TECH TODAY

MASTERING OPENTELEMETRY™ AND OBSERVABILITY

Enhancing Application and Infrastructure Performance and Avoiding Outages

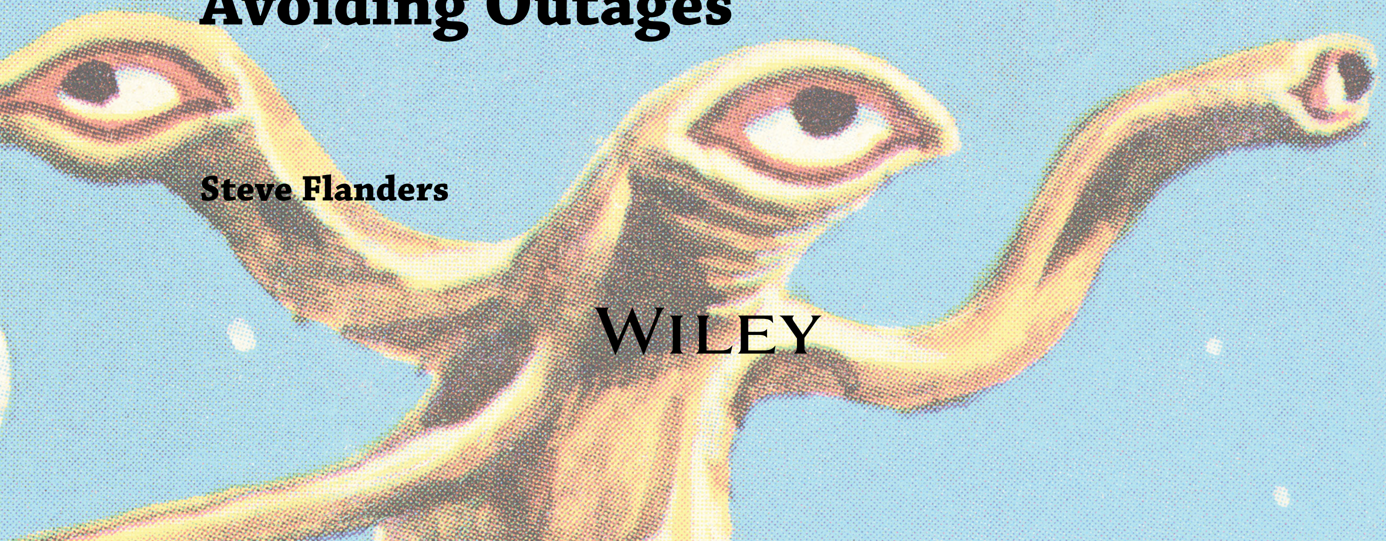
STEVE FLANDERS

WILEY

Mastering

OpenTelemetry™ and Observability

Enhancing Application and Infrastructure Performance and Avoiding Outages



Steve Flanders

WILEY

Copyright © 2025 by John Wiley & Sons, Inc. All rights, including for text and data mining, AI training, and similar technologies, are reserved.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada and the United Kingdom.

ISBNs: 9781394253128 (paperback), 9781394253142 (ePDF), 9781394253135 (ePub)

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at www.wiley.com/go/permission.

Trademarks: WILEY and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. OpenTelemetry is a trademark of The Linux Foundation. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Further, readers should be aware that websites listed in this work may have changed or disappeared between when this work was written and when it is read. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993. For product technical support, you can find answers to frequently asked questions or reach us via live chat at https://support.wiley.com.

If you believe you've found a mistake in this book, please bring it to our attention by emailing our Reader Support team at wileysupport@wiley.com with the subject line "Possible Book Errata Submission."

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic formats. For more information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Control Number: 2024944897

Cover image: © CSA Images/Getty Images
Cover design: Wiley

To my kids, Haven and Addison Flanders—May you always have the courage to chase your dreams with unwavering determination. Never let anyone dim the light of your aspirations or tell you what you can or cannot achieve. Believe in yourself, follow your heart, and remember that your potential is limitless. Your dreams are your own; only you can bring them to life.

About the Author

Steve Flanders is a founding member of the OpenCensus and OpenTelemetry projects and has over a decade of hands-on experience in the monitoring and observability space. As a Senior Director of Engineering at Splunk, a Cisco company, he oversees the Splunk Observability Cloud Platform, including the metrics engine and analytics capabilities. Steve also spearheads Splunk's OpenTelemetry contributions. He was previously instrumental in building what is now the Splunk APM product at Omnition and the Log Insight product at VMware. A sought-after speaker and blogger, Steve frequently shares his insights at prominent conferences like KubeCon and on his blog at `https://sflanders.net`. He holds an MBA from MIT, underscoring his blend of technical acumen, strategic vision, and entrepreneurial spirit.

Acknowledgments

Writing this book has been a labor of love, a journey marked by both immense challenges and profound rewards. The countless hours spent researching, writing, and revising have culminated in a work that I hope will provide valuable insights and guidance to everyone, regardless of background or experience. It was not easy to distill a complex topic like observability, but it was worth the effort. This book would not have been possible without the support and encouragement of many remarkable individuals and teams.

First and foremost, I would like to express my deepest gratitude to my partner, Lily Wang, for her unwavering support and understanding during the long hours and late nights dedicated to this project. Your patience and encouragement kept me going even when the task seemed daunting.

I am profoundly thankful to my colleagues, collaborators, and friends whose expertise and insights have greatly enriched this book. Your willingness to share your knowledge and feedback while engaging in stimulating discussions has been invaluable. Thank you, Fabrizio Ferri-Benedetti, for always providing timely feedback and suggestions. It is because of you the idea of Riley was created! Thank you, Jason Plumb, Pablo Collins, and Antoine Toulme, for reading early drafts and providing valuable initial feedback. Thank you, Alpesh Sheth, for supporting me in writing this book.

I also want to express my sincere gratitude to the technical reviewers who provided meticulous and constructive feedback on short notice. Your attention to detail and commitment to accuracy have been instrumental in shaping the contents of this book. Thank you, Tigran Najaryan and Morgan McLean, for providing an extensive review of the material on short notice. Thank you, Tyler Yahn, for providing a thorough review both technically and grammatically across multiple chapters, especially Chapters 1, 2, and 4. Thank you, Dmitrii Anoshin and Siim Kallas, for your thorough technical review of Chapters 5 and 6, respectively.

