no need to perform this calculation manually in your application code.

Both `Timer` and `DistributionSummary` always publish a count of events in addition to other measurements. There should never be a reason to count the number of executions of a block of code. They should be timed instead.



Which Meter Type to Choose?

Never gauge something you can count, and never count something you can time.

As a general rule, select a `LongTaskTimer` whenever timings exceed two minutes, when you need to monitor in-flight requests, and especially when an operation's failure may inflate the expected time from a few minutes to many minutes or hours.

Controlling Cost

The cost of metrics grows as you instrument more and more pieces of your application. Perhaps at the beginning you only start with shipping basic statistics like memory and processor utilization, and grow to additional areas like HTTP request monitoring, cache performance, database interactions, and connection pool saturation,. In fact, many of these basic use cases are going to be increasingly automatically instrumented by frameworks like Spring Boot.

Expanding instrumentation to additional components isn't the likely source of high cost in telemetry, however. To examine the cost of an individual metric, think about the unique permutations of a metric name and all of its key-value tag combinations. These form a set of time series on the backend, and we refer to the size of this set as the *cardinality* of the metric.

It is essential that you carefully bound a metric's cardinality by considering all the possible key-value tags combinations that could result in a high number of unique values and limit them in some way. The cardinality of a metric is the product of the

unique tag values for each tag key. The reasonable limit you place on tag cardinality varies from monitoring system to monitoring system, but in general keeping it in the thousands is a reasonable upper bound. At Netflix, the general advice was to keep metric cardinality under one million time series. This is probably at the outside edge of what most organizations would find responsible.

Remember, metrics are intended to present an aggregate view of an indicator, and they shouldn't be used to try to examine event-level or request-level performance. Other forms of telemetry like distributed tracing or logging are more appropriate for individual event-level telemetry.

Framework-provided telemetry is generally carefully crafted to limit tag cardinality. For example, Spring Boot's timing of Spring WebMVC and WebFlux adds a handful of tags but carefully limits how each contributes to the cardinality of the metric. The following list bexplains what each tag key means:

Method

There are a small number of possible values for HTTP method, based on the HTTP specification, e.g., GET or POST.

Status

HTTP status codes come from the HTTP specification, so are naturally limited in the possible values. Also, most API endpoints are going to realistically only return one of a few values: 200–202 success/created/accepted, 304 not modified, 400 bad request, 500 internal server error, and maybe 403 forbiddden. Most won't return even this amount of variation.

Outcome

A summarization of status code, e.g., SUCCESS, CLIENT_ERROR, or SERVER_ERROR.

URI

This tag is a good demonstration of how tag cardinality could quickly get out of hand. For 200–400 status codes, the URI is going to be the path that the request was mapped to. But when the endpoint contains a path variable or request parameters, the framework is careful to use the *unsubstituted* path here rather than the raw path of the request, e.g., /api/customer/{id} instead of /api/customer/123 and /api/customer/456. Not only does this help limit the metric's cardinality, but it is also more useful to reason about the performance of all requests to /api/customer/{id} in the aggregate as opposed to groups of requests that retrieved the particular customer with ID 123 or 456. In addition to path substitution, the URI should be further constrained in cases where the server is ultimately going to return a 404. Otherwise, every mistyped URI /api/doesntexist/1, /api/doesntexistagain, etc., results in a new tag. So Spring Boot uses a URI tag value of NOT_FOUND whenever the status code is 404. Similarly, it uses a REDIRECT value when the status code is 301 because the server may

always redirect unauthenticated requests to the authentication mechanism, even when the requested path doesn't exist.

Exception

When the request results in a 500 internal server error, this tag contains the exception class name, which is just a simple way of grouping the general classes of failures—for example, null pointer exceptions versus a connection timeout on a downstream service request. Again, there are generally only a few possible classes of failures based on the implementation of the endpoint, so this is naturally bounded.

For HTTP server request metrics then, the total cardinality might be something like [Equation 2-7](#).

Equation 2-7. Cardinality of HTTP server request metric for a single endpoint

$$2 \text{ methods} \times 4 \text{ statuses} \times 2 \text{ outcomes} \times 1 \text{ URI} \times 3 \text{ exceptions} = 48 \text{ tagvalues}$$

Resist the urge to overoptimize for cost. In particular, there is no need to try to limit the publication of zero values, because in some application states in theory all metrics could be nonzero for a particular publishing interval. It is at that moment of greatest saturation where you would be publishing the most nonzero metrics that will govern the cost of metrics to you, not the low points where you could publish some lesser set of metrics by excluding zero values. Also, reporting a zero value is a useful signal that something is just not happening as opposed to the service not reporting at all.

In some cases, where tag cardinality cannot be limited easily by out-of-the-box instrumentation, you will find the configuration of a set of metrics makes you define how to limit the tags responsibly. A good example is Micrometer's instrumentation for the Jetty `HttpClient`. A typical request using the Jetty `HttpClient` looks like [Example 2-45](#). The `HttpClient` API doesn't provide a mechanism for providing a path variable to the `POST` call and a collection of variable values to substitute later, so by the time Micrometer's Jetty `Request.Listener` intercepts the request, path variables have already been substituted irreversibly.

Example 2-45. Jersey HTTP client call with path variable string concatenation

```
Request post = httpClient.POST("https://customerservice/api/customer/" + customerId);
post.content(new StringContentProvider("{\"detail\": \"all\"}"));
ContentResponse response = post.send();
```

Jetty client metrics should use the unsubstituted path for the `uri` tag for the same reason the `URI` tag on Spring Boot's instrumentation of WebMVC and WebFlux was based on an unsubstituted value. It becomes the responsibility of the engineer using

Jetty HttpClient to specify how to unsubstiute path variables for the purpose of tagging, as shown in [Example 2-46](#).

Example 2-46. Jersey HTTP client metrics configuration

```
HttpClient httpClient = new HttpClient();
httpClient.getRequestListeners().add(
    JettyClientMetrics
        .builder(
            registry,
            result -> {
                String path = result.getRequest().getURI().getPath();
                if(path.startsWith("/api/customer/")) {
                    return "/api/customer/{id}";
                }
                ...
            }
        )
        .build()
);
```

Coordinated Omission

My first job in high school was as a fast-food worker. Running the drive-through window gave me some early firsthand experience about how to lie with statistics.

Our performance as a store was evaluated on the basis of the *average* duration from the time a customer placed an order at the menu to when they departed with their order. Our store was fairly understaffed typically, so at times the average time exceeded our goal. We simply waited until a lull in activity and did laps around the building with one of our cars. A couple dozen three-to-four-second service times will lower the average quickly (again, average is a problem statistic more often than not)! We could make our service time look arbitrarily good. At some point, the corporate headquarters added minimum service time to the average statistic they were evaluating, and our cheating was over.

In one bizarre case, a bus came through the drive-through and each window of the bus ordered. Our service time was only driven by a pressure plate near the menu and after the service window, so it was unaware of service time per order, only per vehicle. Clearly we didn't serve the bus order as quickly as we would serve a typical car, and the ripple effect of the bus on other service times illustrates a concept called *coordinated omission*, in which if we aren't careful, we monitor some definition of service time but exclude wait time. There are two examples of coordinated omission with this bus incident, illustrated in [Figure 2-20](#).

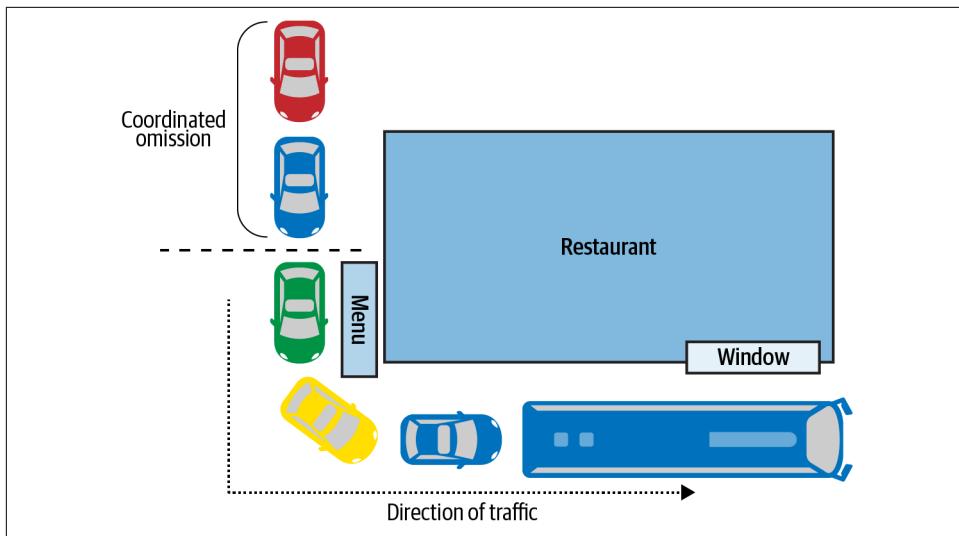


Figure 2-20. Coordinated omission caused by a bus in the drive-through

- At the point where the bus is at the window receiving its orders, there are only three other cars whose service time is being recorded (the three that have activated the pressure plate at the menu). The two cars behind the menu aren't being observed by the system. The actual impact of the bus's obstruction was on five other customers, but we will only see a service time impact on three.
- Suppose instead of service time being determined from the menu through leaving the order window, we instead monitored service time as just the time spent at the order window. The average service time then was only affected by how long it took to serve the bus alone, and not the compounding effect it had on the cars behind it.

These effects are similar to the effect that thread pools have on request timing. If a response time is calculated based on when a request handler begins processing a request and ends when the response is committed, there is no accounting for the time that a request sat in a queue waiting for an available request handler thread to begin processing it.

The other consequence of coordinated omission illustrated by this example is that blocking the drive-through lane prevents the restaurant from being totally overwhelmed by a steady state of customer orders. In fact, the appearance of a long line at the drive-through may have discouraged would-be customers from even attempting an order. Thread pools can have this effect as well. Coordinated omission arises from several sources:

Serverless functions

Measuring the execution of a serverless function from the perspective of that function of course doesn't record the time required to launch the function.

Pauses

Pauses come in many forms—for example, the JVM pauses due to garbage collection, a reindexing database becomes momentarily unresponsive, and cache buffers flush to disk. Execution pauses are reported as higher latencies on in-flight timings, but operations for which timings haven't yet started will report unrealistically low latencies.

Load testers

Conventional load testers back up in their own thread pools before truly saturating a service.

The need for accurate load tests is so common that we'll go into a little more detail about them.

Load Testing

Some conventional load testers like Apache Bench generate requests at a particular rate. Aggregated statistics are generated from the set of all response times. When responses don't fit inside the collection bucket interval, the next request will be delayed.

In this way, a service that is becoming oversaturated doesn't get pushed over the edge because the unintentional coordination of a longer response time causes the load test to back off. This coordination comes from the fact that these types of tests use a blocking request model that effectively limits concurrency to less than or equal to the number of cores on the machine running the test.

Real users don't have this kind of coordination. They interact with your service independently of one another, and so can saturate the service to a far greater degree. One effective way to simulate user behavior is to saturate your service with a nonblocking load test. Gatling and JMeter each act this way (but Apache Bench does not). But to illustrate how an effective load test should work, we can use Spring's nonblocking WebClient and Project Reactor to create a simple nonblocking load test. The result is shown in [Example 2-47](#). It's in fact so easy to build these nonblocking load tests now that maybe it's not worth the extra cognitive overhead of dedicated tools like JMeter and Gatling. You'll have to decide this for yourself and your team.

Example 2-47. A nonblocking load test with WebClient and Project Reactor

```
public class LoadTest {
    public static void main(String[] args) {
        MeterRegistry meterRegistry = ...; ①
        Counter counter = meterRegistry.counter("load.test.requests");

        WebClient client = WebClient.builder()
            .baseUrl("http://" + args[0])
            .build();

        Flux
            .generate(AtomicLong::new, (state, sink) -> { ②
                long i = state.getAndIncrement();
                sink.next(i);
                return state;
            })
            .limitRate(1) ③
            .flatMap(n -> client.get().uri("/api/endpoint").exchange())
            .doOnNext(resp -> {
                if (resp.statusCode().is2xxSuccessful())
                    counter.increment();
            })
            .blockLast();
    }
}
```

- ① Configure a registry to ship metrics from the load test's perspective to a monitoring system of choice.
- ② Generate an infinite flux, or if you want to run this for a particular number of requests, you can use `Flux.range(0, MAX_REQUESTS)` instead.
- ③ Clamp the rate at which you want the test to send requests to the service.

The difference between a reactive load test like this and a conventional load test is significant, as seen in [Figure 2-21](#). Conventional load tests (because they are blocking, and therefore can only have a concurrency level equal to or less than the number of cores on the machine running the test) show a max latency less than 10 ms. A non-blocking reactive load test shows a tower of latencies all the way up to greater than 200 ms, with a strong band greater than 200 ms as the application becomes saturated.

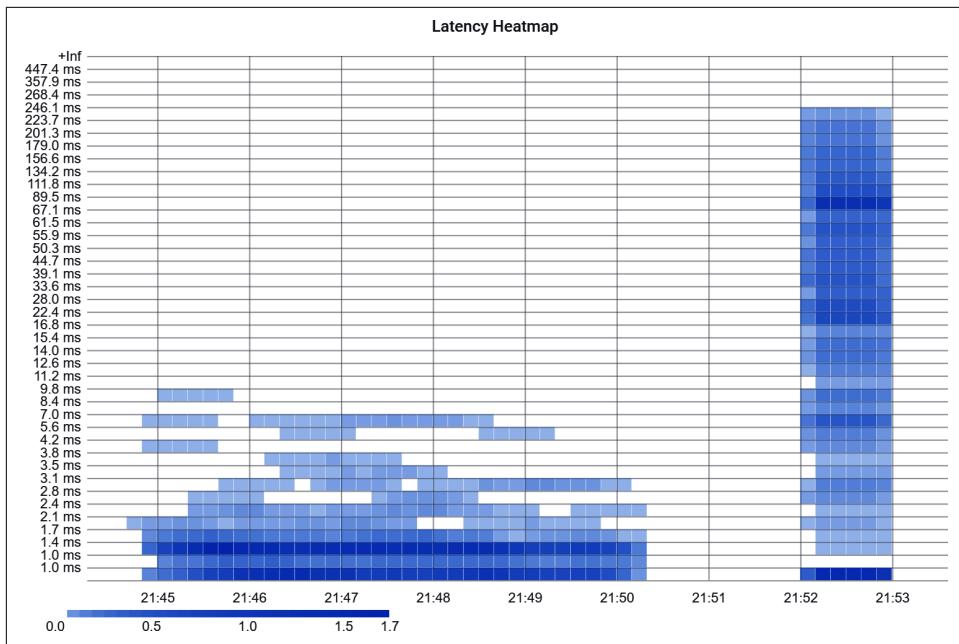


Figure 2-21. Blocking load test versus a nonblocking (reactive) load test

The effect (shown in [Figure 2-22](#)) on the alert criteria suggested in “[Latency](#)” on page [153](#) is noticeable, as expected.

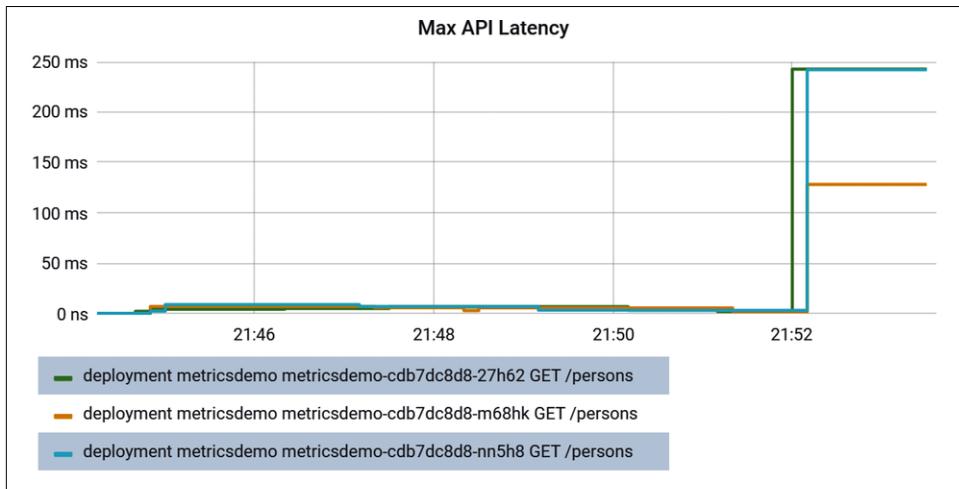


Figure 2-22. Difference in max latency

The effect is also noticeable on the 99th percentile latency indicator, shown in [Figure 2-23](#), so it would show up on the key indicator we use to *compare* the response-time performance of two versions of the same microservice (see “[Automated Canary Analysis](#)” on page 205).

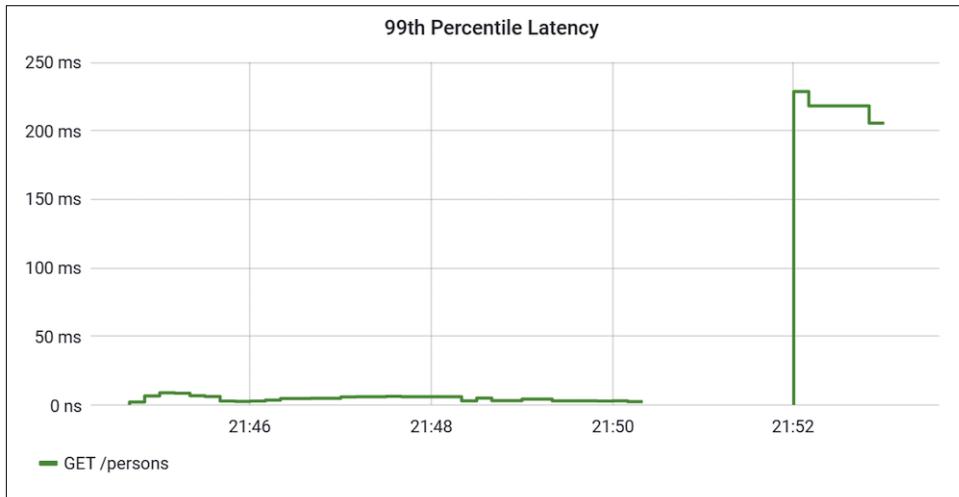


Figure 2-23. Difference in 99th percentile

This example also shows why a server’s view of its own throughput can be misleading. Throughput, shown in [Figure 2-24](#), only increased from approximately 1 ops/second to 3 ops/second between the two tests, but this represents how many requests are being *completed*. In fact, during the period of the reactive load test where the service was oversaturated, many more requests were being queued up on the service’s Tomcat thread pool and not being handled in a timely fashion.

If monitoring throughput (and latency) for the sake of alert criteria in production, it would be better to monitor both from the perspective of the client when possible, as discussed in “[Latency](#)” on page 153.

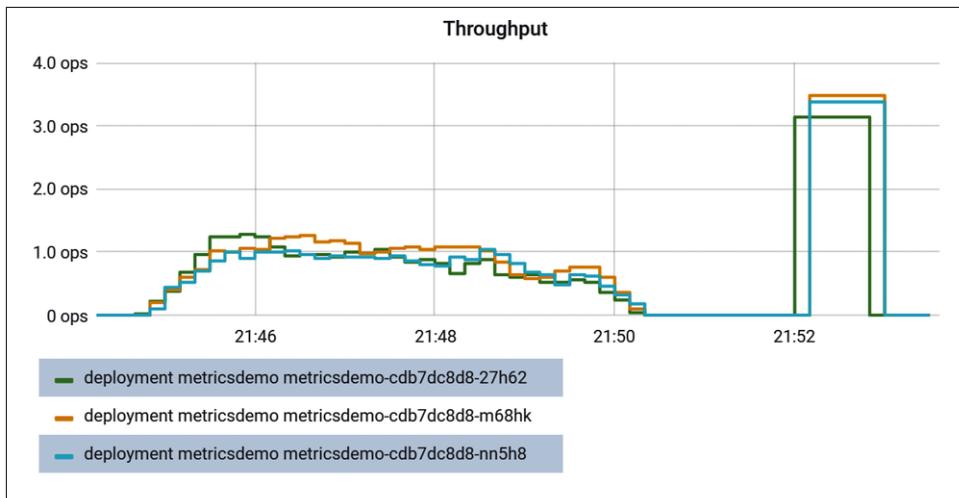


Figure 2-24. Difference in throughput

Lastly, average latency is shown in [Figure 2-25](#). Even though the average has gone up by an order of magnitude between the two tests, average is still around 60 ms, which probably doesn't seem all that bad. Average hides so much of what is really happening that it simply isn't a useful measure.

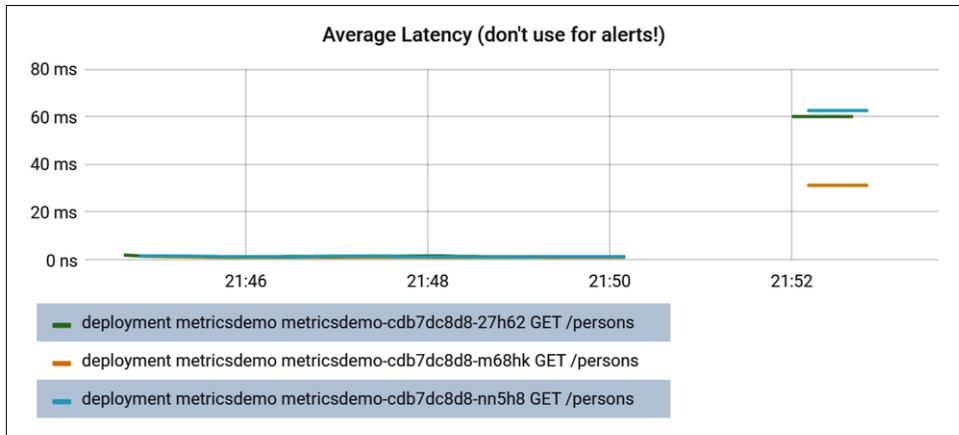


Figure 2-25. Difference in average latency

Now that we've seen many of the meter building blocks and some of the ways they are used, let's turn our focus to application-wide customization of how metrics are shipped.

Meter Filters

The more instrumentation that is added to various parts of the Java stack, the greater the necessity of somehow controlling which metrics are published and at what fidelity. I first became aware of this need chatting with one of the engineers on the Netflix rating team (the ones that control the thumbs up/star rating system on the Netflix UI). He showed me how a particular metric instrumented in a core platform library included by most user-facing microservices was recording roughly 70,000 time series per API endpoint! This was absurdly wasteful, since the vast majority of these time series were not used for many product teams in dashboards and alerts. Instrumentation had been developed for the highest-possible-fidelity use case. This underscores how little choice core library producers really have. They need to instrument for the high fidelity case and rely on somebody downstream of them to tune down this fidelity to something useful for them. From this experience, Micrometer meter filters were born.

Each registry can be configured with meter filters, which allow a great degree of control over how and when meters are registered and what kinds of statistics they emit. Meter filters serve three basic functions:

1. *Deny* (or accept) meters from being registered.
2. *Transform* meter IDs (e.g., changing the name, adding or removing tags, changing description or base units).
3. *Configure* distribution statistics for some meter types.

Implementations of `MeterFilter` are added to the registry programmatically, as in [Example 2-48](#).

Example 2-48. Applying meter filters

```
registry.config()
    .meterFilter(MeterFilter.ignoreTags("too.much.information"))
    .meterFilter(MeterFilter.denyNameStartsWith("jvm"));
```

Meter filters are applied in order, and the results of transforming or configuring a meter are chained.



Apply Meter Filters Early in the Application Life Cycle

For performance reasons, meter filters only influence meters registered *after* the filter.

Deny/Accept Meters

The verbose form of an accept/deny filter is shown in [Example 2-49](#).

Example 2-49. Most verbose form of an accept/deny filter

```
MeterFilter filter = new MeterFilter() {  
    @Override  
    public MeterFilterReply accept(Meter.Id id) {  
        if(id.getName().contains("test")) {  
            return MeterFilterReply.DENY;  
        }  
        return MeterFilterReply.NEUTRAL;  
    }  
}
```

`MeterFilterReply` has three possible states:

DENY

Do not allow this meter to be registered. When you attempt to register a meter against a registry and the filter returns DENY, the registry will return a NOOP version of that meter (e.g., `NoopCounter`, `NoopTimer`). Your code can continue to interact with the NOOP meter, but anything recorded to it is discarded immediately with minimal overhead.

NEUTRAL

If no other meter filter has returned DENY, then registration of meters proceeds as normal.

ACCEPT

If a filter returns ACCEPT, the meter is immediately registered without interrogating any further filters' accept methods.

`MeterFilter` provides several convenience static builders for deny/accept type filters:

`accept()`

Accept every meter, overriding the decisions of any filters that follow.

`accept(Predicate<Meter.Id>)`

Accept any meter matching the predicate.

`acceptNameStartsWith(String)`

Accept every meter with a matching prefix.

`deny()`

Deny every meter, overriding the decisions of any filters that follow.

```
denyNameStartsWith(String)
```

Deny every meter with a matching prefix. All out-of-the-box `MeterBinder` implementations provided by Micrometer have names with common prefixes to allow for easy grouping visualization in UIs, but also to make them easy to disable/enable as a group with a prefix. For example, you can deny all JVM metrics with `MeterFilter.denyNameStartsWith("jvm")`.

```
deny(Predicate<Meter.Id>)
```

Deny any meter matching the predicate.

```
maximumAllowableMetrics(int)
```

Deny any meter after the registry has reached a certain number of meters.

```
maximumAllowableTags(String meterNamePrefix, String tagKey, int maximumTagValues, MeterFilter onMaxReached)
```

Place an upper bound on the number of tags produced by matching series.

Allowlisting only a certain group of metrics is a particularly common case for monitoring systems that are *expensive*. This can be achieved with the static:

```
denyUnless(Predicate<Meter.Id>)
```

Deny all meters that *don't* match the predicate.

Meter filters are applied in the order in which they are configured on the registry, so it is possible to stack deny/accept filters to achieve more complex rules. In [Example 2-50](#), we're explicitly accepting any metric prefixed with `http`, and denying everything else. Because the first filter gives an accept decision on a meter like `http.server.requests`, the universal deny filter is never asked to provide an opinion.

Example 2-50. Accept only HTTP metrics and deny everything else

```
registry.config()
    .meterFilter(MeterFilter.acceptNameStartsWith("http"))
    .meterFilter(MeterFilter.deny());
```

Transforming Metrics

Meter filters can also transform a meter's name, tags, description, and base units. One of the most common applications of transforming metrics is to add common tags. The author of a common Java library like the RabbitMQ Java client cannot possibly guess how you wish to identify RabbitMQ metrics arriving at your monitoring system by application, the deployed environment, the instance that the code is running on, the version of the application, etc. The possibility of applying common tags to all metrics streaming out of an application means that low-level-library authors can keep their instrumentation simple, adding only tags related to the piece they are

instrumenting, e.g., the queue name for RabbitMQ metrics. The application developer can then enrich this instrumentation with other identifying information.

A transform filter is shown in [Example 2-51](#). This filter adds a name prefix and an additional tag conditionally to meters starting with the name “test.”

Example 2-51. Transform meter filter

```
MeterFilter filter = new MeterFilter() {  
    @Override  
    public Meter.Id map(Meter.Id id) {  
        if(id.getName().startsWith("test")) {  
            return id.withName("extra." + id.getName()).withTag("extra.tag", "value");  
        }  
        return id;  
    }  
}
```

`MeterFilter` provides convenience builders for many common transformation cases:

`commonTags(Iterable<Tag>)`

Add a set of tags to all metrics. Adding common tags for app name, host, region, etc., is a highly recommended practice.

`ignoreTags(String...)`

Drop matching tag keys from every meter. This is particularly useful when a tag’s cardinality probably becomes too high and starts stressing your monitoring system or costing too much, but you can’t change all the instrumentation points quickly.

`replaceTagValues(String tagKey, Function<String, String> replacement, String... exceptions)`

Replace tag values according to the provided mapping for all matching tag keys. This can be used to reduce the total cardinality of a tag by mapping some portion of tag values to something else.

`renameTag(String meterNamePrefix, String fromTagKey, String toTagKey)`

Rename a tag key for every metric beginning with a given prefix.

Ignoring one or more of the tags mentioned at the beginning of this section on Netflix core platform instrumentation, which yielded tens of thousands of tags per API endpoint, could have significantly cut down on cost.

Configuring Distribution Statistics

Timer, LongTaskTimer, and DistributionSummary contain a set of optional distribution statistics in addition to the basics of count, total, and max that can be configured through filters. These distribution statistics include precomputed “Percentiles/Quantiles” on page 60, “Service Level Objective Boundaries” on page 69, and “Histograms” on page 65. Distribution statistics can be configured through a MeterFilter, as shown in Example 2-52.

Example 2-52. Configuring distribution statistics

```
new MeterFilter() {
    @Override
    public DistributionStatisticConfig configure(Meter.Id id,
        DistributionStatisticConfig config) {
        if (id.getName().startsWith(prefix)) {
            return DistributionStatisticConfig.builder()
                .publishPercentiles(0.9, 0.95)
                .build()
                .merge(config);
        }
        return config;
    }
};
```

Generally, you should create a new DistributionStatisticConfig with just the pieces you wish to configure and then merge it with the input configuration. This allows you to drop-down on registry-provided defaults for distribution statistics and to chain multiple filters together, each of which configures some part of the distribution statistics (e.g., maybe you want a 100 ms SLO for all HTTP requests but only percentile histograms on a few critical endpoints).

MeterFilter provides convenience builders for the following:

`maxExpected(Duration/long)`

Governs the upper bound of percentile histogram buckets shipped from a timer or summary.

`minExpected(Duration/long)`

Governs the lower bound of percentile histogram buckets shipped from a timer or summary.

Spring Boot offers property-based filters for configuring SLOs, percentiles, and percentile histograms by name prefix, as shown in the following list:

`management.metrics.distribution.percentiles-histogram`

Whether to publish a histogram suitable for computing aggregable (across dimension) percentile approximations.

`management.metrics.distribution.minimum-expected-value`

Publish less histogram buckets by clamping the range of expected values.

`management.metrics.distribution.maximum-expected-value`

Publish less histogram buckets by clamping the range of expected values.

`management.metrics.distribution.percentiles`

Publish percentile values computed in your application.

`management.metrics.distribution.sla`

Publish a cumulative histogram with buckets defined by your SLAs.

Meter filters show up in ways that demonstrate how organizational culture can drive even the lowest level of software configuration. Several organizations use them to separate platform and application metrics, for example.

Separating Platform and Application Metrics

Conway’s Law suggests that “you ship your org chart.” This means roughly that the way in which your system is written, deployed, and works bears some resemblance to your organization.

I’ve found a common pattern that I think is a good positive illustration of this principle. At Netflix, the operations engineering organization (what we are calling platform engineering in this book) built tools that solved problems otherwise undifferentiated among individual microservices. This organization was very much customer-engineer-focused, but it exercised no oversight or control over what individual teams did because of the overarching “freedom and responsibility” culture. But at many organizations (and I don’t think this is necessarily worse or better), platform teams do act centrally on behalf of product teams. So while the monitoring of an individual microservice at Netflix was wholly the responsibility of the product team (with as much advice and assistance as requested from the central team), at many organizations the responsibility for monitoring applications may reside wholly with a central platform team.

In other cases, the responsibility is split, with a central platform team responsible for monitoring certain types of signals (usually resource metrics like processor, memory, and maybe closer-to-business metrics like API error ratio) and application teams responsible for any custom indicators. You may notice how the responsibilities in this case roughly break down along the lines of what black box and white box instrumentation each excel at (with platform teams monitoring black box signals and product

teams monitoring white box signals). This is a good illustration of how agent-based instrumentation shipped to some particular SaaS product may serve the needs of the platform team while product teams use an entirely different monitoring system for other signals—i.e., how these types of instrumentation can be complementary rather than competitive.

Let's consider an organization like this, and the impact it has on how metrics telemetry is configured in the application. To make this concrete, assume this organization is using Prometheus as its monitoring system. A platform engineer and a product engineer have different goals in this case:

Platform engineer

A platform engineer wants to monitor all microservices across the organization in the same way. This means that every microservice should publish a set of metrics the platform team intends to monitor. These metrics should have a consistent way of determining common tags (e.g., for the stack like test/dev/prod, region, cluster, server group name, application version, and an instance identifier). There is no value to the platform team in shipping anything more than the set of metrics the platform team intends to monitor. This set of metrics is unlikely to change significantly over time, because by the nature of the responsibility of the platform team, they represent general indicators that are useful to all applications and aren't tied to specific business functions or features.

Product engineer

A product engineer wants to monitor their microservice(s) alone. The same set of common tags applicable to the platform engineer are likely beneficial to the product engineer as well, but there may be additional common tags that further differentiate individual instances at a level that isn't relevant to the platform team. For example, the application team may deploy several clusters of its microservice that could contain the same application code but serve different segments of its end users (e.g., internal users versus external users). They will likely want to add a common tag for this distinction to their metrics as well, as there may be different SLOs for different end-user populations. The metrics that a product engineer should focus most on will be more specific to end-user experience. They are also likely to be feature-focused. As new features change, the set of metrics changes as well.

If metrics for a microservice were published through a single `MeterRegistry`, there is a chance that product engineer customizations to the registry would impact the observability of the microservice for the platform team. And, depending on how metrics tags are being aggregated for display by the product team, platform engineers adding additional common tags could impact the alerts and dashboards of product engineers.

Because the engineering organization is structured in this way with this division of responsibilities across team boundaries, it should come up with a way to split the publishing of metrics into distinct meter registries that best serve the individual responsibilities of platform and product teams. “You ship your org chart.”

Since this sort of division of responsibility is fairly common, let’s consider how this would be achieved. First, the platform team becomes responsible for shipping a common library, a JAR binary dependency that every microservice can include that contains this common configuration. Because presumably microservice teams will be on different release cycles, the platform team will naturally have to evolve this common configuration slowly, relative to the pace of change of any individual microservice. In this common platform JAR, we’d expect to see autoconfiguration like in [Example 2-53](#).

Example 2-53. Platform team’s autoconfiguration of metrics shared with product teams

```
@Configuration
public class PlatformMetricsAutoConfiguration {
    private final Logger logger = LoggerFactory.getLogger(
        PlatformMetricsAutoConfiguration.class);

    private final PrometheusMeterRegistry prometheusMeterRegistry =
        new PrometheusMeterRegistry(PrometheusConfig.DEFAULT); ①

    @Value("${spring.application.name:unknown}")
    private String appName;

    @Value("${HOSTNAME:unknown}")
    private String host;

    public PlatformMetricsAutoConfiguration() { ②
        new JvmGcMetrics().bindTo(prometheusMeterRegistry);
        new JvmHeapPressureMetrics().bindTo(prometheusMeterRegistry);
        new JvmMemoryMetrics().bindTo(prometheusMeterRegistry);
        new ProcessorMetrics().bindTo(prometheusMeterRegistry);
        new FileDescriptorMetrics().bindTo(prometheusMeterRegistry);
    }

    @Bean
    @ConditionalOnBean(KubernetesClient.class)
    MeterFilter kubernetesMeterFilter(KubernetesClient k8sClient) { ③
        MeterFilter k8sMeterFilter = new KubernetesCommonTags();
        prometheusMeterRegistry.config().meterFilter(k8sMeterFilter);
        return k8sMeterFilter;
    }

    @Bean
    @ConditionalOnMissingBean(KubernetesClient.class)
    MeterFilter appAndHostTagsMeterFilter() {
```

```

        MeterFilter appAndHostMeterFilter = MeterFilter.commonTags(
            Tags.of("app", appName, "host"));
        prometheusMeterRegistry.config().meterFilter(appAndHostMeterFilter);
        return appAndHostMeterFilter;
    }

    @RestController
    class PlatformMetricsEndpoint {
        @GetMapping(path = "/platform/metrics", produces = TextFormat.CONTENT_TYPE_004)
        String platformMetrics() { ④
            return prometheusMeterRegistry.scrape();
        }
    }
}

```

- ❶ In the platform team's configuration, a Prometheus meter registry can be created privately, without adding it to the Spring application context, so it is not subject to `MeterFilter`, `MeterBinder`, and other customizations that the product team might configure via the application context.
- ❷ The metrics that the platform team cares about are added directly to the privately configured registry.
- ❸ The platform team provides a common tag meter filter that works for every microservice running in Kubernetes. This practice is a concern cutting all micro-service teams, and they all benefit from the same set of common tags (though they may add their own additionally). The meter filter is also applied right after construction to the private platform meter registry. A fallback set of common tags is provided for when the application isn't running in Kubernetes.
- ❹ The platform team sets up an API endpoint for itself, common to every application and distinct from the typical `/actuator/prometheus` endpoint that Spring would autoconfigure, leaving the actuator endpoint to be under the full control of the product team for its own purposes.

The implementation of Kubernetes common tags, shown in [Example 2-54](#), is applicable to the kinds of annotations that Spinnaker's Kubernetes implementation configures to be placed on pods. Spinnaker will be discussed in greater detail in [Chapter 5](#).

Example 2-54. Kubernetes common tags, assuming the service was deployed by Spinnaker

```

public class KubernetesCommonTags implements MeterFilter {
    private final Function<Meter.Id, Meter.Id> idMapper;

    public KubernetesCommonTags(KubernetesClient k8sClient) {
        try {

```

```

Map<String, String> annotations = k8sClient.pods()
    .withName(host)
    .get()
    .getMetadata()
    .getAnnotations();

for (Map.Entry<String, String> annotation : annotations.entrySet()) {
    logger.info("Kubernetes pod annotation <" + annotation.getKey() +
        "=" + annotation.getValue() + ">");
}

idMapper = id -> id.withTags(Tags.of(
    "revision",
    annotations.getOrDefault("deployment.kubernetes.io/revision", "unknown"),
    "app",
    annotations.getOrDefault("moniker.spinnaker.io/application", appName),
    "cluster",
    stream(annotations
        .getOrDefault("moniker.spinnaker.io/cluster", "unknown")
        .split(" "))
        .reduce((first, second) -> second).orElse("unknown"),
    "location",
    annotations.getOrDefault("artifact.spinnaker.io/location", "unknown"),
    "host", host
));
} catch (KubernetesClientException e) {
    logger.warn("Unable to apply kubernetes tags", e);
    idMapper = id -> id.withTags(Tags.of(
        "app", appName,
        "host", host,
        "cluster", "unknown")
    );
}
}

@Override
public Meter.Id map(Meter.Id id) {
    return idMapper.apply(id);
}
}

```

Another use case of meter filters involves adding a layer of resiliency to your publication of metrics themselves.

Partitioning Metrics by Monitoring System

Suppose your organization has selected Prometheus as its primary monitoring system, and a dedicated platform engineering team is operating a series of Prometheus instances which should be pulling metrics from every application in the company's inventory. Perhaps this platform team even contributes to monitoring some set of

common indicators, on behalf of product teams, known to be broadly applicable to every Java microservice running in the organization.

How do these engineers prove that the Prometheus instances they think they are operating effectively are in fact scraping all the deployed assets successfully? There are different failure modes. In one case, Prometheus could simply time out attempting to pull metrics from a given application. This would be visible from Prometheus's monitoring of itself. But another failure mode might be that we have misconfigured Prometheus so that it isn't even *attempting* to scrape this application. In this case, Prometheus dutifully scrapes all the applications it knows about and reports nothing about the applications it doesn't.

Suppose also your organization is running its infrastructure on AWS. We could choose to dual-publish metrics, both to our primary monitoring system Prometheus and to AWS Cloudwatch. A little investigation shows that Cloudwatch (like other public cloud-provider-hosted monitoring solutions) is fairly expensive, billed by the number of time series sent. But we really only want to use Cloudwatch to verify that Prometheus is doing what it should.

The overall process looks like [Figure 2-26](#). The Prometheus team routinely queries the state of the deployed asset inventory (maybe through a stateful delivery automation solution as described in [Chapter 5](#)). For each application listed, the team can check for the counter that Micrometer maintains of Prometheus's scrape attempts for the application. An application that isn't being scraped will have a zero or empty counter.

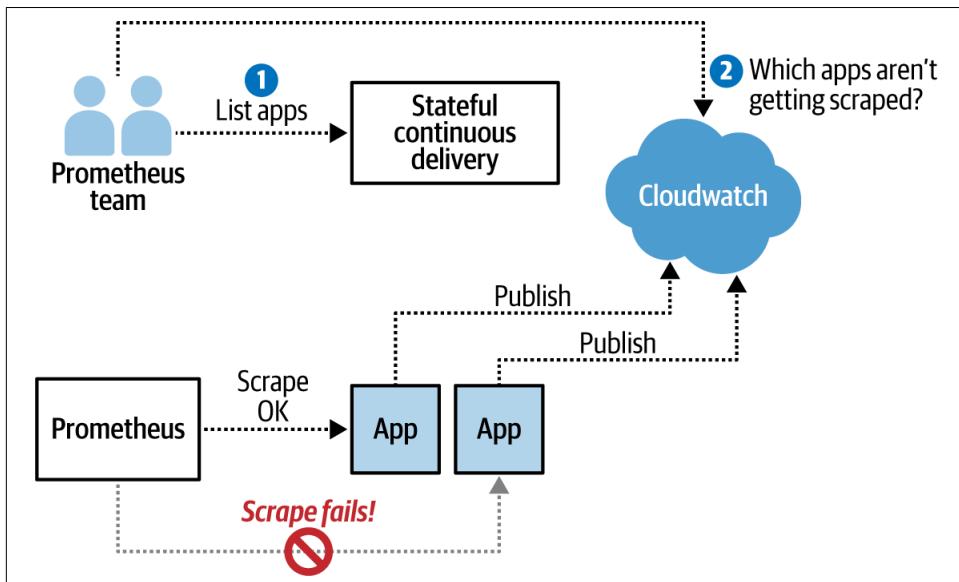


Figure 2-26. Shipping metrics to both Prometheus and Cloudwatch

Example 2-55 shows how we can use an accept and deny MeterFilter pair to cost-effectively ship to Cloudwatch only the Prometheus metrics that are relevant in helping the platform team determine that the Prometheus scrape configuration is working as expected. It uses a Spring Boot feature called `MeterRegistryCustomizer` that allows us to add filter and other registry customization to specific registry types rather than to all of them.

Example 2-55. Using Cloudwatch MeterRegistryCustomizer

```
@Bean
MeterRegistryCustomizer<CloudwatchMeterRegistry> cloudwatchCustomizations() {
    return registry -> registry.config()
        .meterFilter(MeterFilter.acceptNameStartsWith("prometheus"))
        .meterFilter(MeterFilter.deny());
}
```

There is one last concept related to the organization of metrics.

Meter Binders

In many cases monitoring some subsystem or library involves more than one meter. Micrometer provides a simple functional interface called a `MeterBinder` that is designed for encapsulating a set of meters together. Spring Boot automatically registers the metrics from any `MeterBinder` bean that is configured to the application context (i.e., `@Bean MeterBinder ...`). **Example 2-56** illustrates a simple meter binder that encapsulates some metrics around a vehicle type.

Example 2-56. Meter binder implementation

```
public class VehicleMeterBinder implements MeterBinder {
    private final Vehicle vehicle;

    public VehicleMeterBinder(Vehicle vehicle) {
        this.vehicle = vehicle;
    }

    @Override
    public void bindTo(MeterRegistry registry) {
        Gauge.builder("vehicle.speed", vehicle, Vehicle::getSpeed)
            .baseUnit("km/h")
            .description("Current vehicle speed")
            .register(registry);

        FunctionCounter.builder("vehicle.odometer", vehicle, Vehicle::readOdometer())
            .baseUnit("kilometers")
            .description("The amount of distance this vehicle has traveled")
            .register(registry);
    }
}
```

```
    }  
}
```

It is best if all metrics registered in a meter binder share some common prefix (in this case “vehicle”), especially when the binder is packed up and shipped as default configuration across a wide array of applications. Some teams may not find the metrics to be useful in their specific case and filter them out to save on cost. Having a common prefix makes it easy to apply a deny meter filter by the common prefix, as in [Example 2-57](#). In this way, you can add and remove metrics from the meter binder over time, and the filter logic still has the effect of broadly including or excluding the metrics produced by this meter binder.

Example 2-57. A property-based deny filter for metrics coming from the vehicle meter binder

```
management.metrics.enable.vehicle: false
```

Summary

In this chapter, you’ve learned how to measure various parts of your application with dimensional metrics. We haven’t yet discussed specifically how to *use* this data. We will eventually be coming back to metrics in [Chapter 4](#), where effective indicators for every Java microservice are presented along with how to build effective charts and alerts.

The organizational commitment you are signing up for to take advantage of all this dimensional metrics data involves the selection of one or more target dimensional monitoring systems, either picking a SaaS offering or standing up one of the available OSS systems on-prem. The impact on your code is limited. When using modern Java web frameworks like Spring Boot, prepackaged instrumentation will provide a great deal of detail without having to write any custom instrumentation code. All you need to do is add a dependency on the monitoring system implementation of your choice and then provide some configuration to ship metrics to it.

In the next chapter, we’ll see how metrics instrumentation compares to debuggability signals like distributed traces and logs. While reading it, keep in mind that the metrics instrumentation we’ve just discussed is designed to provide fixed-cost telemetry that helps you to understand what is happening to your system in the *aggregate*. These other telemetry sources will provide detailed information about what is happening at an individual event (or request) level.

Debugging with Observability

As mentioned at the beginning of [Chapter 2](#), observability signals can be roughly broken down into two categories, based on the value they bring: availability and debuggability. Aggregated application metrics provide the best availability signal. In this chapter, we will discuss the other two main signals, distributed tracing and logs.

We'll show one approach to correlating metrics and traces using only open source tooling. Some commercial vendors also work to provide this unified experience. Like in [Chapter 2](#), the purpose in showing a specific approach is to develop an expectation about the minimum level of sophistication you should be able to *expect* from your observability stack when it is fully assembled.

Lastly, distributed tracing instrumentation, given that it needs to propagate context across a microservice hierarchy, can be an efficient place to govern behavior deeper in a system. We'll discuss a hypothetical failure injection testing feature as an example of the possibilities.

The Three Pillars of Observability...or Is It Two?

As discussed in [*Distributed Systems Observability*](#) by Cindy Sridharan (O'Reilly), three different types of telemetry form the “three pillars of observability”: logs, distributed traces, and metrics. This three pillars classification is common, to such an extent that it's difficult to pinpoint its origin.

While logs, distributed traces, and metrics are three distinct forms of telemetry with unique characteristics, they roughly serve two purposes: proving availability and debugging for root cause identification.

The operational cost of maintaining all this telemetry data would be high unless the volume of data is reduced in some way. Clearly we can only maintain telemetry data for a certain amount of time, so there is a need for other reduction strategies.

Aggregation

Precalculating statistics from every measurement. Data from timers (see “[Timers](#)” on page 45), for example, could be presented as a sum, a count, and some limited distribution statistics.

Sampling

Selecting only certain measurements for retention.

Aggregation effectively compacts the representation at the expense of request-level granularity, where sampling retains request-level granularity at the expense of the holistic view of the system’s performance. Except in low-throughput systems, the cost of both full request-level granularity and a full representation of all requests is too expensive.

Retaining some information at full granularity is essential to *debugging* with observability tools, the focus of this chapter. Availability signals derived from metrics will point to a problem. Dimensional exploration of this data may in some cases be enough to identify the root cause of the issue. For example, breaking a particular signal into individual signals by instance may reveal a particular instance that is failing. There could be a whole region failing or an application version. In the rare cases where a pattern isn’t obvious, representative failures in distributed traces or logs will be the key to root cause identification.

Considering the characteristics of each of the “three pillars” shows that logging and tracing as event-level telemetry serve the purpose of debugging and metrics serve the purpose of proving availability.

Logs

Logs are ubiquitous in the software stack. Regardless of their structure and where they are ultimately stored, logs have some defining characteristics.

Logs grow proportionally to throughput through a system. The more times a log-instrumented code path is executed, the more log data is emitted. Even if log data is sampled, this proportional relationship in size still holds.

The context of a log is scoped to an event. Log data provides context into the execution behavior of a particular interaction. When data from several independent log events is aggregated together to reason about the overall performance of a system, the aggregate is effectively a metric.

Logs are obviously geared toward debugging. Sophisticated log analytics packages help to prove availability from logging data only through aggregation. There is a cost

to performing this aggregation, to persisting the data subject to aggregation, and to allocating the payload that was persisted.

Distributed Tracing

Tracing telemetry, like logs, is recorded per instrumented execution (i.e., it is event-driven) but links individual events across disparate parts of a system causally. A distributed tracing system can reason about a user interaction end to end across the whole system. So for a given request that was known to exhibit some degradation, this end-to-end view of the satisfaction of a user request shows which part of the distributed system was degraded.

Tracing telemetry is even more commonly sampled than logs. Nevertheless, tracing data still grows proportionally to throughput through a system in much the same way that log data grows proportionally to throughput.

Tracing can be difficult to retrofit into an existing system, as each collaborator in an end-to-end process must be configured to propagate trace context forward.

Distributed tracing shines especially for a particular type of performance problem where the entire system is slower than it should be but there is no obvious hotspot to quickly optimize. Sometimes you simply have to see the contribution of many subsystems to the performance of a system as whole to recognize that systemic “death by a thousand cuts” that needs to be visualized in order to build the organizational will to address, and thus the attention of time and resources.

“It’s slow” is the hardest problem you’ll ever debug. “It’s slow” might mean one or more of the number of systems involved in performing a user request is slow. It might mean one or more of the parts of a pipeline of transformations across many machines is slow. “It’s slow” is hard, in part, because the problem statement doesn’t provide many clues to the location of the flaw. Partial failures, ones that don’t show up on the graphs you usually look up, are lurking in a dark corner. And, until the degradation becomes very obvious, you won’t receive as many resources (time, money, and tooling) to solve it. Dapper and Zipkin were built for a reason.

—Jeff Hodges

In organizations with a large set of microservices, distributed tracing helps to understand the service graph (the dependencies between services themselves) involved in the processing of a certain type of request. This assumes, of course, that each service in the graph has tracing instrumentation in one form or another. In the narrowest sense, the last tier of services can be uninstrumented and still appear in the service graph if named by a span wrapping a call on the client side.

Distributed tracing, like logging, is inherently event-driven and so is best suited to act as a debugging signal, but one that carries important interservice relational context in addition to its tags.

Metrics

Logs and distributed traces are more similar to each other than they are to metrics, which were discussed in detail in [Chapter 2](#), since in some ways they are both sampled to control cost. Metrics are presented in aggregate and are used to understand some service level indicator (SLI) as a whole, rather than providing detail about the individual interactions that, taken together, are measured as an SLI.

Retrofitting an existing codebase with metrics is partially a manual effort, and partially comes out of the box with improvements in common frameworks and libraries which are increasingly shipping with instrumentation.

Metrics SLIs are purposefully collected to be tested against a service level objective, so they are geared toward proving availability.

Which Telemetry Is Appropriate?

Given this context about what each form of observability is intended for, consider how they overlap. Where they overlap, which form do we emphasize over another?

The idea that both tracing and logging are debugging signals implies that they can be redundant, though not equal. All things being equal about retrieving and searching for them, a trace with effective tags and metadata is superior to a log line when it also provides useful context about the chain of calls that led to it (and propagates this context further along as well).

Tracing instrumentation exists at all the same logical places where metrics timers do. Note that distributed traces *only* measure executions, though. Where execution timing is concerned, metrics and tracing instrumentation may both be appropriate because they complement one another. Metrics provide an aggregated view of all executions of the instrumented piece of code (and without caller context), and distributed tracing provides sampled examples of individual executions. In addition to timing executions though, metrics also count and gauge things. There are no tracing equivalents for these kinds of signals.

To make this more concrete, let's take a look at excerpts from a typical application log in [Example 3-1](#). Much of the beginning of this log excerpt contains one-time events about which components were configured and features were enabled. These pieces of information may be important in understanding why the application isn't functioning as expected (for example, if a component was expected to be configured but wasn't), but they don't make sense as metrics because they aren't recurring events that need to be aggregated over time to understand the overall performance of the system. They don't make sense as distributed traces either, because these are events that are specific to the state of this service alone and have nothing to do with the coordinated satisfaction of an end-user request across multiple microservices.

There are other log lines that could be replaced with tracing or metrics, as noted in the callouts following the example.

Example 3-1. A typical application log demonstrating telemetry choices

```
. /\\ / ____' - _ _ - _(_)_ - _ _ - \ \ \ \ \
( ( )\__| '_| '_| | '_\ \ / _` | \ \ \ \
\ \ \ \_)| |_)| | | | | | ( | | ) ) ) )
' |____| ._|_|_|_|_|_|_|_\_, | / / /
=====|_|=====|_|=/_|/_/
:: Spring Boot ::      (v...RELEASE)

:56:56 main INFO c.m.MySampleService - Starting MySampleService on
HOST with PID 12624
:56:56 main INFO c.m.MySampleService - The following profiles are active: logging
:56:56 main INFO o.s.b.c.e.AnnotationConfigEmbeddedWebApplicationContext - Refresh
    org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplication
    Context@2a5c8d3f: startup date [Tue Sep 17 14:56:56 CDT]; root of context
:56:57 background-preinit INFO o.h.v.i.util.Version - HV000001: Hibernate Validator
    5.3.6.Final
:57:02 main INFO o.s.b.c.e.t.TomcatEmbeddedServletContainer - Tomcat initialized
    with port(s): 8080 (http)
:57:03 localhost-startStop-1 INFO i.m.c.i.l.LoggingMeterRegistry - publishing
    metrics to logs every 10s
:57:07 localhost-startStop-1 INFO o.s.b.a.e.m.EndpointHandlerMapping - Mapped
    "[/{env}/{name: *}],methods=[GET],produces=[application/
    vnd.spring-boot.actuator.v1+json || application/json]}" onto public
    java.lang.Object org.springframework.boot.actuate.endpoint.mvc.
    EnvironmentMvcEndpoint.value(java.lang.String)
:57:07 localhost-startStop-1 INFO o.s.b.w.s.FilterRegistrationBean - Mapping filter:
    'metricsFilter' to: [/*]
:57:11 main INFO o.mongodb.driver.cluster - Cluster created with settings
    {hosts=[localhost:27017], mode=SINGLE, requiredClusterType=UNKNOWN,
    serverSelectionTimeout='30000 ms', maxWaitQueueSize=500}
:57:12 main INFO o.s.b.a.e.j.EndpointMBeanExporter - Registering beans for JMX
    exposure on startup
:57:12 main INFO o.s.b.a.e.j.EndpointMBeanExporter - Located managed bean
    'healthEndpoint': registering with JMX server as MBean
    [org.springframework.boot:type=Endpoint,name=healthEndpoint]
:57:12 main INFO o.s.b.c.e.t.TomcatEmbeddedServletContainer - Tomcat started on
    port(s): 8080 (http)
:57:13 cluster-ClusterId{value='5d813a970df1cb31507adbc2', description='null'}-
    localhost:27017 INFO o.mongodb.driver.cluster - Exception in monitor thread
    while connecting to server localhost:27017
com.mongodb.MongoSocketOpenException: Exception opening socket ①
    at c.m.c.SocketStream.open(SocketStream.java:63)
    at c.m.c.InternalStreamConnection.open(InternalStreamConnection.java:115)
    at c.m.c.DefaultServerMonitor$ServerMonitorRunnable.run(
        DefaultServerMonitor.java:113)
    at java.lang.Thread.run(Thread.java:748)
```

```

Caused by: j.n.ConnectException: Connection refused: connect
at j.n.DualStackPlainSocketImpl.waitForConnect(Native Method)
at j.n.DualStackPlainSocketImpl.socketConnect(
    DualStackPlainSocketImpl.java:85)
at j.n.AbstractPlainSocketImpl.doConnect(AbstractPlainSocketImpl.java:350)
at j.n.AbstractPlainSocketImpl.connectToAddress(
    AbstractPlainSocketImpl.java:206)
at j.n.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:188)
at j.n.PlainSocketImpl.connect(PlainSocketImpl.java:172)
at j.n.SocksSocketImpl.connect(SocksSocketImpl.java:392)
at j.n.Socket.connect(Socket.java:589)
at c.m.c.SocketStreamHelper.initialize(SocketStreamHelper.java:57)
at c.m.c.SocketStream.open(SocketStream.java:58)
... 3 common frames omitted
:57:13 main INFO c.m.PaymentsController - [GET] Payment 123456 retrieved in 37ms. ②
:57:13 main INFO c.m.PaymentsController - [GET] Payment 789654 retrieved in 38ms
... (hundreds of other payments retrieved in <40ms)
:57:13 main INFO c.m.PaymentsController - [GET] Payment 567533 retrieved in 342ms.
:58.00 main INFO c.m.PaymentsController - Payment near cache contains 2 entries. ③

```

- ➊ *Both metrics and logging, no tracing.* Mongo socket connection attempts could easily be timed with metrics, with a tag indicating success/failure and a tag with a summarized exception tag like `exception=ConnectException`. This summarized tag may be enough to understand the problem without viewing the whole stack trace. In other cases, where the summarized exception tag is something like `exception=NullPointerException`, logging the stack trace helps identify the specific problem once the monitoring system alerts us to a grouping of exceptions that have failed an established service level objective.
- ➋ *Both traces and metrics, no logging.* The log statement in the code could be removed entirely. Metrics and distributed traces capture all the interesting information about this payment in a way that allows us to reason about the retrieval of all payments holistically, as well as representative retrievals of individual payments. The metrics, for example, will show us that while most payments are retrieved in less than 40 ms, some will take an order of magnitude longer to retrieve.
- ➌ *Metrics, no traces or logs.* A near cache of frequently retrieved payments could be monitored strictly with a gauge metric. There is no equivalent to a gauge in tracing instrumentation, and logging this is redundant.



