Software Performance Engineering

A comprehensive guide for high-performance development

Alon Rotem

www.bpbonline.com

First Edition 2025

ISBN: 978-93-65895-445

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true and correct to the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but the publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

Dedicated to

Antonia and *Benji*

About the Author

Alon Rotem is a soulful geek and musician. His encounters with code go back to his teenage years in the mid-1980s, where he discovered the ATARI 800 8-bit computers and the BASIC programming language. His days in the actual tech industry, all around software engineering, go back to the mid-1990s, the days of DOS, Windows 3.1, and prehistoric Red Hat distributions.

Since then, he has worked as a quality assurance engineer, a software engineer, a lecturer, an educational manager, a solutions consultant, a team lead, a tech lead, a solution enterprise architect and a senior engineer at one of the most well-known database companies, the **MariaDB Foundation**. He has been managing a team of software architects for one of the big four global accounting companies, **KPMG**. He also established and created a certification program for one of the biggest enterprise-level content management systems, **Sitefinity**, being one of the senior engineers who had built it firsthand for the most successful Bulgarian software company, **Telerik**.

Apart from his work, he is an active hacker and developer, always exploring technological solutions, workarounds, free alternatives, and hacks, and is an avid supporter of Linux and open-source software.

He studied computer science at the *Open University of Israel*, and in recent years has been living and working on both ends of East and West Europe.

He is also an electro-acoustic musician whose works can be found on all major streaming platforms, as well as on his personal site.

About the Reviewers

- **Martin Yanev** is a highly accomplished software engineer with nearly a decade of experience across diverse industries, including aerospace and medical technology. Throughout his illustrious career, Martin has carved out a niche for himself in developing and integrating cutting-edge software solutions for critical domains, including air traffic control and chromatography systems. Renowned as an esteemed instructor and computer science professor at Fitchburg State University, he possesses a deep understanding of the full spectrum of OpenAI APIs and exhibits mastery in constructing, training, and fine-tuning AI systems. As a widely recognized author, Martin has shared his expertise to help others navigate the complexities of AI development. With his exceptional track record and multifaceted skill set, Martin continues to propel innovation and drive transformative advancements in the field of software engineering.

- **Tareq** is a performance engineering senior architect with nearly 15 years of experience in quality engineering, including a decade specializing in performance engineering. He focuses on optimizing system performance, API tuning, and architectural design. His expertise spans tools like JMeter, LoadRunner, AppDynamics, DataDog, and New Relic. He was a speaker at the Xpand Conference 2024, where he talked about Observability in an Age of Microservices. He is an active reader, enjoying comics, manga, science fiction, and IT-related books, and continuously shares best practices and insights on software performance through various platforms.

Acknowledgement

As someone who has been deeply immersed in the tech industry for almost four decades, I have taken on many roles and undertaken a wide variety of tasks. I have written blog posts, recorded training videos, and tech videos, but I have never written an entire book. This was a golden opportunity, brought to me by BPB Publications, and I am immensely grateful for it. Their guidance and expertise in bringing this book to reality cannot be emphasized enough. Their patient support and assistance were invaluable in navigating the complexities of the publishing process. My experience was intriguing, interesting, and educational, all of which I am grateful for.

I would like to acknowledge the reviewers, technical experts, and editors who provided valuable feedback and contributed to the refinement of the book's contents through their insights and suggestions, which significantly improved its quality. Not to mention their patience with my constant corrections and additions, as technology has been running forward, even while I was researching and writing.

Special thanks to my literary editor, to whom I have given a whole lot of painstaking, fine-tuned work, which she has executed meticulously and flawlessly, and made this book so much better.

Lastly, I would like to express my gratitude to the readers who have shown interest in this book. Your support and encouragement are warmly appreciated.

Thank you to everyone who has played a part in making this book a reality.