To the illustrator, publishers, and editorial team, thank you for your guidance and support throughout the publication process. Your professionalism and dedication ensured that this book reached its highest potential. Special thanks to Kenyon Brown for the opportunity to write this book, Tom Dinse for all the editing and formatting suggestions, and Navin Vijayakumar for keeping me on track. Also, a special thanks to Emily Griffin for bringing Riley, Jupiterian, and Watchwhale to life through relatable illustrations.

To the OpenTelemetry community, thank you for your innovative work and commitment to excellence, which continues to inspire and push the boundaries of what is possible. Being the second-most active project in the Cloud Native Computing Foundation is an amazing accomplishment and speaks to the need for an open standard and framework for telemetry data. To all the previous, current, and future authors, bloggers, and speakers on the topic of OpenTelemetry and observability, thank you for creating relevant information and sharing your knowledge. For anyone considering contributing to the project or sharing your experience, please do! The OpenTelemetry community is friendly and welcoming. We need your help to make the project even better. In the spirit of giving back, you will find a list of issues and pull requests (PRs) I submitted when writing this book in the appendix of this book.

Finally, I want to acknowledge all the readers and practitioners in the field of observability. Your passion for continuous improvement and innovation drives the evolution of this domain. It is my hope that this book serves as a valuable resource in your ongoing journey to master OpenTelemetry and achieve excellence in observability.

Thank you all for being part of this rewarding journey.

Contents

Foreword . . . *xiii*

Introduction . . . *xiv*
The Mastering Series . . . xvi

Chapter 1 • What Is Observability? . . . 1
Definition . . . 1
Background . . . 4
Cloud Native Era . . . 4
Monitoring Compared to Observability . . . 5
Metadata . . . 8
Dimensionality . . . 9
Cardinality . . . 9
Semantic Conventions . . . 10
Data Sensitivity . . . 10
Signals . . . 10
Metrics . . . 10
Logs . . . 13
Traces . . . 14
Other Signals . . . 20
Collecting Signals . . . 20
Instrumentation . . . 21
Push Versus Pull Collection . . . 22
Data Collection . . . 23
Sampling Signals . . . 26
Observability . . . 27
Platforms . . . 27
Application Performance Monitoring . . . 28
The Bottom Line . . . 28
Notes . . . 30

Chapter 2 • Introducing OpenTelemetry! . . . 31
Background . . . 31
Observability Pain Points . . . 31
The Rise of Open Source Software . . . 34
Introducing OpenTelemetry . . . 35
OpenTelemetry Components . . . 37
OpenTelemetry Concepts . . . 48
Roadmap . . . 50
The Bottom Line . . . 50
Notes . . . 51

Chapter 3 • Getting Started with the Astronomy Shop 53
Background 53
Architecture 54
Prerequisites 54
Getting Started 55
Accessing the Astronomy Shop 57
Accessing Telemetry Data 57
Beyond the Basics 58
Configuring Load Generation 58
Configuring Feature Flags 59
Configuring Tests Built from Traces 60
Configuring the OTel Collector 60
Configuring OTel Instrumentation 62
Troubleshooting Astronomy Shop 62
Astronomy Shop Scenarios 63
Troubleshooting Errors 63
Troubleshooting Availability 69
Troubleshooting Performance 70
Troubleshooting Telemetry 74
The Bottom Line 75
Notes 76

Chapter 4 • Understanding the OpenTelemetry Specification 77
Background 77
API Specification 79
API Definition 80
API Context 80
API Signals 81
API Implementation 82
SDK Specification 82
SDK Definition 83
SDK Signals 83
SDK Implementation 84
Data Specification 84
Data Models 86
Data Protocols 88
Data Semantic Conventions 88
Data Compatibility 89
General Specification 90
The Bottom Line 91
Notes 92

Chapter 5 • Managing the OpenTelemetry Collector 93
Background 94
Deployment Modes 95
Agent Mode 96
Gateway Mode 98
Reference Architectures 100

The Basics . . . 101
  The Binary . . . 103
  Sizing . . . 103
  Components . . . 104
Configuration . . . 106
  Receivers and Exporters . . . 115
  Processors . . . 116
  Extensions . . . 126
  Connectors . . . 127
Observing . . . 128
  Relevant Metrics . . . 128
  Health Check Extension . . . 131
  zPages Extension . . . 131
Troubleshooting . . . 134
  Out of Memory Crashes . . . 134
  Data Not Being Received or Exported . . . 134
  Performance Issues . . . 135
Beyond the Basics . . . 135
  Distributions . . . 135
  Securing . . . 137
  Management . . . 138
The Bottom Line . . . 140
Notes . . . 141

Chapter 6 • Leveraging OpenTelemetry Instrumentation . . . 143

Environment Setup . . . 144
Python Trace Instrumentation . . . 149
  Automatic Instrumentation . . . 150
  Manual Instrumentation . . . 157
  Programmatic Instrumentation . . . 163
  Mixing Automatic and Manual Trace Instrumentation . . . 166
Python Metrics Instrumentation . . . 167
  Automatic Instrumentation . . . 168
  Manual Instrumentation . . . 169
  Programmatic Instrumentation . . . 174
  Mixing Automatic and Manual Metric Instrumentation . . . 176
Python Log Instrumentation . . . 178
  Manual Metadata Enrichment . . . 179
  Trace Correlation . . . 181
Language Considerations . . . 183
  .NET . . . 184
  Java . . . 184
  Go . . . 184
  Node.js . . . 185
  Deployment Models . . . 185
Distributions . . . 185
The Bottom Line . . . 186
Notes . . . 187

Chapter 7 • Adopting OpenTelemetry 189

The Basics 189
Why OTel and Why Now? 190
Where to Start? 191
General Process 192
Data Collection 193
Instrumentation 195
Production Readiness 196
Maturity Framework 197
Brownfield Deployment 198
Data Collection 198
Instrumentation 200
Dashboards and Alerts 202
Greenfield Deployment 204
Data Collection 204
Instrumentation 208
Other Considerations 208
Administration and Maintenance 208
Environments 211
Semantic Conventions 212
The Future 213
The Bottom Line 213
Notes 214

Chapter 8 • The Power of Context and Correlation 215

Background 215
Context 217
OTel Context 219
Trace Context 221
Resource Context 223
Logic Context 224
Correlation 225
Time Correlation 225
Context Correlation 226
Trace Correlation 228
Metric Correlation 230
The Bottom Line 230
Notes 231