Which Observability Tool Should You Choose?

Tracing is preferable to logging whenever possible because it can contain the same information but with richer context. Where tracing and metrics overlap, start with metrics because the first task should be *knowing* when some system is unavailable. Adding additional telemetry to help remediate problems can come later. When you do add tracing, start at places where timed metrics instrumentation exists, because it is likely also worth tracing with a superset of the same tags.

Supposing then that you are ready to add distributed tracing, let's next consider what makes up a trace and how it might be visualized.

Components of a Distributed Trace

A full distributed trace is a collection of individual *spans*, which contain information about the performance of each touchpoint in the satisfaction of an end-user request. These spans can be assembled into an “icicle” graph that shows relatively how much time was spent in each service, as shown in [Figure 3-1](#).

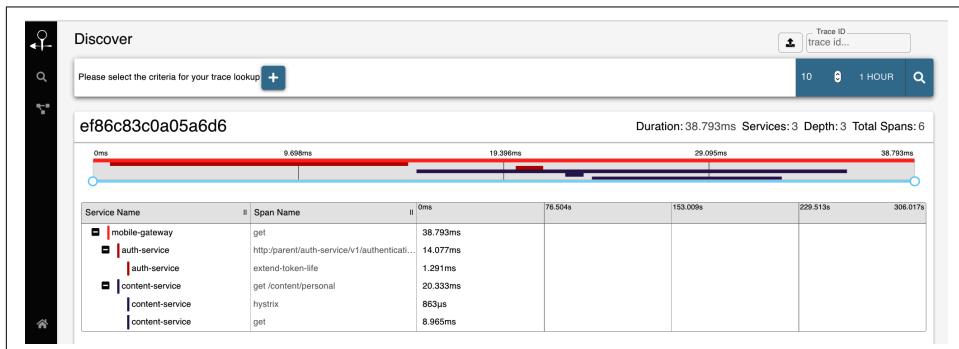


Figure 3-1. Zipkin icicle graph

Spans contain a name and set of key-value tag pairs, much like metrics instrumentation does. Many of the principles we covered in [“Naming Metrics” on page 31](#) apply equally to distributed traces. So if a trace span is named `http.server.requests`, then tags may identify region (in the public cloud sense), API endpoint, HTTP method, response status code, etc. Keeping metric and trace naming consistent is the key to allowing for correlation of telemetry (see [“Correlation of Telemetry” on page 118](#)).

Unlike in metrics, the Zipkin span data model contains special fields for service name (used in the Zipkin Dependencies view, displaying a service graph). This is equivalent to tagging metrics with an application name, where most metrics backends don't set aside a reserved tag name for this concept. Span name is also a defined field on the

Zipkin data model. Both are indexed for lookup, so unbounded value set cardinality on span and service name should be avoided.

Unlike metrics, it is not necessary to control tag cardinality of a trace in every circumstance. This has to do with the way traces are stored. [Table 2-1](#) showed how metrics are stored logically in rows by unique ID (combination of name and key/value tags). Additional measurements are stored as samples in an existing row. The cost of metrics is then the product of the total number of IDs and the amount of samples maintained per ID. Distributed trace spans are stored individually without any regard to whether another span had the same name and tags. The cost of distributed traces is then the product of the throughput through the system and the sampling rate (viewed as a percentage).

While tag cardinality does not influence storage cost in a distributed tracing system, it does influence *lookup* cost. And in tracing systems, tags can be marked as indexable by the tracing backend (and autocompletable in the Zipkin UI). Clearly, these tag value sets should be bounded for index performance.

It's best to overlap as many tags as possible between metrics and traces so that they can later be correlated. You should also tag distributed traces with additional high-cardinality tags that can be used to locate a request from a particular user or interaction, as in [Table 3-1](#). Strive for the value to match wherever tag keys match.

Table 3-1. Overlap of distributed trace and metrics tagging

Metric tag key	Trace tag key	Value
application	application	payments
method	method	GET
status	status	200
uri	uri	/api/payment/{paymentId}
	detailedUri	/api/payment/abc123
	user	user123456

It should be clear at this point that a trace is designed to give you insight into the end-to-end performance of a request. It shouldn't be surprising then that the Zipkin UI is focused on searching for traces given a set of parameters, as shown in [Figure 3-2](#). This kind of listing trades off an understanding of the overall distribution of end-to-end performance for a matching set of traces for a particular set of parameters. Building a correlation between the overall distribution and this view is the subject of "[Correlation of Telemetry](#)" on page 118.

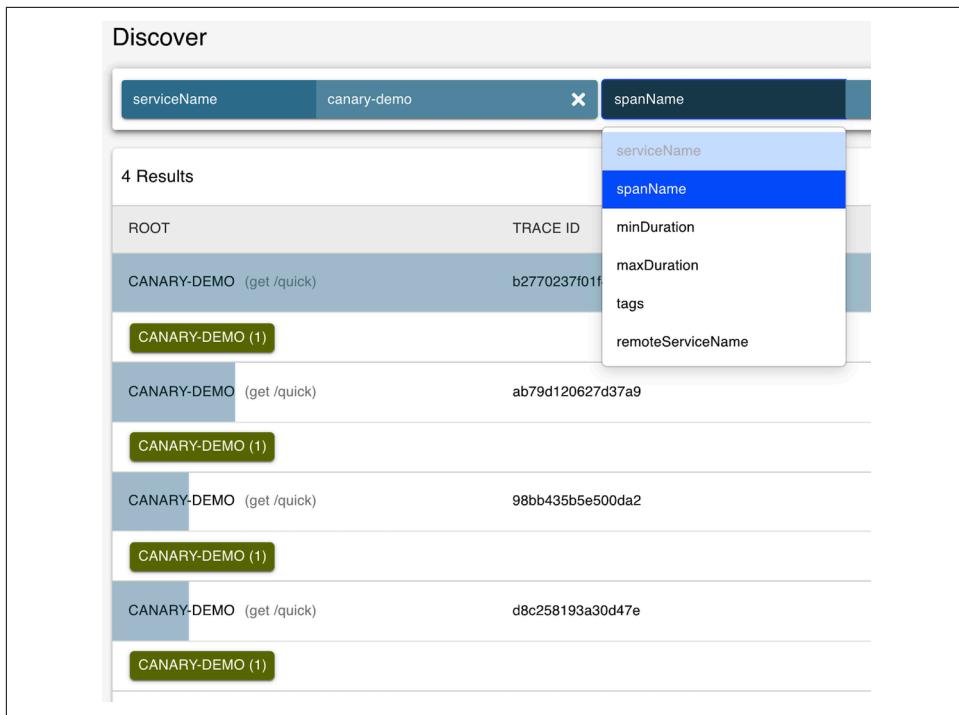


Figure 3-2. Searching for traces in the Zipkin Lens UI

As with metrics, there are multiple ways of adding distributed tracing to your application. Let's consider some of the advantages of each.

Types of Distributed Tracing Instrumentation

Everything discussed in “[Black Box Versus White Box Monitoring](#)” on page 24 with respect to metrics also applies to distributed tracing instrumentation. Tracing instrumentation is available at various architectural levels (from infrastructure to individual components of an application).

Manual Tracing

Libraries like Zipkin’s Brave or OpenTelemetry allow you to instrument your application explicitly in code. In an ideally traced distributed system, some level of manual tracing will certainly be present. Through it, key business-specific context can be added to traces that other forms of prepackaged instrumentation couldn’t possibly be aware of.

Agent Tracing

Like with metrics, agents (typically vendor-provided) can automatically add tracing instrumentation without making code changes. Attaching an agent is a change to your application delivery pipeline, and this complexity cost shouldn't be ignored.

This cost is real no matter which level of abstraction your platform operates at:

- For an infrastructure-as-a-service platform like Amazon EC2, you will have to add the agent and its configuration to your base Amazon Machine Image.
- For a container-as-a-service (CaaS) platform, you will need another container level between a basic image like `openjdk:jre-alpine` and your application. This impact can leak into your build then. If you were using the Gradle `com.bmuschko.docker-spring-boot-application` plug-in to package Spring Boot applications for deployment to a CaaS, you now need to override the default container image with one including the agent. Also, any time the base image (which may very well be the default of `com.bmuschko.docker-spring-boot-application`) of the container image containing the agent is updated, you have to publish a new image.
- For a platform as a service (PaaS) like Cloud Foundry or Heroku, you have to use a custom base unless integration with the agent is specifically supported by the PaaS vendor already.

Framework Tracing

Frameworks can also include telemetry out of the box. Because frameworks are included in an application as a binary dependency, this form of telemetry is technically a black box solution. Framework-level instrumentation can have a white box feel to it when it allows for user-provided customizations to its automatically instrumented touchpoints.

Frameworks are aware of their own implementation idiosyncrasies, so they can provide rich contextual information as tags.

To give an example, framework instrumentation for an HTTP request handler can tag the span with a parameterized request URI (i.e., `/api/customers/(id)` versus `/api/customers/1`). Agent instrumentation would have to be aware of and switch over all supported frameworks to provide the same level of richness and keep up with changes to frameworks individually.

Another complication comes from increasingly prevalent asynchronous workflows in modern programming paradigms like reactive. A proper tracing implementation requires in-process propagation, which can be tricky in reactive contexts where

you can't just stuff context into a `ThreadLocal`. Also, dealing with **mapped diagnostic contexts** to correlate logs and tracing can be tricky in those same contexts.

Retrofitting an existing application with framework-level instrumentation can be relatively low touch. For example, Spring Cloud Sleuth adds tracing telemetry to an existing Spring Cloud—based application. You just need an additional dependency as in [Example 3-2](#) and a bit of configuration as in [Example 3-3](#), the latter of which can be done cross-organizationally if you are already using a centralized dynamic configuration server like Spring Cloud Config Server.

Example 3-2. Sleuth runtime dependency in Gradle build

```
dependencies {  
    runtimeOnly("org.springframework.cloud:spring-cloud-starter-zipkin") ❶  
}
```

- ❶ Note that the `io.spring.dependency-management` plug-in is responsible for adding the version to this dependency specification.

Example 3-3. Sleuth configuration in Spring Boot's application.yml

```
spring.zipkin.baseUrl: http://YOUR_ZIPKIN_HOST:9411/
```

Service Mesh Tracing

The service mesh is an infrastructure layer outside of application code that manages interaction between microservices. Many implementations accomplish this through sidecar proxies in some way associated with the application process.

In some ways, this form of instrumentation isn't much different from how the framework might accomplish it, but don't be fooled into thinking they are equal. They are similar in the point of instrumentation (decorating an RPC call). The framework will certainly have *more* information than the service mesh. For example, for REST endpoint tracing, the framework has access to exception details that are mapped in a lossy way to one of a small set of HTTP status codes. The service mesh would only have access to the status code. The framework has access to the unsubstituted path of the endpoint (e.g., `/api/person/{id}` instead of `/api/person/1`).

Agents also have more potential for richness than a sidecar because they can reach down into individual method invocations, a finer level of granularity than RPC calls.

Adding service mesh not only changes your delivery pipeline, it also comes at an additional resource and complexity cost in managing sidecars and their control plane as well.

Still, instrumenting at a service mesh layer means you don't have to retrofit existing applications with framework instrumentation like Spring Cloud Sleuth, change your base images like with agent-based instrumentation, or perform manual instrumentation. Because of the lack of information available to the service mesh relative to frameworks, introducing a service mesh primarily to achieve telemetry instrumentation incurs the significant cost of maintaining the mesh for what ultimately will be less-rich telemetry. For example, a mesh will observe a request to `/api/customers/1` and not have the context that the framework does, that this is a request to `/api/customers/{id}`. As a result, telemetry streaming out of mesh-based instrumentation will be harder to group by parameterized URI. Adding the runtime dependency may very well be significantly easier in the end.

Blended Tracing

White box (or framework telemetry that, because it is autoconfigured, *feels* like white box) and black box options aren't mutually exclusive. They can in fact complement each other well. Consider the REST controller in [Example 3-4](#). Spring Cloud Sleuth is designed to automatically create a span around the request handler `findCustomerById`, tagging it with pertinent information. By injecting a `Tracer`, you can add a finer-grained span on just the database access. This breaks down the end-to-end user interaction into an even-finer-grained trace. Now we can identify where the database specifically is the cause of a service degradation in the satisfaction of the request in this particular microservice.

Example 3-4. Blended black box and white box tracing instrumentation

```
@RestController
public class CustomerController {
    private final Tracer tracer;

    public CustomerController(Tracer tracer) {
        this.tracer = tracer;
    }

    @GetMapping("/customer/{id}") ❶
    public Customer findCustomerById(@PathVariable String id) {
        Span span = tracer.nextSpan().name("findCustomer"); ❷
        try (Tracer.SpanInScope ignored = tracer.withSpanInScope(span.start())) {
            Customer customer = ... // Database access to lookup customer
            span.tag("country", customer.getAddress().getCountry()); ❸
            return customer;
        }
        finally {
            span.finish(); ❹
        }
    }
}
```

```
}
```

- ➊ Spring Cloud Sleuth will automatically instrument this endpoint, tagging the span with useful context like `http.uri`.
- ➋ Starting a new span adds another distinct element to the trace icicle graph. We can now reason about the cost of just this method `findCustomerById` in the context of an entire end-to-end user interaction.
- ➌ Adding business-specific context that black box instrumentation would lack. Maybe your company just launched a service recently in a new country as part of a global expansion, and because of its recency, customers in that country lack a long activity history with your product. Seeing a surprising difference in lookup times between older and newer customers might suggest a change to how activity history is loaded in this situation.
- ➍ The whole data access operation is manually instrumented with white box instrumentation.

Supposing database access was traced across the application stack (potentially by encapsulating the wrapping of database access with tracing instrumentation into a common library and sharing it across the organization), the database would be effectively traced without adding any form of white box or black box monitoring to the database layer itself. Instrumenting something like an IBM DB2 database running on a z/OS mainframe with Zipkin Brave seems like an impossible task initially, but it can be accomplished from the caller perspective.



Tracing All Calls to a Subsystem Effectively Traces the Subsystem

By tracing all *calls* to a subsystem, the subsystem is covered with tracing in the same way as if it were instrumented itself. Many component frameworks (database, cache, message queue) offer some sort of event system for you to hook into. In many cases, the task of coating all callers with instrumentation can be reduced to ensuring all calling applications have a binary dependency on their runtimes that's capable of automatically injecting an event handler into the component framework you want to instrument.

The other ramification of tracing from a caller is that it includes latency (e.g., network overhead) between the two systems. In a scenario where a series of callers is executing requests to a downstream service that is serving requests at a fixed maximum concurrency level (e.g., a thread pool), the downstream service may not even be aware of a

request until it starts to be processed. Instrumenting from the caller side includes the time a request sat in a queue waiting to be processed by the downstream.

All this contextual information can get expensive. To control cost, at some point we have to sample tracing telemetry.

Sampling

As mentioned in [“The Three Pillars of Observability...or Is It Two?” on page 101](#), tracing data generally must be sampled to control cost, meaning some traces are published to the tracing backend and others are not.

No matter how smart the sampling strategy, it is important to remember that data is being *discarded*. Whatever collection of traces you get as a result are going to be skewed in some way. This is perfectly fine when you are pairing distributed tracing data with metrics data. Metrics should alert you to anomalous conditions and traces used to do in-depth debugging when necessary.

Sampling strategies fall into a few basic categories, ranging from not sampling at all to propagating sampling decisions down from the edge.

No Sampling

It is possible to retain every tracing sample. Some organizations even do this at a large scale, often at extraordinary cost. For Spring Cloud Sleuth, configure the `Sampler` via a bean definition, as shown in [Example 3-5](#).

Example 3-5. Configuring Spring Cloud Sleuth to always sample

```
@Bean
public Sampler defaultSampler() {
    return Sampler.ALWAYS_SAMPLE;
}
```

Rate-Limiting Samplers

By default, Spring Cloud Sleuth retains the first 10 samples per second (a configurable rate limit threshold) and downsamples probabilistically thereafter. Since rate-limiting sampling is the default in Sleuth, the rate limit can be set by a property, as in [Example 3-6](#).

Example 3-6. Configuring Spring Cloud Sleuth to retain the first 2,000 samples per second

```
spring.sleuth.sampler.rate: 2000
```

The logic behind this is that there is some rate of throughput for which it is reasonably cost-effective to not discard anything. This is largely going to be dictated by the nature of your business and the throughput to your applications. One regional property and casualty insurer receives 5,000 requests per minute through its flagship app, generated from the interactions of approximately 3,500 insurance agents in the field. Since the pool of insurance agents is not going to suddenly grow by an order of magnitude overnight, a stable capacity plan for a tracing system that accepts 100% of traces for this system is determinable.

Even if your organization is like this insurer, it's important to keep in mind where further investments in application observability are made, often in open source at tech companies with significant scale and at monitoring system vendors that can't assume their customers all have such a stable capacity plan. Considering something like the high-percentile calculation of the latency of a service endpoint, it can still make sense to leverage high-percentile approximations from bucketed histograms over trying to calculate an exact percentile from tracing data, even though this is mathematically possible with 100% data.

The point is to avoid inventing new methods of calculating distribution statistics when similar methods are available from metrics telemetry designed to operate at scale.

One challenge of rate-based sampling is holes. When you have several microservices in a call chain, each independently making a decision about whether to retain a trace, holes are going to develop in the end-to-end picture of a given request. Put another way, rate-based samplers don't make a consistent sampling decision given a trace ID. The moment any individual subsystem exceeds the rate threshold, a hole develops in traces involving this subsystem.

When making capacity planning decisions based off of rate-based samplers, be careful to recognize that these rates are on a *per-instance* basis. The rate of samples reaching the tracing system is the product of instances in the cluster times and the sampling rate in the worst case.

Probabilistic Samplers

Probabilistic samplers count to see how many out of 100 traces should be retained. They guarantee that if you select a 10% probability, 10 out of 100 traces will be retained, but it likely won't be the first 10 or the last 10.

In the presence of a probability property, Spring Cloud Sleuth configures a probabilistic sampler instead of a rate-limiting sampler, as in [Example 3-7](#).

Example 3-7. Configuring Spring Cloud Sleuth to retain 10% of traces

```
spring.sleuth.sampler.probability: 0.1
```

Probabilistic samplers are rarely the right choice for a couple of reasons:

Cost

No matter what probability you choose, your tracing cost grows linearly in proportion to traffic. Maybe you never expect an API endpoint to receive more than 100 requests per second and you've sampled to 10%. If there is a sudden increase in traffic to 10,000 requests per second, you'll suddenly be shipping 1,000 traces per second rather than 10. Rate-limiting samplers cap the cost in a way that places a fixed upper bound on cost, irrespective of throughput.

Holes

Like rate-based samplers, probabilistic samplers don't look at trace IDs and headers to make their sampling decisions. Holes will develop in the end-to-end picture. In the case of relatively low throughput systems, a rate-based sampler may practically have no holes because no individual subsystem exceeded the rate threshold, but a probabilistic sampler has a uniform probability of holes per unit of throughput, so holes will likely exist even for low-throughput systems.

Boundary Sampling

Boundary samplers are a variant of probabilistic sampler that solves the problem of holes by making the sampling decision only once at the edge (the first interaction with your system) and propagating the sampling decision downstream to other services and components. The trace context in each component contains a sampling decision that is added as an HTTP header and extracted into trace context by the downstream component, as shown in [Figure 3-3](#).

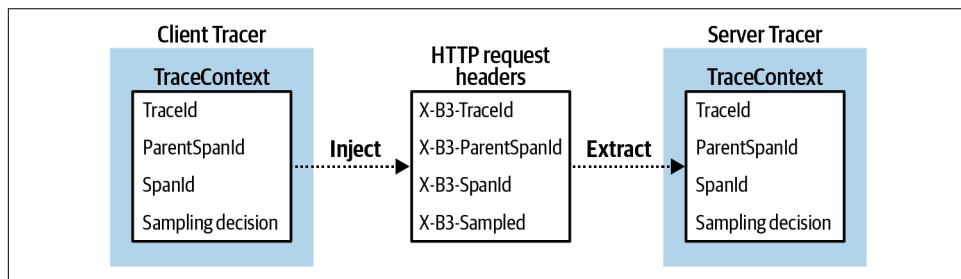


Figure 3-3. B3 trace headers propagate the sampling decision to downstream components

Impact of Sampling on Anomaly Detection

Let's consider specifically the impact probabilistic sampling would have on anomaly detection. A similar effect would occur for any sampling strategy really, but we'll use probabilistic sampling to make this concrete.

Anomaly detection systems built on sampled traces are generally misguided unless your organization assumes the cost of 100% sampling. To show why, let's consider a hypothetical sampling strategy that makes an up-front decision about whether to preserve a trace based on a weighted random number at the beginning of each request (as Google's Dapper did originally). If we are sampling 1% of requests, then an outlier above the 99th percentile, like all other requests, has a 1% chance of surviving the sampling. There's a 0.01% chance of seeing any of these individual outliers. Even at 1,000 requests/second, you could have 10 outliers happening per second but only see 1 every 5 minutes, as shown in [Figure 3-4](#) (a plot of $(1 - 0.99^N) * 100\%$).

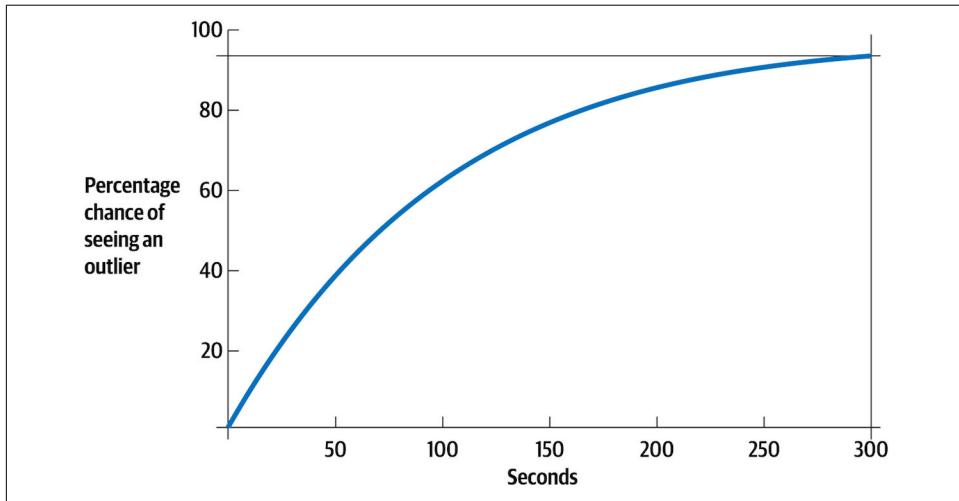


Figure 3-4. Chance of seeing an outlier in tracing data over time

There can be a significant range of outliers above the 99th percentile, as shown in [Figure 4-20](#). You could have a huge business-critical outlier (above P99.9) happening once per second and only see it *once* in tracing data in any given hour! For the purposes of debuggability, having one or a small set of outliers survive over a given period is fine—we still get to examine in detail the nature of what's happening in the outlier case.

Distributed Tracing and Monoliths

Don't be fooled by the name *distributed* tracing. It is absolutely reasonable to use this form of observability in a monolith as well. In the purest microservices architecture, tracing around RPC calls could be implemented in a black box fashion at either the framework level (like Spring) or in a sidecar such as those we find in service mesh technologies. Given the single-responsibility nature of a microservices architecture, tracing RPC could actually give you quite a bit of information about what's going on;

i.e., microservice boundaries are effectively business logic functional boundaries as well.

Inside a monolithic application that receives a single end-user request and performs many tasks to satisfy that request, framework-level instrumentation has a diminished value, of course, but you can still write tracing instrumentation at key functional boundaries inside the monolith in much the same way you write logging statements. In this way, you'll be able to select specific tags that allow you to search for spans with business context that framework or service-mesh instrumentation will certainly lack.

In fact, white box instrumentation with business-specific tagging winds up being essential even in a pure microservices architecture. In many cases, our key pieces of business functionality aren't *completely* broken in production, but rather broken along specific (often unusual) business-specific fault lines. Maybe an insurance company's policy administration system is failing to rate classic cars in a particular county in Kentucky. Having vehicle class, county, and state on both metrics and tracing telemetry allows an engineer to drill down dimensionally on a metric and find the problem area and then hop to debugging signals like tracing and logs to see example failures once the failing dimension is known.



Business Context Makes White Box Tracing as Important in Monoliths as in Distributed Systems

The density of white box distributed tracing instrumentation per bounded context of business functionality should roughly be the same in a microservices or monolithic architecture, because black box instrumentation will not tag spans with business-specific context that aids in later lookup.

So the only difference between microservices and monoliths is that you have more bounded contexts of business functionality packed into one process. And with each additional piece of business functionality comes all the trappings that support its existence. Observability is not an exception.

Additionally, even a microservice with a single responsibility can do a handful of things, including data access, in the satisfaction of a user request.

Correlation of Telemetry

Since metrics data is a strong availability signal and tracing and logging data is useful for debugging, anything we can do to link them together makes the transition from an alert indicating a lack of availability to the debugging information that would best identify the underlying issue. In the case of latency, we will have a chart on a dashboard and an alert on a decaying max. Presenting a view of the latency distribution as

a heatmap of the latency histogram is an interesting information-dense visualization but isn't something we can plot an alert threshold on.

Metric to Trace Correlation

We can plot example traces (exemplars), as shown in [Figure 3-5](#), on the heatmap and make the heatmap more interactive by making heatmap cells into links that take us directly to the tracing UI, where we can see a set of traces that match these criteria. So an engineer responsible for a system receives an alert on a latency condition, looks at the set of latency charts for this application, and can immediately click through to the distributed tracing system.

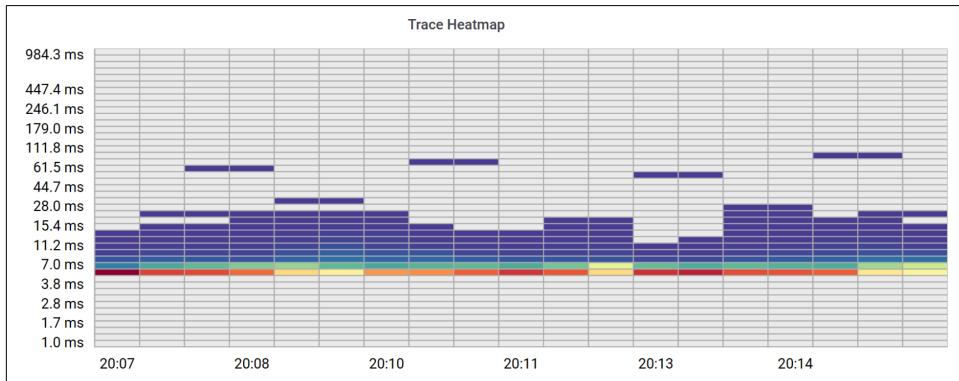


Figure 3-5. Zipkin trace data plotted on top of a Prometheus histogram represented as a heatmap in Grafana

This sort of correlative plot makes metrics and tracing together more valuable. We'd normally lose through aggregation an understanding of what happened in particular cases when looking at metrics data. Traces, on the other hand, lack the understanding of the big picture that metrics provide.

Also, since it is certainly possible that trace sampling (again, to control cost) has thrown away all the traces that would match a particular latency bucket, we still get to understand what latencies end users experienced even if we aren't able to drill into the trace detail.

This visualization is built through the combination of independent Prometheus and Zipkin queries, as shown in [Figure 3-6](#). Notice that the tags don't have to strictly line up between the metrics and tracing instrumentation. Micrometer Timer called `http.server.requests` (which yields a set of time series which are called `http_server_requests_second_bucket` in Prometheus when histograms—“Histograms” on page 65—are turned on) is collected with a tag called `uri`. Spring Cloud

Sleuth instruments Spring in a similar way but tags traces with `http.uri`. These are of course logically equivalent.

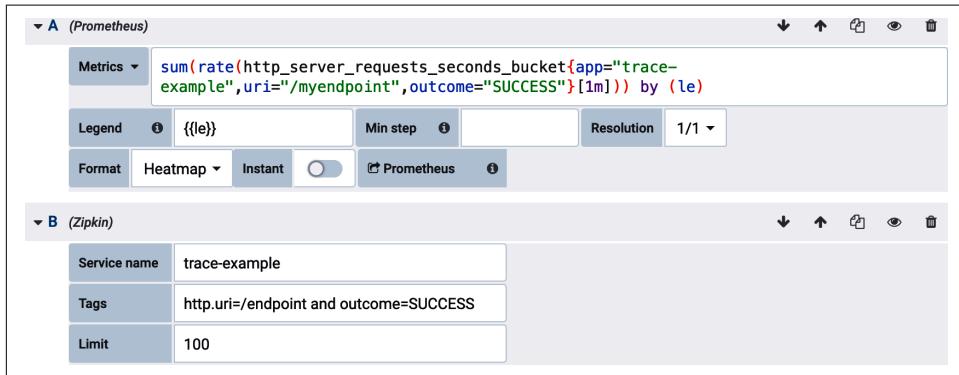


Figure 3-6. Independent Prometheus and Zipkin queries form the combined trace exemplar heatmap

It should be clear, however, that even though the tag keys (and even values) don't have to be identical, if you want to filter the heatmap to a metrics tag that has no logical equivalent in the tracing data, then it won't be possible to accurately find exemplars to match what is seen on the heatmap (there will be some false positives). For example, Spring Cloud Sleuth didn't initially tag traces with HTTP status code or outcome, while Spring's Micrometer instrumentation did. Often we want to limit a latency visualization to either successful or unsuccessful outcomes because their latency characteristics can be quite different (e.g., failures occur abnormally fast due to an external resource unavailability or abnormally slow due to a timeout).

So far, our look at distributed tracing has strictly been related to observability, but it can serve other purposes that influence or govern the way traffic is handled.

Using Trace Context for Failure Injection and Experimentation

Earlier, when discussing sampling methods for distributed tracing, we covered boundary sampling (see “[Boundary Sampling](#)” on page 116). In this method, a sampling decision is made up front (i.e., at the edge), and this decision is propagated downstream to the microservices that are involved in satisfying a request. There is an interesting opportunity to make other up-front decisions and leverage trace context to pass along other information unrelated to sampling decisions to downstream services as well.

A well-known example of this is *failure injection testing* (FIT), a specific form of chaos engineering. The overall discipline of chaos engineering is broad and covered in detail in [Chaos Engineering](#).

Failure injection decisions can be added by the API gateway up front in coordination with rules provided by a central FIT service and propagated downstream as a trace tag. Later, a microservice in the execution path can use this information about a failure test to unnaturally fail a request in some way. [Figure 3-7](#) shows the whole process, end to end.

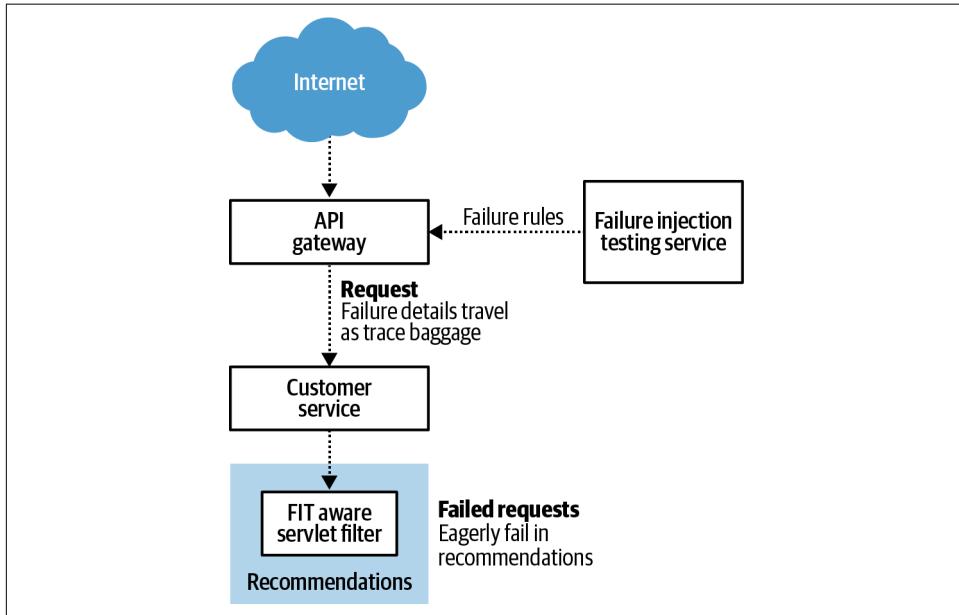


Figure 3-7. Failure injection testing process from user request to failure

Attaching this kind of decision to telemetry has the added benefit that any sampled traces that also are part of a failure injection are tagged as such, so you can differentiate between real and intentional failures when looking at telemetry later. [Example 3-8](#) shows a simplified example of a Spring Cloud Gateway application (that also has the Spring Cloud Sleuth starter applied) looking up and adding a FIT decision as “baggage” to the trace context, which can automatically be converted to a trace tag by setting the property `spring.sleuth.baggage.tag-fields=failure.injection`.

Example 3-8. Spring Cloud Gateway adding failure injection testing data to trace context

```
@SpringBootApplication
public class GatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }
}

@RestController
class GatewayController {
    private static final String FAILURE_INJECTION_BAGGAGE = "failure.injection";

    @Value("${remote.home}")
    private URI home;

    @Bean
    BaggagePropagationCustomizer baggagePropagationCustomizer() {
        return builder -> builder.add(BaggagePropagationConfig.SingleBaggageField
            .remote(BaggageField.create(FAILURE_INJECTION_BAGGAGE)));
    }

    @GetMapping("/proxy/path/**")
    public Mono<ResponseEntity<byte[]>> proxyPath(ProxyExchange<byte[]> proxy) {
        String serviceToFail = "";
        if (serviceToFail != null) {
            BaggageField.getByName(FAILURE_INJECTION_BAGGAGE)
                .updateValue(serviceToFail);
        }

        String path = proxy.path("/proxy/path/");
        return proxy.uri(home.toString() + "/foos/" + path).get();
    }
}
```

Then, add an incoming request filter (in this case a WebFlux `WebFilter`) to all micro-services that might participate in failure injection tests, as shown in [Example 3-9](#).

Example 3-9. WebFlux WebFilter for failure injection testing

```
@Component
public class FailureInjectionTestingHandlerFilterFunction implements WebFilter {
    @Value("${spring.application.name}")
    private String serviceName;

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain) {
        if (serviceName.equals(BaggageField.getByName("failure.injection")
            .getValue())) {
            exchange.getResponse().setStatus(HttpStatusCode.INTERNAL_SERVER_ERROR);
        }
    }
}
```

```

        return Mono.empty();
    }

    return chain.filter(exchange);
}
}

```

We can also add a failure injection test decision as a tag to HTTP client metrics, as shown in [Example 3-10](#). It may be useful to filter out failure injection tests from our notion of the error ratio of the HTTP client interaction with downstream services. Or perhaps they are left in to alert criteria to validate the engineering discipline of being alerted to and responding to unexpected failure, but the data will still be present so that the investigating engineer can dimensionally drill down to determine if the alert was caused by failure injection or by real issues.

Example 3-10. Adding the failure injection testing decision as a Micrometer tag

```

@Component
public class FailureInjectionWebfluxTags extends DefaultWebFluxTagsProvider {
    @Value("${spring.application.name}")
    private String serviceName;

    @Override
    public Iterable<Tag> httpRequestTags(ServerWebExchange exchange, Throwable ex) {
        return Tags.concat(
            super.httpRequestTags(exchange, ex),
            "failure.injection",
            serviceName.equals(BaggageField
                .getByName("failure.injection").getValue()) ? "true" : "false"
        );
    }
}

```

This is just a sketch, of course. It is up to you how you would define a failure injection service, and under what conditions to select requests for failure injection. For a simple set of rules, this service could even be an integral part of your gateway application.

In addition to failure injection, trace baggage could be used to propagate decisions about whether a request is participating in A/B experiments as well.

Summary

In this chapter, we've shown the difference between monitoring for availability and monitoring for debugging. The event-based nature of debugging signals means they tend to want to grow proportionally with increased throughput through a system, a cost-limiting measure is necessary. Different methods of sampling to control cost were discussed. The fact that debugging signals are typically sampled should give us

pause about trying to build aggregations around them, since every form of sampling discards some part of the distribution and thus skews the aggregation in one form or another.

Lastly, we showed how in addition to its chief function in publishing debugging information, we can piggyback on trace context propagation to propagate behaviors down a deep microservice call chain.

In the next chapter, we return to metrics, showing which availability signals you should start with for basically every Java microservice.

Charting and Alerting

Monitoring doesn't have to be an all-in proposition. If you only add a measure of error ratio for end-user interactions where you have no monitoring (or only resource monitoring like CPU/memory utilization), you've already taken a huge step forward in terms of understanding your software. After all, CPU and memory can look good but a user-facing API is failing 5% of all requests, and failure rate is a much easier idea to communicate between engineering organizations and their business partners.

While Chapters 2 and 3 covered different forms of monitoring instrumentation, here we present the ways we can *use* that data effectively to promote action via alerting and visualization. This chapter covers three main topics.

First, we should think about what makes for a good visualization of an SLI. We're only going to show charts from the commonly used [Grafana](#) charting and alerting tool, because it is a freely available open source tool that has datasource plug-ins for many different monitoring systems (so learning a little Grafana is a largely transferable skill from one monitoring system to another). Many of the same suggestions apply to charting solutions integrated into vendor products.

Next, we'll discuss specifics about the measurements that generate the most value and how to visualize and alert on them. Treat these as a checklist of SLIs that you can add incrementally. Incrementalism may even be preferable to implementing them all at once, because by adding an indicator at a time, you can really study and understand what it means in the context of your business and shape it in little ways to generate the most value to you. If I walked into the network operation center of an insurance company, I'd be much more relieved to see only indicators on the error ratio of policy rating and submissions than I would be to see a hundred low-level signals and no measure of business performance.

Taking an incremental approach to introducing alerts is also an important trust-building exercise. Introducing too many alerts too quickly risks overwhelming engineers and leading to “alert fatigue.” You want engineers to feel comfortable subscribing to more alerts, not mute them! This also gives you a bit of time, if you are not already accustomed, to working through the on-call process, and training engineers how to respond to one alert condition at a time helps the team build a reservoir of knowledge about how to address anomalies.

So the focus in this chapter will be providing advice about those SLIs that are as close to business performance (e.g., API failure rate and the response times users see) as possible without being tied to any particular business. To the extent we cover things like heap utilization or file descriptors, they will be a select group of indicators that are most likely to be the direct cause of business performance degradation.

Recreating NASA mission control (Figure 4-1) should not be the end result of a well-monitored distributed system. While arraying screens across a wall and filling them with dashboards may look visually impressive, screens are not actions. They require somebody to be looking at them to respond to a visual indicator of a problem. I think this makes sense when you’re monitoring a single instance of a rocket with exorbitant costs and human lives on the line. Your API requests, of course, don’t have the same per-occurrence importance.



Figure 4-1. This is not a good role model!

Almost every metrics collector will collect more data than you will find useful at any given time. While every metric may have usefulness in some circumstance, plotting every one is not helpful. However, several indicators (e.g., max latency, error ratio, resource utilization) are strong reliability signals for practically every Java microservice (with some tweaks to the alert thresholds). These are the ones we'll focus on.

Lastly, the market is eager to apply artificial intelligence methods to monitoring data to automate the delivery of insights into your systems without requiring much understanding of alert criteria and key performance indicators. In this chapter, we'll survey several traditional statistical methods and artificial intelligence methods in the context of application monitoring. You should have a solid understanding of the strength and weakness of each method so that you can cut through the marketing hype and apply the best methods for your needs.

Before going any further, it's worth considering the breadth of variation in monitoring systems on the market and the impact that has on your decisions for how to instrument code and get data to these systems.

Differences in Monitoring Systems

The point of discussing differences in monitoring systems here is that we are about to see specifics about how to chart and alert with Prometheus. A product like Datadog has a very different query system than Prometheus. Both are useful. More products are going to emerge in the future with capabilities we aren't yet imagining. Ideally, we want our monitoring instrumentation (what we will put in our applications) to be portable across these monitoring systems with no changes in application code required (other than a new binary dependency and some registry-wide configuration).

There tends to be quite a bit more consistency in the way distributed tracing backend systems receive data than the way metrics systems receive data. Distributed tracing instrumentation libraries may have different propagation formats, requiring a degree of uniformity in the selection of an instrumentation library across the stack, but the data itself is fundamentally similar from backend to backend. This intuitively makes sense because of what the data is: distributed tracing really consists of per-event timing information (contextually stitched together by trace ID).

Metrics systems could potentially represent not only aggregated timing information, but also gauges, counters, histogram data, percentiles, etc. They don't agree on the way in which this data should be aggregated. They don't have the same capabilities for performing further aggregation or calculation at query time. There is an inverse relationship between the number of time series a metrics instrumentation library must publish and the query capabilities of a particular metrics backend, as shown in [Figure 4-2](#).

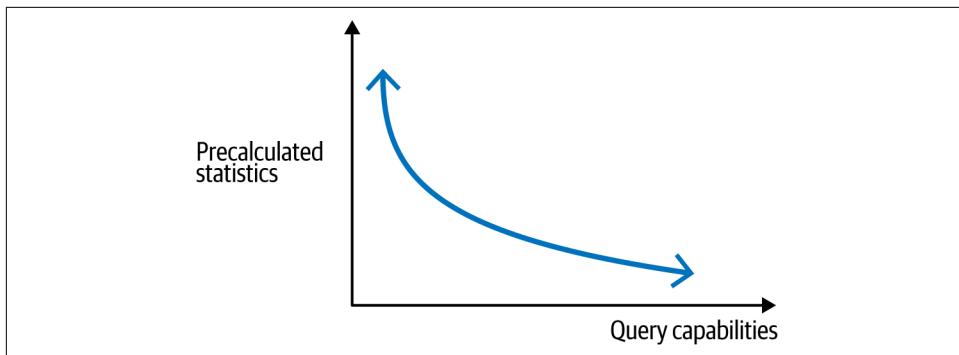


Figure 4-2. Inverse relationship between published time series and query capabilities

So, for example, when Dropwizard Metrics was initially developed, the popular monitoring system was Graphite, which didn't have rate-calculating functions available in modern monitoring systems like Prometheus. As a result, when publishing a counter, Dropwizard had to publish cumulative count, 1-minute rate, 5-minute rate, 15-minute rate, etc. And because this was inefficient if you never needed to look at a rate, the instrumentation library itself distinguished between `@Counted` and `@Metered`. The instrumentation API was designed with the capabilities of its contemporary monitoring systems in mind.

Fast forward to today, and a metrics instrumentation library intending to publish to multiple destination metrics systems needs to be aware of these subtleties. A `Micrometer Counter` is going to be presented to Graphite in terms of a cumulative count and several moving rates, but to Prometheus only as a cumulative count, because these rates can be computed at query time with a PromQL `rate` function.

It's important to the design of the API of any instrumentation library today to not simply lift all concepts found in earlier implementations forward, but to consider the historical context behind why these constructs existed at that time. [Figure 4-3](#) shows where Micrometer has overlap with Dropwizard and Prometheus simple client predecessors, and where it has extended capabilities beyond those of its predecessors. Significantly, some concepts have been left behind, recognizing the evolution in the monitoring space since. In some cases, this difference is subtle. Micrometer incorporates histograms as a feature of a plain `Timer` (or `DistributionSummary`). It is often unclear at the point of instrumentation deep in a library where an operation is being timed whether the application incorporating this functionality views this operation as critical enough to warrant the extra expense of shipping histogram data. (So the decision should be left up to the downstream application author rather than the library author.)

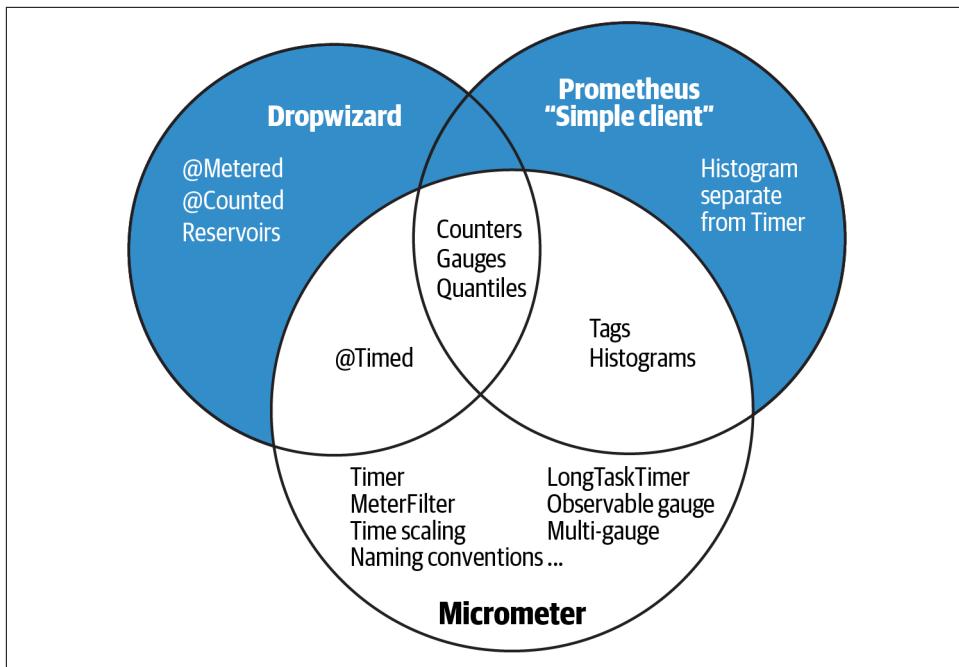


Figure 4-3. Metrics instrumentation capability overlap

Similarly, in the Dropwizard Metrics era, monitoring systems didn't include query functionality that helped to reason about timing data (no percentile approximations, no latency heatmaps, etc.). So this concept of "don't gauge something you can count, don't count something you can time" wasn't applicable yet. It wasn't uncommon to add `@Counted` to a method, where now `@Counted` is almost never the right choice for a method (which is inherently timeable, and timers always publish with a count as well).

While at the time of this writing OpenTelemetry's metrics API is still in beta, it hasn't changed substantially in the last couple years, and it appears the meter primitives won't do a sufficient job to build usable abstractions for timing and counting. [Example 4-1](#) shows a Micrometer Timer with varying tags, depending on the outcome of an operation (this is the most verbose a timer gets in Micrometer).

Example 4-1. A Micrometer timer with a variable outcome tag

```
public class MyService {
    MeterRegistry registry;

    public void call() {
        try (Timer.ResourceSample t = Timer.resource(registry, "calls")
            .description("calls to something")
            .publishPercentileHistogram()
            .serviceLevelObjectives(Duration.ofSeconds(1))
            .tags("service", "hi")) {
            try {
                // Do something
                t.tag("outcome", "success");
            } catch (Exception e) {
                t.tags("outcome", "error", "exception", e.getClass().getName());
            }
        }
    }
}
```

Even trying to get close to this with the OpenTelemetry metrics API right now is difficult, as shown in [Example 4-2](#). No attempt has been made to record something similar to percentile histograms or SLO boundary counts like in the Micrometer equivalent. That would of course substantially increase the verbosity of this implementation, which is already getting lengthy.

Example 4-2. OpenTelemetry timing with variable outcome tags

```
public class MyService {
    Meter meter = OpenTelemetry.getMeter("registry");
    Map<String, AtomicLong> callSum = Map.of(
        "success", new AtomicLong(0),
        "failure", new AtomicLong(0)
    );

    public MyService() {
        registerCallSum("success");
        registerCallSum("failure");
    }

    private void registerCallSum(String outcome) {
        meter.doubleSumObserverBuilder("calls.sum")
            .setDescription("calls to something")
            .setConstantLabels(Map.of("service", "hi"))
            .build()
            .setCallback(result -> result.observe(
                (double) callSum.get(outcome).get() / 1e9,
                "outcome", outcome));
    }
}
```

```

public void call() {
    DoubleCounter.Builder callCounter = meter
        .doubleCounterBuilder("calls.count")
        .setDescription("calls to something")
        .setConstantLabels(Map.of("service", "hi"))
        .setUnit("requests");

    long start = System.nanoTime();
    try {
        // Do something
        callCounter.build().add(1, "outcome", "success");
        callSum.get("success").addAndGet(System.nanoTime() - start);
    } catch (Exception e) {
        callCounter.build().add(1, "outcome", "failure",
            "exception", e.getClass().getName());
        callSum.get("failure").addAndGet(System.nanoTime() - start);
    }
}
}

```

I believe the problem for OpenTelemetry is an emphasis on polyglot support, which naturally puts pressure on the project to want to define a consistent data structure for meter primitives like the “double sum observer” or “double counter.” The impact on the resulting API forces the end user to compose from lower-level building blocks the constituent parts of a higher-level abstraction like a Micrometer Timer. This not only leads to exceedingly verbose instrumentation code, but also leads to instrumentation that is specific to a particular monitoring system. For example, if we attempt to publish a counter to an older monitoring system like Graphite while we gradually migrate to Prometheus, we need to explicitly calculate per-interval moving rates and ship those too. The “double counter” data structure doesn’t support this. The reverse problem exists as well, the need to include the union of all possibly usable statistics for a “double counter” in the OpenTelemetry data structure to satisfy the widest array of monitoring systems, even though shipping this extra data is pure waste to a modern metrics backend.

As you get into exploring charting and alerting, you may want to experiment with different backends. And making a selection today based on what you know, you may find yourself transitioning with more experience in a year. Make sure your metrics *instrumentation* permits you to move fluidly between monitoring systems (and even publish to both while you transition).

Before we get into any particular SLIs, let’s first go over what makes for an effective chart.

Effective Visualizations of Service Level Indicators

The recommendations offered here are naturally subjective. I'm going to state a preference for bolder lines and less "ink" on the chart, both of which deviate from Grafana's defaults. To be honest, I'm a little embarrassed to offer these suggestions, because I don't want to presume that my aesthetic sense is in some way greater than that of the excellent design team at Grafana.

The stylistic sensibility I will offer is derived from two significant influences over my last few years of work:

Watching engineers stare and squint at charts

I worry when an engineer looks at a chart and squints. I worry especially that the lesson they take from an overly complex visualization is that monitoring itself is complex, and maybe too complex for them. Most of these indicators are *really* simple when presented correctly. It should feel that way.

The Visual Display of Quantitative Information

For a time, I asked the same question of every member of the rare population of user experience designers I met who focus on operations engineering and developer experience: which book(s) did they find were the greatest influence on them? *The Visual Display of Quantitative Information* by Edward Tufte (Graphics Press) was always among their answers. One of the ideas most relevant to time series visualization that comes from this book is "data-ink" ratio, specifically to increase it as much as possible. If "ink" (or pixels) on a chart isn't conveying information, it is conveying complexity. Complexity leads to squinting. Squinting leads to me worrying.

Let's think then from this perspective that data-ink ratio needs to go *up*. The specific recommendations that follow change the default styling of Grafana to maximize this ratio.

Styles for Line Width and Shading

Grafana's default chart contains a 1 px solid line, a 10% transparency fill under the line, and interpolation between time slices. For better readability, increase the solid line width to 2 px and remove the fill. The fill reduces the data-ink ratio of the chart, and the overlapping colors of fills get disorienting with more than a couple lines on a chart. Interpolation is a little misleading, since it implies to a casual observer that the value may have briefly existed at intermediate points along the diagonal between two time slices. The opposite of interpolation is called "step" in Grafana's options. The chart on the top in [Figure 4-4](#) uses the default options, and the chart on the bottom is adjusted with these recommendations.

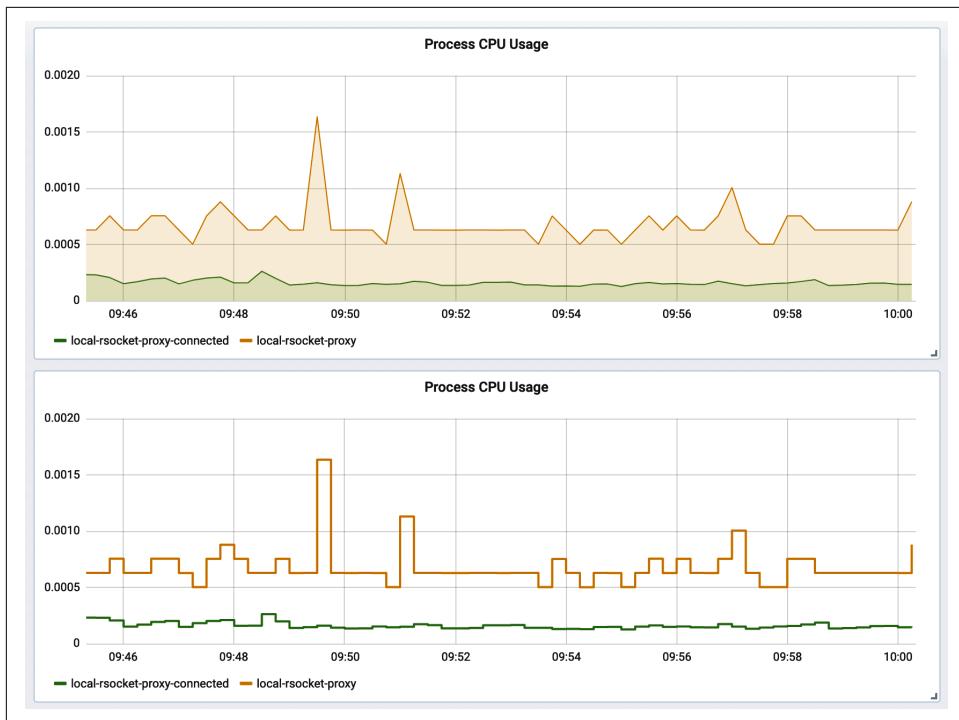


Figure 4-4. Grafana chart style default versus recommended

Change the options in the “Visualization” tab of the chart editor, as shown in Figure 4-5.