Preface

Performance engineering is a broad and elusive subject on which several books can be written, and indeed, quite a few have been. Commonly, users do not give a lot of thought to how fast their app runs before they intuitively sense it is slow. Many moving parts contribute to how well an app performs: from the architecture of the CPU cores and the hardware setup, through the optimization of the code, the underlying runtime environment, efficiency of algorithms, data structures, databases, storages and strategies, all the way to the broad worldwide deployment and scaling schemes of services, networking components and security constraints. This renders performance and efficiency complex cross-cutting concerns throughout the entire process of producing and running software in our modern era.

Since ENIAC, the very first digital computer, was introduced just 80 years ago, in the 1940s, the performance and capabilities of computers have been evolving astronomically, in the endless race to accommodate ever-heavier tasks. Unlike in the old times, software today is expected to be operated by hundreds of millions of users, literally around the world, to process vast amounts of data records, while remaining pleasing, easy, and quick to use, all the way down to the single user's experience. The evolution of big data, with machine learning and artificial intelligence, keeps pushing even the most powerful supercomputers to the edge of their abilities, while on the other hand, quantum computers advance in lightning speed towards a completely new world of computing. We really are living in the future that science fiction has charted for us.

As the topic is so broad and versatile, this book covers some of the practical aspects of developing and delivering performant software in the modern world of technology, while mentioning other areas as knowledge pointers, in order to help direct the curious readers to acquire more knowledge, with the intention to provide a wholistic understanding of performance engineering, right from the standard processes of inception, production, testing and delivery, to the high level runtime view and analysis. This book is current. It talks about the past, but looks into the future, demonstrating current trends, tools, programming languages, frameworks, and platforms, to provide a strong base for people who are interested in learning the basics of performance engineering.

Chapter 1: Introduction to Performance Engineering- This chapter presents some of the aspects of performance engineering to understand its broad meaning in today's world, in order to acquaint the readers with the concept itself, which not many are aware

of, at least not in depth. We look at how it is integrated in our modern-day software development lifecycle processes, we discuss principles of modern software delivery, in light of the evolution of computers, and some of the performance concerns software must accommodate nowadays, and why they count.

Chapter 2: Performance Driven Development- In many ways, performance-oriented development is baked into the software delivery methodology itself. In this chapter, we get into more refined details of the current-day processes of software delivery. We look at the evolution of various methodologies, each with its advantages and flaws, and learn how applications are being planned, built, provisioned, and delivered with modern tools, in order to get the most out of them. We also mention different types of tests that help us evaluate the quality of our software and to understand how well it is performing in comparison to our expectations and requirements.

Chapter 3: Non-functional Requirements Definition and Tracking- Performance planning is a first-class citizen in the functionality of our application, how it is perceived, how it functions, how it reacts, and responds. However, it comes with a detailed underlying, well-defined set of requirements that we need to plan and take into account, attributes such as security, maintainability, and compliance, as well as considerations related directly to runtime performance features. This chapter looks at ways we can structure our non-functional requirements' definitions as part of our product plan.

Chapter 4: Workload Modeling and Projection- Continuing the analysis and breakdown of the features in our software, in this chapter, we look at how we identify and map use cases, usage flows, and workloads, in order to understand how and by whom our application is expected to be used. We discuss future load projection methods and performance measurements.

Chapter 5: High Performance Design Patterns- In this detailed technical chapter, we take a closer look at software and system design. We review the evolution of software architecture and approaches throughout the decades and how modern architectural design accommodates modern requirements. We discuss and review in detail several software design principles, which are directly related to scalability and performance enhancements.

Chapter 6: Performance Antipatterns- In continuation of the discussion of useful design patterns, in this chapter, we look at potential pitfalls of common design, which are sometimes overlooked by software architects and developers. Accompanied by concrete code examples, we take a deep technical look at a number of antipatterns, discuss their flaws, and why and how to avoid and mitigate them, in order to improve the performance and overall quality of our application.