Chapter 9 • Choosing an Observability Platform 233

Primary Considerations 233
Platform Capabilities 235
Marketing Versus Reality 237
Price, Cost, and Value 238
Observability Fragmentation 241
Primary Factors 242
Build, Buy, or Manage 242

Licensing, Operations, and Deployment 244
OTel Compatibility and Vendor Lock-In 244
Stakeholders and Company Culture 245
Implementation Basics 246
Administration 247
Usage 248
Maturity Framework 248
The Bottom Line 250
Notes 250

Chapter 10 • Observability Antipatterns and Pitfalls 251
Telemetry Data Missteps 251
Mixing Instrumentation Libraries Scenario 253
Automatic Instrumentation Scenario 253
Custom Instrumentation Scenario 254
Component Configuration Scenario 255
Performance Overhead Scenario 255
Resource Allocation Scenario 256
Security Considerations Scenario 256
Monitoring and Maintenance Scenario 257
Observability Platform Missteps 258
Vendor Lock-in Scenario 260
Fragmented Tooling Scenario 260
Tool Fatigue Scenario 261
Inadequate Scalability Scenario 261
Data Overload Scenario 262
Company Culture Implications 264
Lack of Leadership Support Scenario 265
Resistance to Change Scenario 266
Collaboration and Alignment Scenario 266
Goals and Success Criteria Scenario 267
Standardization and Consistency Scenario 268
Incentives and Recognition Scenario 268
Feedback and Improvement Scenario 269
Prioritization Framework 270
The Bottom Line 272
Notes 273

Chapter 11 • Observability at Scale 275
Understanding the Challenges 275
Volume and Velocity of Telemetry Data 276
Distributed System Complexity 278
Observability Platform Complexity 281
Infrastructure and Resource Constraints 281
Strategies for Scaling Observability 282
Elasticity, Elasticity, Elasticity! 282

Leverage Cloud Native Technologies 284
Filter, Sample, and Aggregate 286
Anomaly Detection and Predictive Analytics 290
Emerging Technologies and Methodologies 291
Best Practices for Managing Scale 292
General Recommendations 292
Instrumentation and Data Collection 293
Observability Platform 293
The Bottom Line 294
Notes 295

Chapter 12 • The Future of Observability 297
Challenges and Opportunities 297
Cost 297
Complexity 299
Compliance 300
Code 301
Emerging Trends and Innovations 302
Artificial Intelligence 303
Observability as Code 304
Service Mesh 305
eBPF 306
The Future of OpenTelemetry 307
Stabilization and Expansion 308
Expanded Signal Support 308
Unified Query Language 310
Community-driven Innovation 310
The Bottom Line 311
Notes 311

Appendix A • The Bottom Line 313
Chapter 1: What Is Observability? 313
Chapter 2: Introducing OpenTelemetry! 315
Chapter 3: Getting Started with the Astronomy Shop 316
Chapter 4: Understanding the OpenTelemetry Specification 317
Chapter 5: Managing the OpenTelemetry Collector 318
Chapter 6: Leveraging OpenTelemetry Instrumentation 320
Chapter 7: Adopting OpenTelemetry 321
Chapter 8: The Power of Context and Correlation 323
Chapter 9: Choosing an Observability Platform 324
Chapter 10: Observability Antipatterns and Pitfalls 326
Chapter 11: Observability at Scale 327
Chapter 12: The Future of Observability 328

Appendix B • Introduction 329
Chapter 2: Introducing OpenTelemetry! 330
  OpenTelemetry Concepts > Roadmap 330
Chapter 3: Getting Started with the Astronomy Shop 330
  Background > Architecture 330
Chapter 5: Managing the OpenTelemetry Collector 332
  Background 332
  The Basics > Components 332
Chapter 12: The Future of Observability 340
  Challenges and Opportunities > Code 340
Notes 341

Index 343

Foreword

To build and operate any complex system, whether it be inventory in warehouses, money in bank accounts, or large computer systems, you need to be able to understand what you have built and how it is currently operating. The observability tools that we rely on today have a long history; in one way or another, they have existed since the beginning of the computing industry. As relatively high-scale (for their era) computing services started to come about in the 1980s and early 1990s, commercial tools that analyzed their performance also became available. As the dot-com boom of the late 1990s and then the proliferation of easily accessible cloud infrastructure drove more and more firms to build high-scale web services, the market and capabilities of what we now call observability tools increased dramatically.

Throughout this period, one of the biggest challenges that these tools faced was how to get the right data into them. This is harder than it seems, as early solutions would capture some combination of logs and metrics, requiring integrations with a handful of operating systems and known technologies like databases and message queues. Getting visibility into a modern microservices environment requires distributed traces, application metrics, profiles, and other types of data that must be captured from every web framework, RPC system, database client, and so forth, each of which are different for each programming language. Each of these integrations must be maintained to ensure that it does not break when the data source gets updated; this is extremely expensive for vendors to build and for customers to set up, leading to poor coverage and for customers to be semipermanently locked in to their vendors.

We created OpenTelemetry to break this logjam. By providing a single set of APIs, agents, and a protocol, we allowed software developers to both emit and capture distributed traces, metrics, profiles, and logs easily and with the strong semantic conventions needed to gain valuable insights from analyzing it. This has fundamentally changed people's relationships with observability tools. Thanks to OpenTelemetry, they are more accessible and widely used than ever before—and of higher quality, as vendors and open source solutions have redirected the effort that they used to spend on data collection to providing better solutions. Both end users and those who want to emit data from shared code are no longer locked in to vendor-specific interfaces, and anyone can take control of creating custom telemetry, filtering their data, and sending it wherever they would like. OpenTelemetry now has over 1,200 developers contributing to it every month, making it one of the largest open source projects in the world—a testament to its utility and how much it has changed things.

That being said, tools are only as good as one's ability to properly use them, and OpenTelemetry is no exception. OpenTelemetry is now an essential part of building and operating services of any scale, and this book will guide you through the problems that it can be used to solve (and those that it should not), OpenTelemetry's various components, best practices and examples of using OpenTelemetry successfully, and how to apply it to your codebase and organization to achieve your goals.