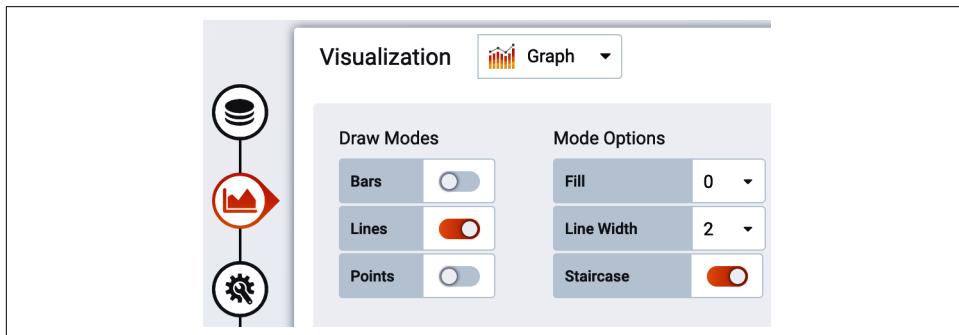


Figure 4-5. Grafana line width options

Errors Versus Successes

Plotting a stacked representation of outcomes (success, error, etc.) is very common for timers, as we'll see in “[Errors](#)” on page 148, and shows up in other scenarios as well. When we think of successes and errors as colors, many of us will immediately think of green and red: stoplight colors. Unfortunately, a significant portion of the population has color vision impairments that affect their ability to perceive color differences. For the most common impairments, deutanopia and protanopia, the difference between green and red is difficult or impossible to distinguish! Those affected by monochromacy cannot distinguish colors at all, only brightness. Because this book is printed monochromatically, we all get to experience this briefly for the stacked chart of errors and successes in [Figure 4-6](#).

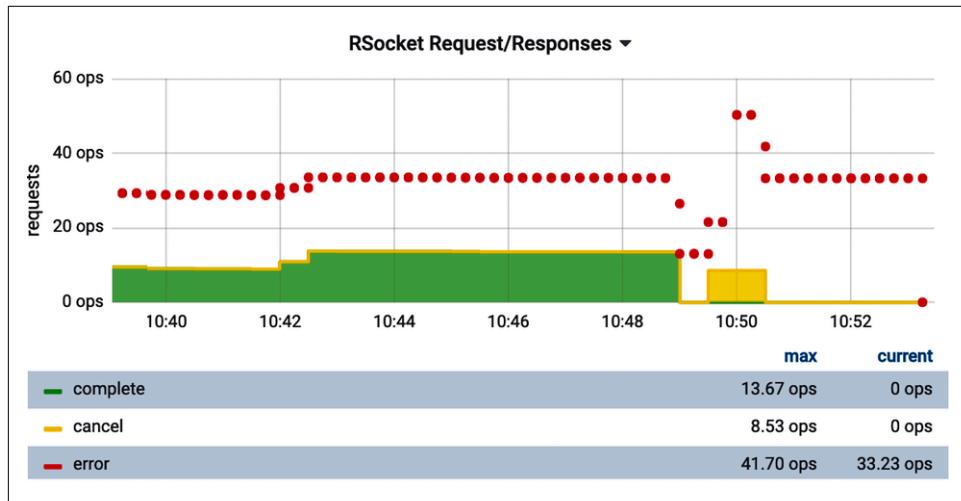


Figure 4-6. Display errors with a different line style for accessibility

We need to provide some sort of visual indicator of errors versus successes other than strictly color. In this case, we've chosen to plot “successful” outcomes as stacked lines and the errors above these outcomes as thick points to make them stand out.

Additionally, Grafana doesn't offer an option to specify the order of time series as they appear in a stacked representation (i.e., “success” on the bottom or top of the stack), even for a limited set of possible values. We can force an ordering of them by selecting each value in a separate query and ordering the queries themselves, as shown in [Figure 4-7](#).

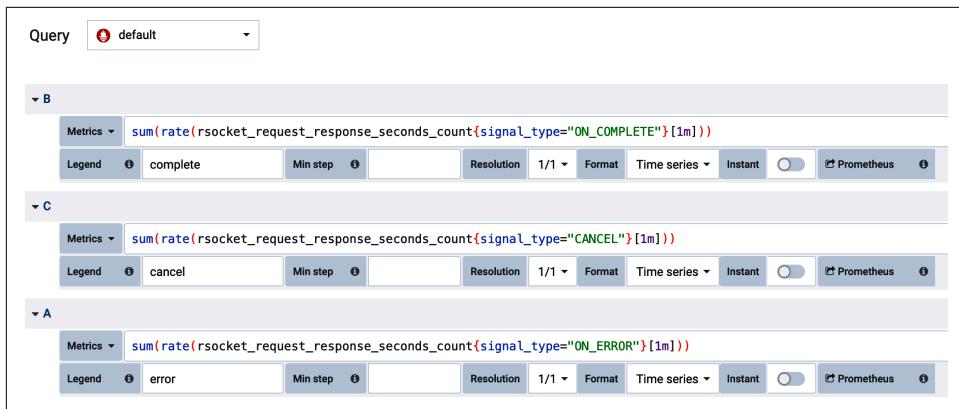


Figure 4-7. Ordering outcomes in a Grafana stack representation

Lastly, we can override the styling of each individual query, as shown in Figure 4-8.

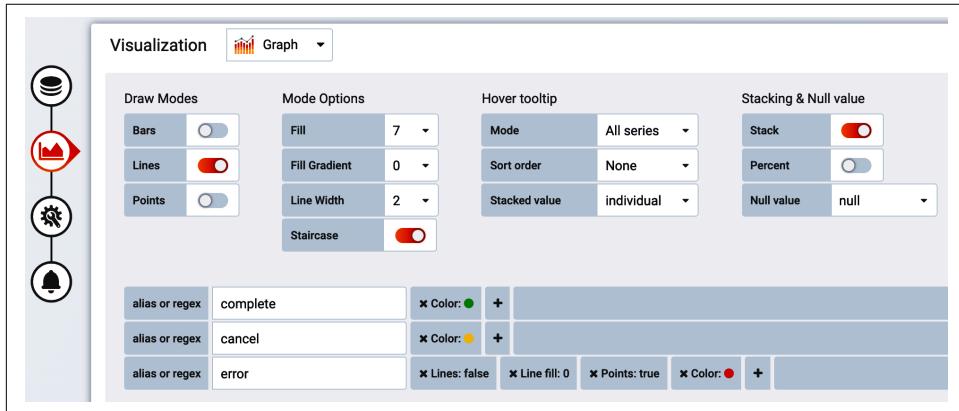


Figure 4-8. Overriding line styles for each outcome

“Top k” Visualizations

In many cases, we want to display some indicator of the “worst” performers by some category. Many monitoring systems offer some sort of query function to select the “top k” time series for some criteria. Selecting “top 3” worst performers doesn’t mean that there will be a maximum of three lines on the chart, however, because this race to the bottom is perpetual, and the worst performers can change over the course of the time interval visualized by the chart. At worst, you are displaying N datapoints on a particular visualization, and there will be 3^*N distinct time series displayed! If you draw a vertical line down any part of Figure 4-9 and count the number of unique colors it intersects, it will always be less than or equal to three because this chart was built with a “top 3” query. But there are six items in the legend.

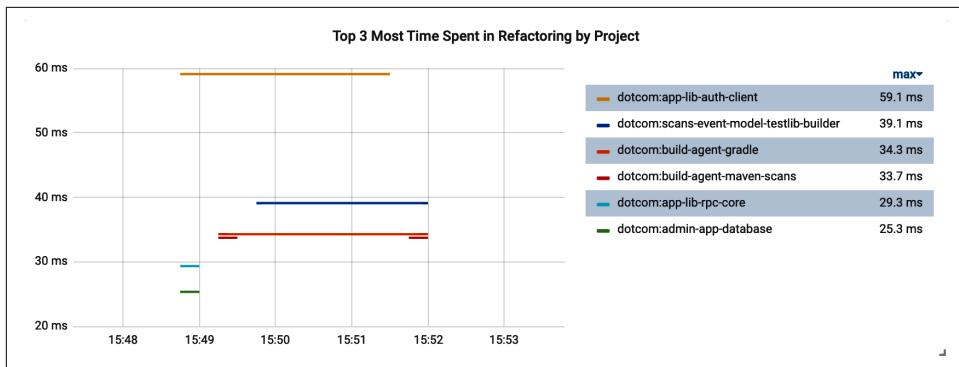


Figure 4-9. Top k visualization with more than k distinct time series

It can get far busier than this very easily. Consider Figure 4-10, which shows the top five longest Gradle build task times over a period of time. Since the set of build tasks running changes rapidly over the time slices shown in this chart, the legend fills up with many more values than simply five.

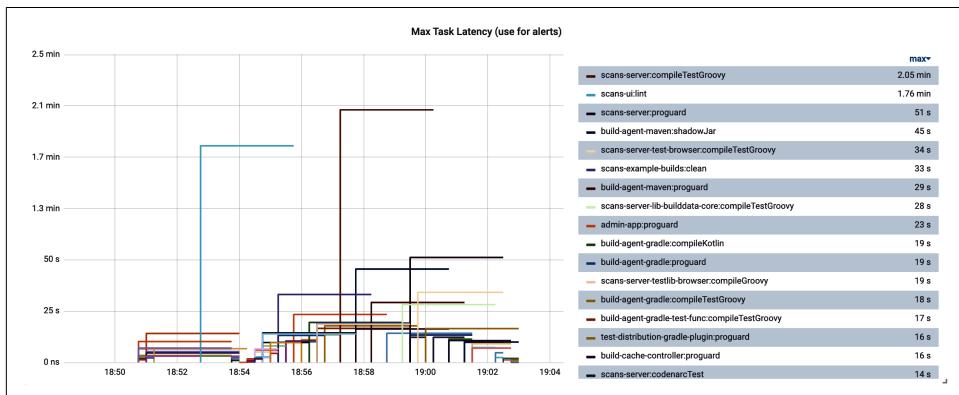


Figure 4-10. Top k can still yield many more items in the legend than k

In such cases, the legend is overwhelmed with labels, to the point where it is illegible. Use the Grafana options to shift the legend to a table on the right, and add a summary statistic like “maximum,” as shown in Figure 4-11. You can then click the summary statistic in the table to sort the legend-as-table by this statistic. Now when we look at the chart, we can quickly see which performers are overall worst for the time range that we are viewing.

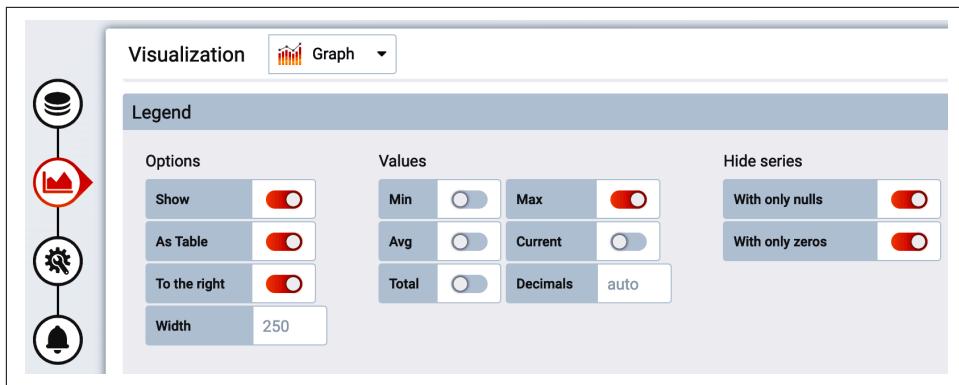


Figure 4-11. Overriding line styles for each outcome

Prometheus Rate Interval Selection

Throughout this chapter, we are going to see Prometheus queries that use [range vectors](#). I highly suggest using range vectors that are at least twice as long as the scrape interval (by default one minute). Otherwise, you risk missing datapoints due to slight variations in scrape timing that may cause adjacent datapoints to be just slightly more than the scrape interval apart. Similarly, if a service is restarted and a datapoint is missing, the rate function will not be able to make a rate during the gap or next datapoint until the interval contains at least two points. Using a higher interval for the rate avoids these problems. Because application startup may be longer than a scrape interval, depending on your application, if it is important to you to totally avoid gaps, you may choose a range vector longer than twice the scrape interval (something in fact closer to whatever application startup plus two intervals would be).

Range vectors are a somewhat unique concept to Prometheus, but the same principle applies in other contexts in other monitoring systems. For example, you'd want to construct a "min over interval" type query to compensate for potential gaps during application restart if you are setting a minimum threshold on an alert.

Gauges

A time series representation of a gauge presents more information about as compactly as an instantaneous gauge. It is just as obvious when a line crosses an alert threshold, and the historical information about the gauge's prior values provides useful context. As a result, the bottom chart is preferable in [Figure 4-12](#).

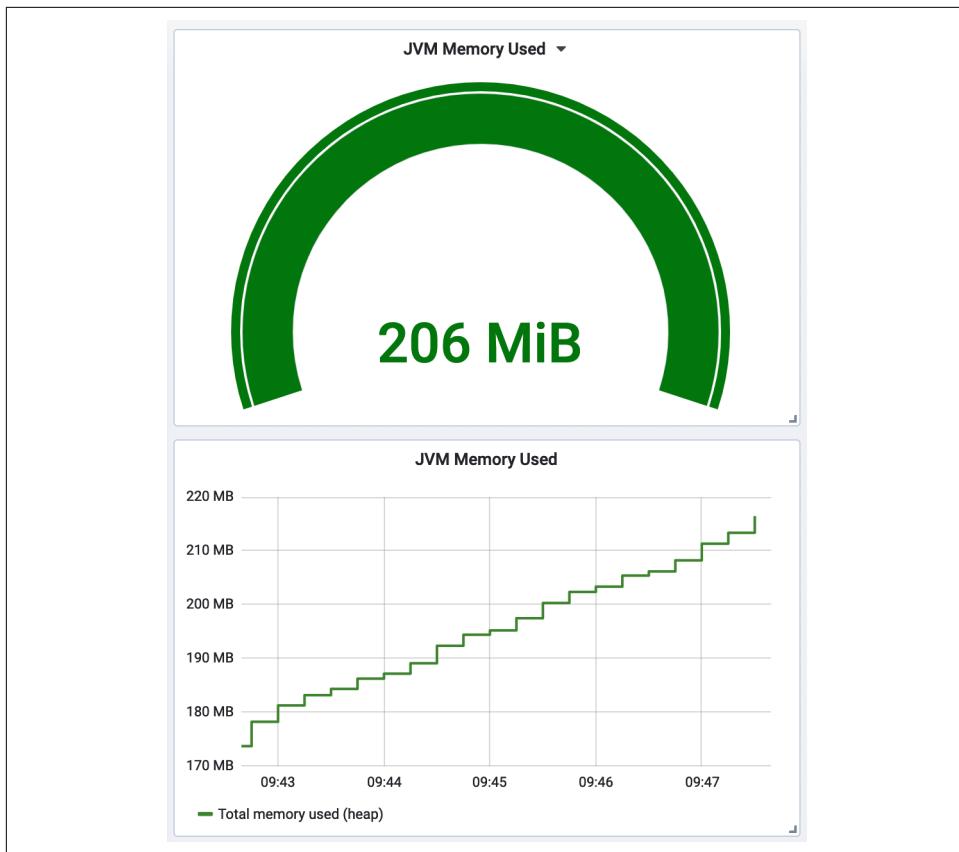


Figure 4-12. Prefer a line chart over an instantaneous gauge

Gauges have a tendency to be spiky. Thread pools can appear to be temporarily near exhaustion and then recover. Queues get full and then empty. Memory utilization in Java is especially tricky to alert on since short-term allocations can quickly appear to fill up a significant portion of allocated space only for garbage collection to sweep away much of the consumption.

One of the most effective methods to limit alert chattiness is to use a rolling count function, the results of which are shown in [Figure 4-13](#). In this way we can define an alert that only fires if a threshold is exceeded more than three times in the last five intervals, or some other combination of frequency and number of lookback intervals. The longer the lookback, the more time will elapse before the alert first fires, so be careful to not look back too far for critical indicators.

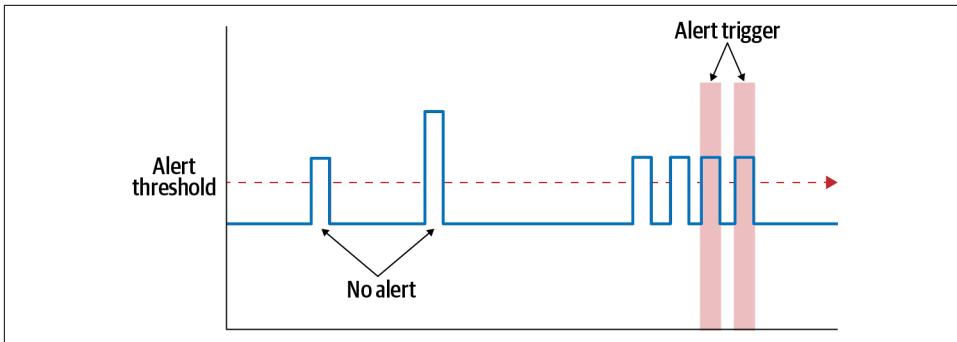


Figure 4-13. Rolling count to limit alert chattiness

Being instantaneous values, gauges are basically just graphed as is on each monitoring system. Counters are a little more nuanced.

Counters

Counters are often tested against a maximum (or less frequently, a minimum) threshold. The need to test against a threshold reinforces the idea that counters should be observed as rates rather than a cumulative statistic, regardless of how the statistic is stored in the monitoring system.

[Figure 4-14](#) shows an HTTP endpoint's request throughput as a rate (yellow solid line) and also the cumulative count (green dots) of all requests to this endpoint since the application process started. Also, the chart shows a fixed minimum threshold alert (red line and area) of 1,000 requests/second that has been set on this endpoint's throughput. This threshold makes sense relative to throughput represented as a rate (which in this window varies between 1,500 and 2,000 requests/second). It makes little sense against the cumulative count though, since the cumulative count is effectively a measure of both the rate of throughput and the longevity of the process. The longevity of the process is irrelevant to this alert.

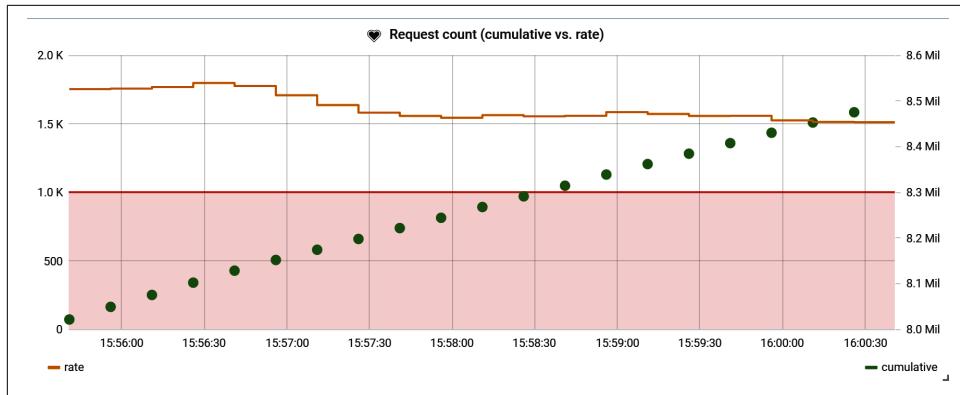


Figure 4-14. A counter with a minimum alert threshold on rate, with cumulative count displayed as well

Sometimes a fixed threshold is difficult to determine a priori. Also, the rate at which an event is occurring may fluctuate periodically based on something like peak and off-peak business hours. This is especially common with a throughput measure like requests/second, as seen in [Figure 4-15](#). If we set a fixed threshold on this service to detect when traffic was suddenly not reaching the service (a minimum threshold), we would have to set it somewhere below 40 RPS, the minimum throughput this service sees. Suppose the minimum threshold is set at 30 RPS. This alert fires when traffic drops below 75% of the expected value during off-peak hours, but only when traffic drops below 10% of the expected value during peak hours! The alert threshold is not equally valuable during all periods.

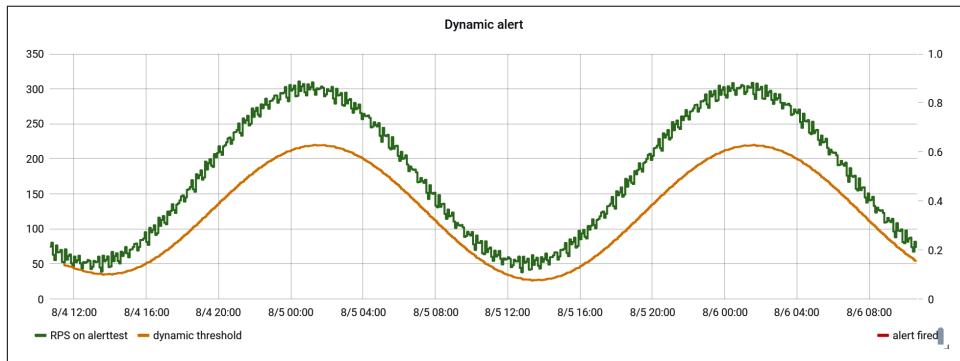


Figure 4-15. A service with periodic increases in traffic based on the time of day

In these cases, consider framing an alert in terms of finding sharp increases or decreases in rate. A good general approach to this, seen in [Figure 4-16](#), is to take the counter rate, apply a smoothing function to it, and multiply the smoothing function by some factor (85% in the example). Because the smoothing function naturally takes

at least a little time to respond to a sudden change in the rate, a test to ensure that the counter rate doesn't fall below the smoothed line detects sudden change without having to know what the expected rate is at all. A much more detailed explanation of statistical methods for smoothing for dynamic alerting is presented in [“Building Alerts Using Forecasting Methods”](#) on page 176.

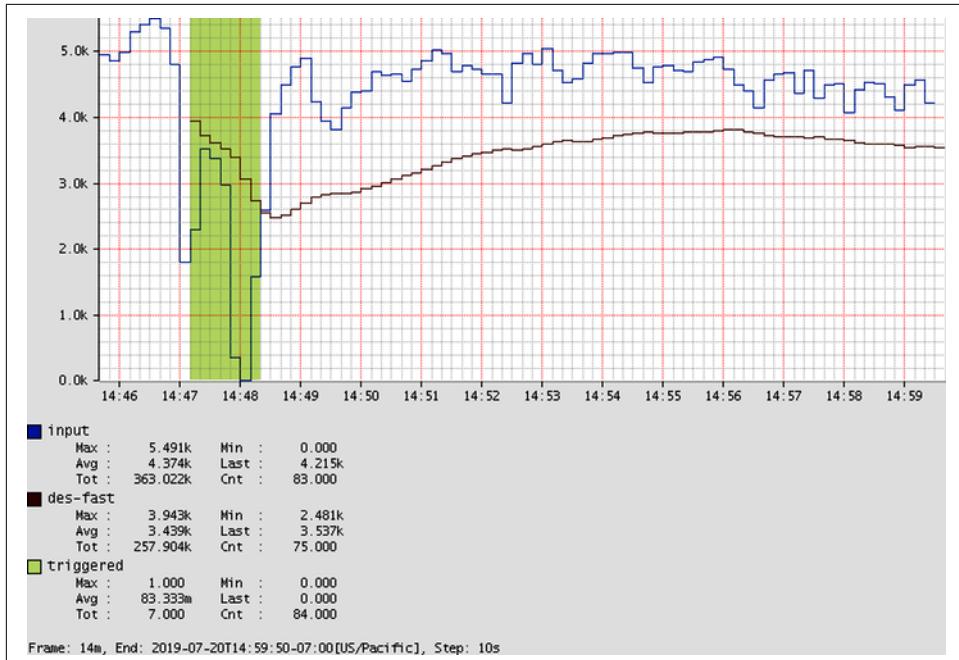


Figure 4-16. A counter with a double-exponentially smoothed threshold, forming a dynamic alert threshold

It is Micrometer's responsibility to ship the data to your monitoring system of choice in such a way that you can draw a rate representation of a counter in your chart. In the case of Atlas, counters are already shipped in a rate-normalized way, so a query for a counter already returns a rate value that can be directly plotted, as shown in [Example 4-3](#).

Example 4-3. Atlas counters are already a rate, so selecting them charts a rate

```
name,cache.gets,:eq,
```

Other monitoring systems expect cumulative values to be shipped to the monitoring system and include some sort of rate function for use at query time. [Example 4-4](#) would display roughly the same rate line as the Atlas equivalent, depending on what you select as the range vector (the time period in the []).

Example 4-4. Prometheus counters are cumulative, so we need to explicitly convert them to a rate

```
rate(cache_gets[2m])
```

There is one problem with the Prometheus rate function: when new tag values are added rapidly inside a chart's time domain, the Prometheus rate function can generate a NaN value as opposed to a zero. In [Figure 4-17](#), we are charting Gradle build task throughput over time. Since in this window, build tasks are uniquely described by project and task name, and once a task is complete it isn't incremented again, several new time series are coming into existence inside of the time domain we've selected for the chart.

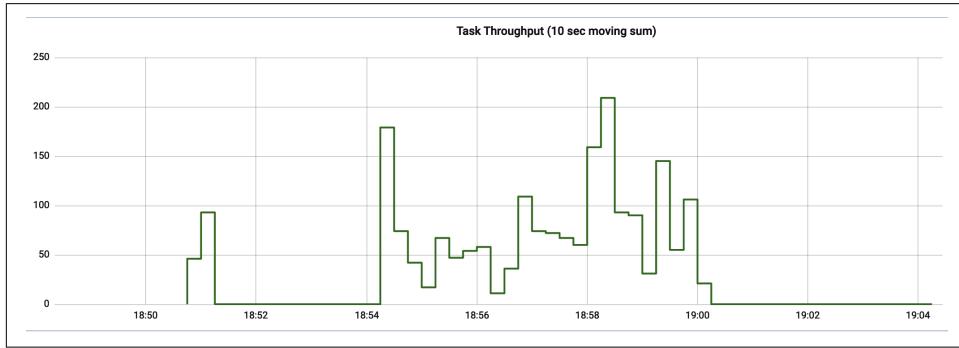


Figure 4-17. Zero-filling Prometheus counter rates when new tag values are coming into existence inside the time domain

The query in [Example 4-5](#) shows the method we can use to zero-fill the gaps.

Example 4-5. The query to zero-filling Prometheus counter rates

```
sum(gradle_task_seconds_count) by (gradle_root_project_name) -  
(  
    sum(gradle_task_seconds_count offset 10s) by (gradle_root_project_name) > 0 or  
    (  
        (sum(gradle_task_seconds_count) by (gradle_root_project_name)) * 0  
    )  
)
```

How to chart counters varies a bit from monitoring system to monitoring system. Sometimes we have to explicitly create rates, and sometimes counters are stored as rates up front. Timers have even more options.

Timers

A `Timer` Micrometer meter generates a variety of different time series with one operation. Wrapping a block of code with a timer (`timer.record(() -> { ... })`) is enough to collect data on throughput through this block, maximum latency (decaying over time), the total sum of latency, and optionally other distribution statistics like histograms, percentiles, and SLO boundaries.

On dashboards, latency is the most important to view, because it is most directly tied to user experience. After all, users care mostly about the performance of *their* individual requests. They care little to nothing about the total throughput the system is capable of, except indirectly to the extent that at a certain throughput level their response time is affected.

Secondarily, throughput can be included if there is an expectation of a certain shape to traffic (which may be periodic based on business hours, customer time zones, etc.). For example, a sharp decline in throughput during an expected peak period can be a strong indicator of a systemic problem where traffic that should be reaching the system is not.

For many cases, it is best to set alerts on maximum latency (in this case meaning maximum observed for each interval) and use high-percentile approximations like the 99th percentile for comparative analysis (see “[Automated Canary Analysis](#)” on [page 205](#)).



Set Timer Alerts on Maximum Latency

It is exceedingly common in Java applications for maximum timings to be an order of magnitude worse than the 99th percentile. It is best to set your alerts on maximum latency.

I didn’t discover the importance of even measuring maximum latency until after I left Netflix and was introduced to a [compelling argument](#) by Gil Tene for alerting on maximum latency. He makes a particularly visceral point about worst case, drawing an analogy to pacemaker performance and emphasizing that “your heart will keep beating 99.9% of the time” is not reassuring.” Always a sucker for a well-reasoned argument, I added maximum latency as a key statistic shipped by Micrometer `Timer` and `DistributionSummary` implementations just in time for the SpringOne conference in 2017. There I met a former colleague from Netflix and sheepishly suggested this new idea, conscious of the fact that Netflix wasn’t actually monitoring max latency. He immediately laughed off the idea and left for a talk, leaving me a bit deflated. A short while later, I got a message from him with the chart shown in [Figure 4-18](#), showing max latency an order of magnitude worse than P99 on a key

internal Netflix service (which he had gone and added max to as a quick experiment to test this hypothesis).

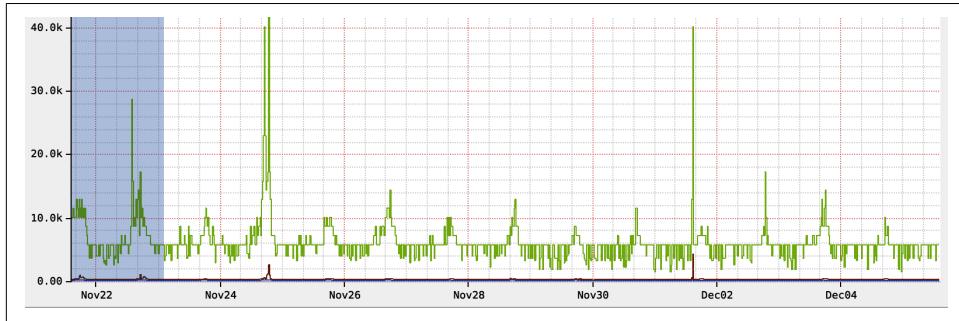


Figure 4-18. Max versus P99 in a Netflix logging service (in nanoseconds)

Even more amazing, Netflix had recently undergone an architectural shift that made P99 a little bit better but made max substantially worse! It's easy to argue it was actually worse off for having made the change. I cherish the memory of this interaction because it is such an acute illustration of how every organization has something it can learn from another: in this case a highly sophisticated monitoring culture at Netflix learned a trick from Domo which in turn learned it from Azul Systems.

In Figure 4-19, we see the order-of-magnitude difference between maximum and 99th percentile. Response latency tends to be tightly packed around the 99th percentile with at least one separate grouping near the maximum reflective of garbage collection, VM pauses, etc.

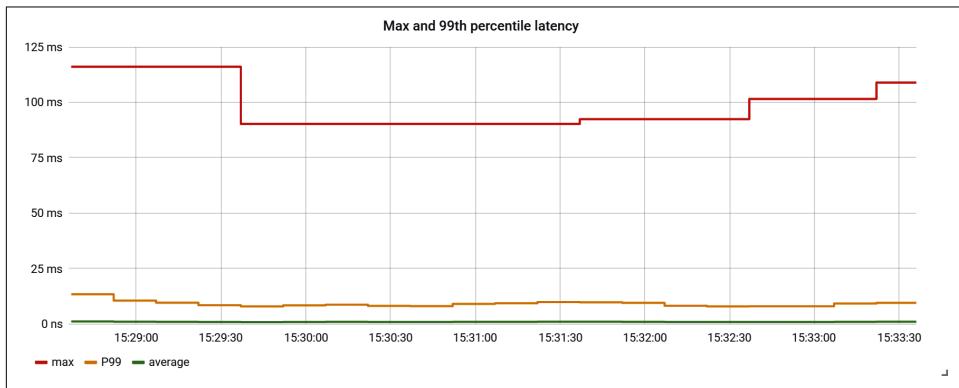


Figure 4-19. Maximum versus P99 latency

In Figure 4-20, a real-world service is exhibiting the characteristic that the average floats above the 99th percentile because requests are so densely packed around the 99th.

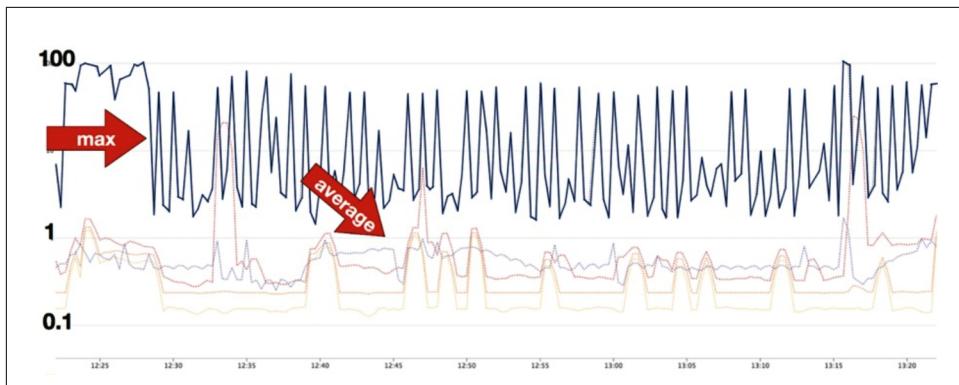


Figure 4-20. Average versus P99 latency

As insignificant as this top 1% may seem, real users are affected by these latencies, so it is important to recognize where that boundary is and compensate for it where needed. One recognized approach to limiting the effect of the top 1% is a client-side load-balancing strategy called hedge requests (see “[Hedge Requests](#)” on page 272).

Setting an alert on max latency is key (we’ll talk more about why in “[Latency](#)” on page 153). But once an engineer has been alerted to a problem, the dashboard that they use to start understanding the problem doesn’t necessarily need to have this indicator on it. It would be far more useful to see the distribution of latencies as a heatmap (as shown in [Figure 4-21](#)), which would include a nonzero bucket where the max is that caused the alert, to see how significant the problem is relative to the normative request coming through the system at that time. In a heatmap visualization, each vertical column represents a histogram (refer to “[Histograms](#)” on page 65 for a definition) at a particular time slice. The colored boxes represent the frequency of latencies that are in a range of times defined on the y-axis. So the normative latency an end user is experiencing should look “hot” and outliers look cooler.

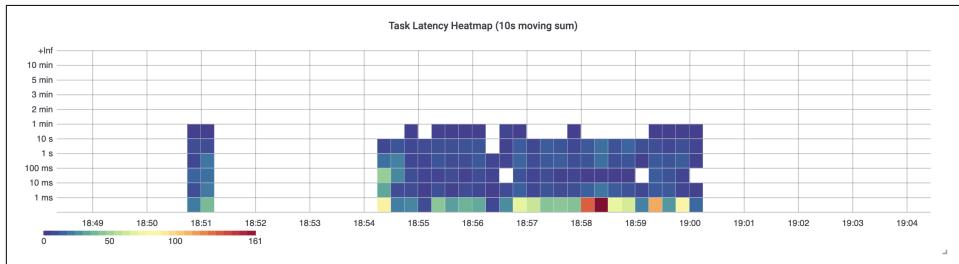


Figure 4-21. A timer heatmap

Are most requests failing close to the max value, or are there just one or a few stray outliers? The answer to this question likely affects how quickly an alerted engineer escalates the issue and brings others in to help. There’s no need to plot both the max

and heatmap on a diagnostic dashboard, as shown in [Figure 4-22](#). Just include the heatmap.



Figure 4-22. Max latency versus heatmap of latency distribution

The latency heatmap is also expensive to draw, since it involves retrieving potentially dozens or hundreds of buckets (which are individual time series in the monitoring system) for each time slice on the chart, for a total that often amounts to thousands of time series. This reinforces the idea that there's no reason to have this chart auto-updating on a prominent display somewhere hanging on a wall. Allow the alerting system to do its job and view the dashboard as needed to limit the load on the monitoring system.

The toolbox of useful representations has now grown to the point that a word of caution is necessary.

When to Stop Creating Dashboards

I visited with a former colleague of mine, now VP of operations at Datadog, in 2019. He lamented that, ironically, a lack of healthy moderation in dashboards built by customers is one of the key capacity problems confronting Datadog. Imagine legions of computer screens and TV displays arrayed around the world, each automatically refreshing at prescribed intervals a series of charts that look nice. I found this to be such a fascinating business problem, because clearly lots of TV displays showing Datadog branding improve the visibility and stickiness of the product while simultaneously producing an operational nightmare for a SaaS.

I've always found the "mission control" dashboard view a bit of a curiosity. After all, what is it about a chart that visually indicates to me a problem? If it's a sharp spike, a deep trough, or simply a value that has crept above all reasonable expectation, then an alert threshold can be created to define where that point of unacceptability is, and the metric can be monitored automatically (and around the clock).

As an on-call engineer, it's nice to receive an alert with an instantaneous visualization of the indicator (or a link to one). Ultimately, when we open alerts, we want to dig for information to discover a root cause (or sometimes determine that the alert isn't worth paying attention to). If the alert links to a dashboard, ideally that dashboard is configured in such a way as to allow immediate dimensional explosion or exploration. In other words, the TV display dashboard treats humans as a sort of low-attention span, notoriously unreliable alerting system.

The visualizations useful for alerting may not be useful to include on a dashboard at all, and not all charts on a dashboard are possible to build alerts on. For example, [Figure 4-22](#) shows two representations of the same timer: a decaying max and a heatmap. The alerting system is going to watch max, but when an engineer is alerted to the anomalous condition, it's much more useful to see the distribution of latencies around that time to know how severe the impact was (and the max should be captured in a latency bucket that's visible on the heatmap).

However, be careful about how you construct these queries! If you look closely you will see that there is no latency around 15 ms on the heatmap. The Prometheus range vector in this case was too close to the scrape interval, and the resulting momentary gap in the chart that is invisible hides the 15 ms latency! Since Micrometer decays max, we still see it on the max chart.

Heatmaps are also much more computationally expensive to render than a simple max line. For one chart this is fine, but add up this cost across many individual displays across business units in a large organization and this can be taxing on the monitoring system itself.

Charts aren't a substitute for alerts. Focus first on delivering them as alerts to the right people when they stray from acceptable levels rather than rushing to set up monitors.



A human constantly watching a monitor is just an expensive alert-ing system polling visually for unacceptable levels.

Alerts should be delivered to on-call personnel in such a way that they can quickly jump to a dashboard and start drilling down on the failing metric dimensionally to reason about where the problem is.

Not every alert or violation of an SLO needs be treated as a stop-the-world emergency.

Service Level Indicators for Every Java Microservice

Now that we have a sense of how to visually present SLIs on charts, we will turn our focus to the indicators you can add. They are presented in approximately the order of importance. So if you are following the incrementalist approach to adding charts and alerts, implement these in sequence.

Errors

When timing a block of code it's useful to differentiate between successful and unsuccessful operations for two reasons.

First, we can directly use the ratio of unsuccessful to total timings as a measure of the frequency of errors occurring in the system.

Also, successful and unsuccessful outcomes can have radically different response times, depending on the failure mode. For example, a `NullPointerException` resulting from making a bad assumption about the presence of some data in request input can fail early in a request handler. It then doesn't get far enough to call other downstream services, interact with the database, etc., where the majority of time is spent when a request is successful. In this case, unsuccessful requests that fail in this way will skew our perspective on the latency of the system. Latency will in fact appear better than it actually is! On the other hand, a request handler that makes a blocking downstream request to another microservice that is under duress and for which the response ultimately times out may exhibit a much higher-than-normal latency (something close to the timeout on the HTTP client making the call). By not segregating errors, we present an overly pessimistic view of the latency of our system.

Status tags (recall “[Naming Metrics](#)” on page 31) should be added to timing instrumentation in most cases on two levels.

Status

A tag that provides a detailed error code, exception name, or some other specific indicator of the failure mode

Outcome

A tag that provides a more course-grained error category that separates success, user-caused error, and service-caused error

When writing alerts, rather than trying to select a tag by matching a status code pattern (e.g., using Prometheus’s not-regex tag selector for `status !~ "2.."`), it is preferable to perform an exact match on the outcome tag (`outcome="SERVER_ERROR"`). By selecting “not 2xx,” we are grouping server errors, like the common HTTP 500 Internal Server Error, with errors caused by the user, like HTTP 400 Bad Request or HTTP 403 Forbidden. A high rate of HTTP 400s may indicate that you recently released code that contained an accidental backward incompatibility in an API, or it could indicate that a new end user (e.g., some other upstream microservice) is trying to onboard onto using your service and hasn’t gotten the payload right yet.



Panera Faced Chatty Alerts Not Distinguishing Client from Server Errors

Panera Bread, Inc., faced an overly chatty alert from an anomaly detector implemented by its monitoring system vendor for HTTP errors. It caused several email alerts in one day because a single user provided the wrong password five times. Engineers discovered that the anomaly detector didn’t differentiate between client and server error ratio! Alerts on client error ratio might be nice for intrusion detection, but the threshold would be much higher than server error ratio (and certainly higher than five errors in a short period of time).

An HTTP 500 basically always is your fault as a service owner and needs attention. At best, an HTTP 500 shines a spotlight on where more up-front validation could have instead yielded a useful HTTP 400 back to the end user. I think “HTTP 500—Internal Server Error” is too passive. Something like “HTTP 500—Sorry, It’s My Fault” feels better.

When you are writing your own timers, a common pattern involves using a `Timer` sample and deferring the determination of tags until it’s known whether the request will succeed or fail, as in [Example 4-6](#). The sample holds the state of the time that the operation started for you.

Example 4-6. Determining an error and outcome tag dynamically based on the result of an operation

```
Timer.Sample sample = Timer.start();
try {
    // Some operation that might fail...

    sample.stop(
        registry.timer(
            "my.operation",
            Tags.of(
                "exception", "none", ①
                "outcome", "success"
            )
        )
    );
} catch(Exception e) {
    sample.stop(
        registry.timer(
            "my.operation",
            Tags.of(
                "exception", e.getClass().getName(), ②
                "outcome", "failure"
            )
        )
    );
}
```

- ① Some monitoring systems like Prometheus expect a consistent set of tag keys to appear on metrics with the same name. So even though there is no exception here, we should tag it with some placeholder value like “none” to mirror what tags are present in the failure cases as well.
- ② Perhaps you have a way of better cataloging the failure conditions and can provide a more descriptive tag value here, but even adding the exception class name can go a long way toward understanding what *kinds* of failures there are. Null PointerException is a very different type of exception than a poorly handled connection timeout on a call to a downstream service. When error ratio spikes, it’s useful to be able to drill down on the exception name to get a brief glimpse at what the error is. From this exception name, you can hop over to your debuggability observability tools like logs and search for occurrences of the exception name around the time of the alert condition.



Be Careful with Class.getSimpleName(), etc., as a Tag Value

Be aware of the fact that `Class.getSimpleName()` and `Class.getCanonicalName()` can return null or empty values, for example in the case of anonymous class instances. If you use one of them as a tag value, at least null/empty check the value and fall back on `Class.getName()`.

For HTTP request metrics, for example, Spring Boot automatically tags `http.server.requests` with a `status` tag indicating the HTTP status code and an `outcome` tag that is one of `SUCCESS`, `CLIENT_ERROR`, or `SERVER_ERROR`.

Based on this tag, it is possible to plot the error *rate* per interval. Error rate is difficult to establish an alert threshold for, because it can fluctuate wildly under the same failure conditions, depending on how much traffic is coming through the system.

For Atlas, use the `:and` operator to select only `SERVER_ERROR` outcomes, as shown in [Example 4-7](#).

Example 4-7. Error rate of HTTP server requests in Atlas

```
# don't do this because it fluctuates with throughput!
name,http.server.requests,:eq,
outcome,SERVER_ERROR,:eq,
:and,
uri,$ENDPOINT,:eq,:cq
```

For Prometheus, use a tag selector, as shown in [Example 4-8](#).

Example 4-8. Error rate of HTTP server requests in Prometheus

```
# don't do this because it fluctuates with throughput!
sum(
  rate(
    http_server_requests_seconds_count{outcome="SERVER_ERROR", uri="$ENDPOINT"}[2m]
  )
)
```

If every 10th request fails, and 100 requests/second are coming through the system, then the error rate is 10 failures/second. If 1,000 requests/second are coming through the system, the error rate climbs to 100 failures/second! In both cases, the error *ratio* relative to throughput is 10%. This error ratio normalizes the rate and is easy to set a fixed threshold for. In [Figure 4-23](#), the error ratio hovers around 10–15% in spite of the fact that throughput, and therefore error rate, spikes.

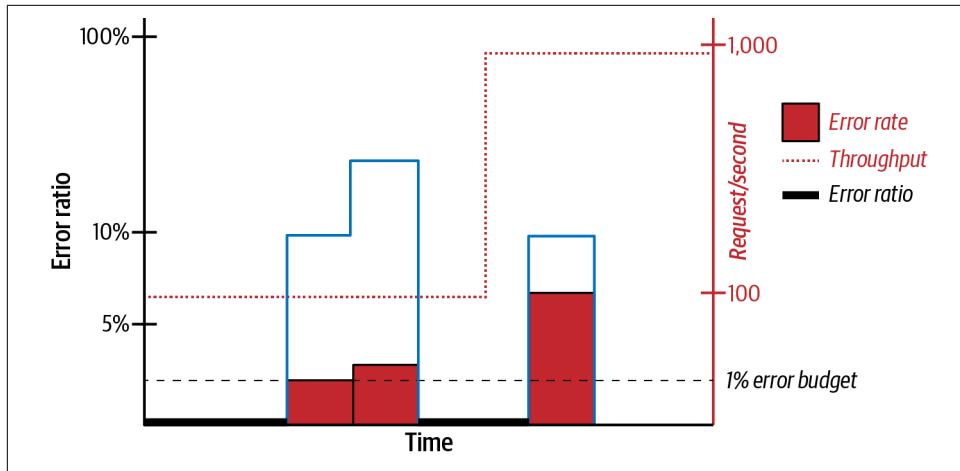


Figure 4-23. Error ratio versus error rate

The coarse-grained outcome tag is used to construct queries that represent the error ratio of the timed operation. In the case of `http.server.requests`, this is the ratio of SERVER_ERROR to the total number of requests.

For Atlas, use the `:div` function to divide SERVER_ERROR outcomes by the total count of all requests, as shown in [Example 4-9](#).

Example 4-9. Error ratio of HTTP server requests in Atlas

```
name,http.server.requests,:eq,
:dup,
outcome,SERVER_ERROR,:eq,
:div,
uri,$ENDPOINT,:eq,:cq
```

For Prometheus, use the `/` operator similarly, as in [Example 4-10](#).

Example 4-10. Error ratio of HTTP server requests in Prometheus

```
sum(
  rate(
    http_server_requests_seconds_count{outcome="SERVER_ERROR", uri="$ENDPOINT"}[2m]
  )
) /
sum(
  rate(
    http_server_requests_seconds_count{uri="$ENDPOINT"}[2m]
  )
)
```



Error Rate Is Better Than Error Ratio for Low-Throughput Services

In general, prefer error ratio to error rate, *unless* the endpoint has a very low throughput. In this case, even a small difference in errors can lead to wild shifts in the error ratio. It is more appropriate in these situations to pick a fixed error rate threshold.

Error rate and ratio are just one view of a timer. Latency is the other essential view.

Latency

Alert on maximum latency (in this case meaning maximum observed for each interval), and use high-percentile approximations like the 99th percentile for comparative analysis, as shown in “[Automated Canary Analysis](#)” on page 205. Popular Java web frameworks, as part of their “white box” (see “[Black Box Versus White Box Monitoring](#)” on page 24) autoconfiguration of metrics, offer instrumentation of inbound and outbound requests with rich tags. I’ll present details of Spring Boot’s automatic instrumentation of requests, but most other popular Java web frameworks have done something very similar with Micrometer.

Server (inbound) requests

Spring Boot autoconfigures a timer metric called `http.server.requests` for both blocking and reactive REST endpoints. If the latency of a particular endpoint(s) is a key indicator of the performance of an application and it will also be used for comparative analysis, then add the `management.metrics.distribution.percentiles-histogram.http.server.requests=true` property to your `application.properties` to export percentile histograms from your application. To be more fine-grained about enabling percentile histograms for a particular set of API endpoints, you can add the `@Timed` annotation in Spring Boot, like in [Example 4-11](#).

Example 4-11. Using @Timed to add histograms to just a single endpoint

```
@Timed(histogram = true)
@GetMapping("/api/something")
Something getSomething() {
    ...
}
```

Alternatively, you can add a `MeterFilter` that responds to a tag, as shown in [Example 4-12](#).

Example 4-12. A MeterFilter that adds percentile histograms for certain endpoints

```
@Bean
MeterFilter histogramsForSomethingEndpoints() {
    return new MeterFilter() {
        @Override
        public DistributionStatisticConfig configure(Meter.Id id,
            DistributionStatisticConfig config) {
            if(id.getName().equals("http.server.requests") &&
                id.getTag("uri").startsWith("/api/something")) {
                return DistributionStatisticConfig.builder()
                    .percentilesHistogram(true)
                    .build()
                    .merge(config);
            }
            return config;
        }
    };
}
```

For Atlas, [Example 4-13](#) shows how to compare max latency against some predetermined threshold.

Example 4-13. Atlas max API latency

```
name,http.server.requests,:eq,
statistic,max,:eq,
:and,
$THRESHOLD,
:gt
```

For Prometheus, [Example 4-14](#) is a simple comparison.

Example 4-14. Prometheus max API latency

```
http_server_requests_seconds_max > $THRESHOLD
```

The tags that are added to `http.server.requests` are customizable. For the blocking Spring WebMVC model, use a `WebMvcTagsProvider`. For example, we could extract information about the browser and its version from the “User-Agent” request header, as shown in [Example 4-15](#). This sample uses the MIT-licensed `Browscap` library to extract browser information from the user-agent header.

Example 4-15. Adding browser tags to Spring WebMVC metrics

```
@Configuration
public class MetricsConfiguration {
    @Bean
    WebMvcTagsProvider customizeRestMetrics() throws IOException, ParseException {
        UserAgentParser userAgentParser = new UserService().loadParser();

        return new DefaultWebMvcTagsProvider() {
            @Override
            public Iterable<Tag> getTags(HttpServletRequest request,
                HttpServletResponse response, Object handler, Throwable exception) {

                Capabilities capabilities = userAgentParser.parse(request
                    .getHeader("User-Agent"));

                return Tags
                    .concat(
                        super.getTags(request, response, handler, exception),
                        "browser", capabilities.getBrowser(),
                        "browser.version", capabilities.getBrowserMajorVersion()
                    );
            }
        };
    }
}
```

For Spring WebFlux (the nonblocking reactive model), configure a `WebFluxTagsProvider` similarly, as in [Example 4-16](#).

Example 4-16. Adding browser tags to Spring WebFlux metrics

```
@Configuration
public class MetricsConfiguration {
    @Bean
    WebFluxTagsProvider customizeRestMetrics() throws IOException, ParseException {
        UserAgentParser userAgentParser = new UserService().loadParser();

        return new DefaultWebFluxTagsProvider() {
            @Override
            public Iterable<Tag> httpRequestTags(ServerWebExchange exchange,
                Throwable exception) {

                Capabilities capabilities = userAgentParser.parse(exchange.getRequest()
                    .getHeaders().getFirst("User-Agent"));

                return Tags
                    .concat(
                        super.httpRequestTags(exchange, exception),
                        "browser", capabilities.getBrowser(),
                        "browser.version", capabilities.getBrowserMajorVersion()
                    );
            }
        };
    }
}
```

```

        );
    }
};

}
}

```

Note that the `http.server.requests` timer only begins timing a request once it is being processed by the service. If the request thread pool is routinely at capacity, requests from users are sitting in the thread pool waiting to be handled, and this elapse of time is very real to the user waiting for a response. The missing information in `http.server.requests` is one example of a larger problem first described by Gil Tene called coordinated omission (see “[Coordinated Omission](#)” on page 80), which comes in several other forms.

It is also useful to monitor latency from the perspective of the caller (client). In this case, by client I generally mean service-to-service callers and not human consumers to your API gateway or first service interaction. A service’s view of its own latency doesn’t include the effects of network delays or thread pool contention (e.g., Tomcat’s request thread pool or the thread pool of a proxy like Nginx).

Client (outbound) requests

Spring Boot also autoconfigures a timer metric called `http.client.requests` for both blocking and reactive *outbound* calls. This allows you to instead (or also) monitor a service’s latency from the perspective of all of its callers, provided they each make the same conclusion about what the name of the called service is. [Figure 4-24](#) shows three service instances calling the same service.

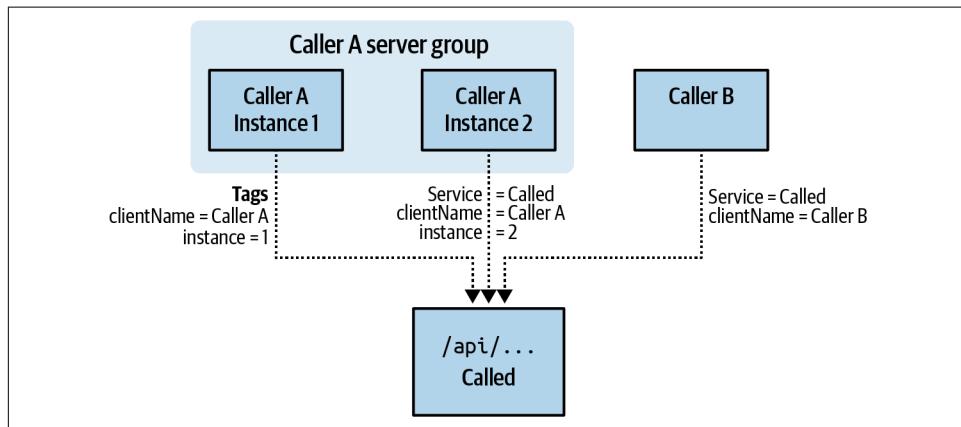


Figure 4-24. HTTP client metrics from multiple callers

We can identify the performance of a particular endpoint for the called service by selecting on the `uri` and `serviceName` tags. By aggregating over all other tags, we see

the performance of the endpoint across all callers. Dimensionally drilling down by the `clientName` tag would show the service's performance from just that client's perspective. Even if the called service processes every request in the same amount of time, the client perspective could vary (e.g., if one client is deployed in a different zone or region). Where there is the possibility for this variance between clients, you can use something like Prometheus's `topk` query to compare against an alert threshold so that the totality of the experience of an endpoint's performance for all clients doesn't wash away the outlier for some particular client, as shown in [Example 4-17](#).

Example 4-17. Max outbound request latency by client name

```
topk(  
    1,  
    sum(  
        rate(  
            http_client_requests_seconds_max{serviceName="CALLED", uri="/api/..."}[2m]  
        )  
    ) by (clientName)  
) > $THRESHOLD
```

To autoconfigure HTTP client instrumentation for Spring's `RestTemplate` (blocking) and `WebClient` (nonblocking) interfaces, you do need to treat path variables and request parameters a certain way. Specifically, you have to let the implementations do path variable and request parameter substitution for you rather than using string concatenation or a similar technique to construct a path, as shown in [Example 4-18](#).

Example 4-18. Allowing RestTemplate to handle path variable substitution

```
@RestController  
public class CustomerController { ①  
    private final RestTemplate client;  
  
    public CustomerController(RestTemplate client) {  
        this.client = client;  
    }  
  
    @GetMapping("/customers")  
    public Customer findCustomer(@RequestParam String q) {  
        String customerId;  
        // ... Look up customer ID according to 'q'  
  
        return client.getForEntity(  
            "http://customerService/customer/{id}?detail={detail}",  
            Customer.class,  
            customerId,  
            "no-address"  
        );  
    }  
}
```

```

}

...

@Configuration
public class RestTemplateConfiguration {
    @Bean
    RestTemplateBuilder restTemplateBuilder() { ②
        return new RestTemplateBuilder()
            .addAdditionalInterceptors(..)
            .build();
    }
}

```

- ① Sounds nefarious?
- ② To take advantage of Spring Boot's autoconfiguration of `RestTemplate` metrics, ensure that you are creating any custom bean wirings for `RestTemplateBuilder` and not `RestTemplate` (and note that Spring also provides a `RestTemplateBuilder` for you automatically with the defaults via autoconfiguration). Spring Boot attaches an additional metrics interceptor to any such beans that it finds. Once the `RestTemplate` is created, it is too late for this configuration to take place.