Chapter 7: Performance in the Clouds- Cloud platforms are all the rage in modern software development and delivery. We discuss advancements in high-power computing, which is made accessible to all through the cloud, as well as a look into the future of quantum computing, which is, in fact, already the present, and becoming available as we speak. We discuss the advantages of running our microservice applications in the cloud, we look at scalability, elasticity, and large data management.

Chapter 8: Designing Performance Monitoring- Once we have our application running, we want to make sure it runs smoothly. Monitoring its performance metrics and telemetry, following detailed logs and runtime traces, gives us real-time insights into issues and helps us design and plan for improvements. This chapter explains the concepts, tools, and methodologies to monitor our app, to ensure it lives up to the promises we had made for it.

Chapter 9: Tools and Techniques for Code Profiling- Looking deeper into the code of our application, in this practical chapter, we take a technical dive into the analysis of our code, in development as well as during execution. We discuss static vs. dynamic profiling, we look at code profiling tools such as cProfile, pyinstrument, line profiler, VisualVM, pprof, and eBPF, by going through detailed code examples and walkthroughs, to understand how to put them to use in the real world.

Chapter 10: Performance Testing, Checklist to Best Practices- After learning about tools for optimizing our code, in this chapter, we look at what we can do once it's already built. Testing is a crucial pillar of software development and improvement, and performance counts for quality. In this chapter, we look at different types of performance tests, how they are executed, and the importance of test environments, conditions, and practices.

Chapter 11: Test Data Management- Still in the realm of testing, data is at the heart of any modern-day application. Providing proper quality data for testing is just as important as the test itself, and the area of test data management (TDM) has been marked as a core emerging technology by Gartner's hype cycle for Agile and DevOps. This chapter explains test data management, strategies for good quality test data, as well as practical demos for automated test data production, in order to achieve the most from our tests.

Chapter 12: Performance Benchmarking- Another aspect of understanding how well our application runs is benchmarking and execution analysis over time. In this chapter, we revisit the different types of performance tests while practically looking at runtime test tools, such as Locust and JMeter. We differentiate benchmarking from baselining and discuss the important aspect of continuous performance monitoring and validation, using an automation server (Jenkins) and containers (Docker).

Chapter 13: Golden Signals, KPI, Metrics, and Tools- As we run tests and benchmarks to monitor and get acquainted with our application's performance, we also want to refer to a well-defined set of measurable metrics, in order to actually know how well we are doing. In this chapter, we understand key performance indicators (KPIs). We map different types of metrics to different levels of roles; we discuss monitoring tools and take a practical example with the ELK stack and the Elastic Application Performance Monitoring suite.

Chapter 14: Performance Behavioral Correlation- Continuing the discussion on metrics and runtime monitoring, in this chapter, we look further into how to better understand the reports of our monitoring tools. While discussing practical examples, we investigate root cause analysis, data correlations, behavioral analytics, as well as strategies for future predictions, and practical code examples of mapping and charting them. We also talk about how to follow up on issues and the process of closing and completing them.

Chapter 15: Post-Production Management- As the previous chapters focused on delivering the software, making sure it complies with our requirements, code quality, performance definitions, test requirements, and runtime measurements, this chapter looks at the next day: once we have our app up and running in a live production environment. We look at managing dashboards and alerts, about the endless, continuous journey of improvement, we discuss the different stakeholders and different levels of ownership and responsibility, and briefly look at predictive analytics for the future.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/rdwrm8j

The code bundle for the book is also hosted on GitHub at
https://github.com/bpbpublications/Software-Performance-Engineering.
In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

1. Introduction to Performance Engineering 1
Introduction 1
Structure 1
Objectives 2
The story of performance engineering 2
About performance engineering 2
Modern principles of software engineering 4
The modern era 4
Development process principles and practices 6
Software design principles 8
Importance of performance engineering in software development 10
Objectives of performance engineering 12
Common performance issues in modern applications 13
Consequences of downtime and performance impact 16
How performance engineering solves problems that software brings in 17
Mindset for performance engineering adoption 18
Common challenges in performance engineering 19
Early problem detection and anticipating failures 20
Creating sustainability in the digital landscape 22
Conclusion 23
Key learnings 23