—Morgan McLean, Senior Director of Product Management, Splunk

Introduction

Welcome to *Mastering OpenTelemetry and Observability*, a comprehensive guide designed to help you navigate the complex and ever-evolving landscape of observability. As organizations increasingly rely on distributed systems and microservices architectures, the need for robust observability solutions has never been greater. OpenTelemetry, or OTel as it is called, is an open source and vendor-agnostic observability framework. It has emerged as a critical technology in this field, providing standardized tools for collecting and analyzing telemetry data across various platforms and technologies. In addition, OTel is extensible, with the ability to handle the telemetry needs and observability platforms of today and the observability landscape of the future.

This book aims to equip you with the knowledge and skills necessary to harness the full potential of OTel and build a solid observability foundation. Whether you are a developer, DevOps engineer, site reliability engineer (SRE), sales engineer, support engineer, information technology (IT) manager, engineering manager (EM), product manager (PM), C-level executive, or really any role that involves software or infrastructure, the insights and practical guidance offered in this book will empower you to observe, diagnose, and optimize your systems effectively.

You will begin by exploring the fundamental concepts of observability, tracing its evolution from traditional monitoring practices to modern, holistic approaches. You will gain a deep understanding of the three pillars of observability—metrics, logs, and traces—and how they interrelate to provide a comprehensive view of system health and performance. The core of this book delves into OpenTelemetry, starting with its architecture and components, including the specification, instrumentation, and the Collector. Next, the OTel demo environment, known as the Astronomy Shop, is explored so you can experience the power of OTel firsthand. Deep dives on all the major components, including step-by-step instructions, are provided on how to instrument your applications and collect, process, and send your telemetry data using OTel. You will also learn about important topics such as context propagation, distributions, and integrating OTel with popular observability platforms like Prometheus and Jaeger.

With a solid foundation in observability and OTel, you will move on to adopting and scaling observability in large and complex environments. From obtaining stakeholder buy-in to handling high volumes of telemetry data to ensuring performance and reliability, you will discover practical solutions to common challenges faced by organizations today. This is followed by considerations for observability platforms, whether existing or new.

Beyond technical guidance, this book also addresses the human and organizational aspects of observability. This is because building a culture of observability within your team and organization is crucial for success. This book discusses strategies for fostering collaboration, continuous improvement, and proactive incident response, ensuring that observability becomes an integral part of your operational practices. Finally, this book explores emerging trends and innovations in observability, including the role of artificial intelligence (AI) and machine learning (ML) in predictive analytics, the evolution of observability standards, and the potential impact of new technologies on the industry.

Mastering OpenTelemetry and Observability is more than just a technical manual; it is a journey into the heart of modern system monitoring and optimization. By the end of this book, you will have the knowledge and confidence to implement robust observability solutions that enhance your system's reliability, performance, and overall user experience.

Before you begin reading, there are a few things to know:

- This book has been written in a way that tries to make it approachable to the largest audience possible. Examples of this include:
  - The book does not use contractions to make it easier for non-native English speakers.
  - Every abbreviation used in every chapter is defined first.
  - Relatable examples and metaphors will be found throughout the book.
  - Hyperlinks to additional information are provided throughout the book so you can learn more about the topics being discussed.
- A fictitious but likely relatable story is embedded into every chapter. Through it, you will learn how an enterprise company migrating to the cloud was struggling to achieve observability. With each challenge experienced, you will see how a determined site reliability engineer (SRE) helps her company embrace OTel and improve observability.
- Some terminology is used throughout this book that you should be aware of, including:
  - **Back end:** The data access layer of an application, which often includes processing and persistence of data.
  - **Framework:** A structure on which other things are built. For example, OTel is a telemetry framework that can be extended to support various use cases.
  - **Front end:** The presentation layer of an application, which is often a user interface (UI) or user-facing way to interact with an application.
  - **Instrumentation:** Software added to an application to generate telemetry data. Various forms of instrumentation are available, including automatic, which is injected at runtime, manual, which is added with the existing code, and programmatic, which is a particular form of manual instrumentation where specific libraries or frameworks have already been instrumented (also called *instrumentation libraries*).
  - **Platform:** An environment in which software is executed. An observability or monitoring platform typically consists of one or more back end and front end components.

    - **Telemetry:** Data used to determine the health, performance, and usage of applications. Examples of telemetry include metrics, logs, and traces. This data is typically sent to a platform or back end.

- The OTel project is constantly evolving, and changes are frequently released. The examples provided in this book were tested against specific versions of OTel. Where possible, they were created in a generic way that should work as the project advances. With that said, it is possible that changes have been made that will result in differences from what is documented. If this occurs, checking the GitHub repository associated with this book (covered next) and reading the latest OTel documentation and release notes is recommended. The minimal recommended and maximum tested versions of OTel components for this book are as follows:
    - OTel Demo, also known as the Astronomy Shop, version 1.11 is the minimum supported version. This is to get OpenSearch support. Up to version 1.11.1 has been tested.
    - Collector (core and contrib) version 0.95.0 is the minimum supported version. This is to get JSON encoding for the OTLP receiver and exporter. Up to version 0.109.0 has been tested.
    - Python instrumentation version 1.23.0/0.44b0 is the minimum supported version. This is to get support for Flask and Werkzeug 3.0 or higher. Up to version 1.27.0/0.48b0 has been tested.
- This book is accompanied by a GitHub repository, which can be found at `https://github.com/flands/mastering-otel-book` and will be updated at least annually. If you notice any issues with the information presented in this book, please open a GitHub issue. The contents of this repository include:
    - All code examples provided in the book
    - Status information about OTel components
    - Post-production modifications
    - Changes to support the latest OTel advancements

The Mastering Series

The *Mastering* series from Sybex provides outstanding instruction for readers with intermediate and advanced skills, in the form of top-notch training and development for those already working in their field and clear, serious education for those aspiring to become pros. Every *Mastering* book includes:

- Real-World Scenarios, ranging from case studies to interviews, that show how the tool, technique, or knowledge presented is applied in actual practice
- Skill-based instruction, with chapters organized around real tasks rather than abstract concepts or subjects
- Self-review test questions, so you can be certain you're equipped to do the job right

What Is Observability?

In modern software development and operations, observability has emerged as a fundamental concept essential for maintaining and improving the performance, reliability, and scalability of complex systems. But what exactly is observability? At its core, observability is the practice of gaining insights into the internal states and behaviors of systems through the collection, analysis, and visualization of telemetry data. Unlike traditional monitoring, which primarily focuses on predefined metrics and thresholds, observability offers a more comprehensive and dynamic approach, enabling teams to proactively detect, diagnose, and resolve issues.