The idea is that the `uri` tag should still contain the requested path with path variables *pre-substitution* so that you can reason about the total number and latency of requests going to that endpoint regardless of what particular values were being looked up. Also, this is essential to controlling the total number of tags that the `http.client.requests` metrics contains. Allowing unbounded growth in unique tags would eventually overwhelm the monitoring system (or get really expensive for you if the monitoring system vendor charges by time series).

The equivalent for the nonblocking `WebClient` is shown in [Example 4-19](#).

Example 4-19. Allowing WebClient to handle path variable substitution

```

@RestController
public class CustomerController { ①
    private final WebClient client;

    public CustomerController(WebClient client) {
        this.client = client;
    }

    @GetMapping("/customers")
    public Mono<Customer> findCustomer(@RequestParam String q) {
        Mono<String> customerId;
        // ... Look up customer ID according to 'q', hopefully in a non-blocking way
    }
}

```

```

        return customerId
            .flatMap(id -> webClient
                .get()
                .uri(
                    "http://customerService/customer/{id}?detail={detail}",
                    id,
                    "no-address"
                )
                .retrieve()
                .bodyToMono(Customer.class)
            );
    }
}

...

```

```

@Configuration
public class WebClientConfiguration {
    @Bean
    WebClient.Builder webClientBuilder() { ②
        return WebClient
            .builder();
    }
}

```

- ➊ Sounds nefarious?
- ➋ Make sure you are creating bean wirings for `WebClient.Builder` and not `WebClient`. Spring Boot attaches an additional metrics `WebClientCustomizer` to the builder, not the completed `WebClient` instance.

While the default set of tags that Spring Boot adds to client metrics is reasonably complete, it is customizable. It is especially common to tag metrics with the value of some request header (or response header). Be sure when you add tag customizations that the total number of possible tag values is well bounded. You shouldn't add tags for things like unique customer ID (when you can have more than maybe 1,000 customers), a randomly generated request ID, etc. Remember, the purpose of metrics is to get an idea of aggregate performance, not the performance of some individual request.

As a slightly different example than the one we used in `http.server.requests` tag customization earlier, we could additionally tag the retrievals of customers by their subscription level, where subscription level is a response header on the retrieval of a customer by ID. By doing so, we could chart the latency and error ratio of the retrieval of premium customers versus basic customers separately. Perhaps the business places a higher level of expectation on the reliability or performance of requests

to premium customers, manifesting in a tighter service level agreement based on this custom tag.

To customize tags for `RestTemplate`, add your own `@Bean RestTemplateExchangeTagsProvider`, as shown in [Example 4-20](#).

Example 4-20. Allowing RestTemplate to handle path variable substitution

```
@Configuration
public class MetricsConfiguration {
    @Bean
    RestTemplateExchangeTagsProvider customizeRestTemplateMetrics() {
        return new DefaultRestTemplateExchangeTagsProvider() {
            @Override
            public Iterable<Tag> getTags(String urlTemplate,
                HttpRequest request, ClientHttpResponse response) {

                return Tags.concat(
                    super.getTags(urlTemplate, request, response),
                    "subscription.level",
                    Optional
                        .ofNullable(response.getHeaders().getFirst("subscription")) ❶
                        .orElse("basic")
                );
            }
        };
    }
}
```

- ❶ Beware that `response.getHeaders().get("subscription")` can potentially return `null!` So whether we use `get` or `getFirst`, we need to `null` check somehow.

To customize tags for `WebClient`, add your own `@Bean WebClientExchangeTagsProvider`, as shown in [Example 4-21](#).

Example 4-21. Allowing WebClient to handle path variable substitution

```
@Configuration
public class MetricsConfiguration {
    @Bean
    WebClientExchangeTagsProvider webClientExchangeTagsProvider() {
        return new DefaultWebClientExchangeTagsProvider() {
            @Override
            public Iterable<Tag> tags(ClientRequest request,
                ClientResponse response, Throwable throwable) {

                return Tags.concat(
                    super.tags(request, response, throwable),

```

```
        "subscription.level",
        response.headers().header("subscription").stream()
            .findFirst()
            .orElse("basic")
    );
}
};
```

To this point we've focused on latency and errors. Now let's consider a common saturation measurement related to memory consumption.

Garbage Collection Pause Times

Garbage collection (GC) pauses often delay the delivery of a response to a user request, and they can be a bellwether of an impending “out of memory” application failure. There are a few ways we can look at this indicator.

Max pause time

Set a fixed alert threshold on the maximum GC pause time you find acceptable (knowing that a GC pause directly contributes to end-user response time as well), potentially selecting different thresholds for minor and major GC types. Plot the max from the `jvm.gc.pause` timer to set your thresholds, as shown in Figure 4-25. A heat-map of pause times may also be interesting if your application undergoes frequent pauses and you want to understand what typical behavior looks like over time.

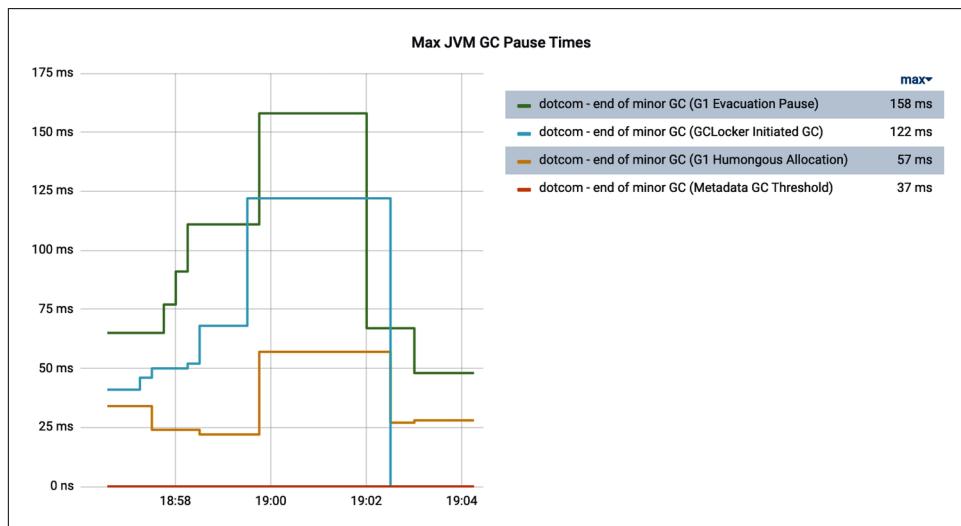


Figure 4-25. Max garbage collection pause times

Proportion of time spent in garbage collection

Since `jvm.gc.pause` is a timer, we can look at its sum independently. Specifically, we can add the increases in this sum over an interval of time and divide it by the interval to determine what proportion of the time the CPU is engaged in doing garbage collection. And since our Java process does nothing else during these times, when a significant enough proportion of time is spent in GC, an alert is warranted. [Example 4-22](#) shows the Prometheus query for this technique.

Example 4-22. Prometheus query for time spent in garbage collection by cause

```
sum( ①
    sum_over_time( ②
        sum(increase(jvm_gc_pause_seconds_sum[2m])[1m:] ③
    )
) / 60 ④
```

- ① Sums over all the individual causes, like “end of minor GC.”
- ② The total time spent in an individual cause in the last minute.
- ③ This is the first time we’ve seen a Prometheus [subquery](#). It allows us to treat the operation on the two indicators as a range vector for input into `sum_over_time`.
- ④ Since `jvm_gc_pause_seconds_sum` has a unit of seconds (and therefore so do the sums) and we’ve summed over a 1-minute period, divide by 60 seconds to arrive at a percentage in the range [0, 1] of the time we’ve spent in GC in the last minute.

This technique is flexible. You can use a tag to select particular GC causes and evaluate, for example, only the proportion of time spent in major GC events. Or, like we’ve done here, you can simply sum over all the causes and reason about overall GC time in a given interval. More than likely, you’re going to find that if you do separate these sums by cause, minor GC events don’t contribute that significantly to the proportion of time spent in GC. The app being monitored in [Figure 4-26](#) was undergoing minor collections every minute and, unsurprisingly, it still only spent 0.0182% of its time in GC-related activities.

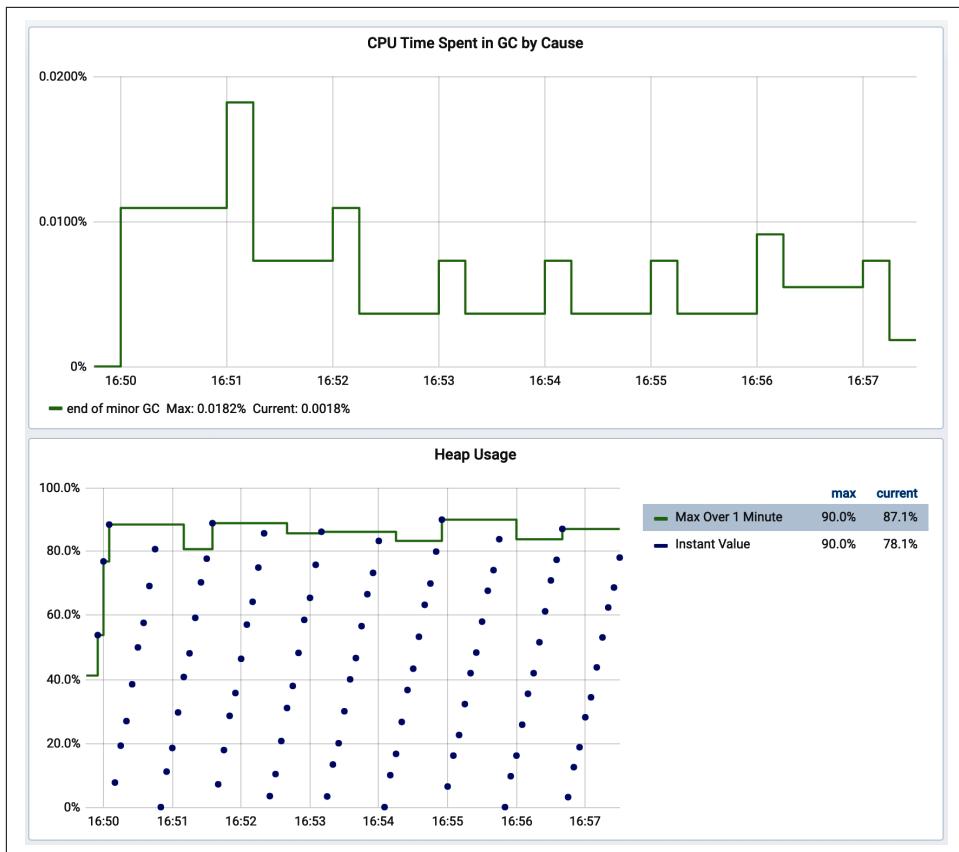


Figure 4-26. The proportion of time spent in minor GC events

If you aren't using a monitoring system that provides aggregation functions like `sum_over_time`, Micrometer provides a meter binder called `JvmHeapPressureMetrics`, shown in [Example 4-23](#), that precomputes this GC overhead and ships a gauge called `jvm.gc.overhead` that is a percentage in the range $[0, 1]$ that you can then set a fixed threshold alert against. In a Spring Boot app, you can simply add an instance of `JvmHeapPressureMetrics` as a `@Bean` and it will be bound to your meter registries automatically.

Example 4-23. Configuring the JVM heap pressure meter binder

```
MeterRegistry registry = ...  
  
new JvmHeapPressureMetrics(  
    Tags.empty(),  
    Duration.ofMinutes(1), ①  
    Duration.ofSeconds(30)  
).register(meterRegistry);
```

- ① Controls the lookback window.

The presence of any humongous allocation

In addition to choosing one of the above forms for monitoring time spent in GC, it's also a good idea to set an alert on the presence of humongous allocation that GC causes in the G1 collector, because it indicates that somewhere in your code you are allocating an object >50% of the total size of the Eden space! More than likely, there is a way to refactor the application to avoid such an allocation by chunking or streaming data. A humongous allocation could occur while doing something like parsing an input or retrieving an object from a datastore that is not yet as big as the application could theoretically see, and a larger object very well could bring the application down.

For this, specifically, you are looking for a nonzero count for `jvm.gc.pause` where the cause tag is equal to `G1 Humongous Allocation`.

Way back in “[Monitoring for Availability](#)” on page 7, we mentioned that saturation metrics are usually preferable to utilization metrics when you have a choice between the two. This is certainly true of memory consumption. The views of time spent in garbage collection as a measure of memory resource problems are easier to get right. There are some interesting things we can do with utilization measurements, too, if we're careful.

Heap Utilization

The Java heap is separated into several pools with each pool having a defined size. Java object instances are created in heap space. The most important parts of the heap are as follows:

Eden space (young generation)

All new objects are allocated here. A minor garbage collection event occurs when this space is filled.

Survivor space

When a minor garbage collection occurs, any live objects (that demonstrably still have references and therefore cannot be collected) are copied to the survivor space. Objects that reach the survivor space have their age incremented, and after an age threshold is met, are promoted to old generation. Promotion may happen prematurely if the survivor space cannot hold all of the live objects in young generation (objects skip the survivor space and go straight to old generation). This last fact will be key in how we measure dangerous levels of allocation pressure.

Old generation

This is where long-surviving objects are stored. When objects are stored in the Eden space, an age for that object is set; and when it reaches that age, the object is moved to old generation.

Fundamentally, we want to know when one or more of these spaces is getting and *staying* too “full.” This is a tricky thing to monitor because JVM garbage collection by design kicks in as spaces get full. So having a space fill up isn’t itself an indicator of a problem. What is concerning is when it stays full.

Micrometer’s `JvmMemoryMetrics` meter binder automatically collects JVM memory pool usage, along with the current total maximum heap size (since this can increase and decrease at runtime). Most Java web frameworks automatically configure this binder.

Several metrics are plotted in [Figure 4-27](#). The most straightforward idea for how to measure heap pressure is to use a simple fixed threshold, such as a percentage of total heap consumed. As we can see, the fixed threshold alert will fire far too frequently. The earliest alert is triggered at 11:44, well before it is apparent that a memory leak is present in this application. Even though the heap temporarily exceeds the percentage-of-total-heap threshold we have set, garbage collection events routinely bring total consumption back under the threshold.

In [Figure 4-27](#):

- The solid vertical bars together are a stack graph of memory consumption by space.
- The thin line around the 30.0 M level is the maximum heap space allowed. Notice how this fluctuates as the JVM attempts to pick the right value between initial heap size (`-Xms`) and max heap size (`-Xmx`) for the process.
- The bold line around 24.0 M level represents a fixed percentage of this maximum allowed memory. This is the threshold. It is a fixed threshold relative to the max, but dynamic in the sense that it is a percentage of the max which itself can fluctuate.

- The lighter bars represent points where actual heap utilization (the top of the stack graph) exceeds the threshold. This is the “alert condition.”

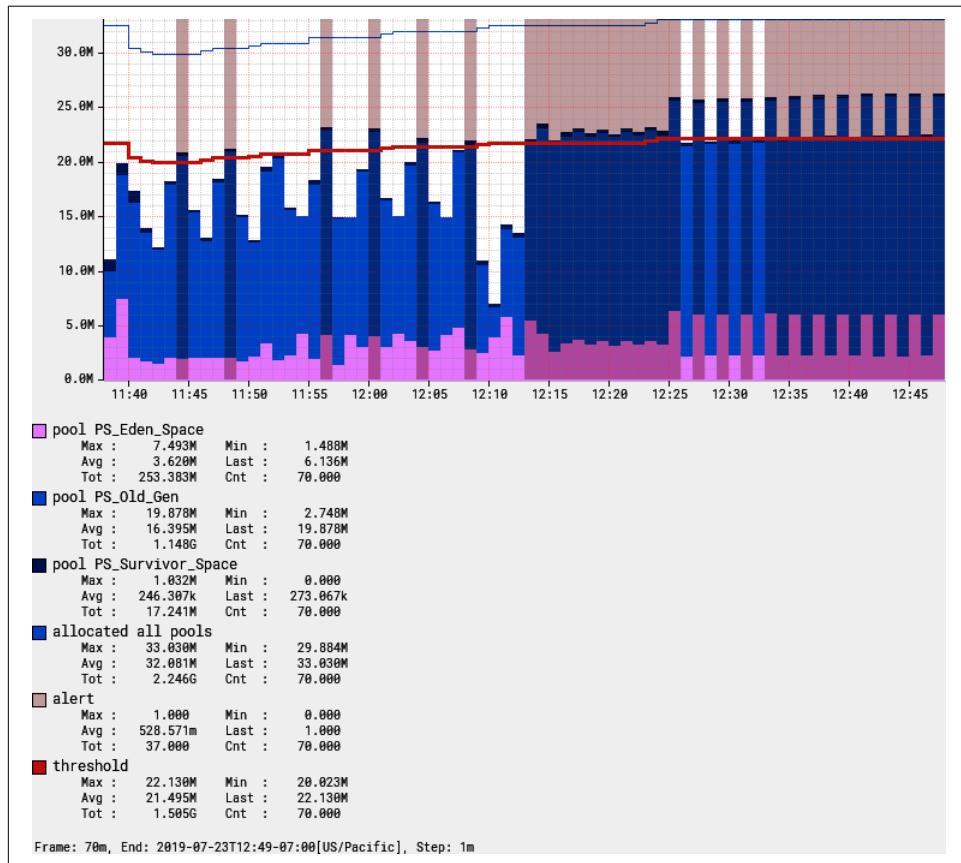


Figure 4-27. An alert on memory utilization with a fixed threshold

So this simple fixed threshold won’t work. There are better options available, depending on the capabilities of your target monitoring system.

Rolling count occurrences of heap space filling up

By using a feature like the rolling count function in Atlas, we can alert only when the heap exceeds the threshold—say, three out of the five prior intervals—indicating that in spite of the garbage collector’s best effort, heap consumption continues to be a problem (see Figure 4-28).

Unfortunately, not many monitoring systems have a function like Atlas’s rolling count. Prometheus can do something like this with its `count_over_time` operation, but it is tricky to get a similar “three out of five” dynamic.

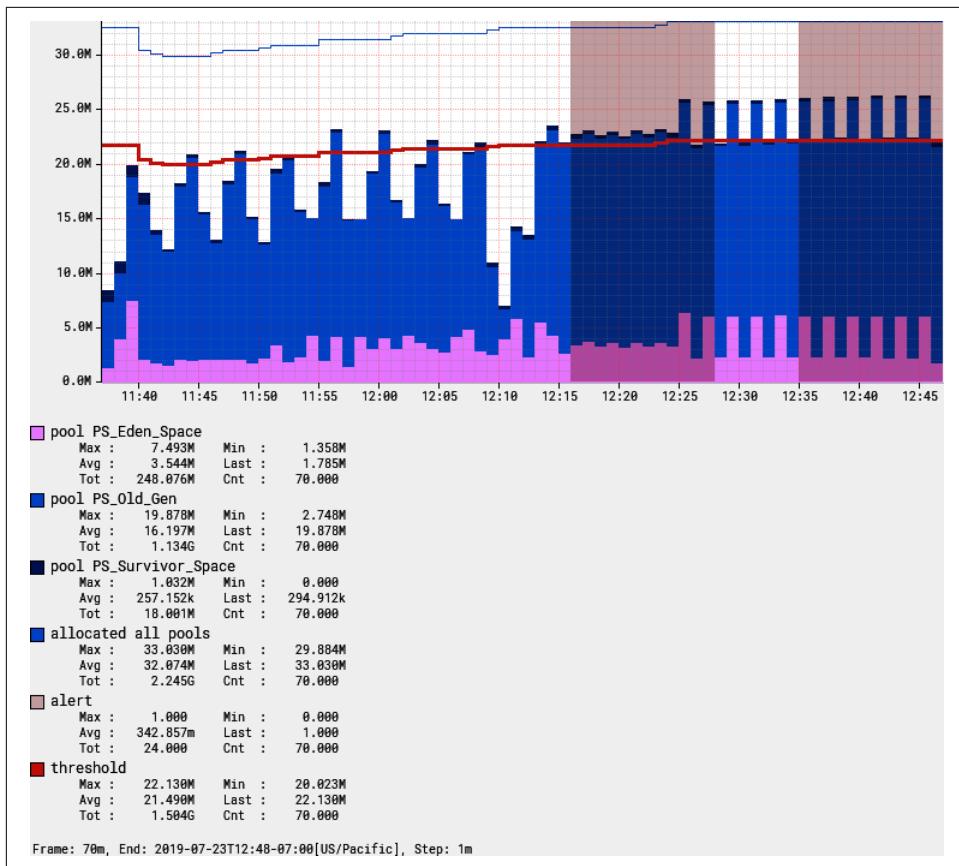


Figure 4-28. Using rolling count to limit alert chattiness

There is an alternative approach that also works well.

Low pool memory after collection

Micrometer's `JvmHeapPressureMetrics` adds a gauge `jvm.memory.usage.after.gc` for the percentage of Old Generation heap used after the last garbage collection event.

`jvm.memory.usage.after.gc` is a percentage expressed in the range [0, 1]. When it is high (a good starting alert threshold is greater than 90%), garbage collection isn't able to sweep up much garbage. So long-term pause events which occur when Old Generation is swept can be expected to occur frequently, and frequent long-term pauses both significantly degrade the performance of the app and ultimately lead to `OutOfMemoryException` fatal errors.

A subtle variation on measuring low pool memory after collection is also effective.

Low total memory

This technique involves mixing indicators from heap usage and garbage collection activity. A problem is indicated when they *both* exceed a threshold:

`jvm.gc.overhead > 50%`

Notice that this is a lower alert threshold than suggested in “[Garbage Collection Pause Times](#)” on page 161 for the same indicator (where we suggested 90%). We can be more aggressive about this indicator because we are pairing it with a utilization indicator.

`jvm.memory.used/jvm.memory.max > 90% at any time in the last 5 minutes`

Now we have an idea that GC overhead is going up because one or more of the pools keep filling up. You could constrain this to just the Old Generation pool as well if your app generates a lot of short-term garbage under normal circumstances.

The alert criteria for the GC overhead indicator is a simple test against the gauge value.

The query for total memory usage is a little less obvious. The Prometheus query is shown in [Example 4-24](#).

Example 4-24. Prometheus query for maximum memory used in the last five minutes

```
max_over_time(  
  (  
    jvm_memory_used_bytes{id="G1 Old Gen"} /  
    jvm_memory_committed_bytes{id="G1 Old Gen"}  
  )[5m:]  
)
```

To understand better what `max_over_time` does, [Figure 4-29](#) shows the total amount of Eden space (`jvm.memory.used{id="G1 Eden Space"}` in this case) consumed at several points in time (the dots) and the result of applying a one-minute `max_over_time` query to the same query (the solid line). It is a moving maximum window over a prescribed interval.

As long as heap usage is going up (and hasn’t been below the current value in the lookback window), the `max_over_time` tracks it exactly. Once a garbage collection event happens, the current view of usage drops and `max_over_time` “sticks” at the higher value for the lookback window.

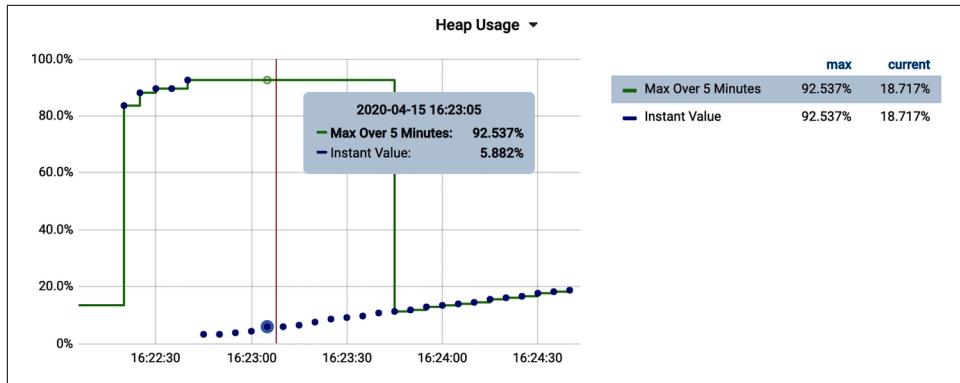


Figure 4-29. Prometheus max_over_time looking at max Eden space used in a one-minute lookback

This is also the first time we've considered an alert that is based on more than one condition. Alerting systems generally allow for the boolean combination of multiple criteria. In [Figure 4-30](#), assuming that the jvm.gc.overhead indicator represents Query A and the usage indicator represents Query B, an alert can be configured in Grafana on both together.

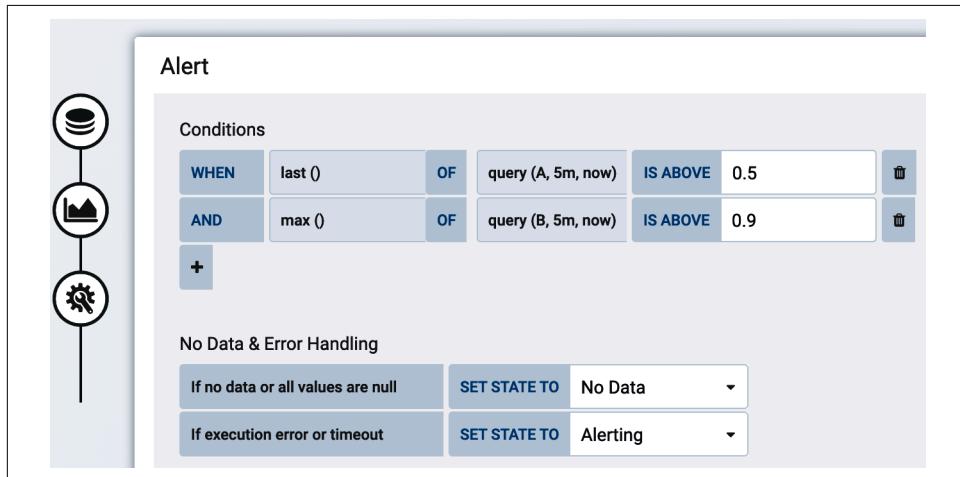


Figure 4-30. Configuring a Grafana alert based on two indicators for low total memory

Another common utilization measurement is CPU, which doesn't have an easy saturation analog.

CPU Utilization

CPU usage is a common utilization alert to set, but unfortunately it is difficult to establish a general rule for what is a healthy amount of CPU because of the different programming models described below—this will have to be determined for each application, depending on its characteristics.

For example, a typical Java microservice running on Tomcat and serving requests using a blocking servlet model will typically consume available threads in the Tomcat thread pool well before overutilizing the CPU. In these types of applications, high memory saturation is far more common (e.g., lots of excess garbage created in the handling of each request or large request/response bodies).

A Java microservice running on Netty and using a reactive programming model all the way down will accept a much higher throughput per instance, and so CPU utilization tends to be much higher. In fact, better saturating available CPU resources is commonly cited as an advantage of the reactive programming model!



On Some Platforms, Consider CPU and Memory Utilization Together Before Resizing Instances

A common feature of platform as a service is the simplification of instance sizing down to the amount of CPU or memory you desire with the other variable growing proportionally as you move up sizes. In the case of Cloud Foundry, this proportionality between CPU and memory was decided at a time when a blocking model of request handling like Tomcat was almost universal. As noted, CPU tends to be underused in this model. I once consulted at a company that had adopted a nonblocking reactive model for its application, and noticing that memory was significantly underutilized, I downsized the company's Cloud Foundry instances to not consume as much memory. But CPU is allocated to instances on this platform proportionally to how much memory is requested. By picking a lower memory requirement, the company also inadvertently starved its reactive app of the CPU it would otherwise have so efficiently saturated!

Micrometer exports two key metrics for CPU monitoring, which are listed in [Table 4-1](#). Both of these metrics are reported from Java's operating system MXBean (`ManagementFactory.getOperatingSystemMXBean()`).

Table 4-1. Micrometer reported processor metrics

Metric	Type	Description
<code>system.cpu.usage</code>	Gauge	The recent CPU usage for the whole system
<code>process.cpu.usage</code>	Gauge	The recent CPY usage for the Java virtual machine process

For the most common case in the enterprise where an application is serving requests via a blocking servlet model, testing against a fixed threshold of 80% is reasonable. Reactive applications will need to be tested empirically to determine their appropriate saturation point.

For Atlas, use the `:gt` function, as shown in [Example 4-25](#).

Example 4-25. Atlas CPU alert threshold

```
name,process.cpu.usage,:eq,  
0.8,  
:gt
```

For Prometheus, [Example 4-26](#) is just a comparison expression.

Example 4-26. Prometheus CPU alert threshold

```
process_cpu_usage > 0.8
```

Process CPU usage should be plotted as a percentage (where the monitoring system should expect an input in the range 0–1 to appropriately draw the *y*-axis). Take note of the *y*-axis in [Figure 4-31](#) for what this should look like.

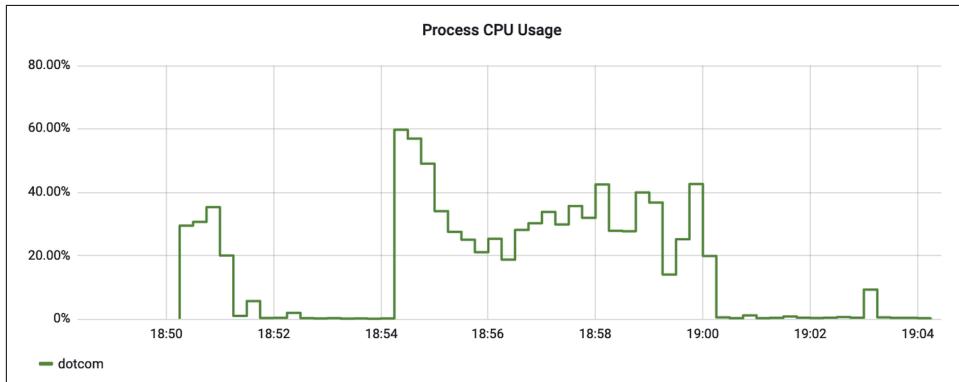


Figure 4-31. Process CPU usage as a percentage

In Grafana, “percent” is one of the units selectable in the “Visualization” tab. Make sure to select the option for “percent (0.0-1.0),” as shown in [Figure 4-32](#).

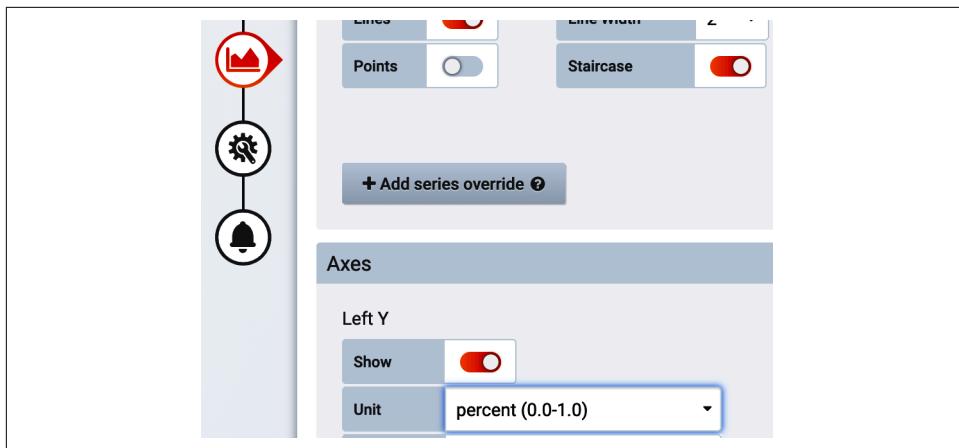


Figure 4-32. Grafana percent unit

There is one more resource-based indicator you should measure on every application related to file descriptors.

File Descriptors

The “ulimits” Unix feature limits how many resources a single user can use, including concurrently open file descriptors. File descriptors are not just consumed for file access, but also for network connections, database connections, etc.

You can view your shell’s current ulimits with `ulimit -a`. The output is shown in [Example 4-27](#). On many operating systems 1,024 is the default limit for open file descriptors. Scenarios like each service request requiring access to read or write a file where the number of concurrent threads can exceed the operating system limit are vulnerable to this issue. Throughput in the thousands of simultaneous requests is not unreasonable for a modern microservice, especially a nonblocking one.

Example 4-27. Output of ulimit -a in a Unix shell

```
$ ulimit -a
...
open files (-n) 1024 ①
...
cpu time (seconds, -t) unlimited
max user processes (-u) 63796
virtual memory (kbytes, -v) unlimited
```

- ① This represents the number of *allowable* open files, not the number of currently open files.

This problem isn't necessarily common, but the impact of reaching the file descriptor limit can be fatal, causing the application to stop responding entirely, depending on how file descriptors are used. Unlike an out-of-memory error or fatal exception, often the application will simply block but appear to be in service still, so this problem is especially pernicious. Because monitoring file descriptor utilization is so cheap, alert on this on every application. Applications using common techniques and web frameworks will probably never exceed 5% file descriptor utilization (and sometimes much lower); but when a problem sneaks in, it is trouble.



Experiencing the File Descriptor Problem While Writing This Book

I've known for some time to monitor this, but never actually experienced a problem myself until writing this book. A Go build step involved in building Grafana from source repeatedly hung, never completing. Evidently the Go dependency resolution mechanism doesn't carefully limit the number of open file descriptors!

An application which may have sockets open to hundreds of callers, HTTP connections to downstream services, connections open to datasources, and data files open could hit the limit of file descriptors. When a process runs out of file descriptors, it tends not to end well. You may see errors in logs like [Example 4-28](#) and [Example 4-29](#).

Example 4-28. Tomcat exhausted file descriptors accepting a new HTTP connection

```
java.net.SocketException: Too many open files
  at java.net.PlainSocketImpl.socketAccept(Native Method)
  at java.net.AbstractPlainSocketImpl.accept(AbstractPlainSocketImpl.java:398)
```

Example 4-29. Java failing to open a file when file descriptors are exhausted

```
java.io.FileNotFoundException: /myfile (Too many open files)
  at java.io.FileInputStream.open(Native Method)
```

Micrometer reports two metrics shown in [Table 4-2](#) to alert you to a file descriptor problem in your applications.

Table 4-2. Micrometer-reported file descriptor metrics

Metric	Type	Description
process.max fds	Gauge	Maximum allowable open file descriptors, corresponding to <code>ulimit -a</code> output
process.open fds	Gauge	Number of open file descriptors

Typically, open file descriptors should remain below the maximum, so a test against a fixed threshold like 80% is a good indicator of an impending problem. This alert

should be set on *every application*, as file limits are a universally applicable hard limit that will take your application out of service.

For Atlas, use the :div and :gt functions, as shown in [Example 4-30](#).

Example 4-30. Atlas file descriptor alert threshold

```
name,process.open.fds,:eq,  
name,process.max.fds,:eq,  
:div,  
0.8,  
:gt
```

For Prometheus, [Example 4-31](#) looks even simpler.

Example 4-31. Prometheus file descriptor alert threshold

```
process_open_fds / process_max_fds > 0.8
```

At this point, we've covered the signals that are applicable to most every Java micro-service. The ones that follow are commonly useful, but not as ubiquitous.

Suspicious Traffic

One other simple indicator that can be derived from metrics like `http.server.requests` involves watching the occurrence of unusual status codes. A rapid succession of HTTP 403 Forbidden (and similar) or HTTP 404 Not Found may indicate an intrusion attempt.

Unlike plotting errors, monitor total occurrences of a suspicious status code as a *rate* and not a ratio relative to total throughput. It's probably safe to say that 10,000 HTTP 403s per second is equally suspicious if the system normally processes 15,000 requests per second or 15 million requests per second, so don't let overall throughput hide the anomaly.

The Atlas query in [Example 4-32](#), is similar to the error rate query we discussed earlier, but looks at the `status` tag for more granularity than the `outcome` tag.

Example 4-32. Suspicious 403s in HTTP server requests in Atlas

```
name,http.server.requests,:eq,  
status,403,:eq,  
:and,  
uri,$ENDPOINT,:eq,:cq
```

Use the Prometheus `rate` function to achieve the same result in Prometheus, as in [Example 4-33](#).

Example 4-33. Suspicious 403s in HTTP server requests in Prometheus

```
sum(  
  rate(  
    http_server_requests_seconds_count{status="403", uri="$ENDPOINT"}[2m]  
  )  
)
```

The next indicator is specialized to a particular kind of application but is still common enough to include.

Batch Runs or Other Long-Running Tasks

One of the biggest risks of any long-running task is that it runs for significantly longer than expected. Earlier in my career, I was routinely on call for production deployments, which were always performed after a series of midnight batch runs. Under normal circumstances, the batch sequence should have completed maybe at 1:00 a.m. The deployment schedule was built around this assumption. So a network administrator manually uploading the deployed artifact (this is before [Chapter 5](#)) needed to be at a computer ready to perform the task at 1:00 a.m. As the representative of the product's engineering team, I needed to be ready to perform a brief smoke test at approximately 1:15 a.m. and be available to help remediate any issues that arose. At this time, I lived in a rural area without internet access, so I traveled along a state highway toward a population center until I could get a reliable enough cell signal to tether to my phone and connect to the VPN. When the batch processes didn't complete in a reasonable amount of time, I sometimes spent hours sitting in my car on some country road waiting for them to complete. On days when production deployments weren't happening, perhaps nobody knew that the batch cycle failed until the next business day.

If we wrap a long-running task in a Micrometer `Timer`, we won't know that the SLO has been exceeded until the task actually completes. So if the task was supposed to take no more than 1 hour, but it actually runs for 16 hours, then we won't see this appear on a monitoring chart until the first publishing interval *after* 16 hours when the sample is recorded to the timer.

To monitor long-running tasks, it is better to look at the running time of in-flight or active tasks. `LongTaskTimer` performs this kind of measurement. We can add this kind of timing to a potentially long-running task, as in [Example 4-34](#).

Example 4-34. An annotation-based long task timer for a scheduled operation

```
@Timed(name = "policy.renewal.batch", longTask = true)
@Scheduled(fixedRateString = "P1D")
void renewPolicies() {
    // Bill and renew insurance policies that are beginning new terms today
}
```

Long task timers ship several distribution statistics: active task count, the maximum in-flight request duration, the sum of all in-flight request durations, and optionally percentile and histogram information about in-flight requests.

For Atlas, test against our expectation of one hour in nanoseconds, as shown in [Example 4-35](#).

Example 4-35. Atlas long task timer maximum alert threshold

```
name,policy.renewal.batch.max,:eq,
3.6e12, ❶
:gt
```

❶ One hour in nanoseconds

For Prometheus, [Example 4-36](#) is tested against one hour in seconds.

Example 4-36. Prometheus long task timer maximum alert threshold

```
policy_renewal_batch_max_seconds > 3600
```

We've seen some examples of effective indicators to look at, and at this point hopefully you have one or more of them plotted on a dashboard and can see some meaningful insights. Next we turn to how to automate alerting when these indicators go awry so that you don't have to watch your dashboards all the time to know when something isn't right.

Building Alerts Using Forecasting Methods

Fixed alert thresholds are often difficult to determine *a priori*, and since system performance is subject to drift over time, can be something that continually needs to be retuned. If performance over time tends to decline (but in such a way that the decline is still within acceptable levels), then a fixed alert threshold can easily become too chatty. If performance tends to improve, then the threshold is no longer as reliable of a measure of expected performance unless it is tuned.

Machine learning is the subject of a lot of hype that the monitoring system will automatically determine alert thresholds, but it hasn't produced the promised results. For

time series data, simpler classical statistical methods are still incredibly powerful. Surprisingly, the paper by S. Makridakis et al., “[Statistical and Machine Learning Forecasting Methods: Concerns and Ways Forward](#)”, shows that statistical methods have a lower prediction error (as shown in [Figure 4-33](#)) than machine learning methods do.

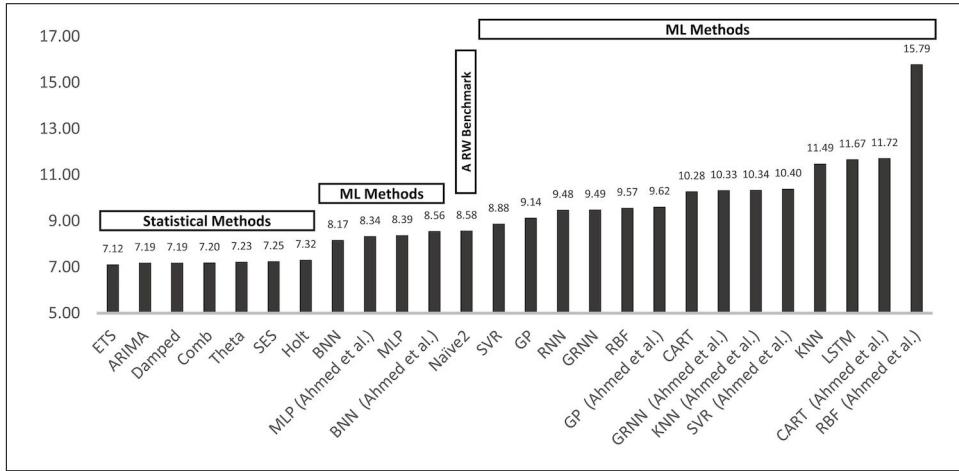


Figure 4-33. One-step forecasting error of statistical versus machine learning techniques

Let’s cover a few of these statistical methods, starting with the least predictive naive method, which can be used with any monitoring system. Later approaches have less universal support from monitoring systems since their math is complicated enough to require built-in query functions.

Naive Method

The [naive method](#) is a simple heuristic that predicts the next value based on the last observed value:

$$\hat{y}_{T+1|T} = \alpha y_T$$

A dynamic alert threshold can be determined with the naive method by multiplying a time series offset by some factor. Then we can test whether the true line ever drops below (or exceeds if the multiplier is greater than one) the forecast line. For example, if the true line is a measure of throughput through a system, a sudden substantial drop in throughput may indicate an outage.

The alert criteria for Atlas is then whenever the query in [Example 4-37](#) returns 1. The query is designed against Atlas’s test dataset, so it’s easy for you to test out and try different multipliers to observe the effect.

Example 4-37. Atlas alert criteria for the naive forecasting method

```
name,requestsPerSecond,:eq,  
:dup,  
0.5,:mul, ①  
1m,:offset, ②  
:rot,  
:lt
```

- ① The tightness of the threshold is set with this factor.
- ② “Look back” to some prior interval for the forecast.

The effect of the naive method can be seen in [Figure 4-34](#). The multiplicative factor (0.5 in the example query) controls how close to the true value we want to set the threshold and also reduces by the same amount the spikiness of the forecast (i.e., the looser the threshold, the less spiky the forecast). Since the method’s smoothing is proportional to looseness of fit, the alert threshold is still tripped four times in this time window (indicated by the vertical bars in the middle of the chart), even though we’ve allowed for a 50% drift from “normal.”

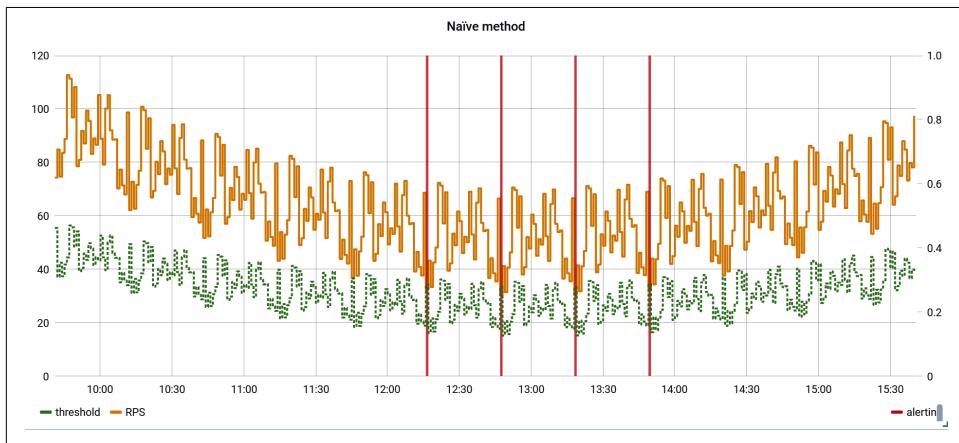


Figure 4-34. Forecasting with the naive method

In order to prevent a chatty alert, we’d have to reduce the tightness of the forecast’s fit to our indicator (in this case a 0.45 multiplier silences the alert for this time window). Of course, doing so also allows for more drift from “normal” before an alert is fired.

Single-Exponential Smoothing

By smoothing the original indicator before multiplying it by some factor, we can fit the threshold closer to the indicator. Single-exponential smoothing is defined by [Equation 4-1](#).

Equation 4-1. Where $0 \leq \alpha \leq 1$

$$\hat{y}_{T+1|T} = \alpha y_T + \alpha(1 - \alpha)y_{T-1} + \alpha(1 - \alpha)^2 y_{T-2} + \dots = \alpha \sum_{n=0}^k (1 - \alpha)^n y_{T-n}$$

α is a smoothing parameter. When $\alpha = 1$, all the terms except the first are zeroed and we are left with the naive method. Values less than 1 suggest how important previous samples should be.

Like for the naive method, the alert criteria for Atlas is whenever the query in [Example 4-38](#) returns 1.

Example 4-38. Atlas alert criteria for single-exponential smoothing

```
alpha,0.2,:set,
coefficient,(,alpha,:get,1,alpha,:get,:sub,),:set,
name,requestsPerSecond,:eq,
:dup,:dup,:dup,:dup,:dup,
0,:roll,1m,:offset,coefficient,:fcall,0,:pow,:mul,:mul,
1,:roll,2m,:offset,coefficient,:fcall,1,:pow,:mul,:mul,
2,:roll,3m,:offset,coefficient,:fcall,2,:pow,:mul,:mul,
3,:roll,4m,:offset,coefficient,:fcall,3,:pow,:mul,:mul,
4,:roll,5m,:offset,coefficient,:fcall,4,:pow,:mul,:mul,
5,:roll,6m,:offset,coefficient,:fcall,5,:pow,:mul,:mul,
:add,:add,:add,:add,:add,
0.83,:mul, ①
:lt,
```

- ① The tightness of the threshold is set with this factor.

The summation $\alpha \sum_{n=0}^k (1 - \alpha)^n$ is a geometric series that converges to 1. For example, for $\alpha = 0.5$, see [Table 4-3](#).

Table 4-3. The convergence to 1 of the geometric series where $\alpha = 0.5$

T	$(1 - \alpha)^T$	$\alpha \sum_{n=0}^k (1 - \alpha)^n$
0	0.5	0.5
1	0.25	0.75
2	0.125	0.88
3	0.063	0.938
4	0.031	0.969
5	0.016	0.984

Since we don't include *all* values of T , the smoothed function is effectively already multiplied by a factor equal to the cumulative sum of this geometric series up to the number of terms we have chosen. [Figure 4-35](#) shows summations of one term and two terms in the series relative to the true value (respectively from bottom to top).

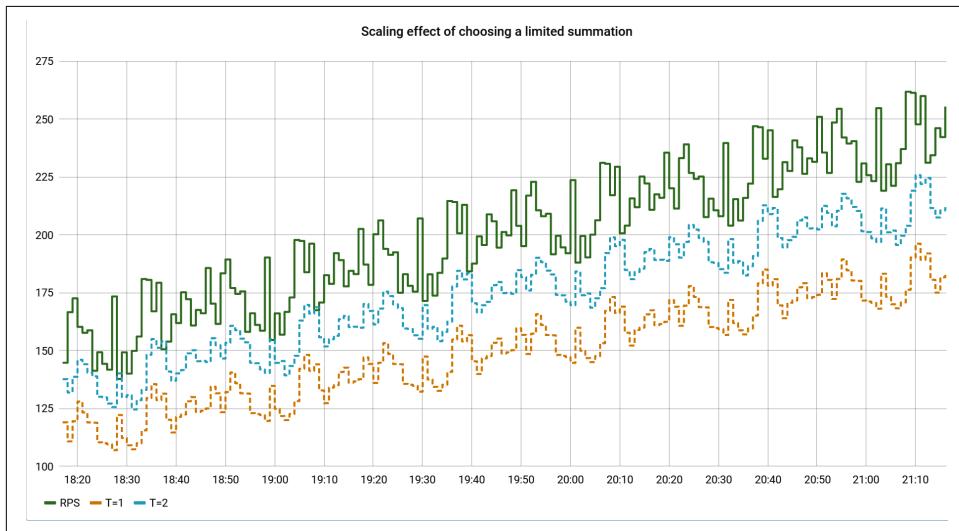


Figure 4-35. Scaling effect of choosing a limited summation

[Figure 4-36](#) shows how different selections of α and T affect the dynamic threshold, in terms of both how smoothed it is and its approximate scaling factor relative to the true indicator.



Figure 4-36. Smoothing and scaling effect when choosing different α and T

Universal Scalability Law

In this section, we are going to shift our mindset entirely from smoothing datapoints that happened in the *past* (which we used as dynamic alert thresholds) to a technique that allows us to predict what *future* performance will look like if concurrency/throughput increases beyond current levels, using only a small set of samples of what performance has looked like at already-seen concurrency levels. In this way, we can set *predictive* alerts as we approach a service-level objective boundary to hopefully head off problems rather than reacting to something already exceeding the boundary. In other words, this technique allows us to test a *predicted* service-level indicator value against our SLO at a throughput that we haven't yet experienced.

This technique is based on a mathematical principle known as Little's Law and the Universal Scalability Law (USL). We'll keep the mathematical explanation here to a minimum. What little is discussed you can skip past. For more details, Baron Schwartz's freely available *Practical Scalability Analysis with the Universal Scalability Law* (VividCortex) is a great reference.



Using Universal Scalability Law in the Delivery Pipeline

In addition to predicting impending SLA violations in production systems, we can use the same telemetry in a delivery pipeline to throw some traffic at a piece of software that doesn't need to be anywhere close to the maximum traffic it might see in production and predict whether production-level traffic will meet an SLA. And we can do this before deploying a new version of the software to production!

Little's Law, [Equation 4-2](#), describes the behavior of queues as a relationship between three variables: queue size (N), latency (R), and throughput (X). If the application of queuing theory to SLI predictions seems a little mind-bending, don't worry (because it is). But for our purposes in predicting an SLI, N will represent the concurrency level of requests passing through our system, X the throughput, and R a latency measure like average or a high-percentile value. Since this is a relationship between three variables, provided any two we can derive the third. Since we care about predicting latency (R), we'd need to forecast this in the two dimensions of concurrency (N) and throughput (X).

Equation 4-2. Little's Law

$$N = XR$$

$$X = N/R$$

$$R = N/X$$

The Universal Scalability Law, [Equation 4-3](#), allows us instead to project latency in terms of only a single variable: either throughput or concurrency. This equation requires three coefficients, which will be derived and updated from a model maintained by Micrometer based on real observations about the system's performance to this point. USL defines κ to be the cost of crosstalk, ϕ to be the cost of contention, and λ to be how fast the system operates under unloaded conditions. The coefficients become fixed values making predictions on latency, throughput, or concurrency dependent on only one of the other three. Micrometer will also publish the values of these coefficients as they change over time, so you can compare the system's major governing performance characteristics over time.

Equation 4-3. Universal Scalability Law

$$X(N) = \frac{\lambda N}{1 + \phi(N - 1) + \kappa N(N - 1)}$$

With a series of substitutions, we get to express R in terms of X or N (see [Equation 4-4](#)). Again, don't think too hard about these relationships, because Micrometer is going to do these calculations for you.

Equation 4-4. Predicted latency as a function of either throughput or concurrency

$$R(N) = \frac{1 + \phi(N - 1) + \kappa N(N - 1)}{\lambda}$$

$$R(X) = \frac{-\sqrt{X^2(\kappa^2 + 2\kappa(\phi - 2) + \phi^2) + 2\lambda X(\kappa - \phi) + \lambda^2} + \kappa X + \lambda - \phi X}{2\kappa X^2}$$

What we'll get is a nice two-dimensional projection instead, as shown in [Figure 4-37](#).

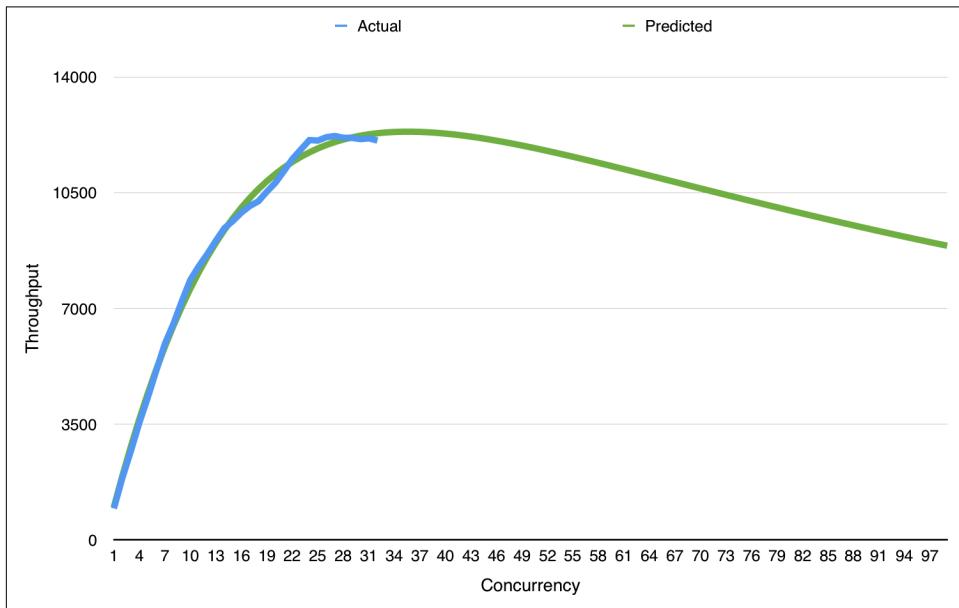


Figure 4-37. USL prediction of latency based on different throughput levels

USL forecasting is a form of “derived” Meter in Micrometer and can be enabled as shown in [Example 4-39](#). Micrometer will publish a set of Gauge meters forming a series of forecasts at various throughput/concurrency levels for each publishing interval. Throughput and concurrency are correlated measurements, so think of them interchangeably from this point on. When you select a related group of timers (which will always have the same name) to publish a forecast for, Micrometer will publish several additional metrics using the common metric name as a prefix:

timer.name.forecast

A series of Gauge meters with a tag throughput or concurrency based on the type of independent variable selected. At a certain time interval, plotting these gauges would generate a visualization like [Figure 4-37](#).

timer.name.crosstalk

A direct measure of the system's crosstalk (e.g., fan-out in a distributed system like that described in the paper by S. Cho et al., [“Moolle: Fan-out Control for Scalable Distributed Data Stores”](#)).

timer.name.contention

A direct measure of the system's contention (e.g., locking on relational database tables and in general any other form of lock synchronization).

timer.name.unloaded.performance

Improvements in ideal unloaded performance (e.g., framework performance improvements) can be expected to yield improvements in loaded conditions as well.

Example 4-39. Universal scalability law forecast configuration in Micrometer

```
UniversalScalabilityLawForecast
    .builder(
        registry
            .find("http.server.requests") ❶
            .tag("uri", "/myendpoint") ❷
            .tag("status", s -> s.startsWith("2")) ❸
    )
    .independentVariable(UniversalScalabilityLawForecast.Variable.THROUGHPUT) ❹
    // In this case, forecast to up to 1,000 requests/second (throughput)
    .maximumForecast(1000)
    .register(registry);
```

- ❶ The forecast will be based on the results of a Micrometer meter search for one or more timers with name `http.server.requests` (remember, there may be several such timers with different tag values).
- ❷ We can further limit the set of timers to base the forecast on by only matching on timers that have a specific key-value tag pair.
- ❸ Like with any search, the tag value can be constrained with a lambda as well. A good example is constraining the forecast to any “2xx” HTTP statuses.
- ❹ The domain of the Gauge histogram will be either `UniversalScalabilityLawForecast.Variable.CONCURRENCY` or `UniversalScalabilityLawForecast.Variable.THROUGHPUT`, defaulting to `THROUGHPUT`.

The latency an application is experiencing at its current throughput in one of these time slices will closely follow the “predicted” latency from the forecast. We can set an alert based on some scaled-up value of whatever the current throughput is to determine if the predicted latency at that scaled-up throughput would still be under our SLO.