2. Performance Driven Development 25
Introduction 25
Structure 26
Objectives 26
Waterfall and Agile ways of delivering software 26
The Waterfall way 27
Advantages 28
Disadvantages 28
The Agile way 29
Advantages 31

Disadvantages ... 32
Scrum ... 32
DevOps ... 32
Plan for performance in sprints ... 34
The right tools at the right place ... 35
Process management and Agile collaboration ... 36
Jira by Atlassian, and tools for SDLC management ... 36
Confluence by Atlassian, or similar, for collaborative documentation ... 36
DevOps tools ... 37
CI/CD tools, Jenkins, or similar ... 37
Controlling infrastructure ... 38
Test automation ... 39
Unit tests ... 39
UI/integration/end-to-end tests ... 40
Performance testing ... 40
Load tests ... 41
Stress tests ... 41
Scalability tests ... 41
Endurance tests ... 41
Volume tests or flood tests ... 42
Backend and code-related tests ... 42
The tools for the job ... 42
Tools for load related tests ... 42
Tools for code execution measurement ... 43
Comprehensive tools ... 43
Requirement engineering ... 43
Design for performance and scalability ... 44
Code optimization ... 45
Performance validation and scalability optimization ... 46
Capacity planning ... 46
Performance engineering using Jenkins ... 47
Velocity of performance bug fixes ... 47
Roles and responsibilities of a performance engineer ... 48
Conclusion ... 49
Key learnings ... 49

3. Non-functional Requirements Definition and Tracking 51
Introduction 51
Structure 52
Objectives 52
Functional and non-functional requirements 52
Types of non-functional requirements 53
Performance *54*
Scalability *54*
Security *55*
Usability *57*
Reliability *58*
Maintainability *59*
Portability *59*
Compliance *60*
Defining attributes related to system performance 61
The user's perspective *63*
The system's perspective *63*
NFR template 64
Introduction and overview *65*
Stakeholders *65*
Definitions and acronyms *65*
System overview *65*
Non-functional requirements *65*
Requirement sections *66*
Dependencies *66*
Approval and sign-off *66*
Guidelines for defining NFRs 66
Managing NFRs through the development lifecycle 67
NFR tracing and management in GitLab 69
Conclusion 70
Key learnings 71

4. Workload Modeling and Projection 73
Introduction 73
Structure 74
Objectives 74

Identifying use cases in a system 74
Understanding use cases *74*
Identifying use cases *76*
The coffee break method *76*
The CRUD method *77*
Event decomposition method *78*
Inferring real use-cases *78*
Defining key business flows and users of a system 79
Key business flows *79*
User analysis *80*
Identifying workloads 81
Usage patterns based on demand and market study 83
Trend analysis 83
Pitfalls in performance measurement and projections 85
Modeling approaches for varying conditions 85
Projection algorithms for business growth and demand 86
Performance test planning 88
Conclusion 90
Key learnings 90

5. High Performance Design Patterns 93
Introduction 93
Structure 94
Objectives 94
Different types of software architectures 94
A brief history of software complexity and architecture *95*
Architectural trends *98*
Design principles 99
Abstraction *99*
SOLID principles *100*
KISS and DRY *100*
Decoupling *101*
Law of least astonishment *101*
Hidden aspects of running software on virtual machines vs. containers 102
Legacy monolithic architecture vs. microservices 104
Design patterns for performance 106