This chapter will explore the principles and components of observability, highlighting its significance in today's distributed and microservices-based architectures. Through a deep dive into the three pillars of observability—metrics, logs, and traces—you will understand the groundwork for how observability can transform the way resilient systems are built and managed.

IN THIS CHAPTER, YOU WILL LEARN TO:

- Differentiate between monitoring and observability
- Explain the importance of metadata
- Identify the differences between telemetry signals
- Distinguish between instrumentation and data collection
- Analyze the requirements for choosing an observability platform

Definition

So, what is observability in the realm of modern software development and operations? While many definitions exist, they all generally refer to observability providing the ability to quickly identify availability and performance problems, regardless of whether they have been experienced before, and help perform problem isolation, root cause analysis, and remediation. Because observability is about making it easier to understand complex systems and address unperceived issues, often referred to in the software industry as *unknown unknowns*,[1] the data collected must be correlated across different telemetry types and be rich enough and immediately accessible to answer questions during a live incident.

The Cloud Native Computing Foundation (CNCF), described more fully later in this chapter, provides a definition for the term *observability*:[2]

> *Observability is a system property that defines the degree to which the system can generate actionable insights. It allows users to understand a system's state from these external outputs and take (corrective) action.*
>
> *Computer systems are measured by observing low-level signals such as CPU time, memory, disk space, and higher-level and business signals, including API response times, errors, transactions per second, etc. These observable systems are observed (or monitored) through specialized tools, so-called observability tools. A list of these tools can be viewed in the Cloud Native Landscape's observability section.*[3]
>
> *Observable systems yield meaningful, actionable data to their operators, allowing them to achieve favorable outcomes (faster incident response, increased developer productivity) and less toil and downtime.*
>
> *Consequently, the observability of a system will significantly impact its operating and development costs.*

While the CNCF's definition is good, it is missing a few critical aspects:

- The goal of observability should be where a system's state can be *fully understood* from its external output *without the need to ship code*. This means you should be able to ask *novel questions* about your observability data, especially questions you had *not* thought of beforehand.
- Observability is not just about collecting data but about collecting *meaningful data*, such as data with context and correlated across different sources, and storing it on a platform that offers rich analytics and query capabilities *across signals*.
- A system is truly observable when you can troubleshoot *without prior knowledge of the system*.

The OpenTelemetry project, which will be introduced in Chapter 2, "Introducing OpenTelemetry!," provides a definition of observability that is worth highlighting:

> *Observability lets you understand a system from the outside, by letting us ask questions about that system without knowing its inner workings. Furthermore, it allows you to easily troubleshoot and handle novel problems—that is, "unknown unknowns." It also helps you answer the question, "Why is this happening?"*
>
> *To ask those questions about your system, your application must be properly instrumented. That is, the application code must emit signals such as traces, metrics, and logs. An application is properly instrumented when developers don't need to add more instrumentation to troubleshoot an issue, because they have all of the information they need.*[4]

In short, observability is about collecting critical telemetry data with relevant context and using that data to quickly determine your systems' behavior and health. Observability goes beyond mere monitoring by enabling a proactive and comprehensive understanding of system behavior, facilitating quicker detection, diagnosis, and resolution of issues. This capability is crucial in today's fast-paced, microservices-driven, distributed environments, where the

complexity and dynamic nature of systems demand robust and flexible observability solutions. Through the lens of the CNCF and OpenTelemetry, you can see observability is not just defined as a set of tools and practices but as a fundamental shift toward more resilient, reliable, and efficient system management.

Real World Scenario

Riley Joins Jupiterian

Riley (she/her) is an experienced site reliability engineer (SRE) with deep observability and operations experience. She recently joined Jupiterian to address their observability problems and work with a new vendor. Riley joined Jupiterian from a large private equity (PE) advertising company, where she was the technical lead of the SRE team and was responsible for a large-scale, globally distributed, cloud native architecture. Before that, she was the founding member of a growth startup where she developed observability practices and culture while helping scale the business to over three million dollars in annual recurring revenue (ARR). Riley was excited about the challenge and opportunity of building observability practices from the ground up at a public enterprise company transitioning to the cloud.

Jupiterian is an e-commerce company that has been around for more than two decades. Over the last five years, the company has seen a massive influx of customers and has been on a journey to modernize its tech stack to keep up with demand and the competition. As part of these changes, it has been migrating from its on-premises monolithic application to a microservices-based architecture running on Kubernetes (K8s) and deployed in the cloud. Recently, outages have been plaguing the new architecture—a problem threatening the company and one that needed to be resolved before the annual peak traffic expected during the upcoming holiday season.

For the original architecture, the company had been using Zabbix, an open source monitoring solution to monitor the environment. The IT team was beginning to learn about DevOps practices and had set up Prometheus for the new architecture. Given organizational constraints and priorities, they did not have the time to develop the skill set to manage it and the ever-increasing number of collected metrics. In short, a critical piece of the new architecture was without ownership. On top of this, engineering teams continued to add data, dashboards, and alerts without defined standards or processes. Not surprisingly, this resulted in the company having difficulty proactively identifying availability and performance issues. It also resulted in various observability issues, including Prometheus availability, blind spots, and alert storms. In terms of observability, the company frequently experienced infrastructure issues and could not tell if it was because of an architecture limitation or an improper use of the new infrastructure. As a result, engineers feared going on-call, and innovation velocity was significantly below average.

The Jupiterian engineering team had been pushing management to invest more in observability and SRE. Instead, head count remained flat, and the product roadmaps, driven primarily by the sales team, continued to take priority. With the service missing its service-level agreement (SLA) target for the last three months, leadership demanded a focus on resiliency. To address the problem, the Chief Technology Officer (CTO) signed a three-year deal with Watchwhale, an observability vendor, so the company could focus on its core intellectual property (IP) instead of managing third-party software. An architect in the office of the CTO vetted the vendor and its technology. Given other organizational priorities, the engineering team was largely uninvolved in the proof of

concept (PoC). The Vice President (VP) of Engineering was tasked with ensuring the service's SLA was consistently hit ahead of the holiday period as well as the adoption and success of the Watchwhale product. He allocated one of his budget IDs (BIDs) for a senior SRE position, which led to Riley being hired.