In addition to predicting an SLI under increased throughput, the modeled values for crosstalk, contention, and unloaded performance are a strong indicator of where performance improvements can be made in an application. After all, decreases in cross-talk and contention and increases in unloaded performance directly impact the system’s predicted and actual latency under various levels of load.

Summary

This chapter has presented you with the tools you need to start monitoring every Java microservice for availability with signals that are included in Java frameworks like Spring Boot. We’ve also discussed more generally how to alert on and visualize classes of metrics like counters and timers.

Though you should strive for finding ways of measuring microservice availability in terms of business-focused metrics, using these basic signals is a huge step forward over simply looking at box metrics in terms of understanding how your service is performing.

Organizationally, you’ve committed to standing up a dashboarding/alerting tool. We showed Grafana in this chapter. Its open source availability and datasources for a wide array of popular monitoring systems make it a solid choice to build on top of without locking yourself in completely to a particular vendor.

In the next chapter, we’re going to transition to delivery automation, where we’ll see how some of these availability signals are used in making decisions about the fitness of new microservice releases. Effective delivery isn’t strictly about the motion of deploying; it turns monitoring into action.

Safe, Multicloud Continuous Delivery

The placement of this chapter in the second half of this book should signal the importance of telemetry in achieving safe and effective delivery practices. This may come as a surprise, because every organization ships software with an emphasis on testing as a means of guaranteeing safety, but not every organization measures it actively in a way that is directly related to end-user experience.

The concepts in this chapter will be introduced with a continuous delivery tool called Spinnaker, but as with earlier chapters, a different tool could achieve similar ends. I'd like to establish a minimum base for what you should expect from a worthy CD tool.

Spinnaker is an open source continuous delivery solution that started at Netflix in 2014 to help manage its microservices in AWS. It was preceded at Netflix by a tool called Asgard, which was really just an alternative AWS console organized with application developers in mind, and built for Netflix's unusual scale of AWS consumption. At one point, I was interacting with an AWS console form that required me to select a security group. The UI element in the console was a plain HTML select (list box) with four visible elements. The available security groups were unsorted in the list box, and there were thousands of them (again, because of Netflix's broad scale in this account)! Usability issues like this led to Asgard, which in turn led to Spinnaker. Asgard was really just an application inventory with some actions (like the AWS console). Spinnaker was conceived as an inventory *plus* pipelines.

In 2015 Spinnaker was open sourced and other, initially IaaS implementations were added to it. At various points, Spinnaker has seen significant contributions from Google, Pivotal, Amazon, and Microsoft, as well as end users like Target. Many of these contributors worked together to write a **separate book** on the topic of Spinnaker.

The practices described in this chapter are applicable to a wide variety of platforms.

Types of Platforms

Different types of platforms have a surprising level of commonality in the high-level concepts that make up a running application. The concepts introduced in this chapter will be mostly platform neutral. Platforms fall into one of the following categories:

Infrastructure as a service (IaaS)

An infrastructure as a service provides virtualized computing resources as a service. An IaaS provider traditionally is responsible for servers, storage, networking hardware, the hypervisor layer, and APIs and other forms of user interface to manage these resources. Originally, to the extent you were using an IaaS, it was as an alternative to having physical hardware. Provisioning resources on an IaaS involves building virtual machine (VM) images. Deploying to an IaaS requires building VM images at some point in the delivery pipeline.

Container as a service (CaaS)

A container as a service is a kind of specialization of an IaaS for container-based workloads rather than VMs. It provides a higher level of abstraction for the deployment of apps as containers. Kubernetes has of course become the de facto standard CaaS offered by public cloud vendors and on-prem. It provides a lot of other services that are outside the scope of this book. Deploying to a CaaS requires the extra step of building containers somewhere in the delivery pipeline (often at build time).

Platform as a service (PaaS)

A platform as a service further abstracts away the details of the underlying infrastructure, generally by allowing you to upload an application binary like a JAR or WAR directly to the PaaS API, which is then responsible for building an image and provisioning it. Contrary to the as-a-service implication of PaaS, sometimes PaaS offerings like Cloud Foundry are layered on top of virtualized infrastructure in customer datacenters. They also can be layered on top of IaaS offerings to provide a further abstraction away from the IaaS resource model, which may serve the purpose of preserving some degree of public cloud provider vendor neutrality or permitting similar delivery and management workflows in a hybrid private/public cloud environment.

These abstractions can be provided to you by another company, or you can build this cloud native infrastructure yourself (as a number of large companies have). The key requirement is an elastic, self-serve, and API-driven platform upon which to build.

A key assumption we will make throughout this chapter is that you are building immutable infrastructure. While nothing about an IaaS, for example, prevents you from building a VM image, launching an instance of it, and dropping an application onto it after launch, we are going to assume that the VM image is “baked” along with

the application and any supporting software in such a way that when a new instance is provisioned, the application should start and run.

A further assumption is that applications deployed in this manner are approximately cloud native. The definition of cloud native varies from source to source, but at a minimum the applications that are amenable to the deployment strategies discussed throughout this chapter are stateless. Other elements of [12-Factor apps](#) aren't as crucial.

For example, I managed a service at Netflix that routinely took over 40 minutes to start, which doesn't look good against the disposability criteria, but was otherwise unavoidable. The same service used exceedingly high memory footprint instance types in AWS, of which we had a small reserved pool. These put constraints on my choices: I couldn't have more than about four instances of this service running at any given time, so I wasn't going to be doing blue/green deployments with several disabled clusters (described in "[Blue/Green Deployment](#)" on page 200).

To further center the discussion around a common language, let's discuss the resource building blocks common to all of these platforms.

Resource Types

To keep our discussion of delivery concepts platform-neutral, we will adopt the abstractions as defined by Spinnaker, which are surprisingly portable across different types of platforms:

Instance

An instance is a running copy of some microservice (that isn't on a local developer machine, because I sincerely hope production traffic isn't finding its way there). The AWS EC2 and Cloud Foundry platforms both call this "instance," conveniently. In Kubernetes, an instance is a pod.

Server group

A server group represents a collection of instances, managed together. Platforms have different ways of managing a collection of instances, but they tend to have a responsibility for ensuring that a certain number of instances are running. We generally assume that all of the instances of a server group have the same code and configuration, because they are immutable (except when they aren't). Server groups can logically be devoid of any instances at all but simply have the potential to scale into a nonzero set of instances. In AWS EC2 a server group is an Auto Scaling Group. In Kubernetes a server group is roughly the combination of a Deployment and ReplicaSet (where a deployment manages rollout of ReplicaSets) or a StatefulSet. In Cloud Foundry, a server group is an Application (not to be confused with Application as defined in this list and as we are going to use the term throughout this chapter).

Cluster

A cluster is a set of server groups that may span multiple regions. Within a single region, multiple server groups may represent different versions of the microservice. Clusters do *not* span cloud providers. You could be running two very similar clusters in different cloud providers, but for our discussion they would be considered distinct. A cluster is a logical concept that doesn't actually have a correlated resource type in any cloud provider. Even more precisely, it doesn't span multiple installations of a particular platform. So a cluster does not span multiple Cloud Foundry foundations or Kubernetes clusters. There is no higher-level abstraction in AWS EC2 that represents a collection of Auto Scaling Groups or in Kubernetes that represents a collection of Deployments. Spinnaker manages cluster membership by naming conventions or additional metadata that it places on resources it creates, depending on the platform's implementation in Spinnaker.

Application

An application is a logical business function and not one particular resource. All of the running instances of an application are included. They may span multiple clusters in multiple regions. They may exist on multiple cloud providers, either because you are transitioning from one provider to another, or because you have some concrete business case for not being locked into one provider, or for any other reason. Wherever there is a running process that represents an instance of this business function, it is part of the grander concept called application.

Load balancer

A load balancer is a component that allocates individual requests to instances in one or more server groups. Most load balancers have a set of strategies or algorithms they can use to allocate traffic. Also, they generally have a health-checking feature that allows the load balancer to determine if a candidate microservice instance is healthy enough to receive traffic. In AWS EC2 a load balancer is an Application Load Balancer (or a legacy Elastic Load Balancer). In Kubernetes, the Service resource is a load balancer. The Cloud Foundry Router is a load balancer.

Firewall

A firewall is a set of rules that govern ingress and egress to a set of server groups. In AWS EC2 these are called Security Groups.

Spinnaker's Kubernetes implementation is a little unique among the providers. Spinnaker can really deploy any Kubernetes resource because it internally uses `kubectl apply` and passes the manifest to the Kubernetes cluster. Furthermore, Spinnaker allows you to treat manifests as templates and provide variable substitution. It then maps some Kubernetes objects like ReplicaSets/Deployments/StatefulSets to server groups and Services to load balancers.

Figure 5-1 shows a Spinnaker view of a series of Kubernetes ReplicaSets. Note how this infrastructure view also contains actions like Edit, Scale, Disable, and Delete for

the selected resource. In this view, `replicaSet helloworldapp-frontend` is the “Cluster” resource (an amalgamation in this case of the Kubernetes resource type and name), representing a set of ReplicaSets in one or more Kubernetes namespaces. `HELLOWORLDWEBAPP-STAGING` is the “Region” corresponding to a Kubernetes namespace of the same name. `helloworldapp-frontend-v004` is a server group (a ReplicaSet). The individual blocks are the “Instances” corresponding to Kubernetes pods.

The screenshot shows the Spinnaker interface for managing Kubernetes resources. On the left, there's a sidebar with filters for ACCOUNT, REGION, STACK, DETAIL, STATUS, and more. The main area displays two clusters: DEV-CLUSTER and PROD-CLUSTER. Each cluster contains a single `replicaSet helloworldwebapp-frontend`. The PROD-CLUSTER section is expanded, showing three instances. A red arrow points to the context menu for the first instance in the PROD-CLUSTER list. The menu includes options like Edit, Scale, Disable, Delete, Namespace (set to `helloworldwebapp-prod`), Kind (set to `replicaSet`), and Images. The Images section lists `gcr.io/cf-spinnaker/github_okundzich_helloworldwebapp:1ec84fe`.

Figure 5-1. Spinnaker view of three Kubernetes ReplicaSets with actions highlighted

Delivery Pipelines

Spinnaker pipelines are just one of many delivery-focused pipeline solutions on the market both commercially and in OSS. They range from the low-level and heavily opinionated Spring Cloud Pipelines to continuous integration pipelining extended to include delivery building blocks like JenkinsX. For the sake of this chapter, we will stick to Spinnaker pipelines, but if you substitute another pipelining solution, look for some key capabilities:

Platform neutrality

A delivery solution doesn't have to support every possible vendor to qualify as a platform-neutral solution, but delivery solutions based on, for example, Kubernetes custom resource definitions are guaranteed to be locked into a given platform. With this kind of lock-in, any heterogeneity in your deployed environment means you are going to be building to multiple tools. Mixed platform use is exceedingly common in enterprises of sufficient scale (as it should be).

Automated triggers

Pipelines should be able to be automatically triggered by events, especially by changes in artifact inputs. We will discuss more about how artifact triggers help you repave your infrastructure in a safe and controlled way in “[Packaging for the Cloud](#)” on page 194.

Scalability

A good pipelining solution accounts for the radically different computational nature of different pipeline stages. “Deploy” stages that are exercising platform API endpoints to provision new resources have very low computational needs, even if the stage may run for several minutes. A single instance of a pipeline execution service can easily run thousands of these in parallel. “Execute script” stages that execute something like a Gradle task are arbitrarily resource-intensive, so the execution of them is best delegated to something like a container scheduler such that the resource utilization of the stage execution doesn’t affect the performance of the pipeline execution service.

When continuous integration products are used to perform deployment operations, they generally inefficiently use resources in a significant way. For one financial institution I once visited, performing delivery operations with the CI system Concourse was costing several million dollars per year. For this organization, running 30 `m4.large` reserved instances in EC2 to support a Spinnaker installation would have cost a little over \$15,000 per year. The resource inefficiency can easily swing the other direction though. Stages of arbitrary computational complexity should not be run on host or in process with Spinnaker’s Orca (i.e., pipeline) service.

The various cloud providers *feel* very different. The deployable resource is a different level of abstraction for each type.

Spinnaker pipelines consist of stages that are approximately cloud neutral. That is, the same basic building blocks will be available for every cloud provider implementation, but the configuration of a stage like “Deploy” will vary from platform to platform.

[Figure 5-2](#) shows the definition of a Spinnaker pipeline that is going to deploy to Kubernetes. Pipelines can be arbitrarily complex, containing parallel stages and multiple triggers defined in the special configuration stage up front.

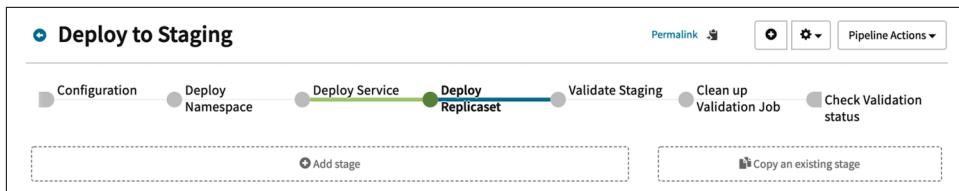


Figure 5-2. A detailed view of a Spinnaker pipeline showing multiple stages

Spinnaker defines several different trigger types. This pipeline is triggered by the publication of a new container image in a Docker registry, as shown in [Figure 5-3](#).

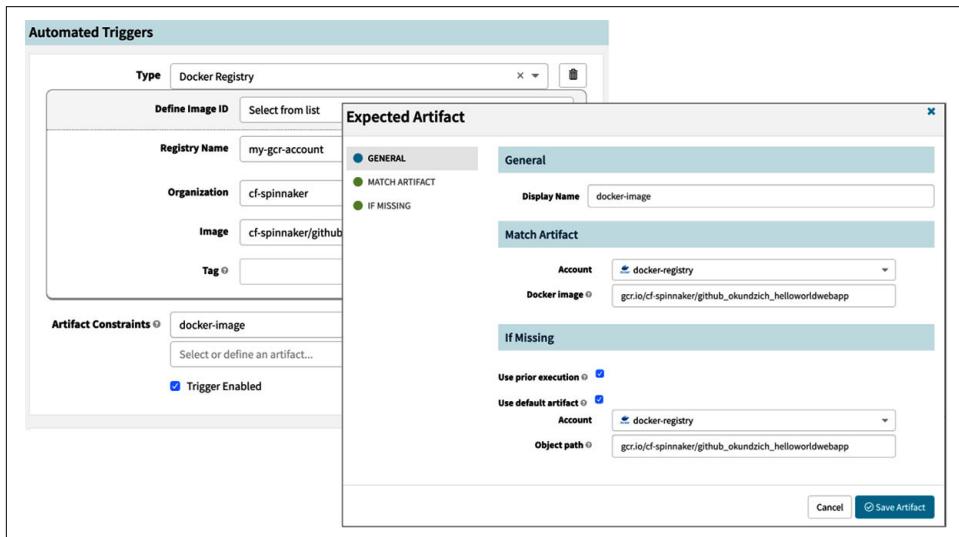


Figure 5-3. Spinnaker expected artifact definition

[Figure 5-4](#) shows the execution history of two Spinnaker pipelines, including the one whose configuration we just saw. The staging pipeline was last executed by a Docker registry trigger (a new container published to a Docker registry). In the other circumstances, the pipeline was manually triggered.

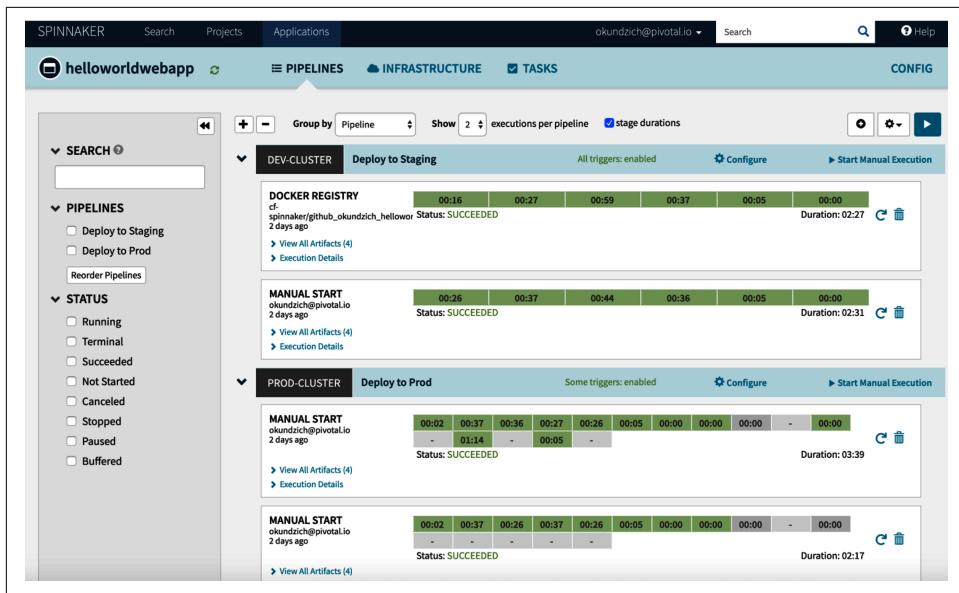


Figure 5-4. Spinnaker view of two different delivery pipelines

The first task of any delivery pipeline is to package the application in an immutable unit of deployment that can be stamped out in instances across a server group.

Packaging for the Cloud

The different abstractions given by the various types of cloud platforms have trade-offs in terms of startup time, resource efficiency, and cost. But as we'll see, there shouldn't be a significant difference in terms of the effort each requires to package a microservice for deployment.

Generating a new application from [start.spring.io](#) includes the generation of a Gradle or Maven build that can generate a runnable JAR. For PaaS platforms like Cloud Foundry and Heroku, this runnable JAR *is* the input unit of deployment. It is the responsibility of the cloud provider to take this runnable JAR and containerize or otherwise package it and then provision some underlying resource to run this package on.

For cloud platforms other than PaaS, the effort required from the application team is surprisingly not much different. The examples included here are implemented with Gradle because open source tooling exists for both IaaS and CaaS uses. There is no reason similar tooling couldn't be produced for Maven.

One of the typical value propositions of a PaaS is that you provide just the application binary as an input to deployment processes and let the PaaS manage the operating

system and package patching on your behalf, even transparently. In practice, it doesn't work out quite like this. In the case of Cloud Foundry, the platform is responsible for a certain level of patching that is achieved in a rolling manner, affecting one instance at a time in any server group (a.k.a. "application" in Cloud Foundry parlance). But such patching comes with a certain degree of risk: an update to any part of the operating system could adversely affect an application running on it. So there is this risk/reward trade-off that carefully circumscribes the types of changes the platform is willing to automate on behalf of users. All other patches/updates are applied to the "type" of image that the platform will layer your application on. Cloud Foundry calls these buildpacks. For example, Java version upgrades involve an update to the buildpack. The platform does not automatically update buildpack versions for every running application that used the Java buildpack. It really is up to the organization then to redeploy every application using the Java buildpack to pick up the update.

For non-PaaS environments, with the extra amount of effort involved in generating another type of artifact (other than a JAR) from your build or having an additional stage in your deployment pipeline comes a dramatically greater degree of control and flexibility over how infrastructure can be patched across your organization. While the type of base image is different between IaaS and CaaS (virtual machine and container image, respectively), the principle of baking your application on top of a base image allows you to define your application binary and the base image it is layered upon as separate inputs to each microservice delivery pipeline. [Figure 5-5](#) shows a hypothetical delivery pipeline for a microservice that first deploys to a test environment, runs tests, and goes through an audit check before finally being deployed to production. Note how Spinnaker supports multiple trigger types in this case: one for a new application binaries and one for a new base image.

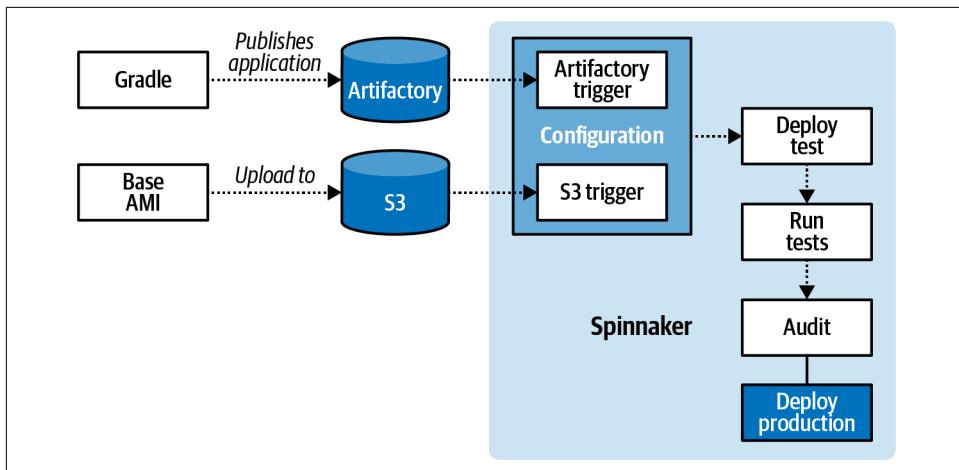


Figure 5-5. Changes to the base image trigger pipelines

In the same organization, a different microservice may have more or fewer stages to verify the fitness of the combination of application artifact and base image before promoting to production. Having changes to the base image *trigger* delivery pipelines is the ideal balance between safety and speed. Microservices whose delivery pipelines contain all fully automated stages may adopt the new base within minutes, where another service with more stringent manual verification and approval stages takes days. Both types of services adopt the change in a way that is best in line with the responsible team's unique culture and requirements.

Packaging for IaaS Platforms

For an IaaS platform, the immutable unit of deployment is a virtual machine image. In AWS EC2, this image is called an Amazon Machine Image. Creating one is a matter of provisioning an instance of the base image (which contains common opinions for all of your microservices like the Java version, common system dependencies, and monitoring and debugging agents.), installing a system dependency on it containing your application binary, snapshotting the resultant image, and configuring new server groups to use this image as the template when provisioning instances of the microservice.

The process of provisioning the instance, installing the system dependency, and snapshotting it is collectively called *baking*. It isn't always necessary to even launch a live copy of the base image to bake. HashiCorp's battle-tested **Packer** provides an open source bakery solution that works for a variety of different IaaS providers.

Figure 5-6 shows where the boundaries of responsibility are for the build tool, the bakery, and the server group managed by the cloud provider. A Spinnaker pipeline stage is responsible for starting the baking process and for creating the server group with the image resulting from the bake stage. It shows that there is one additional requirement of each microservice's build, the production of a system dependency, meaning the production of a Debian package on an Ubuntu or Debian base image, an RPM on a Red Hat base image, etc. Ultimately, the bakery will in some way be invoking the operating-system-level package installer to layer your application binary onto the base image (e.g., `apt-get install <system-package>`).

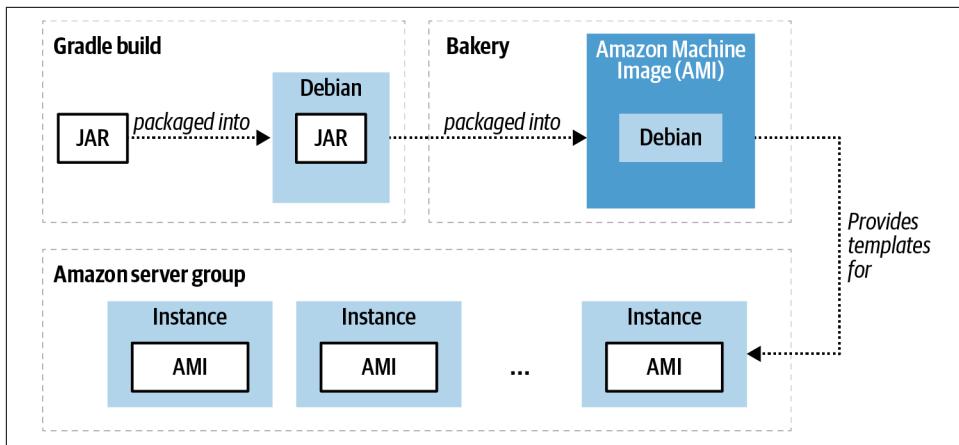


Figure 5-6. Participants in IaaS packaging

Producing a Debian or RPM system dependency is fortunately straightforward with the application of a Gradle plug-in from Netflix’s Nebula suite of Gradle plug-ins, as shown in [Example 5-1](#). This adds a Gradle task called `buildDeb` that does all the work necessary to output a Debian package for a Spring Boot application. It is a one-line change to the build file!

Example 5-1. Using a Nebula Gradle plug-in to produce a Debian package

```

plugins {
    id("org.springframework.boot") version "LATEST"
    id("io.spring.dependency-management") version "LATEST"
    id("nebula.ospackage-application-spring-boot") version "LATEST" ❶
}

...

```

- ❶ Replace LATEST with whatever the latest version is on the [Gradle plug-in portal](#), because LATEST isn’t actually valid for Gradle plug-in version specifications.

The `ospackage` plug-in contains a variety of options for adding start scripts, configuring output locations for configuration files and runnable artifacts, etc. Ultimately though, wherever and whatever happens with these files, there should be enough commonality between microservices in an organization to encapsulate these opinions in a similar way to what Netflix has done with `nebula.ospackage-application-spring-boot` and distribute them as a build tool plug-in that makes adoption trivial.

Packaging for Container Schedulers

Preparing a microservice for deployment to a container scheduler like Kubernetes can be similar. Opinionated tooling is again available in open source to package for common frameworks like Spring Boot, as shown in [Example 5-2](#). This plug-in also understands how to publish to Docker registries, given a little more configuration (which can easily be encapsulated and shipped as a common build tool plug-in across an organization).

Example 5-2. Using a Nebula Gradle plug-in to produce and publish a Docker image

```
plugins {  
    id("org.springframework.boot") version "LATEST"  
    id("io.spring.dependency-management") version "LATEST"  
    id("com.bmuschko.docker-spring-boot-application") version "LATEST"  
}  
  
if (hasProperty("dockerUser") && hasProperty("dockerPassword")) {  
    docker {  
        registryCredentials {  
            username = dockerUser  
            password = dockerPassword  
            email = "bot@myorg.com"  
        }  
  
        springBootApplication {  
            tag = "$dockerUser/${project.name}:${project.version}"  
            baseImage = "openjdk:8"  
        }  
    }  
}
```



Rely on Open Source Build Tooling, but Be Careful About Consuming Base Images Without Validation

It's great to have tools like Ben Muschko's Gradle Docker plug-in for producing an image containing an application built on top of some base. But you should expect that somebody in your organization is validating and creating approved images, known to perform well and be free of known defects and security vulnerabilities. This is applicable to both VM and container images.

This approach does have the disadvantage that operating system and other system package updates are part of the base Docker image used to produce the application container image. Propagating a base container image change across the whole organization then requires us to rebuild the application binary, which can be inconvenient. After all, to effect a change to *only* the base image and not the application code

(which may have a set of further source code changes since the last time it was built), we have to check out the application code at the hash of the version in production and rebuild with the new image. This process, since it involves the build again, is fraught with the potential for nonreproducibility in the application binary when all we want to do is update the base image.

Adding a bake stage to containerized workloads simplifies the build by removing the need to publish a container image at all (just publish the JAR to a Maven artifact repository) and allows for mass updating of the base image, again with the same process and safety guarantees that we received by making the base image an artifact trigger for IaaS-based workloads above. Spinnaker supports baking container images with [Kaniko](#), removing the need for container image building/publishing to be part of the build workflow. One of the advantages of doing so is that you can rebake the same application binary on a more recent base (say, when you fix a security vulnerability in the base), operating effectively with an immutable copy of the application code.

Surprisingly then, the desire for safe base updates across all three cloud abstractions (IaaS, CaaS, and PaaS) leads to a remarkably similar workflow for all three (and similar application developer experience). In effect, ease of deployment is no longer a decision criterion between these level of abstractions, and we are left with considering other differentiators like startup time, vendor lock-in, cost, and security.

Now that we've discussed packaging, let's turn our attention to deployment strategies that can be used to stand up these packages on your platform.

The Delete + None Deployment

If the name *delete + none* sounds ugly, that's because I'm about to describe a hack that may only be useful in certain narrow situations but helps set the framework for other deployment strategies to follow.

The basic idea is to simply delete the existing deployment and deploy a new one. The obvious ramification of this is downtime, however short it may be. The existence of downtime suggests that API compatibility across versions isn't strictly required, provided you coordinate deployments to all callers of a service with a changing API at the same time.

Every deployment strategy that follows will be zero-downtime.

To tie this concept to deployment practices you may be familiar with that are *not immutable*, a delete + none deployment strategy is in use when, on an always-running virtual machine, a new application version is installed and started (replacing the previously running version). Again, this chapter focuses strictly on immutable

deployments, and no other deployment strategy that follows has an obvious mutable counterpart.

The strategy is also in use when performing a basic `cf push` on Cloud Foundry and an operation on AWS EC2 that reconfigures an Auto Scaling Group to use a different Amazon Machine Image. The point is, often basic CLI or console-based deployment options do accept downtime and operate more or less with this strategy.

The next strategy is similar, but with zero downtime.

The Highlander

Despite the odd name, the Highlander strategy is the most common zero-downtime strategy in practice today. The name comes from a slogan from the *Highlander* movie: “there can be only one.” In other words, when you deploy a new version of a service, you replace the old version. There can be only one. Only the new version is running at the end of the deployment.

The Highlander strategy is zero-downtime. In practice it involves deploying a new version of the application and adding it to the load balancer, which causes traffic to be served to both for a short period of time while the old version is destroyed automatically. So maybe the more accurate slogan for this deployment strategy is “there is usually only one.” Required API compatibility across versions follows from the existence of this brief overlap.

The Highlander model is simple, and its simplicity can make it an attractive option for many services. Since there is only one server group at any given time, there is no need to worry about coordinating to prevent interference from “other” running versions that are not supposed to be in service.

Going back to a previous version of code under a Highlander strategy involves reinstalling an old version of the microservice (which receives a new server group version number). Therefore, the time to completion for this pseudorollback action is the amount of time it takes to install and initialize the application process.

The next strategy offers faster rollback at the expense of some coordination and complexity.

Blue/Green Deployment

The blue/green deployment strategy involves having at least two copies of the micro-service provisioned (whether in an enabled or disabled state), involving server groups for old and new versions. At any given time, production traffic is being served from one of these versions. Rolling back is a matter of switching which copy is considered live. Rolling forward to the newer version has the same experience. How this

switching logic is achieved is cloud-platform-dependent (but orchestrated by Spinnaker), but at a high level it involves influencing the cloud platform's load balancer abstraction to send traffic to one version or the other.



kubectl apply Is a Specific Kind of Blue/Green by Default

`kubectl apply` that updates a Kubernetes Deployment (from the CLI and not using Spinnaker) is by default a rolling blue/green deployment, allowing you to roll back to a ReplicaSet representing a previous version. Because it is a container deployment type, the rollback operation involves pulling back the image. The Kubernetes Deployment resource is implemented as a controller on top of ReplicaSet that manages rolling blue/green deployment and rollback. Spinnaker offers more control for Kubernetes ReplicaSets, enabling blue/green functionality with N disabled versions, canary deployments, etc. So think of a Kubernetes Deployment as a limited, opinionated blue/green deployment strategy.

Load balancer switching can have implications for the structure of deployed assets. For example, on Kubernetes, blue/green basically requires that you use the ReplicaSet abstraction. The blue/green strategy requires that *running* resources are somehow edited to influence traffic. For Kubernetes, we can do this with label manipulation, and this is what the Spinnaker Kubernetes implementation does to achieve blue/green. If we instead tried to edit the Kubernetes Deployment object, it would trigger a rollout. Spinnaker automatically adds special labels to ReplicaSets that indirectly cause them to be considered enabled or disabled and a label selector to the service to only route traffic to enabled ReplicaSets. If you aren't using Spinnaker, you'll need to create some sort of similar process that mutates labels in place on ReplicaSets and configure your services to be aware of these labels.

The colors blue/green imply that there are two server groups and either the blue or the green server group is serving traffic. Blue/green strategies aren't always binary either, and the coloring should not suggest that these server groups need to be long-lived, mutating with new service versions as they become available.

A blue/green deployment is more generally a 1:N relationship in any given cluster, where one server group is live and N server groups are not live. A visual representation of such a 1:N blue/green cluster is shown in [Figure 5-7](#).

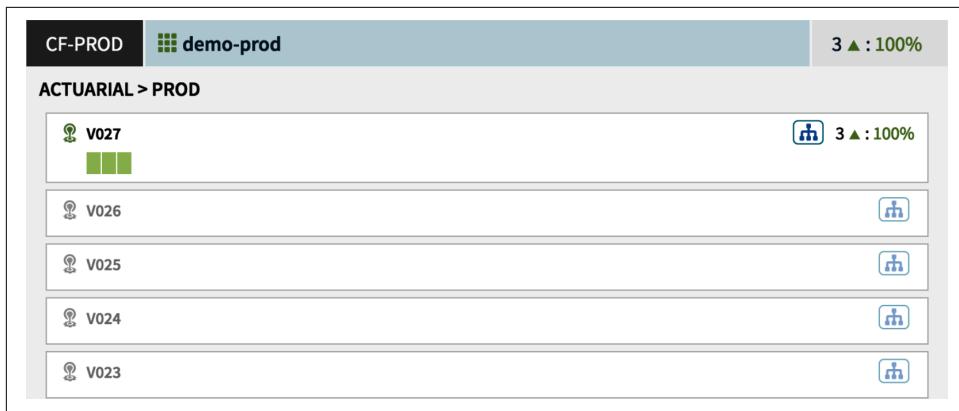


Figure 5-7. Spinnaker blue/green cluster

The rollback server group action, shown in [Figure 5-8](#), allows for the selection of any one of these disabled server group versions (V023–V026). At the completion of the rollback, the current live version (V027) will still be around, but disabled.

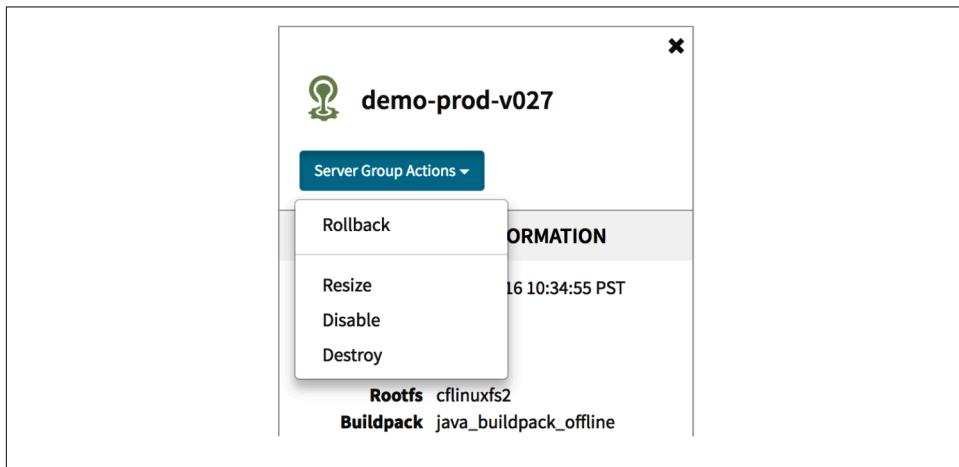


Figure 5-8. Spinnaker rollback server group action

Depending on what the underlying cloud platform can support, a disabled cluster can retain live running instances that aren't receiving any traffic, or it may be reduced to zero instances, ready at rollback to be scaled back up. To achieve the fastest form of rollbacks, disabled clusters should be left with active instances. This of course increases the expense of running the service, as you now pay not only for the cost of the set of instances serving live production traffic, but also the set of instances remaining from prior service versions that may potentially be rolled back to.

Ultimately, you need to evaluate the trade-off between rollback speed and cost, the spectrum of which is shown in [Figure 5-9](#). This should be done on a microservice basis rather than across the organization. The extra operational cost of maintaining fully scaled disabled server groups in a blue/green deployment of a microservice that needs to run hundreds of live instances is not equivalent to the extra cost for such a service that only needs a handful of instances.

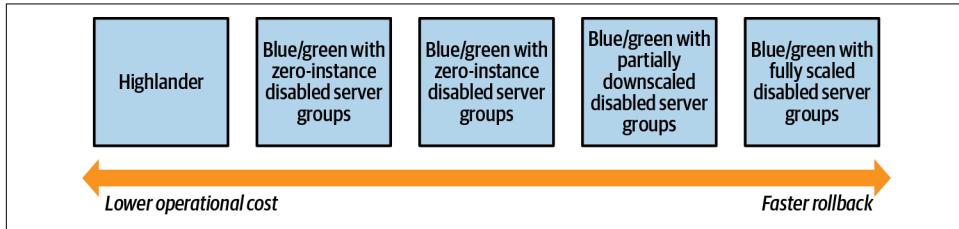


Figure 5-9. The trade-off between operational cost and rollback speed by deployment strategy

When a microservice is not purely RESTful, the blue/green deployment strategy that doesn't completely scale to zero disabled clusters has implications for the application code itself.

Consider for example an (at least partially) event-driven microservice that reacts to messages on a Kafka topic or RabbitMQ queue. Shifting the load balancer from one server group to another has no effect on such a service's connection to their topic/queue. In some way, the application code needs to respond to being placed out of service by some external process, in this case a blue/green deployment.

Similarly, an application process running on an instance that is part of a disabled server group needs to respond to being placed back *in service* by an external process such as a Rollback server group action in Spinnaker, in this case reconnecting to queues and beginning to process work again. The Spinnaker AWS implementation of the blue/green deployment strategy is aware of this problem, and when [Eureka](#) service discovery is also in use, it influences service availability using Eureka's API endpoints for taking instances in and out of service, as shown in [Table 5-1](#).

Notice how this is done on a per-instance basis. Spinnaker's awareness of what instances exist (through polling the state of deployed environment regularly) helps it build this kind of automation.

Table 5-1. Eureka API endpoints that externally affect service availability

Action	API	Notes
Take instance out of service	PUT /eureka/v2/apps/appID/ instanceID/status? value=OUT_OF_SERVICE	
Move instance back into service (remove override)	DELETE /eureka/v2/apps/appID/ instanceID/status?value=UP	The value=UP is optional; it is used as a suggestion for the fallback status due to removal of the override

This supposes that your application is using the Eureka service discovery client to register with Eureka. But doing so means that you can add a Eureka status-changed event listener, as shown in [Example 5-3](#).

Example 5-3.

```
// For an application with a dependency on
// 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
@Bean
ApplicationInfoManager.StatusChangeListener statusChangeListener() {
    return new ApplicationInfoManager.StatusChangeListener() {
        @Override
        public String getId() {
            return "blue.green.listener";
        }

        @Override
        public void notify(StatusChangeEvent statusChangeEvent) {
            switch(statusChangeEvent.getStatus()) {
                case OUT_OF_SERVICE:
                    // Disconnect from queues...
                    break;
                case UP:
                    // Reconnect to queues...
                    break;
            }
        }
    };
}
```

Naturally the same sort of workflow could be achieved with [Consul](#), a dynamic configuration server that allows for tagging (i.e., tag by server group name, or cluster), or any other central source of data that has two characteristics:

- Application code can respond to change events in near real time via some sort of event listener.
- The data is groupable by at least server group, cluster, and application, and your application code is able to determine which server group, cluster, and application it belongs to.

The same requirement to respond to external modifications of a service's availability is applicable also to microservices using persistent RPC connections like [RSocket](#) or streaming/bidirectional [GRPC](#), where a disabled server group needs to terminate any persistent RPC connections either outbound or inbound.

There's a hidden and significant point embedded in having to listen to discovery status events (or any other external indicator service availability): the application is aware of its participation in service discovery. A goal of service mesh (see "[Implementation in Service Mesh](#)" on page 285) is to take this kind of responsibility away from the application and externalize it to a sidecar process or container, generally for the sake of achieving polyglot support for these patterns quickly. We'll talk about other problems with this model later, but blue/green deployment of message-driven applications where you'd like to preserve live instances in disabled server groups is an example of where a language-specific binding (in this case for service discovery) is necessary.



What's in a Name?

A blue/green deployment is the same thing as a red/black deployment. They are just different sets of colors, but the techniques have precisely the same meaning.

Blue/green deployments are something every team should practice before considering a more complex strategy, such as automated canary analysis.

Automated Canary Analysis

Blue/green deployments achieve a great deal of reliability at a reasonably low cost most of the time. Not every service needs to go any further than this. Yet, there is an additional level of safety we can pursue.

While blue/green deployments allow you to quickly roll back a code or configuration change that causes unanticipated issues, canary releases provide an additional level of risk reduction by exposing a small subset of users to a new version of the service that runs alongside the existing version.

Canaries aren't appropriate for every service. Services with low throughput make it difficult, but not impossible, to send just a small percentage of traffic to a canary

server group without prolonging the determination of the canary's fitness to a long period of time. There isn't a right amount of time that a canary fitness determination needs to take. It may very well be acceptable for you to run a canary test for days on a relatively low throughput service to make a determination.

There is a noticeable bias in many engineering teams to underestimate how much traffic their service actually receives and thus assume that techniques like canary analysis couldn't possibly work for them. Recall the real-world team mentioned in “[Learning to Expect Failure](#)” on page 12, whose business application was receiving over 1,000 requests per minute. This throughput is significantly higher than most engineers on this team would guess. This is yet another reason that real production telemetry should be the first priority. Building up even a short history of what is happening in production helps you make better decisions about which kinds of techniques, in this case deployment strategies, are appropriate later.



My Service Can Never Fail Because It Is Too Important

Be cautious of a line of reasoning that eschews strategies like automated canary analysis strictly on the basis that a particular micro-service is too important to fail. Rather, adopt the mindset that failure is not only possible, but will happen on every service regardless of its importance to the business, and act accordingly.

The fitness of a canary deployment is determined by comparing service level indicators of the old and new versions. When there is a significant enough worsening in one or more of these SLIs, all traffic is routed to the stable version, and the canary is aborted.

Ideally, canary deployments consist of three server groups, as shown in [Figure 5-10](#).

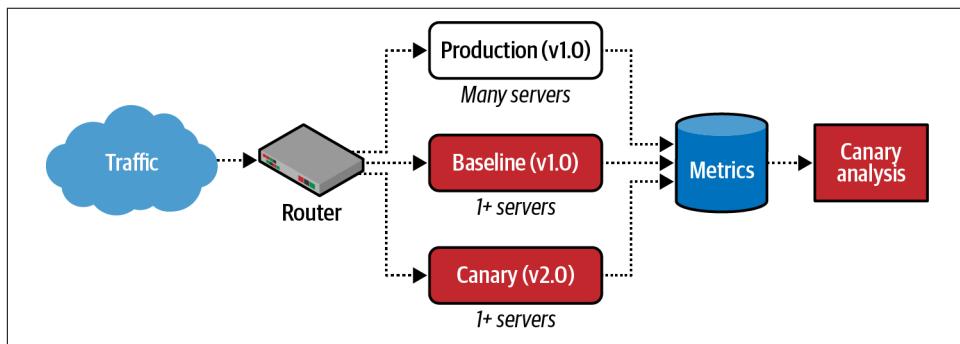


Figure 5-10. Canary release participants

These canary deployments can be described as follows:

Production

This is the existing server group prior to the canary deployment, containing one or more instances.

Baseline

This server group runs the same version of code and configuration as the production server group. While it may seem counterintuitive to run another copy of the old code at first, we need a baseline that is launched at roughly the same time as the canary because a production server group, by virtue of the fact that it has been running for some amount of time, may have different characteristics like heap consumption or cache contents. It is important to be able to make an accurate comparison between the old code and new, and the best way to do that is to launch copies of each at roughly the same time.

Canary

This server group consists of the new code or configuration.

The fitness of the canary is entirely determined by comparing a set of metrics indicators relative only to the baseline (not the production cluster). This implies that applications undergoing canaries are publishing metrics with a `cluster` common tag so that the canary analysis system can aggregate over indicators coming from instances belonging to the canary and baseline clusters and compare the two aggregates relative to each other.

This relative comparison is much preferable to testing a canary against a set of fixed thresholds because fixed thresholds tend to make certain assumptions about the amount of throughput going through the system at test time. [Figure 5-11](#) shows the problem. The application exhibits a higher response time during peak business hours when most traffic is flowing through the system (which is typical). So for the fixed threshold, perhaps we try to set a value that would fail a canary if response time was more than 10% worse than the normative case. During peak business hours, the canary would fail because it has a worse than 10% degradation from the baseline. But if we ran the canary test after hours, it could be significantly worse than 10% from the baseline and still be under the fixed threshold, which was set to 10% worse than what we expect under *different operating conditions*. There is a greater chance that a relative comparison would be able to catch a performance degradation whether the test runs during or after business hours.

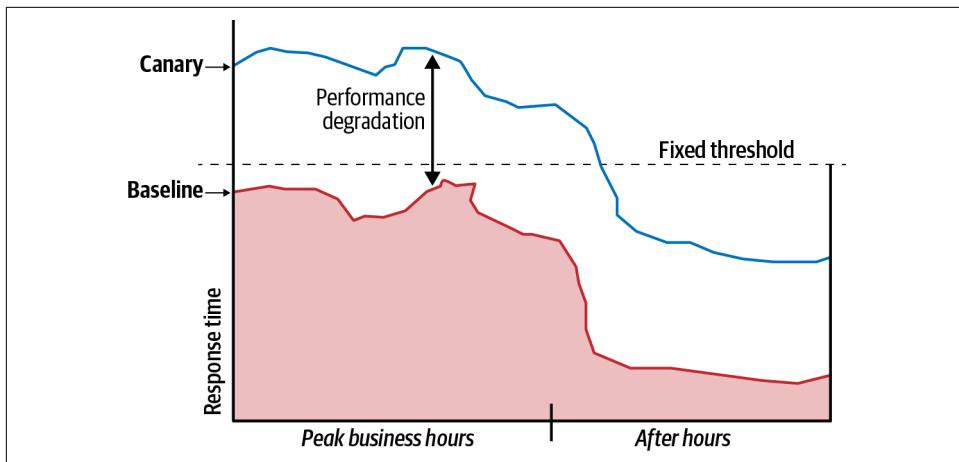


Figure 5-11. Canary test against fixed threshold

On several occasions, I've met with organizations about automated delivery practices where the conversation came about because someone heard about canary deployments and the idea sounds so compelling that it stimulates an interest in the topic. Commonly, these organizations didn't have dimensional metrics instrumentation in place or an automated release process resembling a blue/green deployment. Maybe because of the allure of the safety of a canary deployment, platforms sometimes include canary deployment features. Often they lack baselining and/or comparative measurement, so evaluate platform-provided canary features from this perspective and decide whether giving up one or both still makes sense. I would suggest it doesn't in many cases.

In the three-cluster setup (production, baseline, canary), most traffic will go to the production cluster, with a small amount going to the baseline and canary. The canary deployment uses load balancer configuration, service mesh configuration, or whatever other platform feature is available to distribute traffic proportionally.

Figure 5-12 shows a Spinnaker infrastructure view of the three clusters participating in a canary test. In this case, they are running on a single Kubernetes cluster that has been named “PROD-CLUSTER” in Spinnaker (“cluster” referring to Kubernetes cluster, not as we've defined the word in the delivery definitions at the beginning of this chapter).

Spinnaker integrates with an open source automated canary analysis service, which encapsulates the evaluation of metrics from baseline and canary clusters.

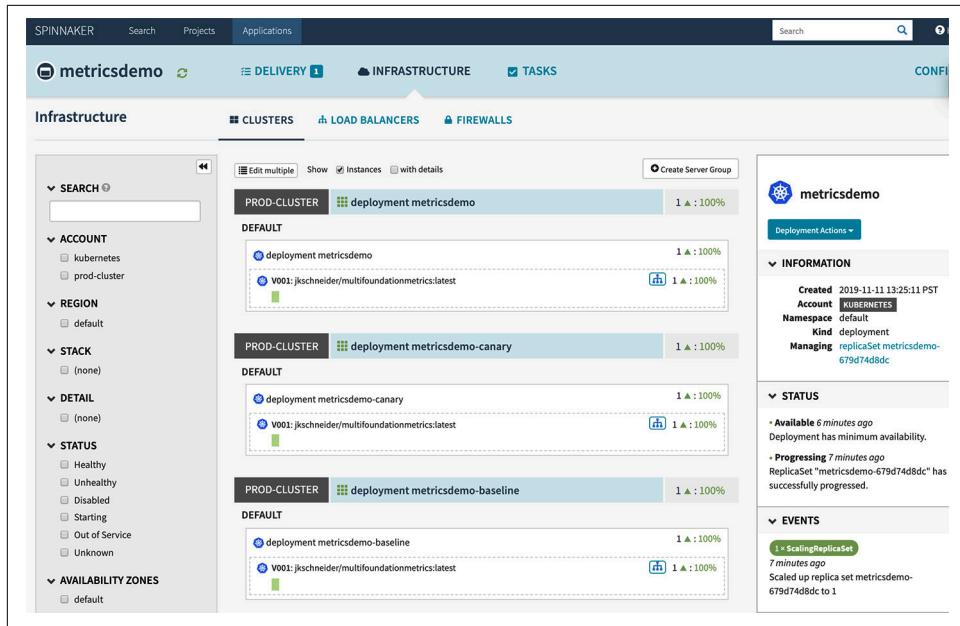


Figure 5-12. Three clusters of an application undergoing a canary

Spinnaker with Kayenta

Kayenta is a stand-alone open source automated canary analysis service that is also deeply integrated into Spinnaker by way of pipeline stages and configuration.

Kayenta determines whether there is a significant difference between the canary and baseline for each metric, yielding a *pass*, *high*, or *low* classification. *High* and *low* are both failing conditions. Kayenta makes this difference comparatively between the two clusters using a [Mann-Whitney U test](#). The implementation of this statistical test is called a judge, and Kayenta could be configured with alternative judges, but they typically involve code that goes beyond what you could achieve through a single query against a metrics system.

Figure 5-13 presents an example of Kayenta's classification decisions for several metrics. This screenshot is from the original Netflix [blog](#) on Kayenta. In this case, latency has failed the test.

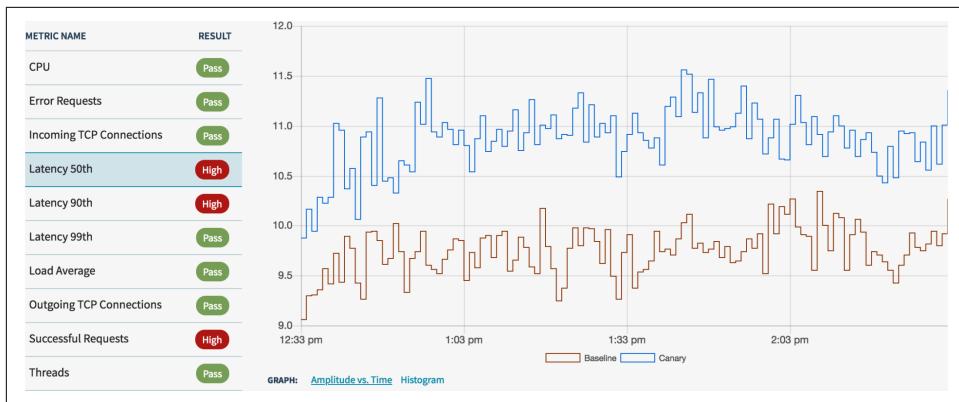


Figure 5-13. Canary metrics

In Spinnaker, the canary metrics for an application can be defined in the “Canary Configs” tab of the application infrastructure view. In the configuration, as shown in [Figure 5-14](#), you can define one or more service level indicators. If enough of these indicators fail, the canary will fail.

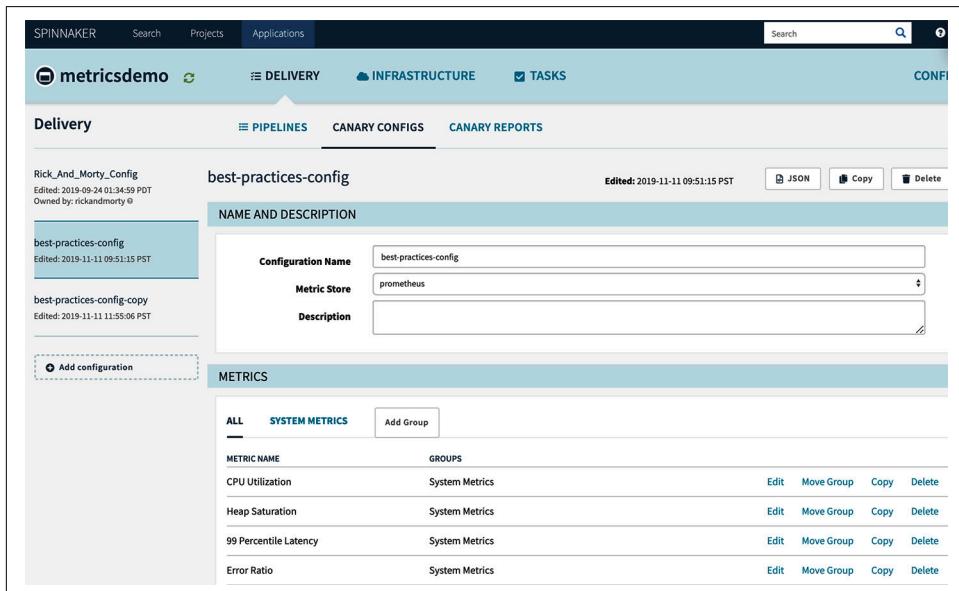


Figure 5-14. Canary configuration for an application in Spinnaker

[Figure 5-15](#) shows the configuration of a single indicator, in this case processor utilization. Notice how the configuration contains a metrics query that is specific to a monitoring system that you have configured Kayenta to poll from (in this case, Prometheus). You then indicate very broadly that an increase or decrease (or a

deviation either way) is considered bad. We would not like to see significantly higher processor utilization in this case, although a decrease would be welcome.

On the other hand, a *decrease* in servicing throughput for an application that should process rather continuously at a certain rate would be a bad sign. The indicator can be marked as critical enough that failure alone should fail the canary.

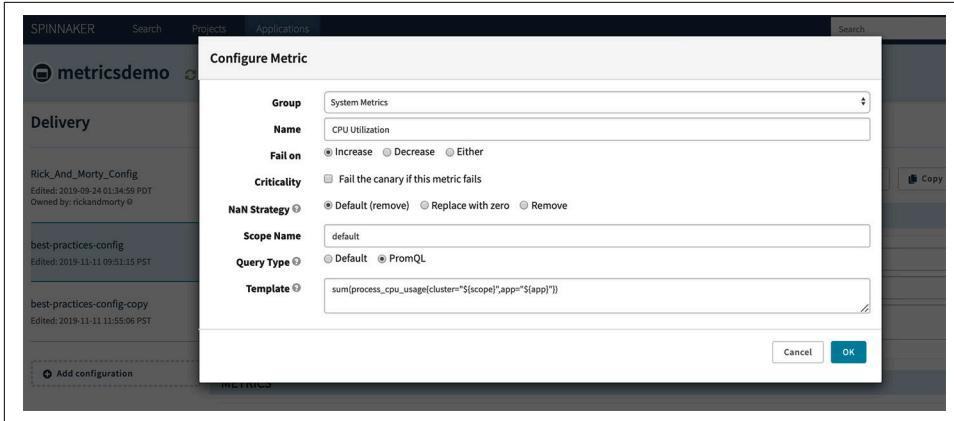


Figure 5-15. Processor utilization canary configuration

Once the canary configuration is built, it can be used in pipelines. A characteristic canary deployment pipeline is shown in Figure 5-16. The “Configuration” stage defines the triggers that begin the process of evaluating the canary. “Set Cluster Name to Canary” sets a variable that is used in the subsequent stage “Deploy Canary” by Spinnaker to name the cluster canary. It is this variable that eventually yields the named canary cluster shown in Figure 5-12.

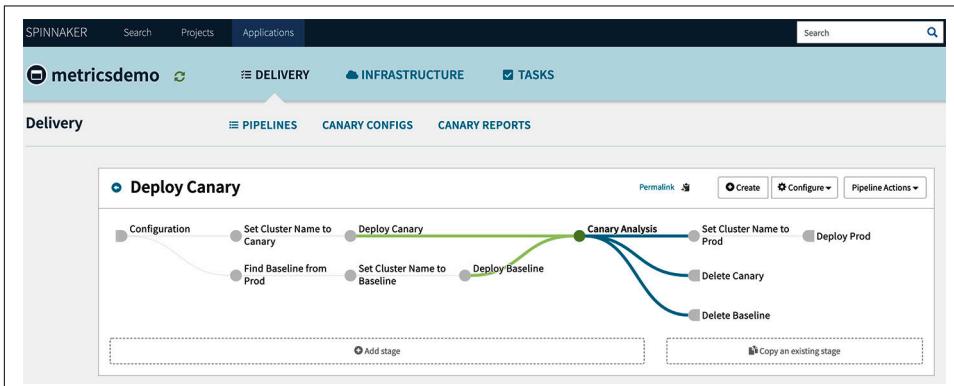


Figure 5-16. A canary deployment pipeline in Spinnaker

In parallel, Spinnaker is retrieving the artifacts that the current production version is based off of and creating a baseline cluster with those artifacts as well. The “Canary Analysis” stage could run for hours or even days, depending on how it is configured. If it passes, we will deploy a new prod cluster (with the same artifact used to created the canary, which may no longer be the latest version available in the artifact repository). In parallel, we can tear down the baseline and canary clusters that are no longer needed. This whole pipeline can be configured in Spinnaker to be run serially so that only one canary is being evaluated at any given time.

The outcome of a canary run is viewable in a few different ways. Spinnaker presents a “Canary Reports” tab that shows the outcome of the judgment for a canary stage, breaking down each service level indicator that goes into the decision individually. Each indicator can be viewed as a time series graph over the interval in which the canary ran, as in [Figure 5-17](#).

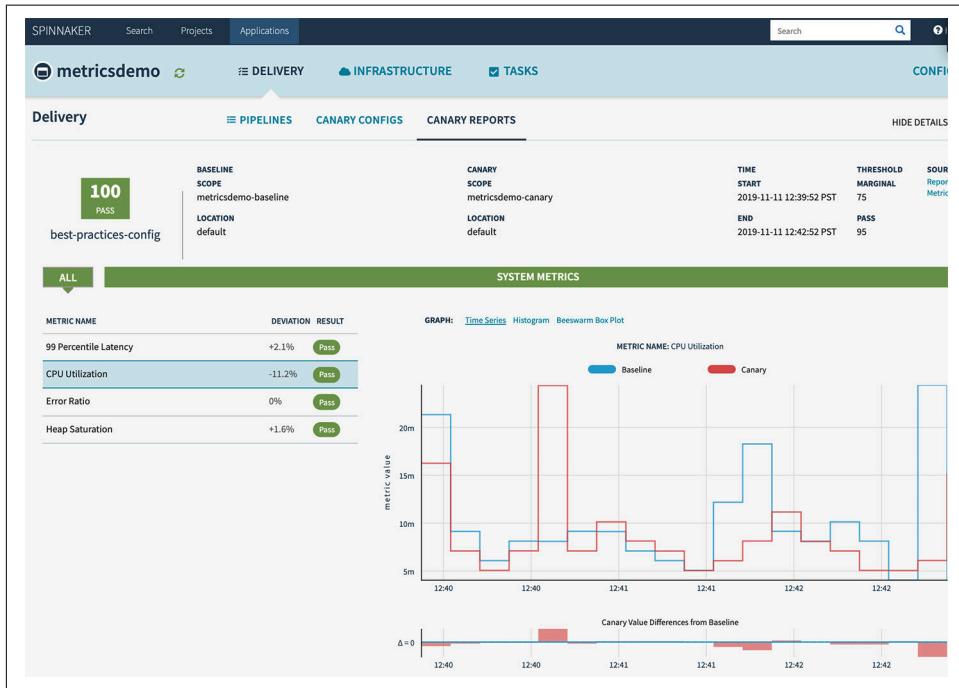


Figure 5-17. Time series visualization of CPU utilization in baseline and canary



Current Production Version Isn't Always Latest

Note that the current production version from which a baseline will be created is *not* always the latest version of the application binary (e.g., JAR or WAR) available in an artifact repository. It may actually be several versions older, in cases where we have attempted to release new versions but they have failed canaries or were otherwise rolled back. One of the values of a stateful continuous delivery solution like Spinnaker is its ability to poll the environment for what is there and to then act upon this information.

Alternatively, the comparison for an indicator can be viewed as a bar chart (or histogram), as in [Figure 5-18](#).

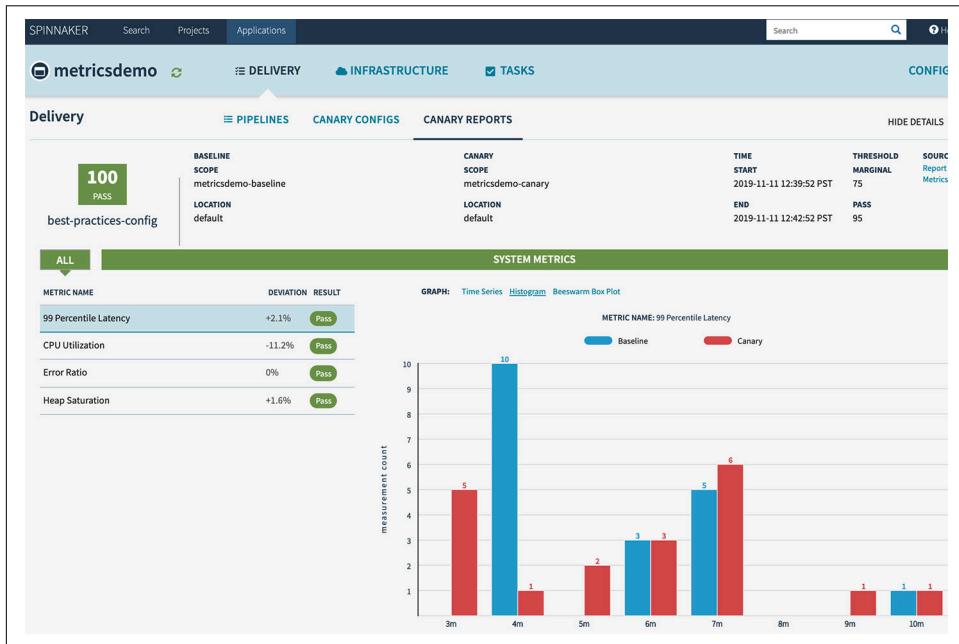


Figure 5-18. Histogram visualization of 99th-percentile latency

Lastly, and perhaps most usefully, the comparison between canary and baseline can be visualized as a beeswarm plot, shown in [Figure 5-19](#). The canary is judged over time, with Kayenta polling the monitoring system for values for the canary and baseline at a prescribed interval. The individual samples here are shown on the beeswarm plot along with a box-and-whisker plot showing the basic quartiles (min, 25th percentile, median, 75th percentile, and max of all samples). The median has certainly increased, but as discussed in [Chapter 2](#), measures of centrality like mean and median aren't really that useful for judging the fitness of a service. This plot really highlights

this fact. The max and even 75% latency haven't changed much at all between versions. So there's a little more variation in the median, but it probably doesn't indicate a performance regression at all.

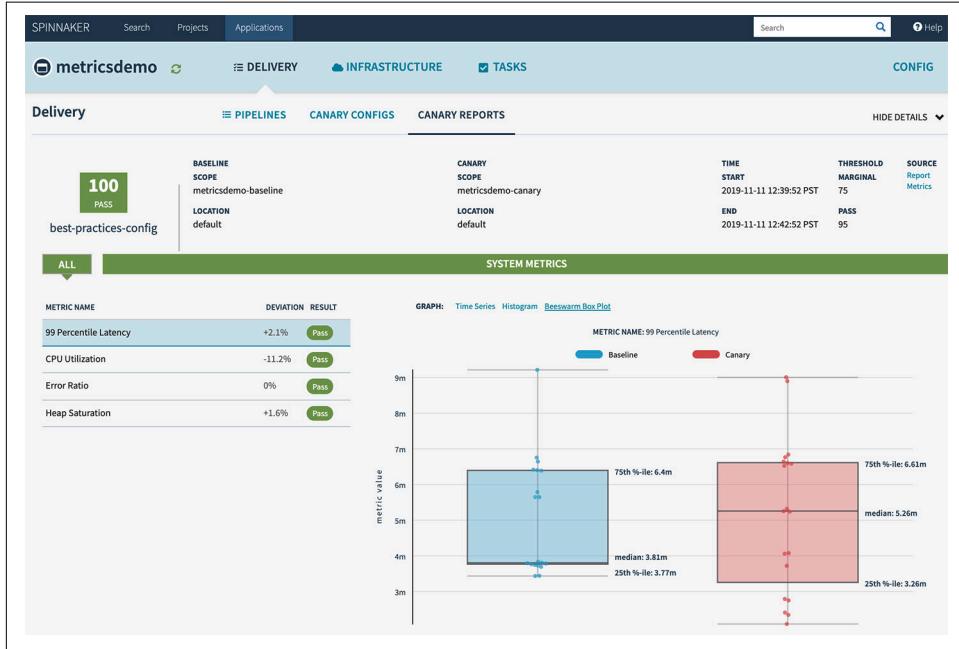


Figure 5-19. Beeswarm visualization of 99th-percentile latency

The key indicators for canary analysis will be different sometimes than the indicators we use to alert, because they are designed for comparative analysis between two clusters rather than against an absolute measure. Even if a new application version remains underneath a service level objective boundary that we've set as the test for an alert, it's still probably best that the general trajectory of the code doesn't continue to regress closer and closer to that service level objective.

General-Purpose Canary Metrics for Every Microservice

Think of the L-USE acronym when considering canary metrics that are useful to start with. In fact, many of the same SLIs that you should alert on for most microservices are also good canary metrics, with a twist.

Let's consider a few good canary metrics, beginning with latency. Really any of the signals described in [Chapter 4](#) are good candidates for canary analysis.

Latency

Latency for some bellwether API endpoint is a good starting point. Constrain the metric to successful outcomes only, since successful outcomes tend to have different latency characteristics than unsuccessful outcomes. Imagine fixing a bug that caused failures for a critical API endpoint in production, only to have the canary fail because the bug caused the API endpoint to fail fast and the canary decides that latency has degraded too much by fixing it!

In [Chapter 4](#), the idea was to measure a timed operation's decaying *maximum* latency against a fixed service level objective, that service level objective being a conservative boundary inside engineering's service level agreement determined with business partners. But maximum latency has a way of being spiky. For example, hypervisor and garbage collection pauses or full connection pools are mostly temporary conditions (and outside of your control) that naturally affect instances at different times. For the purposes of measuring an application's fitness relative to a service level objective, we want to make sure that performance is still acceptable even under these conditions. But because of the staggered nature in which they occur on different instances, these effects lead to bad *comparative* measures.

For canaries, it is best to look at a distribution statistic like 99th-percentile latency, which shaves off the top 1% where these temporary conditions exhibit. 99th percentile (or some other high percentile) is generally a better measure of a piece of code's performance *potential* minus temporary environmental factors.

Recall from [“Histograms” on page 65](#) that in order to compute a high-percentile latency across a cluster (and constrained to successful outcomes for a particular endpoint), we need to use a method like percentile approximation based off of histogram data that can be summed across the instances in this cluster and any other tag variation that exists for successful outcomes for this critical API endpoint. Only a handful of monitoring systems at this point support aggregable percentile approximation. If your monitoring system can't do percentile approximation, do not attempt to aggregate individual percentiles from instances (we showed why the math for this doesn't work in [“Percentiles/Quantiles” on page 60](#)). Also, avoid the temptation to use another measure like average. Look at the beeswarm plot in [Figure 5-18](#) to see how measures of centrality like median and mean can vary widely between versions (and realistically even over time with the same version!) without any real change in performance.

Average: a random number that falls somewhere between the maximum and 1/2 the median. Most often used to ignore reality.

—Gil Tene

To compute an aggregable percentile approximation for Atlas, use the `:percentiles` function, as in [Example 5-4](#).

Example 5-4. Atlas percentile latency for canaries

```
name,http.server.requests,:eq,  
uri,$ENDPOINT,:eq,  
:and,  
outcome,SUCCESS,:eq,  
:and,  
(,99,),:percentiles
```

For Prometheus, use the `histogram_quantile` function, as in [Example 5-5](#).

Example 5-5. Prometheus percentile latency for canaries

```
histogram_quantile(  
    0.99,  
    rate(  
        http_server_requests_seconds_bucket{  
            uri="$ENDPOINT",  
            outcome="SUCCESS"  
        }[2m]  
    )  
)
```

In a similar way, you should include latency metrics for interactions with critical downstream resources like a database. Consider relational database interactions. It is not uncommon for new code to inadvertently cause an existing database index to go unused (increasing latency substantially, as well as load on the database), or for a new index to not perform as well as expected once it gets to production. Regardless of how well we try to replicate and test these new interactions in lower-level environments, there is nothing quite like production.

Error ratio

Error ratio ([Example 5-6](#) on Atlas and [Example 5-7](#) for Prometheus) on some bellwether API endpoint (or all of them) can be incredibly useful as well, as this will determine whether you have introduced semantic regressions into your code that may have not been caught by tests but are causing issues in production.

Example 5-6. Error ratio of HTTP server requests in Atlas

```
name,http.server.requests,:eq,  
:dup,  
outcome,SERVER_ERROR,:eq,  
:div,  
uri,$ENDPOINT,:eq,:cq
```

Example 5-7. Error ratio of HTTP server requests in Prometheus

```
sum(  
  rate(  
    http_server_requests_seconds_count{outcome="SERVER_ERROR", uri="$ENDPOINT"}[2m]  
  )  
) /  
sum(  
  rate(  
    http_server_requests_seconds_count{uri="$ENDPOINT"}[2m]  
  )  
)
```

Carefully consider whether to include more than one API endpoint in a single canary signal. Suppose you had two separate API endpoints that receive significantly different throughput, one receiving 1,000 requests/second and the other receiving 10 requests/second. Because our service isn't perfect (what is?), the old code is failing the high throughput endpoint at a fixed rate of 3 requests/second, but all requests to the low throughput endpoint succeed. Now imagine we make a code change that causes 3 out of 10 requests to the low throughput endpoint to fail but doesn't change the error ratio of the other endpoint. If these endpoints are considered together by the canary judge, the judge will probably pass the regression because it only resulted in a small uptick in error ratio (0.3% to 0.6%). Considered separately, however, the judge would certainly fail the error ratio on the low throughput endpoint (0% to 33%).

Heap saturation

Heap utilization can be compared in two ways: for total consumption relative to the maximum heap and allocation performance.

Total consumption is determined by dividing used by max, shown in [Example 5-8](#) and [Example 5-9](#).

Example 5-8. Atlas heap consumption canary metric

```
name,jvm.memory.used,:eq,  
name,jvm.memory.max,:eq,  
:div
```

Example 5-9. Prometheus heap consumption canary metric

```
jvm_memory_used / jvm_memory_max
```

Allocation performance can be measured by dividing allocations by promotions, shown in Examples [5-10](#) and [5-11](#).

Example 5-10. Atlas allocation performance canary metric

```
name,jvm.gc.memory.allocated,:eq,  
name,jvm.gc.memory.promoted,:eq,  
:div
```

Example 5-11. Prometheus allocation performance canary metric

```
jvm_gc_memory_allocated / jvm_gc_memory_promoted
```

CPU utilization

Process CPU utilization can be compared rather simply, shown in [Example 5-12](#) and [Example 5-13](#).

Example 5-12. Atlas CPU utilization canary metric

```
name,process.cpu.usage,:eq
```

Example 5-13. Prometheus CPU utilization canary metric

```
process_cpu_usage
```

Add canary metrics incrementally, since failed canary tests block the production path, potentially slowing down the delivery of features and bug fixes unnecessarily. Canary failures should be tuned to block dangerous regressions.

Summary

This chapter introduced continuous delivery concepts at a high level with Spinnaker as its example system. You don't need to rush to adopt Spinnaker in order to find a way to gain some of the benefits. For many enterprises, I believe clearing two hurdles would go a long way in improving release success:

Blue/green capability

Have a blue/green deployment strategy that supports N active disabled clusters for fast rollback and takes into account the unique needs of event-driven applications (as switching a load balancer isn't enough to effectively take an event-driven application out of service).

Deployed asset inventory

Have some means of querying the live state of your deployed assets. It's easier (and likely more accurate) to accomplish this by actually polling the state of your deployed environments periodically than by trying to make every possible mutating action pass through some central system like a CI server and trying to rebuild the state of the system from all the individual mutations that have occurred.

A further goal would be to have sufficient access and quality controls in place in your delivery system (again, whether that is Spinnaker or something else) to allow for some variation in deployments between teams. Some deployments, especially for static assets or internal tools, aren't going to benefit from blue/green deployment significantly. Others may release frequently enough that multiple disabled server groups in a blue/green deployment strategy are needed. Some start up fast enough that having active instances in disabled clusters represents a cost inefficiency. A platform engineering team thinking in terms of "guardrails not gates" will favor allowing this pipeline diversity over organizational consistency, maximizing the safety/cost trade-off uniquely for each team.

In the next chapter, we'll use the presence of a deployed asset inventory as an assumption for building an artifact provenance chain for your deployed assets that reaches down to the source code running in each environment.

Source Code Observability

Achieving safe delivery via pipelines or some other repeatable process is a step forward. It helps, though, to look forward to how you can observe the state of the running system beginning with deployed assets. Too much of a focus on just the pipeline itself could leave you without a means to inventory your suite of deployed assets later.

Source code is just as important to monitor as live processes. In an organization's source code, dependencies between internal components and third-party libraries are specified. Small changes in dependencies can render an application unusable. Patterns are found to repeat across an organization as developers emulate the work they see done elsewhere. Even patterns that expose attack vectors into your organization are emulated until an awareness of a vulnerability is developed. In codebases of a significant enough size, even the smallest API change can seem like an insurmountable task.

In the Netflix codebase, we found Guava version drift across deep dependency trees to be almost crippling at times. An attempt to make a shift from one logging library to another across the whole codebase had taken years and was still not achieved until an organization-wide refactoring solution was developed.

Another significant challenge is identifying what code is actually deployed where across a myriad of environments. As your organization makes advances in continuous delivery, for example, the possibility of rollbacks means that the latest release version available in your artifact repository for a given microservice may not be the version that is running in production. Or in the case of a canary test, or a blue/green deployment with greater than one active cluster, you won't even have one particular version running exclusively in production!

The need to map deployed assets to source code is particularly important for organizations that aren't monorepo-based (and most aren't). Any pressures present that

limit the adoption of new internal dependencies in your organization effectively place a ceiling on how effectively you *continuously integrate* as well.

This chapter is all about build tools. Examples given here are for the **Gradle** build tool. The patterns described here could be implemented in Maven or any other Java build tool, but the Gradle ecosystem provides the most readily available concrete examples, especially with its recent advances in binary dependency management and the Netflix Nebula plug-in ecosystem that was developed for an organization skilled at delivering microservices at scale. We'll also assume that binary artifacts for microservices, along with any core platform dependencies that they include, are published to a Maven-style artifact repository like **JFrog Artifactory** or **Sonatype Nexus**.

The focus on build tools may seem strange, given that in the software delivery life cycle, build tools come much earlier than continuous delivery, which has just been covered in [Chapter 5](#). A sense of what a stateful continuous delivery tool can do for you in terms of inventorying production assets, however, is an important prerequisite to a consideration of versioning strategies and the kinds of metadata we need to include as a microservice's artifacts are produced. Given the right data, the production asset inventory presented by your delivery solution should be the first step in a provenance chain. We're going to cover a few pieces that need to be injected into a conventional software delivery life cycle in order for you to ultimately be able to map deployed assets back to the source code that is running in them, as shown in [Figure 6-1](#). The provenance chain should lead at least down to an immutable artifact version and commit hash, and possibly all the way down to source code method-level references.

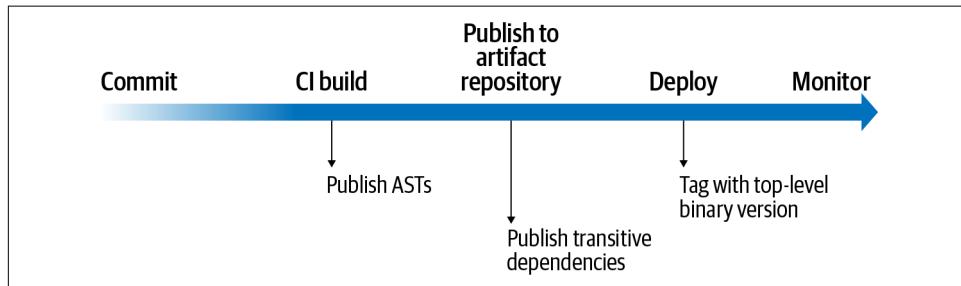


Figure 6-1. The provenance chain leading from a code change to production

Suppose we start with a continuous delivery tool like Spinnaker (introduced in [Chapter 5](#)) telling us that an application representing a microservice is spread over multiple cloud platforms, with multiple versions of the code running in different clusters. If the deployed resources are tagged with the artifact information that went into them, we consider what we can do to ensure that this artifact information leads us to a uniquely identifiable place in the history of this microservice's code.

It begins with the characteristics we need from our delivery system.



Meaning of Terms Used in This Chapter

This chapter will use terms like *instance*, *server group*, and *cluster* as defined in “Resource Types” on page 189 at the start of [Chapter 5](#).

The Stateful Asset Inventory

A queryable stateful asset inventory of code all the way to deployed assets allows you to answer questions about the current state of your systems.

The first objective in building this inventory is to have some system of record that we can query to itemize our deployed resources (in both production and lower-level nonproduction testing environments). How difficult this is depends on how many and what types of places your code can be deployed. Some organizations, even with virtualized hardware in the datacenter, have a fixed set of virtual machine names that don’t change much (numbered in the dozens or hundreds), each dedicated to a particular application. You could quite reasonably maintain a static list of application names and the virtual machines that host them. In an IaaS or CaaS where resources are provisioned more elastically, we really need to query the cloud provider for the list of currently deployed assets.



GitOps Doesn’t Achieve a Deployed Asset Inventory

With GitOps, you store the state of what you *want* to happen in Git. What materializes in the environment can be something very different. The easiest example of this divergence comes from Kubernetes. When you commit a manifest in Git, and this causes a deployment action, that manifest is mutated by Kubernetes controllers in the target cluster into something potentially different. The scope and number of mutations is only growing as vendors and projects layer more and more CRDs on top of Kubernetes. What you get with a `kubectl get pod -o yaml` is not the same manifest as what you committed in Git. And in fact, it can be different from one Kubernetes cluster to another! The point is, even if you successfully gate all *intention actions* through Git, you still don’t really have a true picture of the deployed environment in Git.

A key benefit of a system like Spinnaker is its model of polling live environments for the state of deployed infrastructure, spanning every cloud provider that Spinnaker supports. In other words, you can retrieve in one API call a consistent representation (in terms of application/cluster/server group/instance) of deployed infrastructure across many cloud platforms. It is a single-pane-of-glass experience for asset inventorying. There are two levels to this benefit:

No need to coordinate through one system

By actively polling, we don't need to centralize all possible mutations of the state of the deployed environment through one system that can maintain state separately. In theory you could require this, for example, by enforcing "gitops," i.e., a management update to the application requires an update in Git. In such a setup, there can be no rollback, load balancer changes, manual launching of server groups, or any other mutation that would result in Git not having a full picture of the state of the deployed environment.

Real-time instance-level status

What a GitOps or similar system cannot do is track the state of individual instances in a server group. AWS EC2, for example, makes no guarantees about an individual virtual machine's survivability, just that the Auto Scaling Group will do its best to maintain the specified number of instances in the ASG at any given time. The same is true in Kubernetes, where an individual pod's survivability is not guaranteed. If metrics telemetry is tagged instance ID, instance ordinal, or pod ID, it is convenient to be able to receive an alert on the violation of some service level objective, be able to drill down to a particular failing instance(s), and navigate to a real-time view of the cluster at an instance level to take some remediating action. For example, you could take a failing instance out of the load balancer, allowing the "server group" mechanism to launch another instance and addressing immediate user impact while allowing you time to investigate root cause on the failing instance before eventually terminating it.

As beneficial as it is to operations, real-time instance-level status is not immediately relevant to our discussion of artifact provenance. We really just need *some* source we can interrogate to list running server groups, clusters, and applications (see [Example 6-1](#)). For the remainder of this chapter, we'll use a Java pseudocode that describes the kinds of insights you should be able to derive by achieving a certain level of provenance information. The first part of the model encapsulates the resource types (see "[Resource Types](#)" on page 189) whose definitions were given earlier. The implementation of the methods, like `getApplications()`, is dependent on the deployment automation you use. For example, `getApplications()` is an API call to Spinnaker's Gate service to the endpoint `/applications`.

Example 6-1. Listing running deployed assets

```
delivery
  .getApplications()
  .flatMap(application -> application.getClusters())
  .flatMap(cluster -> cluster.getServerGroups())

// Where...
class Application {
  String name;
  Team owner;
  Stream<Cluster> clusters;
}

class Cluster {
  String cloudProvider;
  String name;
  Stream<ServerGroup> clusters;
}

class ServerGroup {
  String name;
  String region; ①
  String stack; ②
  String version;
  boolean enabled;
  Artifact artifact;
}
```

- ① For example, us-east-1 in AWS or namespace in K8S.
- ② The promotion level of the environment, like test or production.

For a GitOps system, `getApplications()` would interrogate the status of one or more Git repositories. For the private datacenter with a set of named virtual machines that rarely changes, this could be a manually maintained static list.

Notice how at various stages, you should be able to drill down on various characteristics of the data in a stateful delivery system. For example, see [Example 6-2](#) to get a list of teams that have applications that have some running footprint in Kubernetes:

Example 6-2. Discovering teams that have applications running in Kubernetes

```
delivery
  .getApplications()
  .filter(application -> application.getClusters()
    .anyMatch(cluster -> cluster.getCloudProvider().equals("kubernetes")))
  .map(application -> application.getTeam())
  .collect(Collectors.toSet())
```

This ability to drill down leads to better actionability. When a full provenance chain is established, we should be able to, for example, look for a recently revealed method-level security vulnerability. For a critical vulnerability, it might be desirable to first address the production stack and later follow up on the lower-level environments that contain code on a path to production eventually.

In order to be able to determine which **Artifact** a server group contains, we need proper immutable release versioning.

Release Versioning

Since the artifact repository is the part of the software delivery life cycle immediately preceding continuous delivery, a unique binary artifact version is the first step in the artifact provenance chain. For a given pipeline run, Spinnaker keeps track of the artifact inputs to that pipeline. Any resulting deployed assets will also be tagged with this provenance information. [Figure 6-2](#) shows how Spinnaker has kept track of the docker image tag and digest that was an input to a pipeline.

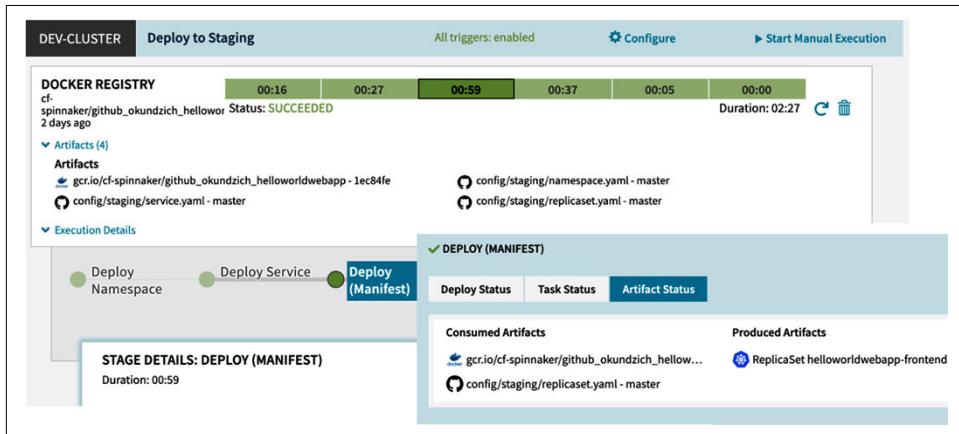


Figure 6-2. Spinnaker resolved expected artifact (notice the artifact is identified by digest instead of tag)

For this image version tag to uniquely identify a piece of code, the tag has to be unique for each unique combination of source code and dependencies. That is, if *either* the source code or dependencies that make up an application change, the version needs to change, and it is this unique version number that needs to be used to retrieve the artifact.



Docker Image Tags Are Mutable

Docker registries don't make any guarantee about tag immutability. And digests are not necessarily 1:1 with tags, though they are immutable. A common convention when publishing a new version of a Docker container to a container registry is to publish the image with the tag "latest" and some fixed version like "1.2.0." The "latest" tag effectively gets overridden, and it shares the same digest as "1.2.0" until the next published version. Despite the fact that image tags are mutable from the registry's perspective, you can build a release process that never actually mutates fixed version numbers so that this more human-readable tag can be used to associate an image with the code that is in it.

The type of artifact that becomes the input to a deployment depends on the target cloud platform. For a PaaS like Cloud Foundry it is a JAR or WAR; for Kubernetes, it is a container image, and for an IaaS like AWS EC2, an Amazon Machine Image is "baked" (see "[Packaging for IaaS Platforms](#)" on page 196) from a system package like a Debian or RPM.

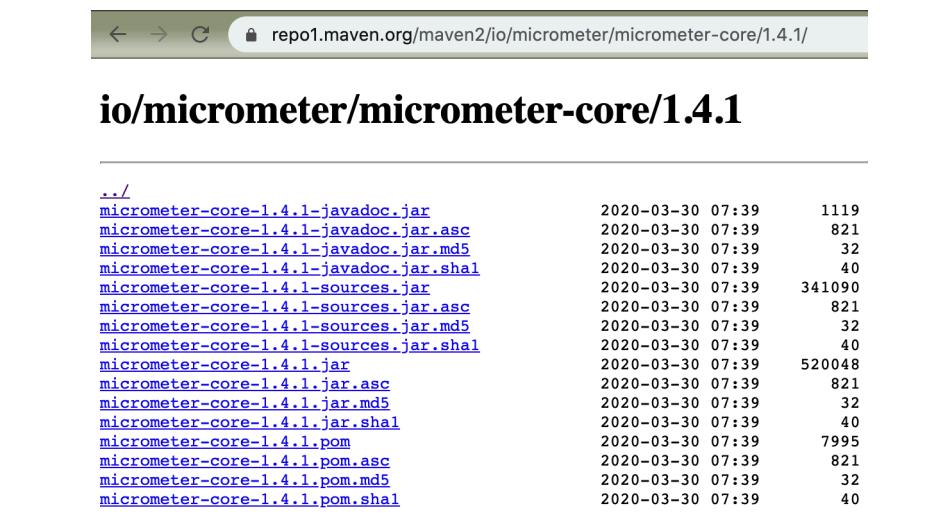
Regardless of whether we are producing a JAR, a container image, or a Debian/RPM, the same versioning scheme considerations apply.

We'll narrow this discussion to JARs published to a Maven repository (the discussion applies equally to Debiants published to a Maven repository), but the end goal of producing an immutable version would be the same for publishing container images to a container registry.

Maven Repositories

In artifact repositories, there are generally two types of Maven repositories: release repositories and snapshot repositories. The structure of the release repository is the repository's base URL plus the artifact's group, artifact, and version. Any dots in the group name are separated by / in the path to arrive at the location of an artifact. In Gradle, a dependency on Micrometer core 1.4.1 can be defined as `implementation io.micrometer:micrometer-core:1.4.1`.

The artifacts for this dependency in Maven Central look like [Figure 6-3](#).



A screenshot of a web browser displaying a Maven repository directory listing. The URL in the address bar is `repo1.maven.org/maven2/io/micrometer/micrometer-core/1.4.1/`. The page title is **io/micrometer/micrometer-core/1.4.1**. The listing shows various artifact files with their last modified date, size, and checksums. The files listed include `micrometer-core-1.4.1-javadoc.jar`, `micrometer-core-1.4.1-javadoc.jar.asc`, `micrometer-core-1.4.1-javadoc.jar.md5`, `micrometer-core-1.4.1-javadoc.jar.shal`, `micrometer-core-1.4.1-sources.jar`, `micrometer-core-1.4.1-sources.jar.asc`, `micrometer-core-1.4.1-sources.jar.md5`, `micrometer-core-1.4.1-sources.jar.shal`, `micrometer-core-1.4.1.jar`, `micrometer-core-1.4.1.jar.asc`, `micrometer-core-1.4.1.jar.md5`, `micrometer-core-1.4.1.jar.shal`, `micrometer-core-1.4.1.pom`, `micrometer-core-1.4.1.pom.asc`, `micrometer-core-1.4.1.pom.md5`, and `micrometer-core-1.4.1.pom.shal`. The sizes of the files range from 32 to 520048 bytes.

<code>.. /</code>			
<code>micrometer-core-1.4.1-javadoc.jar</code>	2020-03-30	07:39	1119
<code>micrometer-core-1.4.1-javadoc.jar.asc</code>	2020-03-30	07:39	821
<code>micrometer-core-1.4.1-javadoc.jar.md5</code>	2020-03-30	07:39	32
<code>micrometer-core-1.4.1-javadoc.jar.shal</code>	2020-03-30	07:39	40
<code>micrometer-core-1.4.1-sources.jar</code>	2020-03-30	07:39	341090
<code>micrometer-core-1.4.1-sources.jar.asc</code>	2020-03-30	07:39	821
<code>micrometer-core-1.4.1-sources.jar.md5</code>	2020-03-30	07:39	32
<code>micrometer-core-1.4.1-sources.jar.shal</code>	2020-03-30	07:39	40
<code>micrometer-core-1.4.1.jar</code>	2020-03-30	07:39	520048
<code>micrometer-core-1.4.1.jar.asc</code>	2020-03-30	07:39	821
<code>micrometer-core-1.4.1.jar.md5</code>	2020-03-30	07:39	32
<code>micrometer-core-1.4.1.jar.shal</code>	2020-03-30	07:39	40
<code>micrometer-core-1.4.1.pom</code>	2020-03-30	07:39	7995
<code>micrometer-core-1.4.1.pom.asc</code>	2020-03-30	07:39	821
<code>micrometer-core-1.4.1.pom.md5</code>	2020-03-30	07:39	32
<code>micrometer-core-1.4.1.pom.shal</code>	2020-03-30	07:39	40

Figure 6-3. Maven Central directory listing for micrometer-core 1.4.1

The directory listing contains the binary JAR (`micrometer-core-1.4.1.jar`), checksums, a Maven POM file describing the module, a set of checksums, and optionally sources and javadoc JARs.



Maven Release Versions Are Immutable

Generally an artifact version like 1.4.1 in a Maven repository is immutable. No other code should ever be published over this version.

Maven snapshot repositories are structured a little differently. [Figure 6-4](#) shows how RSocket's snapshots are structured in a Maven repository for the dependency implementation `io.rsocket:rsocket-core:1.0.0-RC7-SNAPSHOT`.

Name	Last modified	Size
<hr/>		
<code>./maven-metadata.xml</code>	19-Apr-2020 08:34	1.37 KB
<code>rsocket-core-1.0.0-RC7-20200204.195351-1-javadoc.jar-></code>	-	-
<code>rsocket-core-1.0.0-RC7-20200204.195351-1-sources.jar-></code>	-	-
<code>rsocket-core-1.0.0-RC7-20200204.195351-1.jar-></code>	-	-
<code>rsocket-core-1.0.0-RC7-20200204.195351-1.module-></code>	-	-
<code>rsocket-core-1.0.0-RC7-20200204.195351-1.pom-></code>	-	-
<code>rsocket-core-1.0.0-RC7-20200222.203307-2-javadoc.jar-></code>	-	-
<code>rsocket-core-1.0.0-RC7-20200222.203307-2-sources.jar-></code>	-	-
<code>rsocket-core-1.0.0-RC7-20200222.203307-2.jar-></code>	-	-
<code>rsocket-core-1.0.0-RC7-20200222.203307-2.module-></code>	-	-
<code>rsocket-core-1.0.0-RC7-20200222.203307-2.pom-></code>	-	-
<code>rsocket-core-1.0.0-RC7-20200225.110351-3-javadoc.jar-></code>	-	-
<code>rsocket-core-1.0.0-RC7-20200225.110351-3-sources.jar-></code>	-	-
<code>rsocket-core-1.0.0-RC7-20200225.110351-3.jar-></code>	-	-
<code>rsocket-core-1.0.0-RC7-20200225.110351-3.module-></code>	-	-
<code>rsocket-core-1.0.0-RC7-20200225.110351-3.pom-></code>	-	-
<code>rsocket-core-1.0.0-RC7-20200308.194304-4-javadoc.jar-></code>	-	-
<code>rsocket-core-1.0.0-RC7-20200308.194304-4-sources.jar-></code>	-	-
<code>rsocket-core-1.0.0-RC7-20200308.194304-4.jar-></code>	-	-
<code>rsocket-core-1.0.0-RC7-20200308.194304-4.pom-></code>	-	-

Figure 6-4. Spring Artifactory snapshot repository for RSocket 1.0.0-RC7-SNAPSHOT

The path to this directory listing has the path segment `1.0.0-RC7-SNAPSHOT`, but notice that none of the actual artifacts do. Instead they replace `SNAPSHOT` with a timestamp when they were published. If we look in `maven-metadata.xml`, we'll see that it maintains a record of the last timestamp that was published to this snapshot repository, as shown in Example 6-3.

Example 6-3. Maven metadata for RSocket 1.0.0-RC7-SNAPSHOT

```
<?xml version="1.0" encoding="UTF-8"?>
<metadata modelVersion="1.1.0">
  <groupId>io.rsocket</groupId>
  <artifactId>rsocket-core</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <versioning>
    <snapshot>
      <timestamp>20200423.184223</timestamp>
      <buildNumber>24</buildNumber>
    </snapshot>
    <lastUpdated>20200423185021</lastUpdated>
    <snapshotVersions>
      <snapshotVersion>
        <extension>jar</extension>
        <value>1.0.0-RC7-20200423.184223-24</value>
        <updated>20200423184223</updated>
      </snapshotVersion>
      <snapshotVersion>
        <extension>pom</extension>
        <value>1.0.0-RC7-20200423.184223-24</value>
        <updated>20200423184223</updated>
      </snapshotVersion>
    </snapshotVersions>
  </versioning>
</metadata>
```

```
...
</snapshotVersions>
</versioning>
</metadata>
```

Each time a new snapshot is published, this `maven-metadata.xml` is updated by the artifact repository. This means that the dependency implementation `io.rsocket:rsocket-core:1.0.0-RC7-SNAPSHOT` is *not* immutable. If we tagged some deployed asset with the version `1.0.0-RC7-SNAPSHOT`, we don't have enough specific information to tie this snapshot back to the particular snapshot timestamp that was latest at the time that the deployment happened.

In other words, Maven snapshot versioning is a problem for artifact provenance because it doesn't uniquely identify a binary dependency in the artifact repository.

Microservice versioning is different than library versioning as well in that at any time a candidate version that is being tested in a lower environment could be *promoted* to production. It's safest if we don't examine a candidate binary with a snapshot-like version in a lower-level environment and, deciding that it is fit for promotion to production, *rebuild* the binary with a "release" version number. Rebuilding the binary is wasteful in both time and artifact repository storage. More significantly, it introduces the possibility that any part of the build (or the conditions of the machine on which the build is occurring, which might influence the resulting binary) is not repeatable.

To summarize everything we've discussed about versioning into two principles for microservice versioning, we need the following:

Uniqueness

A version number for each unique combination of source code and dependencies

Immutability

A guarantee that artifact versions are never overwritten in the artifact repository

Maven snapshots fail the uniqueness test.

While there are a variety of release versioning schemes that could work, there is a relatively simple approach using open source build tooling that takes quite a bit of the toil out of microservice versioning while meeting both of these tests.

Build Tools for Release Versioning

The Netflix Nebula suite of plug-ins contains a release plug-in that provides a convenient workflow for calculating a version number that meets both of these tests. It contains a set of Gradle tasks for versioning your project: `final`, `candidate`, and `devSnapshot`. When working on libraries, it is typical to build snapshots until you are close to a release, then maybe do one or more release candidates, and finally produce a final release. For microservice versioning, the situation is a bit different, again

because at any time a particular iteration of code running in a lower-level environment could be promoted to production. An iteration cycle is shown in [Figure 6-5](#). In this example workflow, tagging at the end of the deployment pipeline advances the minor release number N for the next development iteration.

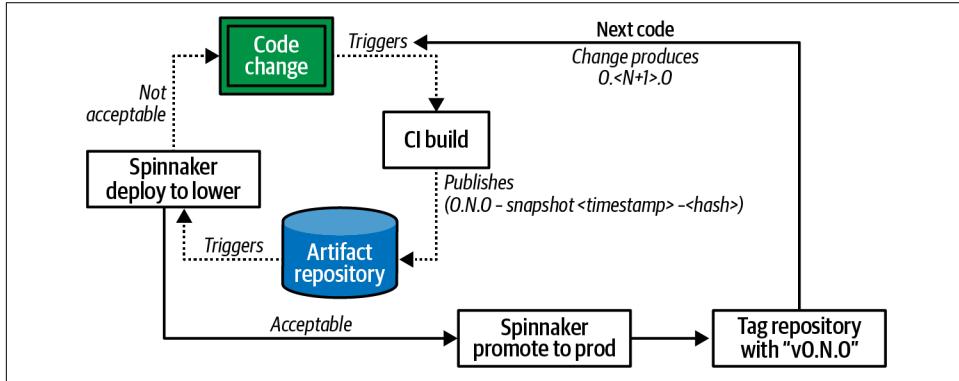


Figure 6-5. Microservice release versioning cycle

This cycle of code change, continuous integration building and producing an artifact and storing it in an artifact repository, and delivery automation provisioning the new artifact in a lower-level environment can be performed potentially many times before finally promoting a deployment to production. The specialized case where there is only one loop is the idealized continuous deployment model. This is the case when any successful deployment to a lower-level environment (and maybe some automated tests against that lower-level environment) results in a production promotion. Whether you get to this level of comfort with immediately shipping changes or not is not important to the versioning scheme.

Whenever the build is run, Nebula release looks at the latest tag on the repository, say, v0.1.0, and selects (by default) to generate snapshots (and candidates, and final builds for libraries) for the *next* minor version. In this case, that would be the minor release 0.2.0.

In the proposed versioning cycle, CI will execute a Gradle build for a repository that uses Nebula release to generate immutable snapshots. The minor revision remains consistent until a deployment is eventually promoted to production, at which point your delivery automation, like a Spinnaker pipeline stage, tags the repository (e.g., a Spinnaker job stage specifically executes `./gradlew final` on the repository, pushing a tag to the Git remote for the current minor release iteration).



Releasing SaaS Versus Packaged Software

Even packaged software can consist of a series of microservices. For example, Spinnaker itself is a suite of microservices that are intended to be deployed together. Generally there is an extra step for packaged software: producing some sort of bill of materials that itself is revisioned containing versions for each individual microservice. These versions are included in the bill of materials to indicate that they are tested *together* and known to work as a group. Bills of materials help to create potentially many running copies of a set of microservices. When running SaaS, the production environment tends to be the only running copy, and the bill of materials isn't necessary.

The Nebula release plug-in is applied to the root project of a Gradle project, as shown in [Example 6-4](#).

Example 6-4. The Nebula release plug-in applied to the root project of a Gradle project

```
plugins {
    java
    id("nebula.release") version "LATEST" ❶
    id("nebula.maven-publish") version "LATEST"
    id("nebula.maven-resolved-dependencies") version "LATEST"
}

project
    .rootProject
    .tasks
    .getByName("devSnapshot")
    .dependsOn(project
        .tasks
        .getByName("publishNebulaPublicationToArtifactory")) ❷

project
    .gradle
    .taskGraph
    .whenReady(object: Action<TaskExecutionGraph> { ❸
        override fun execute(graph: TaskExecutionGraph) {
            if (graph.hasTask(":snapshot") ||
                graph.hasTask(":immutableSnapshot")) {

                throw GradleException("You cannot use the snapshot or" +
                    "immutableSnapshot task from the release plugin. " +
                    "Please use the devSnapshot task.")
            }
        }
    })

publishing {
```

```

repositories {
    maven {
        name = "Artifactory" ④
        url = URI.create("https://repo.myorg.com/libs-services-local")
    }
}

```

- ❶ Replace LATEST with the version listed on the [Gradle plug-in portal](#).
- ❷ Whenever the CI build executes `devSnapshot`, build and publish an artifact to Artifactory. Because we *haven't* attached `final` to publishing, running `./gradlew final` in a stage after promotion to production will tag the repository with the current minor release (e.g., v0.1.0) and push the tag to the repository, but not unnecessarily upload another artifact to the artifact repository. The existence of this tag completes the development cycle. Any subsequent code pushes, and therefore runs of `./gradlew devSnapshot`, then generate snapshots for the next minor release (e.g., `0.2.0-snapshot.<timestamp>+<commit_hash>`).
- ❸ Optionally, ensure that developers don't accidentally use the other types of snapshot tasks made available by Nebula release that would generate snapshots with different version number semantics.
- ❹ Define the repository that is referenced in the `dependsOn` clause of `devSnapshot`.

For the continuous deployment model of every commit (that passes automated tests) resulting in a production deployment, provided that there is no need for an automated test suite in a lower environment, there will be precisely one snapshot per minor release. If all your checks were run prior to artifact publishing, and you trust the outcome enough to promote to production immediately, you could easily use `./gradlew final` and avoid immutable snapshots altogether. Few enterprises are going to be comfortable with this, and there is no pressure to ever reach this level of automation. As mentioned in [“Separating Platform and Application Metrics” on page 92](#), you “ship your org chart” to some extent.

Associating every deployment with an immutable release version is the first phase in the artifact provenance chain. When you have a combination of the ability to interrogate your delivery service for all current deployments *and* your delivery automation somehow tags each deployment with an immutable release version (whether this is stored in the Auto Scaling Group name in EC2, a Kubernetes tag, etc.), you unlock the ability to iterate over all production deployment resources and map to artifact coordinates that are retrievable from your artifact repository.

The pseudocode showing the extent of the artifact provenance chain to this point is in [Example 6-5](#).

Example 6-5. Mapping deployed resources to artifact versions

```
delivery
    .getApplications()
    .flatMap(application -> application.getClusters())
    .flatMap(cluster -> cluster.getServerGroups())
    .map(serverGroup -> serverGroup.getArtifact()) ❶

// Where...
@EqualsAndHashCode(includes = {"group", "artifact", "version"}) ❷
class Artifact {
    String group;
    String artifact;
    String version;
    Set<Artifact> dependencies; ❸
}
```

- ❶ Type is `Stream<Artifact>` because there is a 1:1 correspondence between a deployment and an artifact.
- ❷ Because `devSnapshot` produces immutable artifact versions that are unique for every unique combination of source code and dependencies, two artifacts with the same group/artifact/version coordinates are guaranteed to have the same dependencies as well.
- ❸ At this stage, we don't yet have the ability to determine dependencies.

We need more configuration up front to provide artifact provenance that includes dependencies as well.

Capturing Resolved Dependencies in Metadata

Extending the provenance chain deeper to include the dependencies of applications allows us to quickly find which deployed assets contain a version of a *library* that may be problematic, for example, because of an identified security vulnerability.

Generally, when we publish a Maven POM file along with the application, it contains a `<dependencies>` block that only lists the first-level dependencies. These are the dependencies that are directly listed in the `dependencies { }` section of the Gradle build as well. For example, for a sample Spring Boot application generated from start.spring.io, the first-level dependencies would be something like [Example 6-6](#). Specifically, `spring-boot-starter-actuator` and `spring-boot-starter-webflux` are the two first-level dependencies.

Example 6-6. First-level dependencies of a sample Spring Boot application

```
dependencies {  
    implementation("org.springframework.boot:spring-boot-starter-actuator")  
    implementation("org.springframework.boot:spring-boot-starter-webflux")  
    testImplementation("org.springframework.boot:spring-boot-starter-test")  
    testImplementation("io.projectreactor:reactor-test")  
}
```

For the sake of establishing a provenance chain, test dependencies are not really important. They are only on the classpath during test execution on local developer machines and on continuous integration builds, and they aren't packed in the final application that is running in the deployed environment. Any problem or vulnerability in test dependencies is safely confined to the continuous integration environment and doesn't cause problems in running deployed applications.

When this application is published to an artifact repository like Artifactory, a Maven POM file is published alongside the binary artifact which contains a `dependencies` section like in [Example 6-7](#).

Example 6-7. First-level dependencies as they appear in the Maven POM

```
<dependencies>  
    <dependency>  
        <groupId>org.springframework.boot</groupId>  
        <artifactId>spring-boot-starter-actuator</artifactId>  
        <version>2.3.0.M4</version>  
        <scope>runtime</scope>  
    </dependency>  
    <dependency>  
        <groupId>org.springframework.boot</groupId>  
        <artifactId>spring-boot-starter-webflux</artifactId>  
        <version>2.3.0.M4</version>  
        <scope>runtime</scope>  
    </dependency>  
</dependencies>
```

First-level dependencies of course bring in *other* dependencies. The set of all other dependencies, resolved recursively from the first level down, is called the transitive closure of dependencies.

It may be tempting to think that the transitive closure of dependencies in this sample application could be determined simply by recursively fetching the POM files of each dependency from the artifact repository. This is unfortunately not the case. Commonly enough, additional constraints are present in the build that influence in some way the transitive dependencies that were resolved and packaged with the application. For example, a particular version could be denied with a replacement because some critical bug is fixed by a later version, as in [Example 6-8](#).

Example 6-8. Denying a version with a replacement

```
configurations.all {  
    resolutionStrategy.eachDependency {  
        if (requested.group == "org.software" &&  
            requested.name == "some-library") {  
  
            useVersion("1.2.1")  
            because("fixes critical bug in 1.2")  
        }  
    }  
}
```



No Process That Determines Dependencies from the Artifact Repository Alone Is Correct

The consequence of build-time features like resolution strategies is that no process that tries to determine the transitive closure of dependencies from standard artifact metadata alone (such as the POM `<dependencies>` block) will be correct.

Vendors trying to build universal component analysis tools strictly from standard metadata can only give you an *approximation* of what dependencies are in use. And this approximation is often not trivially divergent from reality.

As a result of resolution strategies, forced versions, etc., the transitive closure must be persisted somehow as the application binary is published. One easy way to do this is to include the resolved transitive closure in a POM `<properties>` element for later consumption by any tooling you build to examine the dependencies in use across your organization.

The [Nebula Info](#) plug-in specializes in just this, attaching build-time metadata to the `<properties>` section of a POM. Out of the box, Nebula Info adds properties for things like Git commit hash and branch, the source and target Java version, and build host.

The `InfoBrokerPlugin` allows us to add new properties at will by key-value pair. [Example 6-9](#) shows how to traverse the transitive dependency closure of the runtime classpath and add the list of dependencies as a property. `nebula.maven-manifest`, included automatically by `nebula.maven-publish`, reads all the properties managed by the info broker and adds them as POM properties.

Example 6-9. List all transitive dependencies sorted in a flat list

```
plugins {
    id("nebula.maven-publish") version "LATEST" ❶
    id("nebula.info") version "LATEST"
}

tasks.withType<GenerateMavenPom> {
    doFirst {
        val runtimeClasspath = configurations
            .getByName("runtimeClasspath")
        val gav = { d: ResolvedDependency ->
            "${d.moduleGroup}:${d.moduleName}:${d.moduleVersion}"
        }
        val indented = "\n" + " ".repeat(6)

        project.plugins.withType<InfoBrokerPlugin> {
            add("Resolved-Dependencies", runtimeClasspath
                .resolvedConfiguration
                .lenientConfiguration
                .allModuleDependencies
                .sortedBy(gav)
                .joinToString(
                    indented,
                    indented,
                    "\n" + " ".repeat(4), transform = gav
                )
            )
        }
    }
}
```

❶ Replace LATEST with whatever the latest version is on the [Gradle Plug-in Portal](#).

This results in a properties listing in the POM that looks something like [Example 6-10](#).

Example 6-10. A flattened transitive dependency closure listed, sorted, and added as a POM property

```
<project ...>
<groupId>com.example</groupId>
<artifactId>demo</artifactId>
<version>0.1.0</version>
<name>demo</name>
...
<properties>
    <nebula_Change>1b0f8d9</nebula_Change>
    <nebula_Branch>master</nebula_Branch>
    <nebula_X_Compile_Target_JDK>11</nebula_X_Compile_Target_JDK>
    <nebula_Resolved_Dependencies>
```

```

ch.qos.logback:logback-classic:1.2.3
ch.qos.logback:logback-core:1.2.3
com.datastax.oss:java-driver-bom:4.5.1
com.fasterxml.jackson.core:jackson-annotations:2.11.0.rc1
com.fasterxml.jackson.core:jackson-core:2.11.0.rc1
com.fasterxml.jackson.core:jackson-databind:2.11.0.rc1
</nebula_Resolved_Dependencies>
</properties>
</project>
```

Now, to build tooling to find all deployments that contain `logback-core` version 1.2.3, we can use something like the pseudocode in [Example 6-11](#) to list all the server groups containing a particular `logback-core` dependency. The population of dependencies on the `Artifact` type consists of downloading the POM from the artifact repository, given the `Artifact`'s group/artifact/version coordinates, and parsing the contents of the `<nebula_Resolved_Dependencies>` POM property.

Example 6-11. Mapping deployed resources to the set of all dependencies included in them

```

delivery
    .getApplications()
    .flatMap(application -> application.getClusters())
    .flatMap(cluster -> cluster.getServerGroups())
    .filter(artifact -> serverGroup
        .getArtifact()
        .getDependencies()
        .stream()
        .anyMatch(d -> d.getArtifact().equals("logback-core") &&
            d.getVersion().equals("1.2.3"))
    )
```

The flattened list representation can be improved upon a little for readability when glancing at an individual POM file without really affecting how we parse the transitive closure for a given artifact. [Example 6-12](#) shows how to instead create a pretty-printed minimum spanning tree of the transitive closure of dependencies.

Example 6-12. A tree view of the transitive dependency closure added as a POM property

```

tasks.withType<GenerateMavenPom> {
    doFirst {
        val runtimeClasspath = configurations.getByName("runtimeClasspath")

        val gav = { d: ResolvedDependency ->
            "${d.moduleGroup}:${d.moduleName}:${d.moduleVersion}"
        }
    }
}
```

```

val observedDependencies = TreeSet<ResolvedDependency> { d1, d2 ->
    gav(d1).compareTo(gav(d2))
}

fun reduceDependenciesAtIndent(indent: Int):
    (List<String>, ResolvedDependency) -> List<String> =
    { dependenciesAsList: List<String>, dep: ResolvedDependency ->

        dependenciesAsList + listOf(" ".repeat(indent)) +
        dep.module.id.toString() + (
            if (observedDependencies.add(dep)) {
                dep.children
                    .sortedBy(gav)
                    .fold(emptyList(), reduceDependenciesAtIndent(indent + 2))
            } else {
                // This dependency subtree has already been printed, so skip
                emptyList()
            }
        )
    }

project.plugins.withType<InfoBrokerPlugin> {
    add("Resolved-Dependencies", runtimeClasspath
        .resolvedConfiguration
        .lenientConfiguration
        .firstLevelModuleDependencies
        .sortedBy(gav)
        .fold(emptyList(), reduceDependenciesAtIndent(6))
        .joinToString("\n", "\n", "\n" + " ".repeat(4)))
}
}
}

```

This generates a resolved dependencies property that looks something like [Example 6-13](#). This looks a bit more readable individually, and consuming such a property from tooling is equivalent to the flattened representation—simply strip the whitespace off the front of each line no matter how much whitespace there is.