Background

The term *observability* has been around since at least the mid-20th century and is mainly credited to Rudolf E. Kálmán, a Hungarian American engineer who used it in a paper about control theory.[5] Since then, the term has been used in various fields, including quantum mechanics, physics, statistics, and perhaps most recently, software development. Kálmán's definition of observability can be summarized as a measure of how well the internal states of a system can be inferred from knowledge of its external outputs.[6]

Observability Abbreviation

Observability is often abbreviated as O11y (the letter *O*, the number *11*, and the letter *y*), as there are 11 characters between the letter *O* and the letter *y*. While it is the number 11, the ones are pronounced as the letter *l*—thus, the abbreviation is pronounced *Ollie*. This abbreviation standard is common for longer words in software. For example, Kubernetes, a popular cloud native open source project, is often referred to as K8s and pronounced *kay-ates* for the same reason.

Cloud Native Era

In software, the term *observability* has become popular due to the rise of cloud native workloads. Since the turn of the century, the software industry has seen a progression that has included

moving from bare metal machines to virtual machines (VMs) to containers. In addition, there has been a shift from owning, deploying, and managing hardware to leasing data center equipment to deploying in the cloud. But what does *cloud native* mean? One way to answer this question is to look to the CNCF. The foundation is part of the Linux Foundation and defines itself as:

> *The open source, vendor-neutral hub of cloud native computing, hosting projects like Kubernetes and Prometheus to make cloud native universal and sustainable.*[7]

Perhaps not surprisingly, the CNCF has created a definition for the term *cloud native*:

> *Cloud native practices empower organizations to develop, build, and deploy workloads in computing environments (public, private, hybrid cloud) to meet their organizational needs at scale in a programmatic and repeatable manner*. They are characterized by loosely coupled systems that interoperate in a manner that is secure, resilient, manageable, sustainable, and observable.
>
> *Cloud native technologies and architectures typically consist of some combination of containers, service meshes, multi-tenancy, microservices, immutable infrastructure, serverless, and declarative APIs—this list is non-exhaustive.*

Monitoring Compared to Observability

Before the cloud native era, it was common to see patterns including on-premises software, monoliths, separate development and operations teams, and waterfall software development with long release cycles. In this prior generation, the term *observability* had not been adopted yet, and instead, the term *monitoring* was used. Sometimes, these terms are used interchangeably, but their meanings are not identical. The Merriam-Webster dictionary defines monitoring as the ability "to watch, keep track of, or check usually for a special purpose."[8] It defines observability as the ability "to come to realize or know especially through consideration of noted facts."[9] The distinction between monitoring and observability is important. With monitoring, you track items but must infer why something occurred or how it is related to another event. With observability, you use information to prove facts and use that knowledge to determine how or why something behaves the way it does. Observability allows for first principle thinking, or the ability to validate assumptions not deduced from another assumption.[10]

In software, both observability and monitoring rely on specific data types—primarily metrics and logs with some tracing—but the usage of the data differs. Before the cloud native era, most software ran on-premises and was often developed and deployed as a monolith or single code base or application. As a result, problem isolation, or where the problem originated, was easy to identify when issues occurred, and scaling typically consisted of adding more resources to the monolith, known as *scaling up* or *scaling vertically*. When issues arose, the problem was either the monolith, the infrastructure the monolith was running on top of, or whatever application was calling into or called by the monolith (see Figure 1.1). To monitor the monolith, operational teams needed the ability to be alerted about specific, known symptoms, sometimes referred to as *known knowns*. Monitoring systems did exactly that.

To provide monitoring, either your application needs to be instrumented to emit health data or you are required to infer the health of the application by watching its external behavior. In either case, the data collected needs to be able to track and answer questions about availability, performance, and security. This data collection needs to be added before issues happen; otherwise, you cannot proactively determine nor quickly resolve the problems as they arise.

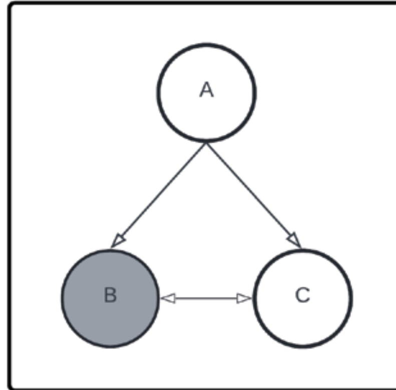


Figure 1.1
An example of a monolithic application experiencing an issue. The square represents the monolith, while the circles represent different functions or features within the monolith. In this example, the B function is experiencing problems, denoted by the service's gray shading. This may or may not result in issues with the A and C functions.

Types of Monitoring

There are two different types of monitoring. First, there is monitoring based on data exposed from the internals of the system. This means the application makes specific data available for external systems to gather. This type of monitoring is sometimes called *white box monitoring* because you can see into the system,[11] though a better name would be *internally provided monitoring*. Second, there is monitoring based on external behavior. This means the application does not make any data available beyond what is required for the application to function. As such, an external system must infer what an application is doing. This type of monitoring is sometimes called *black box monitoring* because you cannot see into the system,[12] though a better name would be *externally provided monitoring*.[13]

In many cases, application developers add instrumentation as necessary, including to measure performance and investigate issues during development and operations. Engineers responsible for monitoring the health and performance of these applications would typically send telemetry data to a monitoring platform. Based on this telemetry data, the engineer would then define alerts with static thresholds. To determine these thresholds, an engineer would need to know what problems to expect beforehand, thus enabling *proactive monitoring*; otherwise, new thresholds would have to be defined after an issue is identified, which is known as *reactive monitoring*. One way to think about monitoring is like a doctor who collects certain pieces of information from a person and compares that data against known baselines to understand the symptoms being experienced and to determine the health of the person. The monitoring of heart rate, blood pressure, and temperature in humans is like the monitoring of CPU (central processing unit), memory, and disk usage in applications.

While monitoring with static thresholds provides some awareness of potential system issues, it is not without its limitations. Take, for example, CPU utilization, which represents the rate at which an application is operating expressed as a percentage. If CPU utilization is very high, this could be a symptom of a system issue and, as such, something you want to be notified about. For example, you could define an alert when the CPU utilization exceeds 95 percent for some period

of time. In fact, such a definition is common in traditional monitoring applications. The problem is, such an alert may not indicate a problem but instead indicate that the application is using its resources efficiently. What is missing from this symptom is context, including how other related components are behaving, and correlation, including changes within the environment. Another limitation of traditional monitoring tools is the difficulty in alerting on issues that do not manifest as high resource consumption or latency.