Example 6-13. A tree view of the transitive dependency closure shown as a POM property

```

<project ...>
    <groupId>com.example</groupId>
    <artifactId>demo</artifactId>
    <version>0.1.0</version>
    <name>demo</name>
    ...
    <properties>
        <n Nebula_Change>1b0f8d9</n>
        <n Nebula_Branch>master</n>
        <n Nebula_X_Compile_Target_JDK>11</n>
    
```

```

<nebula_Resolved_Dependencies>
  org.springframework.boot:spring-boot-starter-actuator:2.3.0.M4
    io.micrometer:micrometer-core:1.3.7
      org.hdrhistogram:HdrHistogram:2.1.11
      org.latencyutils:LatencyUtils:2.0.3
  org.springframework.boot:spring-boot-actuator-autoconfigure:2.3.0.M4
    com.fasterxml.jackson.core:jackson-databind:2.11.0.rc1
    com.fasterxml.jackson.datatype:jackson-datatype-jsr310:2.11.0.rc1
      com.fasterxml.jackson.core:jackson-annotations:2.11.0.rc1
      com.fasterxml.jackson.core:jackson-core:2.11.0.rc1
      com.fasterxml.jackson.core:jackson-databind:2.11.0.rc1
</nebula_Resolved_Dependencies>
</properties>
</project>

```

You can see a hint in this Nebula info output of how we can go further to the commit (<nebula_Change>).

Capturing Method-Level Utilization of the Source Code

Going a step further, we can capture the source code level for a given artifact version. This is the last phase in the provenance chain. Once complete, the chain leads from a conceptual “application,” consisting of clusters spread across potentially multiple cloud providers, all the way down to the method declarations and invocations of the source code in these applications. For example, for an AWS EC2 IaaS-based deployment footprint, the provenance chain now looks like [Example 6-14](#).

Example 6-14. An AWS EC2 full provenance chain from application to method-level source code

```

Application
-> Owner (team)
-> Clusters
-> Server groups
-> Instances
-> Amazon Machine Image (AMI)
-> Debian
-> JAR (or WAR)
-> Git commit
-> Source abstract syntax tree ①
-> Classes
-> Method declarations
-> Method invocations
-> Dependencies
-> Git commit
-> Source abstract syntax tree
-> Classes
-> Method declarations
-> Method invocations

```

- ① How this is constructed is the topic of this section.

Since this chain is linked top-to-bottom with just enough uniquely identifying metadata, we can answer almost any question about our running production environment. Each of these scenarios now has an exact answer that can be updated at will against the current state of a deployed footprint:

Zero-day exploit in an open source third-party library

The security team learns of a vulnerability in a method in an open source third-party library that might be used in the company. This exposure could lead to sensitive personally identifiable customer information leaking if exploited and have significant legal liability and brand implications for the company. Because this vulnerability is widespread, the security team wants to address this problem in two phases, first focusing on production public-facing assets invoking the vulnerable method, and then later addressing internal tools and application code in lower environments on a path to eventual production deployment. Which teams does the security team need to communicate with in the first phase that have potential execution paths leading to the vulnerable method?

Platform team wants to deprecate or change an API

A centralized tools team, responsible for providing a library that every microservice team uses to check whether a given request is subject to a running A/B experiment, would like to make an API change in the way its library is used. The tools team used the source code search mechanism on its Github Enterprise instance to see where the existing API was used and noticed that some of the results pointed to dead code. If the team was to go ahead with the API change, which actively developed and deployed code would be affected? What is the set of teams affected by this potential change? If it is a small number of teams, the tools team can likely meet individually with its affected users or hold a small meeting and feedback session and proceed with the change. If it affects a much broader swath of the organization, perhaps the change is approached more incrementally or reconsidered altogether.

When it comes to identifying which methods are being used in the execution paths of running microservices, there are two approaches:

Live monitoring of execution paths

An agent can be attached to a running Java process to monitor which methods are actually executed. The set of all observed method signatures over a period of time approximates the reachable execution paths. For example, [Snyk](#), which specializes in security vulnerability analysis, provides a Java agent that monitors demonstrable execution paths to match against its method-level vulnerability research and alert organizations to known vulnerabilities. Live monitoring like

this naturally *underreports* the real set of execution paths to a point, since some execution paths may be rarely exercised (e.g., exception-handling paths).

Statically analyzing potential execution paths from the source code

Several Java tools exist to construct an **abstract syntax tree** (an intermediate representation of the source code). This tree can be walked to look for method invocations. The set of all method invocations across the abstract syntax trees for every class in a source repository represents the set of all potential executions. This naturally *overreports* the real set of execution paths, since some method invocations are going to exist buried in a set of conditionals that never evaluate to true or request mappings that are never exercised. An example of one of these tools available in the open is given in “[Structured Code Search with OpenRewrite](#)” on page 243.

Because of the under/overreporting nature of these approaches, it can be useful to combine them. Any applications which are reporting the use of a method through live monitoring are certain to require a change. Potential execution paths can be evaluated on a second pass.

Static analysis tools need to evaluate source code. The combination of `nebula.info` and `nebula.maven-publish`, shown again in [Example 6-15](#), gives us a Git commit hash and branch, the right amount of detail to connect an artifact version known to be running in a given server group to the source code that is included in it. Also, you can follow the application artifact’s transitive dependency closure, looking at a POM file for each dependency and peeking at what commit hash and branch each dependency used to examine their source code in turn.

Example 6-15. Gradle plug-ins that generate POM properties for Git commit hash and branch

```
plugins {
    id("nebula.maven-publish") version "LATEST"
    id("nebula.info") version "LATEST"
}
```

The POM properties this generates, shown in [Example 6-16](#), are easily scraped by tools.

Example 6-16. Maven POM properties for Git commit hash and branch

```
<project ...>
  <groupId>com.example</groupId>
  <artifactId>demo</artifactId>
  <version>0.1.0</version>
  <name>demo</name>
  ...
  <properties>
    <nebula_Change>1b0f8d9</nebula_Change>
    <nebula_Branch>master</nebula_Branch>
  </properties>
</project>
```

To set expectations about what kinds of capabilities a structured code search tool should have that could analyze execution paths, consider OpenRewrite.

Structured Code Search with OpenRewrite

The [OpenRewrite](#) project is a mass refactoring ecosystem for Java and other source code, designed to eliminate technical debt across an engineering organization. Rewrite is designed to be plugged into various workflows, including the following:

- Discover and fix code as a build tool task (e.g., Gradle and Maven).
- Subsecond organization-wide code search for a pattern of arbitrary complexity.
- Mass pull-request issuance to fix a security vulnerability, eliminate the use of a deprecated API, migrate from one technology to another (e.g., JUnit asserts to AssertJ), etc.
- Mass organization-wide Git commits to do the same.

It builds on a custom abstract syntax tree (AST) that encodes the structure and formatting of source code. The AST is printable to reconstitute the source code, including its original formatting. Rewrite provides high-level search and refactoring functions that can transform the AST, as well as utilities for unit testing refactoring logic.

Two key capabilities of the Rewrite AST make it suited for the purpose of the provenance chain:

Type-attributed

Each AST element is imbued with type information. For a field reference, for example, the source code may just refer to myField. The Rewrite AST element for myField would also contain information about what the type of myField is, even if it isn't defined in the same source file or even the same project.

Acyclic and serializable

Most ASTs containing type information are potentially cyclic. Cycles usually come from generic type signatures like class A<T extends A<T>>. This kind of pattern is generally found in things like abstract builder types in Java. Rewrite cuts these cycles off and adds serialization annotations to its types so the AST can be serialized/deserialized with libraries like Jackson.

Type attribution is necessary for accurate matching of patterns. How do we know if `logger` is an SLF4J or a Logback logger when looking at a statement like [Example 6-17](#)? The class type of the instance that a method is invoked against is called the receiver type.

Example 6-17. A logging statement with an ambiguous receiver type

```
logger.info("Hi");
```

The production of type-attributed ASTs for a whole organization is arbitrarily computationally complex, since it requires dependency resolution, parsing of the source code, and type attribution (basically Java compilation up to the point of bytecode generation). Since Rewrite ASTs are serializable, we can store them off centrally as a byproduct of compilation in continuous integration environments and then operate on them en masse later.

Once we have a serialized AST for a particular source file, and because it also contains type information, it can be refactored/searched completely independently of other source files in the same source package or repository. This makes mass search and refactoring a truly linearly scalable operation.

Creating a Rewrite AST from Java source code

To build a Rewrite AST for Java source code, construct a `JavaParser` either with or without the runtime classpath using one of the constructor signatures shown in [Example 6-18](#).

Example 6-18. Constructing a JavaParser instance

```
JavaParser();  
JavaParser(List<Path> classpath);
```

Providing a classpath is optional, because type attribution is a *best effort* for each element. If we are storing ASTs in a datastore for organization-wide search, ideally they are stored fully type-attributed, because you don't know what kinds of searches will be made in advance. The kinds of searches include the following:

No types needed at all

If you are applying a refactoring rule like autoremediation for Checkstyle's `White spaceBefore` rule, we're strictly looking at source formatting, and it's OK if none of the AST elements have types on them, as it doesn't influence the outcome.

Partial types needed

If searching for occurrences of deprecated Guava methods, it is OK to construct a `JavaParser` with a path to a Guava binary. It doesn't even have to be the Guava version that the project is using! The resulting ASTs will have limited type information, but just enough to search for what we want.

Full types needed

When ASTs are emitted as a side effect of compilation to a central datastore for later arbitrary code search, they need to have full type information, because we can't be sure in advance what kinds of searches people will attempt.

`JavaParser` contains a convenience method for building a `JavaParser` from the runtime classpath of the Java process that is constructing the parser, shown in [Example 6-19](#).

Example 6-19. Giving the parser the compile dependencies necessary to do type attribution

```
new JavaParser(JavaParser.dependenciesFromClasspath("guava"));
```

This utility takes the “artifact name” of the dependency to look for. The artifact name is the artifact portion of `group:artifact:version` coordinates. For example, for Google's Guava (`com.google.guava:guava:VERSION`), the artifact name is `guava`.

Once you have a `JavaParser` instance, you can parse all the source files in a project with the `parse` method, which takes a `List<Path>`. [Example 6-20](#) shows the process.

Example 6-20. Parsing a list of Java source paths

```
JavaParser parser = ...;
List<J.CompilationUnit> cus = parser.parse(pathsToSourceFiles);
```

`J.CompilationUnit`

This is the top-level AST element for Java source files, which contains information about the package, imports, and any class/enum/interface definitions contained in the source file. `J.CompilationUnit` is the basic building block upon which we'll build refactoring and search operations for Java source code.

JavaParser

This contains `parse` method overloads for constructing an AST from a string, which is useful for quickly constructing unit tests for different search and refactoring operations.

For JVM languages like Kotlin that support multiline strings, this can be especially convenient, as shown in [Example 6-21](#).

Example 6-21. Parsing Java source

```
val cu: J.CompilationUnit = JavaParser().parse("""
    import java.util.Collections;
    public class A {
        Object o = Collections.emptyList();
    }
""")
```

Notice how this returns a single `J.CompilationUnit`, which can be immediately acted upon. Ultimately, [JEP-355](#) will bring multiline strings to Java as well, so beautiful unit tests for Rewrite operations will be possible to write in plain Java code.

The `dependenciesFromClasspath` method demonstrated in [Example 6-22](#) is especially useful for building unit tests, as you can place a module for which you are affecting some transformation on the test runtime classpath and bind it to the parser. In this way, any references to classes, methods, etc., in that dependency are type-attributed in ASTs produced for unit tests.

Example 6-22. Parsing source with a classpath for type attribution

```
val cu: J.CompilationUnit = JavaParser(JavaParser.dependenciesFromClasspath("guava"))
    .parse("""
        import com.google.common.io.Files;
        public class A {
            File temp = Files.createTempDir();
        }
""")
```

Performing a search with Rewrite

Extending on the previous example, we can search for uses of Guava's `Files#createTempDir()`, shown in [Example 6-23](#). The argument for `findMethodCalls` takes the [AspectJ syntax](#) for pointcut matching on methods.

Example 6-23. Performing a search with Rewrite

```
val cu: J.CompilationUnit = JavaParser(JavaParser.dependenciesFromClasspath("guava"))
    .parse("""
        import com.google.common.io.Files;
        public class A {
            File temp = Files.createTempDir();
        }
""")
```

```

import com.google.common.io.Files;
public class A {
    File temp = Files.createTempDir();
}
""")
```

`val calls: List<J.MethodInvocation> = cu.findMethodCalls("java.io.File com.google.common.io.Files.createTempDir()");`

Many other search methods exist on `J.CompilationUnit`, among them the following:

`boolean hasImport(String clazz)`
Looks for imports

`boolean hasType(String clazz)`
Checks whether a source file has a reference to a type

`Set<NameTree> findType(String clazz)`
Returns all the AST elements that are type-attributed with a particular type

You can also move down a level to individual classes (`cu.getClasses()`) inside a source file and perform additional operations:

`List<VariableDecls> findFields(String clazz)`
Finds fields declared in this class that refer to a specific type.

`List<JavaType.Var> findInheritedFields(String clazz)`
Finds fields that are inherited from a base class. Note that since they are inherited, there is no AST element to match on, but you'll be able to determine if a class has a field of a particular type coming from a base class and then look for uses of this field.

`Set<NameTree> findType(String clazz)`
Returns all AST elements inside this class referring to a type.

`List<Annotation> findAnnotations(String signature)`
Finds all annotations matching a signature as defined in the AspectJ pointcut definition for annotation matching.

`boolean hasType(String clazz)`
Checks whether a class refers to a type.

`hasModifier(String modifier)`
Checks for modifiers on the class definition (e.g., public, private, static).

`isClass()/isEnum()/isInterface()/isAnnotation()`
Checks the type of declaration.

More search methods are available further down the AST.

You can build custom search visitors by extending `JavaSourceVisitor` and implementing any `visitXXX` methods that you need to perform your search. These don't have to be complex. `FindMethods` only extends `visitMethodInvocation` to check whether a given invocation matches the signature we are looking for, as shown in [Example 6-24](#).

Example 6-24. The implementation of the `FindMethods` operation in Rewrite

```
public class FindMethods extends JavaSourceVisitor<List<J.MethodInvocation>> {
    private final MethodMatcher matcher;

    public FindMethods(String signature) {
        this.matcher = new MethodMatcher(signature);
    }

    @Override
    public List<J.MethodInvocation> defaultTo(Tree t) {
        return emptyList();
    }

    @Override
    public List<J.MethodInvocation> visitMethodInvocation(J.MethodInvocation method) {
        return matcher.matches(method) ?
            singletonList(method) :
            super.visitMethodInvocation(method);
    }
}
```

Invoke a custom visitor by instantiating the visitor and calling `visit` on the root AST node, as shown in [Example 6-25](#). `JavaSourceVisitor` can return any type. You define a default return with `defaultTo` and can provide a custom reduction operation by overriding `reduce` on the visitor.

Example 6-25. Invoking a custom Rewrite visitor

```
J.CompilationUnit cu = ...;

// This visitor can return any type you wish, ultimately
// being a reduction of visiting every AST element
new MyCustomVisitor().visit(cu);
```

Refactoring Java source

One of the benefits of establishing this artifact provenance chain all the way down to the method invocation level on source code is you can actually perform some *remediating* action on the source code in a targeted way: first iterating over deployed assets

and mapping them to binaries, then mapping to a commit, then to the ASTs built from that commit.

Refactoring code starts at the root of the AST, which for Java is `J.CompilationUnit`. Call `refactor()` to begin a refactoring operation. We'll detail the kinds of refactoring operations that you can do in a moment, but at the end of this process you can call `fix()`, which generates a `Change` instance that allows you to generate git diffs and print out the original and transformed source. [Example 6-26](#) shows the whole process end-to-end.

Example 6-26. End to end: parsing Java source code to printing a fix

```
JavaParser parser = ...;
List<J.CompilationUnit> cus = parser.parse(sourceFiles);

for(J.CompilationUnit cu : cus) {
    Refactor<J.CompilationUnit, J> refactor = cu.refactor();

    // ... Do some refactoring

    Change<J.CompilationUnit> change = refactor.fix();

    change.diff(); // A string representing a git-style patch
    // Relativize the patch's file reference to some other path
    change.diff(relativeToPath);

    // Print out the transformed source, which could be used
    // to overwrite the original source file
    J.CompilationUnit fixed = change.getFixed();
    fixed.print();

    // Useful for unit tests to trim the output of common whitespace
    fixed.printTrimmed();

    // This is null when we synthesize a new compilation unit
    // where one didn't exist before
    @Nullable J.CompilationUnit original = change.getOriginal();
}
```

`rewrite-java` packs with a series of refactoring building blocks that can be used to perform low-level refactoring operations. For example, to change all fields from `java.util.List` to `java.util.Collection`, we could use the `ChangeFieldType` operation, as shown in test form in [Example 6-27](#).

Example 6-27. A unit test for changing a field type

```
@Test
fun changeFieldType() {
```

```

val a = parse"""
    import java.util.List;
    public class A {
        List collection;
    }
""".trimIndent()

val fixed = a.refactor()
    .visit(ChangeFieldType(
        a.classes[0].findFields("java.util.List")[0],
        "java.util.Collection"))
    .fix().fixed

assertRefactored(fixed, """
    import java.util.Collection;

    public class A {
        Collection collection;
    }
""")
}

```

The `rewrite-java` module comes with basic refactoring building blocks that resemble many of the individual refactoring tools you would find in an IDE:

- Add annotation to a class, method, or variable.
- Add a field to a class.
- Add/remove an import, which can be configured to expand/collapse star imports.
- Change field name (including its references, even across other source files that *use* this field, not just where the field is defined).
- Change a field type.
- Change a literal expression.
- Change a method name, including anywhere that method is referenced.
- Change a method target to a static from an instance method.
- Change a method target to an instance method from a static.
- Change a type reference anywhere it is found in the tree.
- Insert/delete method arguments.
- Delete any statement.
- Generate constructors using fields.
- Rename a variable.
- Reorder method arguments.

- Unwrap parentheses.
- Implement an interface.

Each one of these operations is defined as a `JavaRefactorVisitor`, which is an extension of `JavaSourceVisitor` designed for mutating the AST, ultimately leading to a `Change` object at the end of the refactoring operation.

Visitors can be cursored or not. Cursored visitors maintain a stack of AST elements that have been traversed in the tree thus far. In exchange for the extra memory footprint, such visitors can operate based on the location of AST elements in the tree. Many refactoring operations don't require this state. [Example 6-28](#) provides an example of a refactoring operation that makes each top-level class final. Since class declarations can be nested (e.g., inner classes), we use the cursor to determine if the class is top level or not. Refactoring operations should also be given a fully qualified name with a package representing the group of operations and a name signifying what it does.

Example 6-28. An example of a refactoring operation that makes each top-level class final

```
public class MakeClassesFinal extends JavaRefactorVisitor {
    public MakeClassesFinal {
        super("my.MakeClassesFinal");
        setCursorOn();
    }

    @Override
    public J visitClassDecl(J.ClassDecl classDecl) {
        J.ClassDecl c = refactor(classDecl, super::visitClassDecl);

        // Only make top-level classes final
        if(getCursor().firstEnclosing(J.ClassDecl.class) == null) {
            c = c.withModifiers("final");
        }

        return c;
    }
}
```

Visitors can be chained together by calling `andThen(anotherVisitor)`. This is useful for building up pipelines of refactoring operations made of lower-level components. For example, when `ChangeFieldType` finds a matching field that it is going to transform, it chains together an `AddImport` visitor to add the new import if necessary, and a `RemoveImport` to remove the old import if there are no longer any references to it.

A platform of open source out-of-the-box remediations continues to grow in open source.

At this point, the provenance chain is complete. Connecting the dots down to source code method-level detail allows you to observe your deployed footprint in a fine-grained way, answering a wide array of questions in real time.

Let's switch focus now to a different aspect of building reliability in your source: managing binary dependencies.

Dependency Management

Binary dependencies (those defined in Gradle build files or Maven POMs) present a series of systemic challenges. We'll go over several of these issues and present strategies for how to solve them. You'll notice that in each case the remediation is something that is applied at the build tool layer.

Some organizations attempt to limit the impact of dependency problems through *curation*, i.e., barring the use of dependencies from public artifact repository sources like Maven Central or JCenter in favor of an approved and curated set of dependencies. Curation of dependencies presents its own series of challenges. Specifically, given how interconnected libraries are, deciding to add another library to the curated set involves adding its entire transitive closure. There also is a natural tendency to avoid the toil necessary to get new artifacts added, which means your organization will skew toward slightly older versions of libraries, increasing the security vulnerability and bug footprint. Ironically, the goal of curation is usually to *improve* security. At the very least, this trade-off is worth evaluating against your stated goal.

Version Misalignments

Version misalignments caused by conflict resolution of dependency families like Jackson cause the family to not function correctly. Curated artifact repositories increase the likelihood of version misalignments.

Jackson is a great example of this. Suppose we brought a new version of Spring Boot and its transitive dependencies into our curated repository. [Figure 6-6](#) uses the Gradle `dependencyInsight` task to show the Jackson dependencies that are included in Spring Boot's transitive dependency closure and the paths by which they are included.

Notably absent from this list are all the other Jackson modules that aren't directly required by the framework, e.g., `jackson-module-kotlin`, `jackson-module-afterburner`, and `jackson-modules-java8`. Any microservice using one of these other modules that updates to the new version of Spring Boot included in the curated repository now has an unresolvable version misalignment (which may or may not

create runtime issues) until the new versions of those modules are also added to the curated set.

```
com.fasterxml.jackson.core:jackson-core:2.10.3
+--- com.fasterxml.jackson.core:jackson-databind:2.10.3
|   +--- org.springframework.boot:spring-boot-starter-json:2.2.6.RELEASE
|   |   \--- org.springframework.boot:spring-boot-starter-web:2.2.6.RELEASE
|   |       \--- compileClasspath [Requested org.springframework.boot:spring-boot-starter-web]
|   +--- com.fasterxml.jackson.datatype:jackson-datatype-jdk8:2.10.3
|   |   \--- org.springframework.boot:spring-boot-starter-json:2.2.6.RELEASE (*)
|   +--- com.fasterxml.jackson.datatype:jackson-datatype-jsr310:2.10.3
|   |   \--- org.springframework.boot:spring-boot-starter-json:2.2.6.RELEASE (*)
|   \--- com.fasterxml.jackson.module:jackson-module-parameter-names:2.10.3
|       \--- org.springframework.boot:spring-boot-starter-json:2.2.6.RELEASE (*)
+--- com.fasterxml.jackson.datatype:jackson-datatype-jdk8:2.10.3 (*)
+--- com.fasterxml.jackson.datatype:jackson-datatype-jsr310:2.10.3 (*)
\--- com.fasterxml.jackson.module:jackson-module-parameter-names:2.10.3 (*)
```

Figure 6-6. Spring Boot transitive Jackson dependencies

Dynamic versions create a different set of problems.

Dynamic Version Constraints

Java build tooling unfortunately still lacks advanced range selectors for semantic versioning like [NPM’s selector](#). Instead, we are left with coarser selectors like `latest.release` and `2.10.+` in Gradle and `RELEASE` and `(,2.11.0]` (range selection) in Maven.

Whenever possible, it is best to avoid `+` type selectors, because they *lexicographically* sort versions. So `2.10.9` is considered later than `2.10.10`.

The Maven-style range selector unfortunately pins you to an upper bound that isn’t static. When a further release comes out, the upper bounds need to be updated everywhere they are defined.

While for the most part, common open source libraries are cautious to only publish releases to public repositories, periodically we still see even heavily used libraries publish release candidates. Unfortunately, the `latest.release` (Gradle) and `RELEASE` (Maven) selectors don’t know how to distinguish between release version numbers and version numbers that to a human are clearly release candidates. For example, in March 2020, Jackson published a `2.11.0.rc1`, which would be selected by `latest.release`. Less than a year earlier, in September 2019, Jackson published a `2.10.0.pr1` version (the unconventional “pr” suffix apparently meaning “pre-release”). Neither one of these versions semantically matches the intent behind “latest release.”

We can bar known patterns for release candidates by adding two Maven repositories to the Gradle build that together form a disjoint subset of resolvable artifacts, in the case of [Example 6-29](#), the set of all nonrelease candidate Jackson modules, and everything other than Jackson modules.

Example 6-29. Barring the use of Jackson release candidates in Gradle

```
repositories {  
    mavenCentral {  
        content {  
            excludeVersionByRegex("com\\.fasterxml\\.jackson\\..*", ".*"  
                ".*rc.*")  
        }  
    }  
    mavenCentral {  
        content {  
            includeVersionByRegex("com\\.fasterxml\\.jackson\\..*", ".*"  
                "(\\d+\\.)*\\d+")  
        }  
    }  
}
```

Unused dependencies pose a different sort of problem.

Unused Dependencies

In addition to bloating the size of a packaged microservice (which is rarely a major issue), unused dependencies can result in implicit autoconfiguration of features, with serious consequences.

A Spring Data REST **vulnerability** caught by surprise many who weren't even using the library but, being present in the runtime classpath, caused Spring to autoconfigure a series of REST endpoints that exposed an attack vector.

Guice-based **Governator** autoconfigures any Guice module in the classpath. Governor's scanning mechanism is not constrained by package name. Modules located in the classpath could have dependencies on other modules, but these dependencies weren't necessarily reliable in the classpath. Frequently, unused but autoconfigured Guice modules caused application failure because dependencies that were previously accidentally on the classpath were removed.

Unused dependencies can be detected and removed automatically by the **Nebula Lint** Gradle plug-in. It can be configured in a Gradle project, as shown in [Example 6-30](#).

Example 6-30. Nebula Lint configuration for unused dependencies

```
plugins {  
    id "nebula.lint" version "LATEST"  
}  
  
gradleLint.rules = ['unused-dependency']
```

When `nebula.lint` is applied, build scripts will be automatically linted by a task called `lintGradle` after the last task in the task graph executes. Results are reported in the console, as shown in [Figure 6-7](#).

```
warning dependency-parentheses  parentheses are unnecessary for dependencies
build.gradle:22
compile('org.codenarc:CodeNarc:latest.release')

warning dependency-parentheses  parentheses are unnecessary for dependencies
build.gradle:23
testCompile('com.netflix.nebula:nebula-test:3.1.0')

* 2 problems (0 errors, 2 warnings)
```

Figure 6-7. Nebula Lint warning about dependency formatting

Run `./gradlew fixGradleLint` to automatically fix your build scripts. The autofix process lists all violations and how they were fixed, as shown in [Figure 6-8](#).

```
:fixGradleLint
fixed dependency-parentheses  parentheses are unnecessary for dependencies
build.gradle:22
compile('org.codenarc:CodeNarc:latest.release')
replaced with:
compile 'org.codenarc:CodeNarc:latest.release'

fixed dependency-tuple      use the shortcut form of the dependency
build.gradle:23
testCompile group: 'com.netflix.nebula', name: 'nebula-test',
version: '+'
replaced with:
testCompile 'com.netflix.nebula:nebula-test:+'

Corrected 2 lint problems

BUILD SUCCESSFUL
```

Figure 6-8. Nebula Lint automatically fixing dependency formatting

The last problem represents the mirror opposite of unused dependencies.

Undeclared Explicitly Used Dependencies

An application class imports a class from a dependency that is defined transitively. The first-order dependency that effectively brought the transitive dependency onto the classpath is either removed or its tree changes such that the transitive dependency is no longer on the classpath.

Undeclared dependencies can also be detected and added automatically by Nebula Lint. It can be configured in a Gradle project, as shown in [Example 6-31](#).

Example 6-31. Nebula Lint configuration for undeclared dependencies

```
plugins {  
    id "nebula.lint" version "LATEST"  
}  
  
gradleLint.rules = ['undeclared-dependency']
```

Once added as a first-order dependency, its visibility as a dependency whose version matters to this application is more important.

Summary

This chapter introduced some of the basic requirements to set up your software delivery life cycle in such a way that you can map deployed assets back to the source code that was included inside of them. With an increase in the number of deployed assets (smaller microservices), the existence of some queryable system of record to determine where particular code patterns are present in production-executable code becomes more important.

In the next chapter, we will discuss traffic management and call resiliency patterns that can be used to compensate for and limit the extent of failure, which will naturally be present in any microservice architecture.

Traffic Management

Cloud native applications expect failure and low availability from the other services and resources they interact with. In this chapter, we introduce important mitigation strategies involving load balancing (platform, gateway, and client-side) and call resilience patterns (retrying, rate limiters, bulkheads, and circuit breakers) that work together to ensure your microservices continue to perform.

These patterns won't be applicable for every organization. Often introducing more complex traffic management trades off operational complexity for more predictable user experience or a lower overall failure rate. In other words, it's easy to make a REST call to a downstream service with your HTTP client of choice; it's a little more complicated to wrap that call in a retry. And a little more complicated still to provide a circuit breaker and fallback. But with greater complexity comes greater reliability.

Organizations should evaluate their need here based on the types of applications they have (for example, where circuit breaking is applicable) and which application frameworks microservices are primarily written in. Java has first-class library support for these patterns and integration into popular frameworks like Spring, but the lack of support in some other languages would make it preferable to use sidecars or service meshes, even if there is some loss of flexibility as a result.

Microservices Offer More Potential Failure Points

As the number of microservices involved in a user interaction grows, the likelihood of encountering a service instance (in any given user interaction) that is in a low availability state increases. A service can put load on a downstream service that it cannot sustain and cause it to fail. Call resiliency patterns protect a service from failures in the downstream services, as well as negatively impact downstream services. They alter the call sequence with a goal of providing a reduced service to end users,

but a service nevertheless. For example, a personalized list of Netflix movie recommendations can be replaced with generic movie recommendations if the personalization service is suffering from low availability.

Microservices are usually deployed in a horizontally scaled way across different availability zones to increase resiliency of the distributed system. Microservices are not static. At any given time several of them can be released (new versions are being deployed or canaried), scaled, moved, or failed over. Some instances may experience failures, but not all. They may be temporarily down or experiencing reduced performance. This dynamic, frequently changed system requires adopting a set of practices for dynamically routing traffic: from discovering where the services are in the first place to picking which instance to send the traffic to. This is covered by different load-balancing approaches.

Two approaches exist for implementing these patterns: the application framework (code) and the supporting infrastructure (platform or gateway load balancers, service mesh). A combination can also be used. Generally, implementing these in application frameworks allows more flexibility and customization that's specialized to the business domain. For example, replacing personalized movie recommendations with generic ones is acceptable, but there is no obvious fallback response to a request to a payment or billing service—an understanding of the business domain matters.

Concurrency of Systems

By “concurrency” I am referring to the number of requests a microservice can service at once. There is a natural bound to concurrency in any system, usually driven by a resource like CPU or memory or the performance of a downstream service when requests are satisfied in a blocking manner. Any attempted requests exceeding this bound cannot be satisfied immediately and must be queued or rejected. In the case of a typical Java microservice running on Tomcat, the number of threads in Tomcat’s thread pool represents an upper bound on its concurrency limit (though system resources may very well be exhausted by a number of concurrent requests less than the Tomcat thread pool). The accept queue maintained by the operating system effectively queues up requests in excess of that concurrency limit.

Services fail when during prolonged periods of time the request rate exceeds the response rate. As the queue grows, so will the latency (since requests don’t even begin getting serviced until they are removed from the queue). Eventually queued requests will start timing out.

In this chapter, we will cover strategies to prevent a cascading failure from occurring because a concurrency limit has been reached. The discussion on load balancing, viewed from this perspective, is really a proactive approach to directing traffic in such a way as to prevent load-related failure in the first place.

Platform Load Balancing

Every modern runtime platform (e.g., IaaS offerings like AWS/GCP/Azure, a CaaS offering such as any Kubernetes distribution, or a PaaS offering like Cloud Foundry) has at least some basic cluster load balancer. These load balancers serve to distribute traffic across the instances in a cluster one way or another (often round-robin), but also have a wide range of other responsibilities. For example, AWS Elastic Load Balancers also serve the interests of TLS termination, content-based routing, sticky sessions, etc.

In on-premises environments, even simpler configurations are still prevalent with IIS, Nginx, Apache, etc., serving as statically configured load balancers in front of a fixed set of named virtual machines or physical machines.

Before discussing more complex options, it's worth noting that there is nothing wrong with this setup for a particular level of scale. One regional casualty/property insurer primarily serves a web application for its captive agents, so capacity requirements for this user pool is incredibly stable. While such an organization can benefit from an active-active deployment for greater resiliency to failure in an individual datacenter, its traffic pattern doesn't warrant the more complex load balancing at a gateway or on the client side.

Gateway Load Balancing

Software-based gateways are readily available in open source. [Spring Cloud Gateway](#) is a reasonably modern incarnation of such a gateway, influenced by experience learned from working with [Zuul](#).

The ability of a runtime platform to load balance traffic to optimize availability is limited. For some availability signals like latency, the caller is the best source of information. The load balancer and calling application are similarly positioned to observe and react to latency as an availability signal. But for other signals, especially those involving utilization, the server itself is the best (and often only) source of this information. Combining these two sources of availability signals yields the most effective load-balancing strategy.

From the perspective of reliability, the goal of load balancing is to direct traffic away from servers that have high error rates. The goal should *not* be to optimize for the fastest response time. Optimizing for response time tends to result in strategies that can *herd* traffic to a healthy instance or group of instances, causing them to become overloaded and unavailable. Avoiding instances with high error rates still allows traffic to be distributed to instances that are not optimally performant, but available enough. If all instances in a cluster are overloaded, choosing one instance over

another offers no benefit no matter how smart the load-balancing strategy. However, in many cases a subset of instances are overloaded because of temporary conditions.

A temporarily overloaded subset is found wherever there is a process whose execution is likely to be staggered across the cluster. For example, not all instances are likely to undergo GC or VM pauses, data updates, or cache swapping at the same time. This staggering tends to be present whenever there is no cluster-wide coordination of these processes. If all instances perform some sort of data update based on a synchronized clock, cluster coordination exists. For an example of a lack of coordination, consider what causes a GC pause to occur. Allocations incurred satisfying any given request eventually lead to a GC event. Since traffic will almost certainly be distributed nonuniformly across the cluster regardless of the load-balancing strategy, allocations will be staggered, leading to staggered GC events.

Another example of a subset of low-availability instances is the set of cold instances post-startup, such as instances brought into service by an autoscaling event or a zero-downtime deployment. With the rising popularity of serverless technologies, focus has been directed at application start time up to the point where health checks pass (effectively when the application instance is placed in service). But it's important to note a second phase of cold start ill performance that begins on the first request, as in [Figure 7-1](#), and ends when runtime optimizations have taken effect (i.e., the JVM's JIT optimization, or application-specific behaviors like memory mapping a working set of data into memory). It's this second phase that is so important to mitigate.

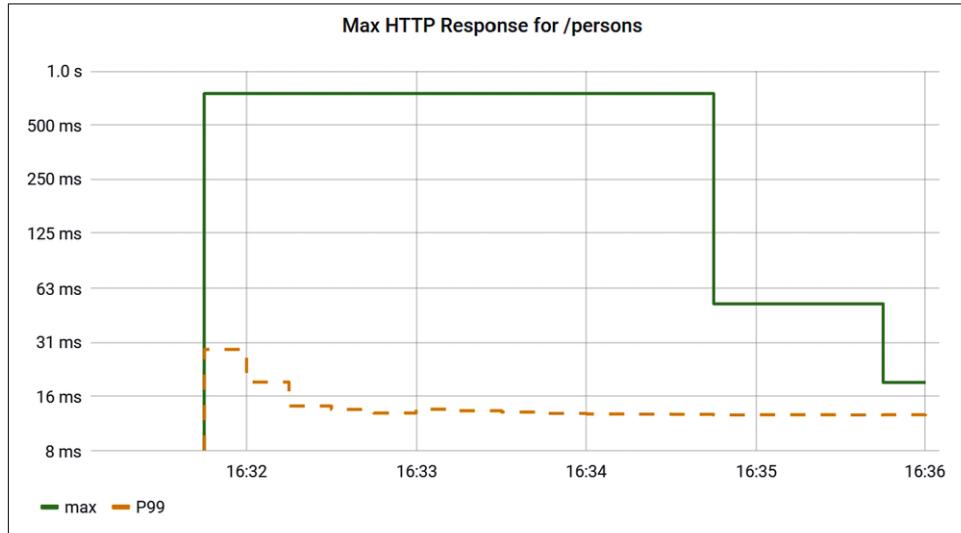


Figure 7-1. The max is more than an order of magnitude worse than P99 in the first few requests

The chart plots the two Prometheus queries shown in [Example 7-1](#).

Example 7-1. Prometheus queries plotting max and P99 latency for a REST endpoint /persons

```
http_server_requests_seconds_max{uri="/persons"}  
histogram_quantile(  
    0.99,  
    sum(  
        rate(  
            http_server_requests_seconds_bucket{uri="/persons"}[5m]  
        )  
    ) by (le)  
)
```

Some instances will run slower than others more or less permanently because of either bad underlying hardware or, increasingly, a noisy neighbor.

Clearly, round-robin load balancing can be improved upon. Architecturally, the logic for this load balancer resides in the edge gateway, as shown in [Figure 7-2](#).

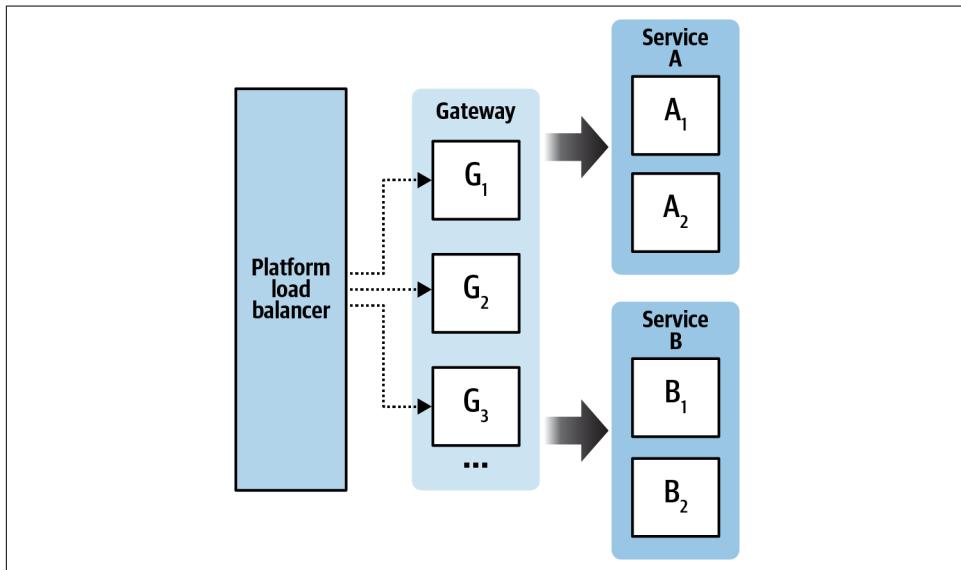


Figure 7-2. Using an API gateway as a smarter load balancer

User-facing traffic comes in through a platform load balancer, which distributes the traffic in a round-robin fashion to a cluster of gateway instances. In this case, we've shown one gateway serving requests to multiple microservices behind the edge. Gateway instances communicate directly with service instances by fetching an instance list from a discovery service like Netflix Eureka or HashiCorp Consul. There is no need

for a platform load balancer in front of individual microservices that are load balanced by the gateway.

With this general setup in mind, we can progressively come up with a load-balancing strategy that takes into account application instances' notion of their own availability. Then we'll consider its unintended side effects. The goal is for you to be exposed to techniques that can be used in combination with domain-specific knowledge to craft a load-balancing strategy that works well for you while learning to think through and anticipate side effects.

Join the Shortest Queue

Perhaps the simplest “adaptive” load balancer that goes beyond simple round-robin is “join the shortest queue.”

Join the shortest queue is implemented by comparing some instance availability signal visible to the load balancer. A good example of this is in-flight requests to each instance that the load balancer is aware of. Suppose the load balancer is directing traffic to three application instances, two of which have an in-flight request. When the load balancer receives a new request, it will direct the request to the one instance that has no in-flight requests, as shown in [Figure 7-3](#). This is computationally cheap (just minimizing/maximizing some statistic) and easy to implement.

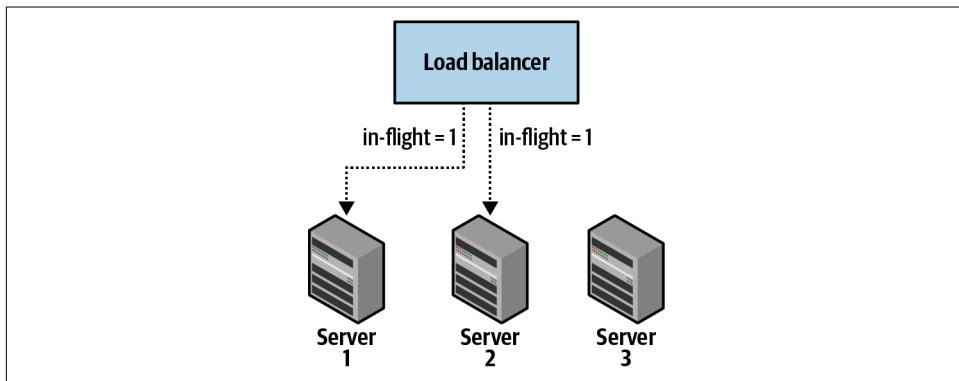


Figure 7-3. Join the shortest queue with one load balancer node

It starts to break down when there is more than one load balancer instance. To this point, the algorithm described makes decisions based on in-flight requests known to any one load balancer instance (those requests that passed through it). In other words, in a pool of load balancers, each load balancer is making its own independent decision, unaware of in-flight requests occurring on the others.

[Figure 7-4](#) shows how Load Balancer 1 will make a bad decision to send an incoming request to Server 3 on the basis of incomplete information it has about in-flight

requests managed by other load balancer nodes. The arrows indicate in-flight requests. So before a new request comes in, Load Balancer 1 has an in-flight request to Service 1 and 2. Load Balancer 2 has an in-flight request to Service 2 and 3. Load Balancer 3 has three in-flight requests to Service 3. As a new request comes in to Load Balancer 1, since it only knows about its own in-flight requests, it will decide to send the new request to Service 3, even though it is the busiest service instance, with four in-flight requests coming from the other load balancers in the cluster.

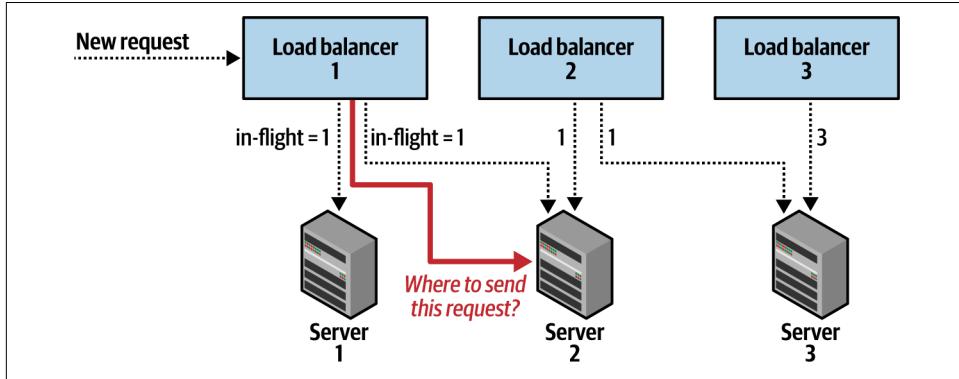


Figure 7-4. Join the shortest queue with several load balancer nodes

Join the shortest queue is an example of load balancing based on only the load balancer's view of the situation. One consequence of this for low-throughput applications is that a load balancer is managing only a few in-flight requests to a subset of the instances in a cluster. The choice of which instance in the cluster is least utilized can lead to a choice between two instances with zero in-flight requests, a random choice since no other information is available.



Avoid the Temptation to Coordinate!

It may be tempting to consider sharing the state of each load balancer's in-flight requests with other load balancers, but distributed coordination like this is difficult and should be avoided whenever possible. You wind up facing an engineering choice between rigging a peer-based distributed state system or choosing a shared datastore with the typical consistency, availability, and partitionability trade-offs.

The next pattern uses information from the instances being load balanced.

Instance-Reported Availability and Utilization

If instead we can inform each load balancer of the instance's perspective on its own availability and utilization, then two load balancers using the same instance have the same information regarding its availability. There are two available solutions:

Poll

Poll each instance's utilization, sampling data from health check endpoint detail.

Passively track

Passively track a header on the responses coming from the server annotated with current utilization data.

Both approaches are equally simple to implement, and each has trade-offs.

Micrometer has a `MeterRegistry` implementation called `HealthMeterRegistry` (available in the `io.micrometer:micrometer-registry-health` module) specifically to convert metrics data into availability signals that can be mapped to health indicators watched by load balancers.

A `HealthMeterRegistry` is configured with a set of service level objectives that are then mapped to framework health indicators and sampled each time the load balancer queries the health check endpoint.

Micrometer provides out-of-the-box service level objectives that are known to be applicable to a broad range of Java applications. These can be manually configured, as in [Example 7-2](#). Spring Boot Actuator also autoconfigures these objectives when `micrometer-registry-health` is present.

Example 7-2. Creating a `HealthMeterRegistry` with recommended service level objectives

```
HealthMeterRegistry registry = HealthMeterRegistry
    .builder(HealthConfig.DEFAULT)
    .serviceLevelObjectives(JvmServiceLevelObjectives.MEMORY)
    .serviceLevelObjectives(JvmServiceLevelObjectives.ALLOCATIONS)
    .serviceLevelObjectives(OperatingSystemServiceLevelObjectives.DISK)
    .build();
```

When this is bound to framework-level health indicators, these objectives are incorporated into the overall determination of an application's health. Spring Boot Actuator's health endpoint is shown configured with this default set of SLOs in [Figure 7-5](#).

```
    "status": "OUT_OF_SERVICE",
    "components": {
        "apiErrorRatio": {
            "status": "OUT_OF_SERVICE",
            "details": {
                "value": "20%",
                "mustBe": "<1%",
                "errorOutcome": "SERVER_ERROR",
                "uriMatches": "/api/**",
                "unit": "percent"
            }
        },
        "diskSpace": {
            "status": "UP",
            "details": {
                "total": 1000240963584,
                "free": 862786252800,
                "threshold": 10485760
            }
        },
        "jvmGcLoad": {
            "status": "UP",
            "details": {
                "value": "0.01%",
                "mustBe": "<50%",
                "unit": "percent CPU time spent"
            }
        },
        "jvmPoolMemory": {
            "status": "UP",
            "details": {
                "value": "0.08%",
                "mustBe": "<90%",
                "unit": "percent used"
            }
        },
        "jvmTotalMemory": {
            "status": "UP",
            "components": {
                "jvmGcOverhead": {
                    "status": "UP",
                    "details": {
                        "value": "0.01%",
                        "mustBe": "<20%",
                        "unit": "percent CPU time spent"
                    }
                },
                "jvmMemoryConsumption": {
                    "status": "UP",
                    "details": {
                        "value": "9.09%",
                        "mustBe": "<90%",
                        "unit": "maximum percent used in last 5 minutes"
                    }
                }
            }
        }
    }
}
```

Figure 7-5. Spring Boot Actuator health endpoint with service level objectives

You can define your own service level objectives as well, and in [Example 7-3](#) we define an `api.utilization` service level objective to support sampling utilization

data from health check endpoint detail on the server. Spring Boot Actuator adds this objective to the `HealthMeterRegistry` that it will automatically create; or if you are wiring your own `HealthMeterRegistry`, you can add it directly at construction time.

Example 7-3. A custom ServiceLevelObjective to report server utilization

```
@Configuration
class UtilizationServiceLevelObjective {
    @Bean
    ServiceLevelObjective apiUtilization() {
        return ServiceLevelObjective
            .build("api.utilization") ①
            .baseUnit("requests") ②
            .failedMessage("Rate limit to 10,000 requests/second.") ③
            .count(s -> s.name("http.server.requests")) ④
                .tag("uri", "/persons")
                .tag("outcome", "SUCCESS")
            )
            .isLessThan(10_000); ⑤
    }
}
```

- ① A name for the service level objective. This can be naming convention normalized just like a meter name when exposing it as the name of a health indicator component. Spring Boot would show this as a health component named `apiUtilization` (camel-cased) based on its convention.
- ② The unit of measure of utilization, which makes the output a little more human readable.
- ③ What it means for this objective to not be met, in plain language.
- ④ We are retrieving a measure of throughput (count) here. Also available are `value` to retrieve a gauge value, `total` to retrieve timer total time, distribution summary total amount, long task timer active tasks, and `percentile`.
- ⑤ A threshold that we are testing the measure against. When this service is receiving more than 10,000 requests/second, it reports itself as out of service to anything monitoring its health endpoint.

When this health indicator is being consumed by a gateway that can contain custom code to respond to different conditions, it's best to always report UP as the status for this health indicator. We could hardcode some fixed threshold in the application and report a different status like OVERLOADED when the utilization exceeds the threshold. Better would be to fetch the threshold from a dynamic configuration server such that the value can be changed on running instances in one stroke by changing the config

server. Best is to leave the determination of what utilization is too much to the load balancer, which could be folding this decision into more complex criteria.

Health Checks

Sometimes, though, our “gateway” is something like a platform load balancer (by platform load balancer I mean something like an AWS Application Load Balancer) that can respond to coarser measures of availability for its decisions. For example, many platform load balancers offer a means to configure a health check path and port. This could easily be configured to `/actuator/health`, but the platform load balancer will be responding only to whether the HTTP status of the response is successful. There isn’t enough configurability to peek at the utilization detail and make a decision relative to a threshold. In this case, it really is up to the application code to set a threshold and return `Health.up()` or `Health.outOfService()`. While there is nothing really inherently wrong with leaving this decision up to the app, it does require some a priori knowledge of performance at the time the app is being written, and of course is less flexible in the deployed environment. As an example of a platform load balancer that looks at health checks, DigitalOcean provides a “health check” configuration for Kubernetes load balancers, as shown in [Example 7-4](#). Health check configurations are also available for AWS Auto Scaling Groups and Google Cloud load balancers. Azure load balancers offer a similar configuration that is called a “health probe.”

Example 7-4. A Kubernetes load balancer configured to look at instance-reported utilization

```
metadata:  
  name: instance-reported-utilization  
  annotations:  
    service.beta.kubernetes.io/do-loadbalancer-healthcheck-port:80  
    service.beta.kubernetes.io/do-loadbalancer-healthcheck-protocol:http  
    service.beta.kubernetes.io/do-loadbalancer-healthcheck-path:/actuator/health  
    service.beta.kubernetes.io/do-loadbalancer-healthcheck-check-interval-seconds:3  
    service.beta.kubernetes.io/do-loadbalancer-healthcheck-response-timeout-seconds:5  
    service.beta.kubernetes.io/do-loadbalancer-healthcheck-unhealthy-threshold:3  
    service.beta.kubernetes.io/do-loadbalancer-healthcheck-healthy-threshold:5
```

When setting up a health indicator like this, the task is to find some key performance indicator that best summarizes the application’s availability. This performance indicator should monitor whatever the weak spot is in the application where an overabundance of traffic will eventually cause trouble. We are choosing to consider the `/persons` API endpoint the key performance indicator of utilization availability in this example. We could have selected more than one endpoint, multiple HTTP response outcomes, or any other combination of factors for HTTP endpoint throughput. Also, there are other measures of utilization that we could have used. If this was an

event-driven application, then the rate of messages consumed from a message queue or Kafka topic would be reasonable. If multiple execution paths in the application all led to an interaction with some utilization-constrained resources like a datasource or the file system, as in [Figure 7-6](#), then measuring the utilization on that resource would also seem reasonable.

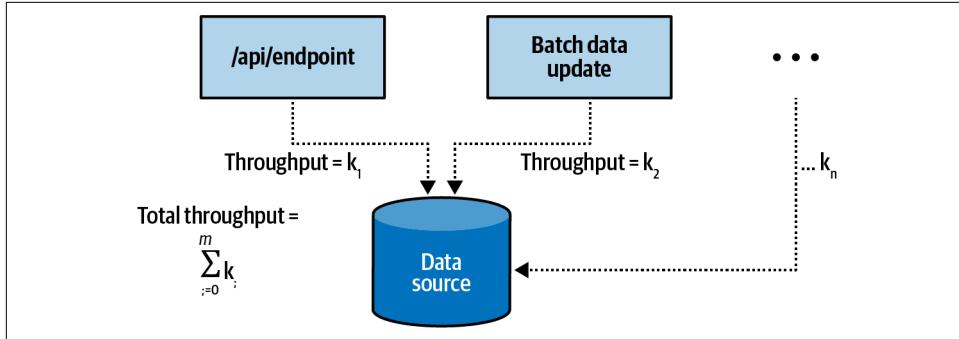


Figure 7-6. Measure throughput on a datasource when multiple execution paths lead it to be a bottleneck

This health indicator can be added to a new Spring Boot application generated from [start.spring.io](#) that includes a runtime dependency on `io.micrometer:micrometer-core` and the configuration found in [Example 7-5](#).

Example 7-5. Required application.yml configuration

```
management:
  endpoints.web.exposure.include: health
  endpoint.health.show-details: always
```

The response from `http://APP_HOST/actuator/health` includes the instance's view of its own utilization. Whether or not this utilization represents near full capacity is not important from the perspective of the “choice of two” algorithm. It is the algorithm’s choice to weight higher the lower of two such figures. It is only when the gateway/load balancer needs to prefilter the list of instances presented to the “choice of two” algorithm that it needs some domain-specific knowledge of a reasonable cutoff threshold to use, being at that point imbued with some understanding of whether a particular utilization level represents *too much* utilization or not.

By actively polling each instance, we are adding additional load on each instance proportional to the number of load balancer nodes. But for a service with low throughput, specifically when the utilization polling rate *exceeds* the request rate through a particular load balancer, polling provides a more accurate picture of utilization.

The passive strategy provides as up-to-date a view of utilization as the last request to arrive at a particular instance. The higher the throughput to an instance, the more accurate the utilization measure is.

We can use instance-reported utilization or health as an input to a more randomizing heuristic, as described next.

Choice of Two

“Choice of two” selects two servers randomly and selects one based on the maximization of some criteria.

Defining the criteria with multiple factors limits bias that could unintentionally lead to herding. For example, suppose one server is failing on every request and that (as is often the case) the failure mode is such that failed responses have a lower response time than a successful response. The instance’s utilization will appear lower. If utilization was the only factor used, then load balancers would start sending *more* requests to the unhealthy instance!

Compute an aggregate of these three factors and maximize on the aggregate for the choice of two selection:

Client health

A measure of connection-related errors for that instance

Server utilization

The most recent utilization measure provided by the instance

Client utilization

Count of in-flight requests to the instance from this load balancer

To make this even more robust, consider prefiltering the list of servers from which the two are chosen and compared. Make sure to bound the filtering in some way to avoid high CPU cost in searching for relatively healthy instances in a cluster with a large pool of instances that are unhealthy (e.g., by only attempting so many times to select a relatively healthy instance). By filtering, we can present the choice of two algorithm with a choice between relatively healthy instances even when some portion of the cluster is persistently unavailable.

We can add one last tweak to our selection algorithm to accommodate cold starts.

Instance Probation

To avoid overloading new instances while they are still undergoing their second-phase warmup, we can simply place a static limit on the number of requests that are allowed to go to that new instance. The probationary period ends when the load balancer receives one or more utilization responses from the new instance.

The concept of statically rate limiting new instances can be extended to include a gradually ramping-up rate limit based on the instance's age. Micrometer includes a `process.uptime` metric out of the box that can be used to calculate instance age.

Now that we have a toolbox of load-balancing strategies, let's think about some of the unintended side effects they can have.