The introduction of cloud native workloads made traditional monitoring even less effective. In this new world, workloads are run in the cloud and often consist of many small applications, called *microservices*, that are isolated to individual functionality. For example, an authentication service or a notification service. Microservices make it easier to deploy more instances, known as *scaling out* or *scaling horizontally*, and allow for specific components to be scaled as needed. These microservices typically run on immutable infrastructure using declarative APIs (application programming interfaces). In addition, they are run with DevOps practices and with the help of site reliability engineers (SREs).[14] Software release cycles are also more frequent and leverage continuous integration and continuous deployment or CI/CD pipelines. The decoupling and elasticity of applications enable developers to reduce duplicated efforts and scale to meet demand, but often at the cost of being able to troubleshoot the system and keep it available. In this era, it is the "unknown unknowns" that need to be addressed.

Due to the difficulty in troubleshooting microservice-based architectures, a popular meme was shared throughout the community:

> *"We replaced our monolith with micro services so that every outage could be more like a murder mystery."* @honest_update[15]

With cloud native workloads, problem isolation became a problem. This is because when one microservice has an issue, it could impact upstream or downstream services, causing them to have problems as well (see Figure 1.2). Using traditional monitoring, the net result is alert storms and the need to investigate every issue on every service in order to get to the root cause and remediation. Of course, there are other issues with cloud native workloads as well. For example, there is an inability to have complete visibility into the infrastructure as it is being managed by a third party and prone to dynamic changes.

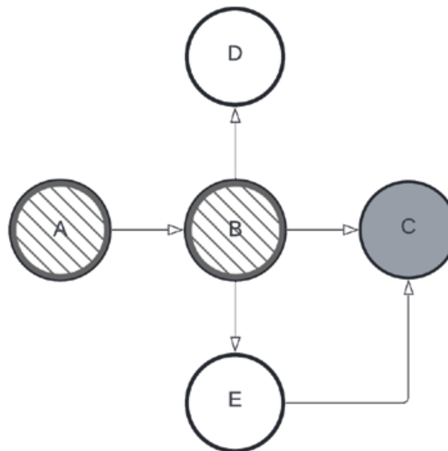


FIGURE 1.2
An example of a microservice-based architecture experiencing an issue. Each circle represents a different microservice. In this example, multiple microservices are experiencing an issue, denoted with gray lines, though one service is the root cause of the problem, denoted with gray shading. Note not all services called by the root cause service are impacted.

Going back to the doctor analogy, assume you have a large group of people who are all part of the same community, and multiple people become sick around the same time. While you may want to help everyone experiencing symptoms concurrently, it requires many doctors and resources. In addition, focusing on the symptoms of the patients does not address the root cause issue, which is that people are getting sick from something, and it is spreading instead of being contained. The sickness may cause other problems to arise as well. For example, doctors may become sick and thus become unable to care for patients, or businesses might need to shut down because they do not have enough employees to work. Without containment, an infectious disease can spread uncontrollably. This analogy is similar to the changes necessary due to the shift to cloud native workloads. For example, instead of paging all service owners during an outage and burning out engineers, the more sustainable approach is to contain the problem and page the root cause service. Observability helps with containment.

When dealing with complex systems, it is ideal when you can address things you are aware of and understand, referred to as *known knowns*, as well as things you are not aware of and do not understand, referred to as *unknown unknowns*, using the same solution. See Table 1.1 for different states of awareness and understanding. A goal of observability is to provide the ability to answer the "unknown unknowns." At the same time, it contains the building blocks necessary to address the "known knowns." As a result, observability may be considered a superset of monitoring.

Table 1.1: A 2×2 matrix showing states of awareness and understanding. Monitoring systems are optimized to address "known knowns" where observability systems can address all aspects but especially "unknown unknowns."

		Awareness **Known**	**Unknown**
Understanding	**Known**	Aware of and understand	Aware of but do not understand
	Unknown	Understand but not aware of	Neither aware of nor understand

Metadata

When you hear the term *observability*, you may initially think about data sources such as metrics, logs, and traces. These terms will be introduced in the next section, but something just as important as the data source information is metadata. While a fancy word, metadata is just data about other data. For example, if you generate and collect a metric, such as the total number of HTTP requests, it may also be helpful to know other information about that metric, such as which host it is running on or what HTTP status code was returned for that request. These additional pieces of information are known as *metadata* and are typically attached to traditional data source information, such as metrics, logs, and traces. Metadata may go by other names as well, including tags, labels, attributes, and resources.

Metadata is powerful because it provides additional information to data sources, which helps with problem isolation and remediation. This information may even contain context and correlation, topics explored in Chapter 8, "The Power of Context and Correlation." Without

metadata, observability is harder to achieve. Metadata is typically represented as a key-value pair, such as `foo="bar"`. The *key* is the name for the piece of metadata and is often referred to as a *dimension*. The *value* can be of various forms, including numbers or strings and the uniqueness of the values is referred to as *cardinality*. Other ways to represent metadata also exist. For example, in unstructured log records, metadata is sometimes presented as just a value where the name is inferred—an example is provided in the "Logs" section later in this chapter.

Dimensionality

In observability, *Dimensionality* refers to the number of unique *keys* (sometimes called *names*) within a set. It is represented by attributes or labels associated with telemetry data, allowing for more granular and detailed analysis. Each piece of telemetry data can have multiple dimensions that provide context about the data. These dimensions enable the grouping, filtering, and slicing of data along various axes, which is crucial for deep analysis and troubleshooting. Examples of dimensions include:

- Time
- Application, such as `service.name` and `service.version`
- Host, such as `host.name` and `host.arch`
- User, such as `enduser.id` and `enduser.role`
- HTTP, such as `http.route` and `http.response.status_code`

These dimensions would allow you to ask your telemetry to show you data such as:

- All `502` errors in the last half hour for host `foo`
- All `403` requests against the `/export` endpoint made by user `bar`

Dimensionality, which may also be referred to as the width of telemetry data, is a foundational concept in observability that greatly enhances the depth and utility of telemetry data. It matters because it enables more detailed, contextualized, and actionable insights, which are essential for maintaining and improving the performance and reliability of modern distributed systems. In practice, dimensions are indexed by observability platforms to support capabilities, including auto-complete and real-time analysis of key-value pairs, that assist with troubleshooting.