Knock-On Effects of Smarter Load Balancing

The goal of developing the choice of two load balancers was to divert traffic away from instances suffering from availability problems. This has some interesting effects:

- When load balancing across two clusters in a blue/green deployment (or rolling blue/green), if one of the clusters has relatively worse performance, then it will receive less than an equal share of traffic.
- In an automated canary analysis setup, the baseline and canary may receive different proportions of traffic for the same reason.
- Anomaly detection may not pick up on outliers as quickly, as early signals of low reliability mean that fewer attempts are made against that instance.
- The request distribution will not be as uniform as a round-robin load balancer.

Availability signals are always decayed over time. In the instance-reported utilization examples ([Example 7-3](#)), this is why we used a rate-per-interval measure of utilization. As soon as the interval rolls over, a period of instability is no longer reflected in utilization data. The end result is if an instance recovers from a period of low availability, it can win a choice-of-two comparison again and receive traffic from the load balancer.

Not every microservice architecture is designed such that inter-microservice requests always pass through a gateway (nor should they be).

Client-Side Load Balancing

A third option is to implement a client-side load balancer. This leaves load-balancing decisions to the caller. Historically, client-side load balancing has been used for novel load-balancing strategies like cloud platform zone avoidance or zone affinity, preference for lowest weighted response times, etc. When these strategies work well in general, they tend to reemerge as features of platform load balancers.

[Figure 7-7](#) shows an interaction between Service A and Service B where Service A is using a client-side load balancer to distribute traffic to Service B. The client-side load balancer is part of Service A's application code. Typically, an instance list will be fetched from a discovery service like Eureka or Consul, and because the client-side load balancer directs traffic to Service B instances picked from the instance list

fetched from discovery, there is no need for a platform load balancer in front of Service B.

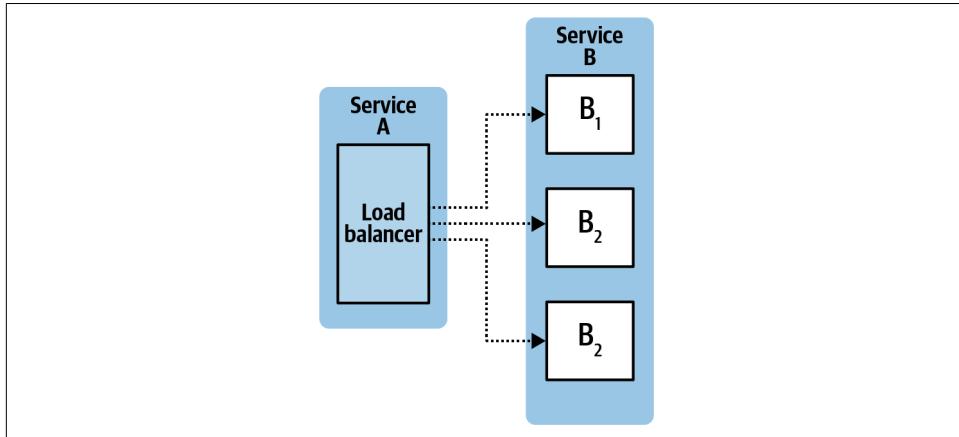


Figure 7-7. Client-side load balancing

Client-side load balancing can be used for different purposes. One of the original purposes was to dynamically source a list of server IPs or host names from a central service discovery mechanism like Eureka or Consul.



Why Service Discovery Instead of a Cloud Load Balancer?

When Netflix first developed Eureka, AWS VPC did not yet exist, and Elastic Load Balancers always had public, internet-facing host names. Not desiring to expose internal microservices to the public internet, Netflix built Eureka to achieve centrally what a private Application Load Balancer (ALBs being a replacement for what is now considered a legacy ELB construct in AWS) can achieve on a per-microservice basis. Perhaps if VPC was around when Netflix first migrated to AWS, Eureka would never have come about. Nevertheless, its use has extended beyond just load balancing to available instances in a cluster. [Table 5-1](#) showed how it is also used in blue/green deployments of event-driven microservices to take the instances in a disabled cluster out of service. Not every enterprise will take advantage of this kind of tooling, and if not, private cloud load balancers are probably simpler to manage.

Spring Cloud Commons has a client-side load-balancing abstraction that makes the configuration of these typical concerns fairly straightforward, as in [Example 7-6](#). Any use of the `WebClient` generated from such a configuration will cache the service listing for a period of time, preferring instances in the same zone, and using the configured `DiscoveryClient` to fetch the list of available names.

Example 7-6. Load balancing health checks

```
@Configuration
@LoadBalancerClient(
    name = "discovery-load-balancer",
    configuration = DiscoveryLoadBalancerConfiguration.class
)
class WebClientConfig {
    @LoadBalanced
    @Bean
    WebClient.Builder webClientBuilder() {
        return WebClient.builder();
    }
}

@Configuration
class DiscoveryLoadBalancerConfiguration {
    @Bean
    public ServiceInstanceListSupplier discoveryClientServiceInstances(
        ConfigurableApplicationContext context) {

        return ServiceInstanceListSuppliers.builder()
            .withDiscoveryClient()
            .withZonePreference()
            .withHealthChecks()
            .withCaching()
            .build(context);
    }
}
```

Not all load balancing is about server selection, however. There is one particular client-side load-balancing strategy used to cut off tail latencies above the 99th percentile.

Hedge Requests

Chapter 2 showed that for N requests, the chance that at least one of these requests is in the top 1% of the latency distribution is $(1 - 0.99^N)^* 100\%$. Furthermore, we saw how latency distributions are almost always multimodal, with the top 1% generally one to two orders of magnitude worse than the 99th percentile. For even 100 individual resource interactions, the chance of encountering one of these top 1% latencies is $(1 - 0.99^{100})^* 100\% = 63.3\%$.

One well-tested strategy to mitigate the effects of the top 1% latency when calling a downstream service or resource is to simply ship multiple requests downstream and accept whichever response comes back first, discarding the others.

This approach may seem surprising because obviously it increases the load on the downstream linearly according to the additional number of requests you include in

your hedge (and potentially this fans out more than linearly beyond the direct downstream, as it in turn makes requests to *its* downstreams and so on). In many enterprises, services experience a throughput that doesn't come close to their total capacity, and the most resilient services are scaled horizontally in some sort of active-active capacity to limit the impact of an outage in any one region. This has the effect of increasing capacity (generally unused) to improve resiliency. At its best, hedge requesting can serve to simply use this excess capacity while improving end-user response times by significantly reducing the frequency of the worst latencies.

Obviously, the decision to employ hedge requesting requires some domain-specific knowledge about the downstream service being called. It wouldn't be appropriate to ship three requests to a third-party payment system to charge a customer's credit card three times! For this reason, hedge requesting is typically performed in application code.



Hedge Requests Can't Be Implemented by Service Mesh Client-Side Load Balancers (!!)

Since domain-specific knowledge essentially requires that the load-balancing decision is made in the calling application, notice that shifting the responsibility for client-side load balancing to a service mesh is unworkable for hedge requesting. Given that hedge requesting is one of the simplest and most effective means of compensating for long-tail latencies above the 99th percentile, the inability of service mesh to replace application code for this purpose should be a trigger to consider whether the service mesh pattern is really appropriate more generally.

We now turn the discussion to patterns that compensate for failure in downstream microservices or lessen the likelihood of such services being overwhelmed in the first place.

Call Resiliency Patterns

Regardless of how well a load balancer makes a predictive decision about which instance *should* handle traffic best, any prediction is based on a projection of past performance. Past performance is never a guarantee of future results, so there still is a need for another level of resiliency to handle failure. Additionally, even a microservice cluster behind a load balancer that perfectly allocates traffic to the most available instances at any given time has a limit for what it can handle. The layers of a micro-service architecture need to be guarded against overloading that could lead to complete failure.

These mechanisms together form different basic “backpressure” schemes.

Backpressure is the signaling of failure from a serving system to the requesting system and how the requesting system handles those failures to prevent overloading itself and the serving system. Designing for backpressure means bounding resource use during times of overload and times of system failure. This is one of the basic building blocks of creating a robust distributed system. Implementations of backpressure usually involve either dropping new messages on the floor, or shipping errors back to users (and incrementing a metric in both cases) when a resource becomes limited or failures occur. Timeouts and exponential backoffs on connections and requests to other systems are also essential. Without backpressure mechanisms in place, cascading failure or unintentional message loss become likely. When a system is not able to handle the failures of another, it tends to emit failures to another system that depends on it.

—Jeff Hedges

The caller can combine four patterns to improve resiliency:

- Retries
- Rate limiters
- Bulkheads
- Circuit breakers

Retries are an obvious first step to overcoming intermittent failure, but we need to be cautious of creating “retry storms,” i.e., overwhelming parts of the system that are already under duress with retries when the original requests begin to fail. The other patterns will help compensate. Still, let’s start with retries.

Retries

Expect transient failure in downstream services, caused by temporarily full thread pools, slow network connections resulting in timeouts, or other temporary conditions that lead to unavailability. This class of faults typically self-correct after a short period of time. Callers should be prepared to handle transient failure by wrapping calls to downstreams in retry logic. Consider three factors when adding retries:

- Whether retries are appropriate. This often requires domain-specific knowledge of the called service. For example, should we retry a payment attempt on a downstream service that returned a timeout? Will the timed-out operation eventually be processed, making a retry a potential double charge?
- The maximum number of retry attempts, and the duration (including backoffs, as shown in [Example 7-7](#)) to use between attempts.
- Which responses (and exception types) warrant a retry. For example, if a downstream returns a 400 because the inputs are malformed for some reason, we cannot expect a different result by retrying the same inputs.

Example 7-7. Setting up an exponential backoff retry with Resilience4J

```
RetryConfig config = RetryConfig.custom()
    .intervalFunction(IntervalFunction.ofExponentialBackoff(
        Duration.ofSeconds(10), 3))
    .maxAttempts(3)
    .retryExceptions(RetryableApiException.class)
    .build();

RetryRegistry retryRegistry = RetryRegistry.of(config);

Retry retry = retryRegistry.retry("persons.api");

retry.executeCallable(() -> {
    Response response = ...
    switch(response.code()) {
        case 401:
            // Authentication flow
        case 502:
        case 503:
        case 504:
            throw new RetryableApiException();
    }

    return response;
});
});
```

Resilience4J has a built-in metric for retry logic, enabled by binding your retry to a Micrometer meter registry, as in [Example 7-8](#).

Example 7-8. Publishing metrics about retries via Micrometer

```
TaggedRetryMetrics
    .ofRetryRegistry(retryRegistry)
    .bindTo(meterRegistry);
```

This exports a single gauge `resilience4j.retry.calls` with a `kind` tag segregating successful (with and without retry) and failed (with and without retry) calls. If you were to set an alert, it would be on a fixed threshold of calls where `kind` equals `failed.with.retry`. In many cases, the code where the call is being made is itself going to be timed. For example, a REST endpoint that, when invoked, does some work, including making downstream service calls with retry logic, is itself going to be timed with `http.server.requests`, and you should already be alerting on failures of that endpoint.

Nevertheless, if your application contains a common component guarding access to a resource or downstream service with retry logic, then alerting on a high failure rate to

that resource can be a good signal that several pieces of your application will be failing.

Rate Limiters

The load on a microservice naturally varies over time based on user activity patterns, scheduled batch processes, etc. An atypical event could result in sudden and overwhelming bursts in activity. If increased load, maybe even for a particular business function served by a microservice, causes a strain on resources that could result in availability levels falling below an established SLO, rate limiting (also known as throttling) can keep the service up and serving requests, albeit at a defined rate of throughput.

Resilience4J implements the rate limiter pattern with several options. In [Example 7-9](#), a rate limiter is used in a microservice that needs to make calls against downstream billing history and payment services. A diagram of the service interaction is shown in [Figure 7-8](#).

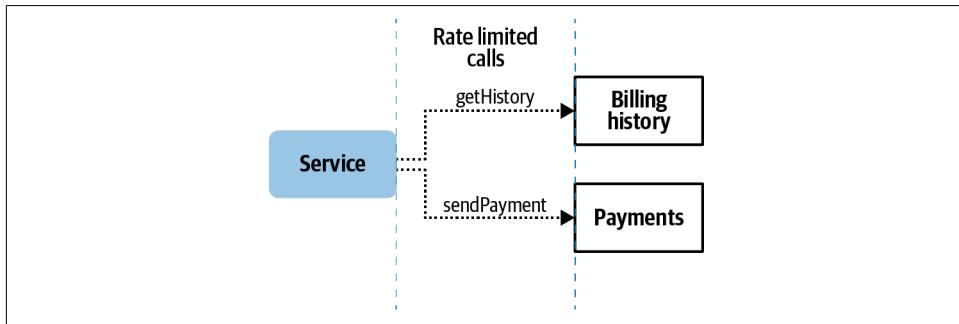


Figure 7-8. Call resiliency example service interaction

Example 7-9. Implementing a rate limiter with Resilience4J

```
RateLimiterConfig config = RateLimiterConfig.custom()
    .limitRefreshPeriod(Duration.ofMillis(1))
    .limitForPeriod(10) ①
    .timeoutDuration(Duration.ofMillis(25)) ②
    .build();

RateLimiterRegistry rateLimiterRegistry = RateLimiterRegistry.of(config);

RateLimiter billingHistoryRateLimiter = rateLimiterRegistry
    .rateLimiter("billingHistory");
RateLimiter paymentRateLimiter = rateLimiterRegistry
    .rateLimiter("payment", config); ③

// Components that would, as part of their implementations,
// execute HTTP requests to downstream services
```

```

BillingHistory billingHistory = ...
Payments payments = ...

// Spring WebFlux Functional route specification
RouterFunction<ServerResponse> route = route()
    .GET("/billing/{id}", accept(APPLICATION_JSON),
        RateLimiter.decorateFunction(
            billingHistoryRateLimiter,
            BillingHistory::getHistory
        )
    )
    .POST("/payment",
        RateLimiter.decorateFunction(
            paymentRateLimiter,
            Payments::sendPayment
        )
    )
    .build();

```

- ① Concurrency limit allowed by the rate limiter.
- ② Timeout for a blocked thread attempting to enter a saturated rate limiter.
- ③ A rate limiter for some service can be created with a configuration different from the global one (e.g., because this service is capable of a higher concurrency level than others).

Resilience4J has built-in metrics for rate limiters. Bind your rate limiter registry to a Micrometer meter registry, as in [Example 7-10](#).

Example 7-10. Publishing metrics about rate limiting via Micrometer

```

TaggedRateLimiterMetrics
    .ofRateLimiterRegistry(rateLimiterRegistry)
    .bindTo(meterRegistry);

```

The metrics in [Table 7-1](#) are then published.

The two metrics have different benefits. Available permissions is an interesting *predictive* indicator. If permissions are reaching zero or near zero (where maybe they previously did not), but waiting threads is low, then end-user experience has not yet been degraded. A high number of waiting threads is a more *reactive* measure that the downstream service may need to be scaled up because end-user experience is being degraded (if response time is important and there are often waiting threads).

Table 7-1. Rate limiter metrics exposed by Resilience4J

Metric name	Type	Description
resilience4j.ratelimiter.available.permissions	Gauge	The number of available permissions, or unused concurrency capacity
resilience4j.ratelimiter.waiting.threads	Gauge	The number of waiting threads

In Atlas, the alert condition for waiting threads tests against a fixed threshold, as shown in [Example 7-11](#).

Example 7-11. Atlas rate limiter alert threshold

```
name,resilience4j.ratelimiter.waiting.threads,:eq,  
$THRESHOLD,  
:gt
```

In Prometheus, the idea is similar, as shown in [Example 7-12](#).

Example 7-12. Prometheus rate limiter alert threshold

```
resilience4j_ratelimiter_waiting_threads > $THRESHOLD
```

Bulkheads

Microservices commonly execute requests on multiple downstream services. When a service suffers from low availability, it can cause dependent services to become unresponsive as well. This is particularly true when the dependent service is blocking and using a thread pool to make requests. For a microservice *A* with multiple downstream services, it could be that only a small portion of the traffic (requests of a certain type) cause a request to microservice *B*. If *A* is using a common thread pool to execute requests not only against *B* but against all of its other downstream services as well, then unavailability in *B* can gradually block requests, saturating threads in the common thread pool to the point where little or no work can happen.

The bulkhead pattern isolates downstream services from one another, specifying different concurrency limits for each downstream service. In this way, only requests that require a service call to *B* become unresponsive, and the rest of the *A* service continues to be responsive.

Resilience4J implements the bulkhead pattern with several options, shown in [Example 7-13](#).

Example 7-13. Implementing the bulkhead pattern with Resilience4J

```
BulkheadConfig config = BulkheadConfig.custom()
    .maxConcurrentCalls(150) ①
    .maxWaitDuration(Duration.ofMillis(500)) ②
    .build();

BulkheadRegistry registry = BulkheadRegistry.of(config);

Bulkhead billingHistoryBulkhead = registry.bulkhead("billingHistory");
Bulkhead paymentBulkhead = registry.bulkhead("payment", custom); ③

// Components that would, as part of their implementations, execute HTTP requests
// to downstream services
BillingHistory billingHistory = ...
Payments payments = ...

// Spring WebFlux Functional route specification
RouterFunction<ServerResponse> route = route()
    .GET("/billing/{id}", accept(APPLICATION_JSON),
        Bulkhead.decorateFunction(
            billingHistoryBulkhead,
            BillingHistory::getHistory
        )
    )
    .POST("/payment",
        Bulkhead.decorateFunction(
            paymentBulkhead,
            Payments::sendPayment
        )
    )
    .build();

```

- ① Concurrency limit allowed by the bulkhead.
- ② Timeout for a blocked thread attempting to enter a saturated bulkhead.
- ③ A bulkhead for some service can be created with a configuration different from the global one (e.g., because this service is capable of a higher concurrency level than others).

Resilience4J has built-in metrics for bulkheads. Bind the bulkhead registry to a Micrometer meter registry, as in [Example 7-14](#).

Example 7-14. Publishing metrics about bulkheads via Micrometer

```
TaggedBulkheadMetrics
    .ofBulkheadRegistry(bulkheadRegistry)
    .bindTo(meterRegistry);
```

Table 7-2 shows the bulkhead metrics shipped by Resilience4J. Alert when available concurrent calls frequently reach zero or near zero.

Table 7-2. Bulkhead metrics exposed by Resilience4J

Metric name	Type	Description
resilience4j.bulkhead.available.concurrent.calls	Gauge	The number of available permissions, or unused capacity
resilience4j.bulkhead.max.allowed.concurrent.calls	Gauge	The maximum number of available permissions

In Atlas, the alert condition for waiting threads tests against a fixed threshold, as shown in [Example 7-15](#). This might also be a good place to use a :roll-count to limit alert chattiness.

Example 7-15. Atlas bulkhead alert criteria

```
name,resilience4j.bulkhead.available.concurrent.calls,:eq,
$THRESHOLD,
:lt
```

In Prometheus, the idea is similar, as shown in [Example 7-16](#).

Example 7-16. Prometheus bulkhead alert criteria

```
resilience4j_bulkhead_available_concurrent_calls < $THRESHOLD
```

Circuit Breakers

Circuit breakers are a further extension of bulkheading with a twist. A circuit breaker maintains a finite state machine, as shown in [Figure 7-9](#), for an execution block that it guards with states of closed, half-open, and open. In the closed and half-open states, executions are allowed. In the open state, a fallback defined by the application is executed instead.

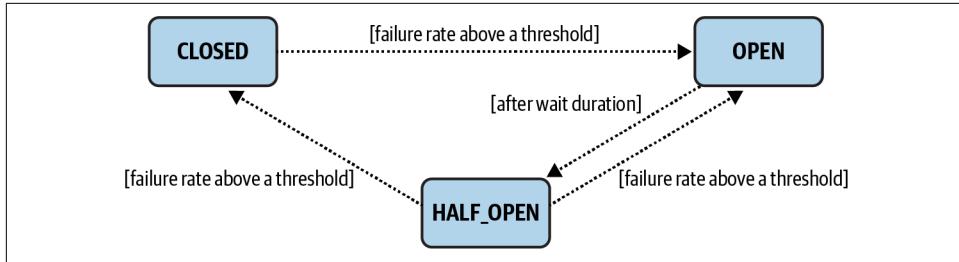


Figure 7-9. The states of a circuit breaker

One classic example of a circuit breaker is Netflix's list of movie recommendations. This is generally personalized to the subscriber based on past viewing history, etc. A circuit breaker guarding a call to the personalization service might respond with a generic list of content as a fallback when the circuit breaker is in an open state.

For some classes of business problems, a fallback that doesn't ultimately present the user with a failure is impossible. There is no reasonable fallback to accepting a payment from a user (assuming it couldn't be stored somewhere for later processing).

Successful and unsuccessful executions are maintained in a ring buffer. When the ring buffer initially fills, the failure ratio is tested against a preconfigured threshold. The state of the circuit breaker changes from closed to open when the failure rate is above a configurable threshold. When the circuit breaker is tripped and opens, it will stop allowing executions for a defined period of time, after which the circuit half-opens, permits a small amount of traffic through, and tests the failure ratio of that small amount of traffic against the threshold. If the failure ratio falls below the threshold, the circuit is closed again.

Netflix Hystrix was the first major open source circuit breaker library, and while still well known, it has now been deprecated. Resilience4J implements the circuit breaker pattern with improvements to library hygiene and support for more threading models. An example is shown in [Example 7-17](#).

Example 7-17. Implementing the circuit breaker pattern with Resilience4J

```

CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.custom()
    .failureRateThreshold(50)
    .waitDurationInOpenState(Duration.ofMillis(1000))
    .ringBufferSizeInHalfOpenState(2)
    .ringBufferSizeInClosedState(2)
    .build();

CircuitBreakerRegistry circuitBreakerRegistry = CircuitBreakerRegistry
    .of(circuitBreakerConfig);

CircuitBreaker billingHistoryCircuitBreaker = circuitBreakerRegistry

```

```

.circuitBreaker("billingHistoryCircuitBreaker");
CircuitBreaker paymentCircuitBreaker = circuitBreakerRegistry
    .circuitBreaker("payment", circuitBreakerConfig); ①

// Components that would, as part of their implementations, execute HTTP requests
// to downstream services
BillingHistory billingHistory = ...
Payments payments = ...

// Spring WebFlux Functional route specification
RouterFunction<ServerResponse> route = route()
    .GET("/billing/{id}", accept(APPLICATION_JSON),
        CircuitBreaker.decorateFunction(
            billingHistoryCircuitBreaker,
            BillingHistory::getHistory
        )
    )
    .POST("/payment",
        CircuitBreaker.decorateFunction(
            paymentCircuitBreaker,
            Payments::sendPayment
        )
    )
    .build();

```

- ① A circuit breaker for some service can be created with a configuration different from the global one.

Resilience4J contains built-in metrics instrumentation for circuit breakers that you should monitor for open circuits. Enable it by binding your circuit breaker registry to a Micrometer meter registry, as in [Example 7-18](#).

Example 7-18. Bind circuit breaker metrics

```

TaggedCircuitBreakerMetrics
    .ofCircuitBreakerRegistry(circuitBreakerRegistry)
    .bindTo(meterRegistry);

```

[Table 7-3](#) shows the two metrics exposed by Resilience4J for *each* circuit breaker.

Table 7-3. Circuit breaker metrics exposed by Resilience4J

Metric name	Type	Description
resilience4j.circuitbreaker.calls	Timer	Total number of successful and failed calls
resilience4j.circuitbreaker.state	Gauge	Set to 0 or 1 depending on whether the state described by the state tag is active (open, closed, etc.)
resilience4j.circuitbreaker.failure.rate	Gauge	The failure rate of the circuit breaker

Since these are gauges, you can alert on whether *any* circuit breaker is currently open by performing a `max` aggregation. At this point, end users are already experiencing a degraded experience by receiving a fallback response or having the failure propagate to them directly.

In Atlas, the alert condition checks for the open state, as shown in [Example 7-19](#).

Example 7-19. Atlas circuit breaker alert threshold

```
name,resilience4j.circuitbreaker.state,:eq,  
state,open,:eq,  
:and,  
:max, ❶  
1,  
:eq
```

- ❶ If any circuit breaker is open, this alert will match on it.

In Prometheus, the idea is similar, as shown in [Example 7-20](#). We can use `sum(..) > 0` or `max(..) == 1` with the same effect.

Example 7-20. Prometheus circuit breaker alert threshold

```
sum(resilience4j_circuitbreaker_state{state="open"}) > 0
```

It's probably OK if circuits briefly open and then close again though, so to limit alert chattiness, it may be better to instead set an error ratio indicator on `resilience4j.circuitbreaker.calls`, allowing for a certain number of failed requests going to the fallback before alerting.

Next we'll discuss how we can improve the flexibility of the alert thresholds themselves by responding to changing conditions in the code and environment.

Adaptive Concurrency Limits

Each of the call resiliency patterns presented so far (rate limiters, bulkheads, and circuit breakers) effectively serve to guard, either proactively or reactively, against load-related problems. They each have their own way of limiting concurrency.

In each case, the pattern was configured with a threshold value determined in advance of the microservice actually running in production. Rate limits are configured to constrain the number of requests that can be executed inside an interval, bulkheads limit instantaneous concurrency, and circuit breakers shed load away from instances that are experiencing failure (including load-related failure). These thresholds can be determined through careful performance testing, but their values tend to

diverge from the true limit over time as code changes, the size of downstream clusters and their availability changes, etc.

A common theme throughout this book is to replace fixed thresholds or manual judgments with adaptive judgments. We saw this in [Chapter 2](#) with thresholds set with forecasting algorithms, and in [Chapter 5](#) with automated canary analysis. It is possible to adopt a similar adaptive approach to concurrency limits.

Choosing the Right Call Resiliency Pattern

In code, the patterns for bulkheads, rate limiters, and circuit breakers look remarkably similar. In fact, the three patterns have overlapping responsibilities, as shown in [Figure 7-10](#).

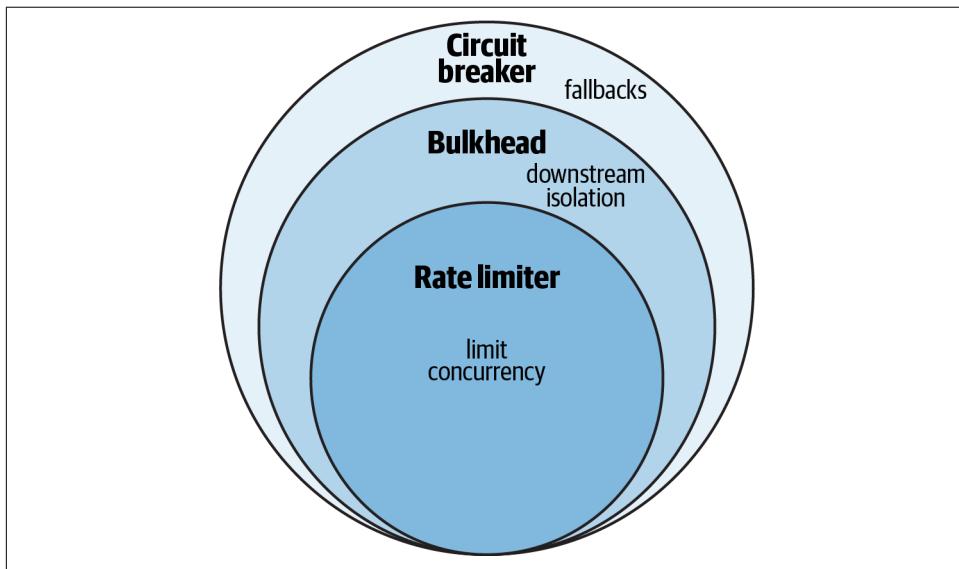


Figure 7-10. Overlapping responsibilities of three call resiliency patterns

All three patterns accomplish rate limiting, but with varying mechanisms, summarized in [Table 7-4](#).

Table 7-4. The rate-limiting mechanisms of different call resiliency patterns

Pattern	Limiting mechanism	Notes
Rate limiter	Limits rate per interval	This does not limit instantaneous concurrency (e.g., a spike of traffic inside the interval) unless that instantaneous concurrency exceeds the limit for the whole interval to be reached.
Bulkhead	Limits instantaneous concurrency level	Limits the concurrency level at the time that a new request is attempted. This does not limit the number of requests to the downstream inside an interval except indirectly since the downstream service responds in a nonzero amount of time.
Circuit breaker	Responds to errors (some of which occur due to the natural limit in the downstream's concurrency)	Either the RPC request guarded by the circuit breaker will time out or the downstream service (or its load balancer) will respond with a failure, such as an HTTP 502 (unavailable). This does not limit either the instantaneous or per-interval rate except indirectly when the downstream begins to be saturated.

For this reason, it isn't common to see a single block of code guarded by more than one of the patterns of rate limiter, bulkhead, or circuit breaker.

The implementations of the call resiliency patterns shown to this point have all employed Resilience4J, making them an application development concern. Let's compare keeping this as an application concern with externalizing the responsibility in service mesh.

Implementation in Service Mesh

Ultimately, the decision of whether to try to pry traffic management away from application concern has to be made in each organization based on the weight it places on several criteria shown in [Table 7-5](#). “Application responsibility” here means that the functionality is achieved through application code or autoconfigured via a binary dependency on a shared library. The weights assigned to each criterion across the header row are just an example and will vary from organization to organization. For example, an organization that has a large number of programming languages in use may assign a much higher weight to language support. This should drive your decision.

Table 7-5. The service mesh versus application responsibility decision matrix for traffic management (higher score = higher cost)

	Service mesh	Application responsibility
Language support = 5	Low: only a thin client needed to connect to mesh ($1 \times 5 = 5$)	High: distinct implementation required for each language ($5 \times 5 = 25$)
Runtime support = 5	High: for example, Istio is a Kubernetes CRD, so bound to a specific runtime ($5 \times 5 = 25$)	Low: only has an impact if the library wants to take advantage of some specific feature of the runtime ($1 \times 5 = 5$)
Deployment complexity = 4	Medium: requires changes to deployment practices ($3 \times 4 = 12$)	Very low: doesn't alter deployment at all ($0 \times 4 = 0$)
Anti-flexibility = 3	Medium: as patterns become known, they are generalized in the mesh, but not immediately ($3 \times 3 = 9$)	Medium: introducing new patterns requires dependency updates across the stack ($4 \times 3 = 12$)
Operational cost = 2	High: often much higher resource consumption ($x 2 = 10$) and operational experience upgrading the mesh independent of application footprint	Low: no additional processes or containers allocated per application ($1 \times 2 = 2$)
Total cost	$5 + 25 + 12 + 9 + 10 = 61$	$25 + 5 + 0 + 12 + 2 = 44$ (best option with these weights)

The choice-of-two load balancer described earlier is an example of a sophisticated load balancer that requires coordination with application code, so it could never be fully encapsulated by a service mesh technology.

There is one other complication of this lack of coordination with application code. Consider a task like **request timeouts**, handled by a sidecar proxy. While the proxy may hang up on a caller after the configured timeout, the application instance is still at work handling the request all the way to completion. If the application is using a conventional blocking thread-pool model like Tomcat, a thread continues to be consumed after the timeout.

Currently, Istio only supports a “circuit breaker” that more closely resembles a bulkhead as we’ve defined it, since it supports limiting instantaneous concurrency to a service by controlling maximum connections or requests. Rather than this being an application concern, with Istio the bulkhead would be applied with YAML configuration from the Istio Kubernetes custom resource definition, as in [Example 7-21](#).

Example 7-21. Istio circuit breaker

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: billingHistory
spec:
  host: billingHistory
  subsets:
```

```
- name: v1
  labels:
    version: v1
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 150
```

This also demonstrates the anti-flexibility of service mesh. The sophistication of your traffic management policy will be limited to what can be expressed in YAML. It's important to consider how this limited expressiveness is a *necessary* condition of not being application code.

If, for example, Istio CRD YAML follows the typical evolution of markup that stretches to meet more diverse expectations, we would expect to see the imposition of boolean logic (appearing already in [multi-match](#)) for assembling more complex rules together, etc. It feels like looping is inevitable.

Again, trends in software engineering are often cyclic. This desire to simplify application development through static configuration or markup has happened before with interesting consequences. Remember, way back in [“Configuration as Code” on page 17](#), the example of how XSLT wandered gradually into becoming a Turing complete language. This is significant, because the moment this happens, it becomes impossible to verify all sorts of characteristics of the configuration (now full-blown software) with static analysis. At that point, we're far from the original stated goal of keeping functionality out of code.

Implementation in RSocket

Reactive Streams provides a standard for asynchronous stream processing with non-blocking backpressure. [RSocket](#) is a persistent bidirectional remote procedure call protocol implementing Reactive Streams semantics. The goal of backpressure was described in 2014 in the [Reactive Manifesto](#):

When one component is struggling to keep up, the system as a whole needs to respond in a sensible way. It is unacceptable for the component under stress to fail catastrophically or to drop messages in an uncontrolled fashion. Since it can't cope and it can't fail, it should communicate the fact that it is under stress to upstream components and so get them to reduce the load. This backpressure is an important feedback mechanism that allows systems to gracefully respond to load rather than collapse under it. The backpressure may cascade all the way up to the user, at which point responsiveness may degrade, but this mechanism will ensure that the system is resilient under load, and will provide information that may allow the system itself to apply other resources to help distribute the load...

—Reactive Manifesto

The concept of backpressure across the network layer may very well eliminate the need for rate limiters, bulkheads, and circuit breakers in application code (or in a sidecar process). As an application instance observes its own decline in availability, it places backpressure on callers. Effectively, callers cannot make a call to an unavailable application instance.

Expect to see the further evolution of infrastructure, extending backpressure up and down the application stack. [R2DBC](#) has extended backpressure down even to database interactions. [Netifi](#) has built an entire control plane around this concept, a sort of alternative to service mesh without many of the disadvantages.

Summary

Failure and degradation of performance should be expected and planned for in any production microservice architecture. In this chapter, we introduced a number of strategies for dealing with these conditions, from load balancing to call resilience patterns.

Your organizational commitment to these patterns is almost entirely in application code. As with other crosscutting concerns that impact application code like metrics instrumentation and distributed tracing, there is an opportunity for an effective platform engineering team to step in and provide some of this cross-organizationally by shipping good default opinions in core libraries and configuration that are consumed by all of the organization's microservices.

This book has described a journey toward more reliable systems. Go as far as you can on this journey, recognizing that at each step your business is better off. It starts with simply measuring the existing state of the system, building a greater degree of awareness about what your end users are experiencing day to day. Continue by adding debuggability signals that allow you to ask questions about why failure is occurring as you become aware of it. Improve your software delivery pipeline to limit the chances that you introduce more failure into the system as you continue to build out your software. Build the capability to observe the state of the deployed assets themselves so that you can begin to reason about how to make change cross-organizationally when needed. These compensations for expected failures are the last step in the journey toward building more reliable distributed systems.

At each step, build guardrails instead of gates!

Index

Symbols

. (dot) character, 32

A

abstract syntax tree (AST), 243-252
accept/deny meters, 88-89
adaptive concurrency limits, 283
agent tracing, 110
aggregable average, 46
aggregation, defined, 102
alerting (see charting and alerting)
anomaly detection, impact of sampling on, 116
anti-correlation of samples, 61
application metrics, 23-99
 black box versus white box monitoring, 24-25
 choosing the right meter type, 77
 classes of meters, 38
 controlling cost, 77-80
 coordinated omission, 80-82
 counters, 42-45
 creating meters, 30
 dimensional metrics, 25
 distribution summaries, 73
 gauges, 39-42
 hierarchical metrics, 26
 load testing, 82-86
 long task timers, 74-77
 meter binders, 98
 meter filters, 87-92
 Micrometer meter registries, 27-99
 naming metrics, 31-38
 partitioning metrics by monitoring system, 96-98

separating platform metrics from, 92-96
timers, 45-73
application platform, 1-21
 chaos engineering and continuous verification, 17
 configuration as code, 17
 delivery, 13
 encapsulating capabilities, 18
 monitoring, 7-13
 platform engineering culture, 2-6
 service mesh, 19-21
 testing automation, 15-17
 traffic management, 15
application, defined, 190
Archaius, 17
AST (abstract syntax tree), 243-252
automated canary analysis, 205-218
automated testing, 15-17
availability
 instance-reported, 264-267
 monitoring for, 7-10
average latency, 60
average, median versus, 60

B

backpressure, 274, 287
baking, 196, 199
base unit of time, 53
batch tasks, 175
beeswarm plot, 213
binary dependency management, 252-256
black box monitoring, 24-25
blended tracing, 112
blue/green deployment, 200-205

boundary sampling, 116

buckets, 65

bulkheads, 278-280, 285

C

CaaS (container as a service), 110, 188

call resiliency patterns, 273-288

adaptive concurrency limits, 283

bulkheads, 278-280

choosing the right pattern, 284

circuit breakers, 280-283

implementation in RSocket, 287

implementation in service mesh, 285

rate limiters, 276-278

retries, 274-276

canary releases

automated canary analysis, 205-218

CPU utilization, 218

error ratio, 216

general-purpose canary metrics for every

microservice, 214-218

heap saturation, 217

latency, 215

Spinnaker with Kayenta, 209-214

cardinality, of metric, 77-79

CD (see continuous delivery)

chaos engineering, 17, 121-123

charting and alerting, 125-185

building alerts using forecasting methods, 176-185

counters, 139-142

differences in monitoring systems, 127-131

effective visualizations of service level indicators, 132-137

gauges, 137-139

latency, 153-161

Prometheus rate interval selection, 137

service level indicators for every Java micro-service, 148-176

timers, 143-146

top k visualizations, 135

when to stop creating dashboards, 147

“choice of two” load balancing, 269

CI (continuous integration), 13

circuit breakers, 280-283, 285

classes, of meters, 38

client (outbound) requests, 156-161

client-side load balancing, 270-272

cloud load balancer, service discovery versus,

271

cloud, packaging for, 194-199

cluster, defined, 190

concurrency

adaptive concurrency limits, 283

traffic management and, 258

configuration as code, 17

configuring distribution statistics, 91

container as a service (CaaS), 110, 188

container schedulers, packaging for, 198

continuous delivery (CD), 187-219

automated canary analysis, 205-218

blue/green deployment, 200-205

continuous deployment versus, 14

continuous integration versus, 13

delete + none deployment, 199

delivery pipelines, 191-194

Highlander deployment, 200

multicloud, 187-219

packaging for the cloud, 194-199

platform types, 188

resource types, 189-191

continuous integration (CI), 13

continuous verification, 17

Conway’s Law, 2, 92

coordinated omission, 80-82

cost, of metrics

controlling, 77-80

probabilistic samplers, 116

count, aggregable average and, 46

counters, 42-45, 139-142

rate of occurrences as measure of throughput, 44

timers versus, 45

CPU utilization, 170-172, 218

cross-functional teams, 3

culture, platform engineering, 2-6

cumulative histograms, 65

curation of dependencies, 252

D

dashboards, when to stop creating, 147

debugging

monitoring as tool for, 10

with observability (see observability)

delete + none deployment, 199

delivery pipelines, 13

canary configuration and, 211

continuous delivery and, 191-194
using universal scalability law in, 182
deny/accept meters, 88-89
dependency management
 capturing resolved dependencies in meta-data, 234-240
 dynamic version constraints, 253
 source code and, 252-256
 undeclared explicitly used dependencies, 255
 unused dependencies, 254
 version misalignments, 252
deployed asset inventory, 218, 223
dimensional metrics, 25
distributed tracing, 103
 agent tracing, 110
 blended tracing, 112
 components of a distributed trace, 107-108
 framework tracing, 110
 instrumentation types, 109-114
 manual tracing, 109
 metrics systems versus, 127
 monoliths and, 117
 service mesh tracing, 111
 using trace context for failure injection and experimentation, 120-123
distribution statistics, configuring, 91
distribution summaries, 56, 73
Docker registries, 227
dot(.) character, 32
dynamic configuration, 17
dynamic version constraints, 253

E

Eden space, 164
encapsulation, 18, 19-21
error ratio
 as canary metric, 216
 error rate versus, 153
errors
 display of errors versus successes, 134-135
 SLIs and, 148-153

F

failure
 learning to expect, 12
 potential failure points from microservices, 257
failure injection testing (FIT), 121-123

file descriptors, 172-174
filters, 87-92
firewall, defined, 190
FIT (failure injection testing), 121-123
forecasting methods
 building alerts using, 176-185
 naive method, 177
 single-exponential smoothing, 179
 universal scalability law, 181-185
framework tracing, 110

G

garbage collection (GC)
 low pool memory after collection, 167
 low total memory, 168-169
 max pause time, 161
 pause times, 161-164
 presence of any humongous allocation, 164
 proportion of time spent in, 162-164
 sum of sums and, 53
gateway load balancing, 259-270
 choice of two, 269
 health checks, 267-269
 instance probation, 269
 instance-reported availability/utilization, 264-267
join the shortest queue, 262-263
knock-on effects of smarter load balancing, 270
gauges, 39-42, 137-139
GC (see garbage collection)
GitOps, 223
Gradle (see source code observability)
Grafana, 132-137
 (see also charting and alerting)
 display of errors versus successes, 134-135
 effective visualizations of service level indicators, 132-137
 top k visualizations, 135
guardrails-not-gates approach, 2, 5, 18

H

health checks, 267-269
HealthMeterRegistry, 264-267
heap saturation, 217
heap utilization, 164-169
 as canary metric, 217
 low pool memory after collection, 167
 low total memory, 168-169

- rolling count occurrences of heap space filling up, 166
heatmaps, 119-120
hedge requests, 272
hierarchical metrics, 26
Highlander deployment strategy, 200
histograms, 65-68
holes
 probabilistic samplers and, 116
 rate-based samplers and, 115
http.client.requests, 156-161
http.server.requests, 153-156
- I**
- inbound (server) requests, 153-156
information hiding, 18
infrastructure as a service (IaaS), 188
 agent tracing, 110
 packaging for, 196
instance probation, 269
instance, defined, 189
instance-reported availability/utilization, 264-267
interval, sum of sums over, 53
Istio, 19
- J**
- join the shortest queue load balancer, 262-263
- K**
- Kayenta, Spinnaker with, 209-214
kubectl apply, 201
Kubernetes, Spinnaker's implementation of, 190
- L**
- latency
 average, 60
 as canary metric, 215
 client (outbound) requests, 156-161
 defined, 8
 hedge requests to mitigate, 272
 measuring (see timers)
 server (inbound) requests, 153-156
 SLIs and, 153-161
 timers and, 143-146
latency distributions, common features of, 59-60
library code, adding timers to, 63
- line width, styles for, 132
Little's Law, 182
load balancing
 client-side, 270-272
 defined, 190
 gateway, 259-270
 join the shortest queue load balancer, 262-263
 platform, 259
load testing, 82-86
logging, tracing versus, 104-107
logs, 102
long task timers, 74-77, 175
long-running tasks, 175
- M**
- machine learning, 176
Mann-Whitney U test, 209
manual tracing, 109
Maven repositories, 227-230
maximum latency, 143-146
maximum, as decaying signal not aligned to push interval, 50-53
median, average versus, 60
metadata, capturing resolved dependencies in, 234-240
meter filters, 87-92
 configuring distribution statistics, 91
 deny/accept meters, 88-89
 SLO boundaries and, 69-73
 transforming metrics, 89
MeterBinder interface, 98
MeterRegistry, 28-30
 gauge creation, 40
 Spring Boot autoconfiguration, 30
 timer creation, 55
metrics, 7
 (see also application metrics)
 general-purpose canary metrics for every microservice, 214-218
 observability and, 7, 104
 tracing versus, 104-107
metrics systems, distributed tracing versus, 127
Micrometer, 27-30
 (see also application metrics)
 meter registries, 27-30
 timers, 143-146
microservices, defined, 1
monitoring, 7-13

(see also charting and alerting)
for availability, 7-10
batch or other long-running tasks, 175
black box versus white box, 24-25
building trust with effective monitoring, 13
as debugging tool, 10
differences in monitoring systems, 127-131
Google approach to SLOs, 9
learning to expect failure, 12
partitioning metrics by monitoring system,
96-98
monolithic applications
distributed tracing and, 117
software release process for, 1
MultiGauge, 41

N

naive forecasting method, 177
naming conventions
common tags, 36-38
metrics and, 31-38
Netflix
and service level objectives, 10
freedom and responsibility culture, 4
platform engineering culture, 2-6
Netflix Archaius, 17
Netflix Nebula, 230-234
nonblocking load test, 82-86
normal histograms, 65

O

object-oriented programming, 18
observability, 101-124
components of a distributed trace, 107-108
correlation of telemetry, 118-120
debugging with, 101-124
determining the appropriate telemetry,
104-107
distributed tracing, 103
distributed tracing and monoliths, 117
logs, 102
metrics and, 7, 104
sampling, 114-117
source code (see source code observability)
three pillars of, 101-107
tool selection, 104-107
using trace context for failure injection and
experimentation, 120-123
old generation, 165

OpenRewrite, 243
(see also Rewrite)
OpenTelemetry, 129-131
outbound (client) requests, 156-161
outcome tags, 149

P

PaaS (platform as a service), 110, 188
packaging
for container schedulers, 198
for IaaS platforms, 196
for the cloud, 194-199
Panera Bread, Inc., 149
percentiles, 60-64
pipelines (see delivery pipelines)
platform (see application platform)
platform as a service (PaaS), 110, 188
platform engineers, monitoring goals of, 93
platform load balancing, 259
platform metrics, 92-96
probabilistic samplers, 115
probation (instance probation), 269
product engineers, monitoring goals of, 93
Prometheus
rate interval selection, 137
SLO boundaries, 71

Q

quantiles, 60-64

R

rate limiters, 276-278, 285
rate-limiting samplers, 114
Reactive Manifesto, 287
Reactive Streams, 287
red/black deployment, 205
(see also blue/green deployment)
refactoring, Java source, 248-252
release repositories, 227
release versioning, 226-234
build tools for, 230-234
Maven repositories, 227-230
releasing SaaS versus packaged software,
232
retries, 274-276
Rewrite
creating a Rewrite AST from Java source
code, 244-246

performing a search with, 246-248
refactoring Java source, 248-252
structured code search with OpenRewrite, 243
ring buffer, 25, 45-53
rollbacks (see blue/green deployment)
rolling count function, 166
RSocket, 287

S

SaaS (software as a service), 232
sampling, 114-117
 boundary sampling, 116
 defined, 102
 impact on anomaly detection, 116
 no sampling, 114
 probabilistic samplers, 115
 rate-limiting samplers, 114
 time correlation of samples, 61
saturation
 defined, 8
 utilization versus, 9
search, with Rewrite, 246-248
security issues
 suspicious traffic, 174
 zero-day exploits, 241
server (inbound) requests, 153-156
server group, defined, 189
service discovery, cloud load balancer versus, 271
service level agreements (SLAs), 7
service level indicators (SLIs), 7
 batch or other long-running tasks, 175
 CPU utilization, 170-172
 effective visualizations of, 132-137
 errors, 148-153
 for every Java microservice, 148-176
 file descriptors, 172-174
 garbage collection pause times, 161-164
 heap utilization, 164-169
 latency, 153-161
 suspicious traffic, 174
service level objectives (SLOs), 7
 Google approach to, 9
 Netflix approach to, 10
 timers and, 69-73
service mesh, 19-21
 call resiliency pattern implementation in, 285

 hedge requests and, 273
 instrumentation as black box, 24
service mesh tracing, 111
service recovery paradox, 12
shading, styles for, 132
sidecars, 19-21, 111
single-exponential smoothing, 179
SLAs (see service level agreements)
SLIs (see service level indicators)
smoothing, single-exponential, 179
snapshot repositories, 228
software as a service (SaaS), 232
software delivery pipeline, 13
source code observability, 221-256
 capturing method-level utilization of source code, 240-252
 capturing resolved dependencies in metadata, 234-240
dependency management, 252-256
release versioning, 226-234
stateful asset inventory, 223-226
structured code search with OpenRewrite, 243

Spinnaker

 continuous delivery with, 187-219
 with Kayenta, 209-214
 origins, 187
Spring Boot
 client (outbound) requests, 156-161
 MeterRegistry autoconfiguration, 30
 server (inbound) requests, 153-156
Spring Cloud Sleuth, 114
standard deviation, 60
stateful asset inventory, 223-226
status tags, 149
sum of sums, 53
sum, aggregable average and, 46
survivor space, 165
suspicious traffic, 174

T

tag values, 35, 71
tags, common, 36-38
telemetry
 correlation of observability to, 118-120
 determining appropriate, 104-107
 metric to trace correlation, 119-120
three pillars of observability, 101-107

- determining the appropriate telemetry, 104-107
logs, 102
metrics, 104
throughput
 count statistic as measure of, 46
 rate of occurrences as measure of, 44
time correlation of samples, 61
TimeGauge, 41
timers, 45-73, 143-146
 adding to library code, 63
 aggregateable average from count and sum
 together, 46
 base unit of time, 53
 common features of latency distributions, 59-60
 count statistic as measure of throughput, 46
 counters versus, 45
 distribution summaries versus, 56
 histograms, 65-68
 long task timers, 74-77
 maximum as decaying signal not aligned to
 push interval, 50-53
 maximum latency, 143-146
 median versus average, 60
 percentiles/quantiles, 60-64
 SLO boundaries and, 69-73
 sum of sum over an interval, 53
 time correlation of samples, 61
 using, 55-58
top k visualizations, 135
tracing
 distributed (see distributed tracing)
 logging or metrics versus, 104-107
 metric to trace correlation, 119-120
 of calls to subsystem, 113
tracing telemetry, 103
traffic management, 15, 257-288
 call resiliency patterns, 273-288
 client-side load balancing, 270-272
 concurrency of systems, 258
 gateway load balancing, 259-270
 hedge requests, 272
 platform load balancing, 259
 potential failure points from microservices, 257
 suspicious traffic, 174
transforming metrics, 89
trust, effective monitoring and, 13
Tufte, Edward, 132
- U**
- U test, 209
undeclared dependencies, 255
universal scalability law (USL), 181-185
unused dependencies, 254
utilization
 defined, 8
 saturation versus, 9
- V**
- version misalignments, 252
versioning (see release versioning)
Visual Display of Quantitative Information, The (Tufte), 132
visualizations
 counters, 139-142
 display of errors versus successes, 134-135
 gauges, 137-139
 Prometheus rate interval selection, 137
 service level indicators, 132-137
 styles for line width/shading, 132
 timers, 143-146
 top k, 135
 when to stop creating dashboards, 147
- W**
- white box monitoring
 black box monitoring versus, 24-25
 monoliths and, 118
- Y**
- young generation (Eden space), 164
- Z**
- zero-day exploits, 241
Zipkin, 107-108

About the Author

Jonathan Schneider is CEO and cofounder of Moderne, where he works to modernize applications and infrastructure through automated source code transformation and asset visibility. Previously, he worked for the Spring team on application monitoring and engineering tools at Netflix. Jonathan has visited and given workshops on monitoring and delivery at a wide variety of enterprises, developing an understanding of the steps organizations need to take to get closer to Netflix-style resiliency, where outages in entire regions don't disrupt service availability and application teams know about problems before customers have a chance to report them.

Colophon

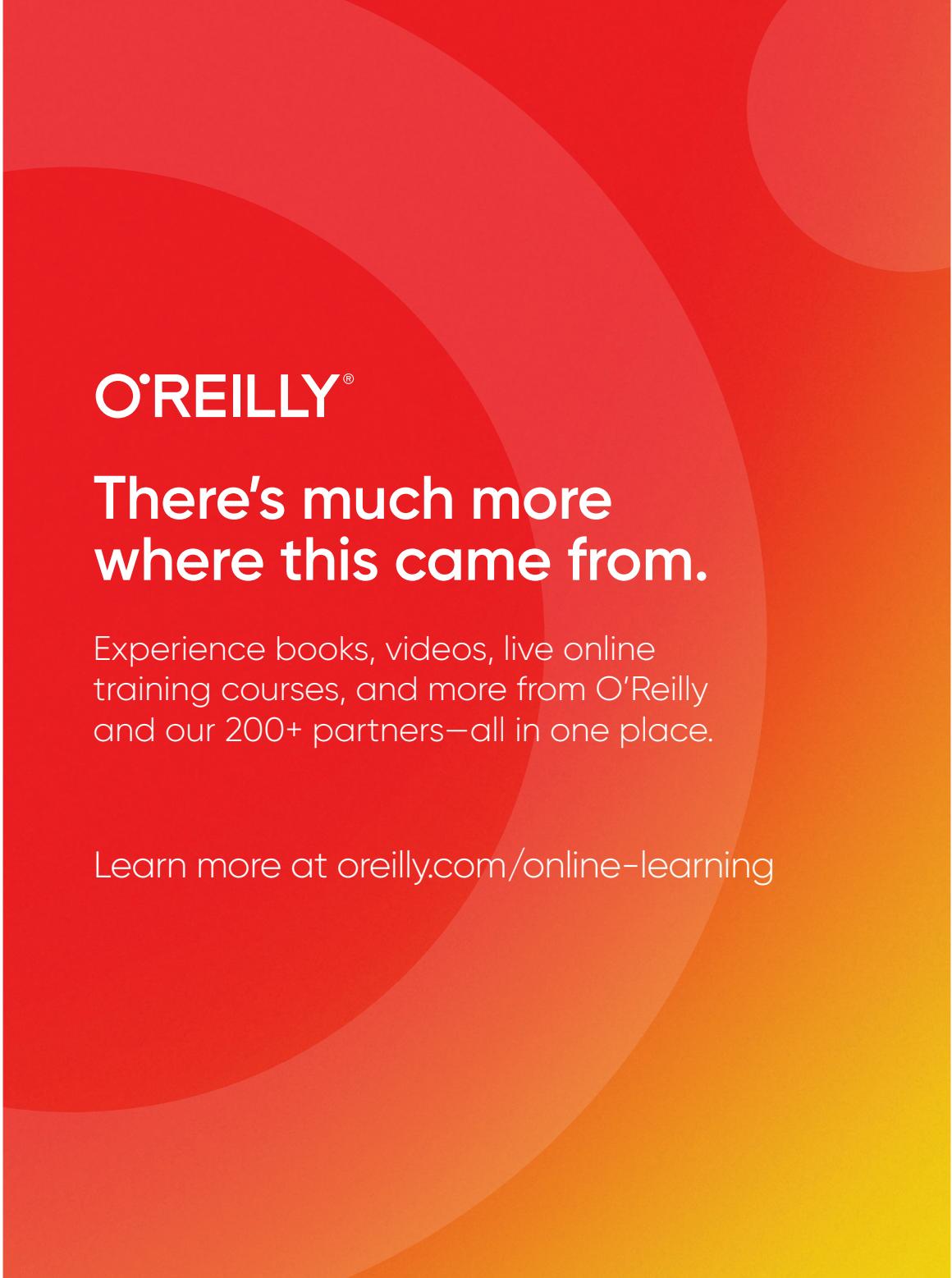
The animal on the cover of *SRE with Java Microservices* is a cream-spotted tiger moth (*Epicallia villica*). This moth inhabits a large range, from southwestern Europe to central Russia, the Middle East, and North Africa, in wooded areas and open grasslands.

The wingspans of the adult moths average two inches across, with females being slightly larger than males. Forewings show a black background and cream spots; hindwings have a light orange background with black spots. Body fur is black in the top third with cream spots on the sides; the rest of the body is orange that gradates into scarlet at the tail, with small black dorsal spots. The antennae are black.

Adult moths of this species fly mostly by night in May and June. Females lay on average 50 eggs in late summer; eggs soon hatch and caterpillars feed through the late summer into fall and winter. The caterpillars have dark brown hair and red feet. They feed on plants such as dandelion, dock, yarrow, and plantain. In spring they form a brown, felt-like cocoon of their own hairs, and after a pupation period of roughly 20 days, the adult moth emerges in spring to begin the cycle again.

The cream-spotted tiger moth is currently listed as “vulnerable” in parts of its range. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The color illustration on the cover is by Karen Montgomery, based on a black-and-white engraving from *Encyclopedie D'Histoire Naturelle: Papillons* (1878). The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.



O'REILLY®

There's much more where this came from.

Experience books, videos, live online training courses, and more from O'Reilly and our 200+ partners—all in one place.

Learn more at oreilly.com/online-learning