Cardinality

In observability, *Cardinality* refers to the number of unique values for a given key within a set. *High cardinality* refers to a large number of unique values, whereas *low cardinality* indicates fewer unique values. For example, a dimension like HTTP status code, which includes values such as `404` or `500`, is bounded and has low cardinality, whereas a dimension like a user, session, or transaction ID is unbounded and is likely to have high cardinality. Monitoring and observability platforms care about cardinality. For example, if a platform supports indexing of keys, it likely needs to return values for those indexed keys quickly. For high cardinality metadata, this can prove challenging to visualize and very expensive to compute. In short, cardinality affects the storage, performance, and usability of telemetry data. High cardinality presents both opportunities for detailed insights and challenges in terms of resource consumption and data management. Effectively managing cardinality is essential for maintaining scalable, efficient, and actionable observability systems.

Semantic Conventions

Another concept you should be aware of is *semantic conventions*, or semconvs for short. These are standardized dimensions, or keys, for metadata and ensure consistency in how data is recorded, labeled, and interpreted across different systems and services. It may also contain standardized cardinality, or values for these dimensions. For example, it is common to have semconvs for HTTP-related data. An example of this may include the key for the HTTP route, such as `http.route`, or the response status code, such as `http.response.status_code`. Semconvs can be grouped into multiple different categories, such as the aforementioned HTTP. Other categories would include databases, exceptions, host metrics, function as a service, and messaging, to name a few. Each category would have multiple semconvs defined. Semconvs may be signal specific or apply to more than one signal type. Semconvs matter because they enable context and correlation and provide data portability. For example, if the same key is used to represent the same data, then it is easy to see its behavior across systems and environments. In addition, if keys are consistently named, they can be leveraged identically across different platforms.

Data Sensitivity

Metadata can contain sensitive information. For example, names or email addresses may be attached to data sources and leak personally identifiable information (PII). In addition, internal business logic, such as Internet protocol (IP) addresses or hostnames, may be considered sensitive information. This information would generally only be sent to the configured observability platforms, but that configuration can change over time. In addition, while only restricted users may have access to the platform data, for example, employees of a company authenticated via Security Assertion Markup Language (SAML), without proper data permissions, such as role-based access control (RBAC), it is possible that sensitive information is exposed to employees who should not have access to such information. Given that metadata can contain virtually anything, care must be taken to ensure proper data configuration, scrubbing, and access control.

Signals

The *three pillars of observability* is an industry phrase that you have likely encountered. The three pillars refer to metrics, logs, and traces. While these pillars are just data sources and do not inherently provide observability, they are recognized as fundamental types of telemetry data needed to understand the behavior and performance of systems. Another comparable term or acronym in the observability space is *MELT*, which stands for metrics, events, logs, and traces. These are the most common data sources, but they are far from exhaustive. Other examples include profiling and sessions. Data sources have a variety of names in the industry, including diagnostics, telemetry, signals, or data sources. For the purposes of this book, and in alignment with OpenTelemetry, the term *signals* will be used going forward. It is important to note that signals do not inherently provide observability, though they are necessary to enable it.

Metrics

A *metric*, sometimes referred to as a *metric record*, *measurement*, or *metric time series* (MTS), is a set of data points represented as a time series with metadata. A *time series* is a set of data points over

some period of time. To generate a time series, an instrument takes one or more measurements. For example, a speedometer measures speed, and a measurement could be taken every tenth of a second but recorded every minute. Metrics also have signal-specific metadata terms. For example, *attributes*, *dimensions*, *labels*, and *resources* are all terms used with metrics that refer to some kind of metadata.

A metric contains a name, value, timestamp, and optionally additional metadata. Note that multiple types of metric values exist. For example, it may be a single value, such as a counter, or a multi-value, such as a histogram. Here is an example of a metric from Prometheus, an open source metric solution that will be described in more detail later:

```
http_requests_total{method="post",code="200"}     1027     | 1395066363000
```

The example Prometheus metric is made up of various components, including:

- Name—`http_requests_total`
- Metadata—`{method="post",code="200"}`
- Value—`1027`
- Timestamp—`1395066363000`

Metrics are one of the primary data sources used to engage on-call engineers as well as troubleshoot availability and performance issues. It is pervasive for alerts and dashboards to be configured based on metric data. Generally, aggregated metrics, like those shown in Figure 1.3, provide the most value because they identify behaviors over time and can be used to determine anomalies. Some popular methods for analyzing aggregated metrics include:

- RED, which stands for requests, error, and duration and was popularized by Tom Wilkie.[16] The idea is for every object to monitor the number of requests, the number of those requests that result in an error, and the amount of time those requests take. In general, this information can be used to determine user experience.
- USE, which stands for utilization, saturation, and errors and was popularized by Brendan Gregg.[17] The idea is for every object to monitor the percentage of time the object was busy, the amount of work (queue size) for the object, and the number of errors. In general, this information can be used to determine object experience.
- Four golden signals, which include latency, traffic, errors, and saturation and was popularized by the Google SRE Handbook.[18] The idea is for every object to monitor the time it takes to service a request, the amount of demand placed on the object, the rate of requests that fail, and the fullness of the object. This is like RED but includes saturation.

In addition, metrics are used to define service-level indicators (SLIs) that measure the performance of applications. These SLIs are used to define and measure service-level objectives (SLOs) which determine whether applications are operating within acceptable bounds. Service-level agreements (SLAs) are also defined and calculated based on metrics to determine whether applications are meeting specified customer expectations.

Figure 1.3 A Grafana dashboard displaying aggregate metric information.

Learning More About SLIs, SLOs, and SLAs

SLIs, SLOs, and SLAs are critical topics that are outside the scope of this book. If you are looking to learn more about these concepts, be sure to read the *Google Site Reliability Engineering (SRE)* book, which is freely available online.[19]

Given the ever increasing number of objects in an environment and the need to collect more and more data, metric platforms need to be able to process and store a large number of metrics quickly. Various techniques are used to control the amount of data generated, processed, and stored. For example, the interval at which metrics are generated within the application or stored within an observability platform can be different from the resolution displayed in charts. Aggregation techniques are used to achieve these different granularities, including aggregation policies and rollups. In short, these strategies provide a summarized view of granular data over specific time intervals. Regardless of the techniques used, end users consume charts or alerts from this collected, analyzed, and queried data.

Several open source metric instrumentation frameworks and standards have become popular over the years. For example, the following solutions were popular in the monitoring era:

- StatsD (`https://github.com/statsd/statsd`)
- Graphite (`https://graphite.readthedocs.io/en/stable/overview.html`)