Cache aside ... *106*
About caching ... *106*
Cache aside ... *107*
Additional considerations ... *108*
Command query responsibility segregation ... *108*
Queue-based throttling ... *111*
Sharding ... *112*
The need for data sharding ... *112*
Enter sharding ... *112*
Challenges of sharding ... *113*
Design patterns for scalability ... 114
Lazy loading ... *114*
Partitioning ... *115*
Horizontal partitioning ... *116*
Vertical partitioning ... *116*
Defining a partitioning key ... *117*
Load balancing ... *117*
Asynchrony ... *118*
Design patterns for high-availability ... 119
Circuit breaker ... *120*
Redundancy ... *121*
Decoupling ... *122*
Dynamically scalable architectures ... 123
Cloud-native designs ... 124
Highly scalable datastores ... 125
Conclusion ... 125
Key learnings ... 126

6. Performance Antipatterns ... 127
Introduction ... 127
Structure ... 127
Objectives ... 128
Introduction to antipatterns ... 128
Performance antipatterns overview ... 129
God object antipattern ... 130
Tight coupling antipattern ... 131

Premature optimization antipattern 137
The root of all evil *137*
Prioritize performance optimizations properly *138*
Blob antipattern 140
Chatty communication antipattern 141
Global interpreter lock antipattern 144
Python and CPython *144*
Disadvantages of GIL *145*
Busy waiting antipattern 146
Refactoring and optimizing performance antipatterns 148
Common pitfalls and best practices 149
Conclusion 150
Key learnings 150

7. Performance in the Clouds 151
Introduction 151
Structure 151
Objectives 152
Architectural considerations 152
High performance computing *152*
Advancements in quantum computing *154*
Performance considerations *157*
Tenancy and virtualization *157*
Infinite options *158*
Cloud-native services *158*
Managed apps *159*
Application architectural considerations *160*
The 12-factor principle of a modern app *160*
Scalability and elasticity 162
Scalability *162*
Elasticity *163*
Microservices performance challenges 165
With great power comes a great electricity bill *165*
Communication issues *165*
Growing complexity of breaking the monolith *166*
Integration overhead and dependency hell *166*

Cloud-native performance optimization 167
Fine orchestration *167*
Migrating to the cloud *167*
Cloud-native apps *167*
The Cloud Native Computing Foundation *168*
Computing performance optimization *168*
Networking performance optimization *169*
Data management and storage 170
Conclusion 172
Key learnings 172

8. Designing Performance Monitoring 173
Introduction 173
Structure 173
Objectives 174
Concepts and tooling 174
Logging *174*
Telemetry *175*
Instrumentation *175*
Application performance monitoring *175*
Monitoring and observability *176*
Benchmarking *176*
Common deployment architectures 177
Infrastructure and software limitations 180
Key components in the architecture 181
Metrics, events, logs, and traces 182
Events *182*
Metrics *183*
Logs *183*
Traces *184*
AWS Distro for OpenTelemetry 186
OpenTelemetry *186*
AWS Distro for OpenTelemetry *188*
List of key attributes to measure from each component 189
General machine metrics *189*
Data collectors and aggregators 190

Data aggregation and processing layer *191*
Data aggregators *191*
Data transformation and normalization *192*
Application performance management 192
Anomaly detection and suspect ranking 192
Predictive analytics 194
User load *195*
Response time *195*
Infrastructure capacity *195*
Conclusion 196
Key learnings 196

9. Tools and Techniques for Code Profiling 197
Introduction 197
Structure 197
Objectives 198
Static vs. dynamic profiling 198
Static code analysis *198*
Dynamic code profiling *199*
Example with Python's cProfile and pyinstrument *200*
Profiler collection methods 204
Sampling *204*
Instrumentation 206
Line profiling *206*
Continuous profiling *207*
Choosing the right profiling tool 207
Common challenges 208
Overhead *209*
Sampling vs. instrumentation *209*
Complexity of analysis, debugging, and troubleshooting *209*
Resource constraints *210*
Profiling across distributed systems *210*
Security and privacy concerns *210*
Profile code and runtime with VisualVM 210
Continuous profiling using pprof 213
Golang *213*

protobuf *214*
pprof *214*
Continuous profiling *217*
Linux kernel profiling using eBPF 218
Automation for gathering profiler snapshots 222
Code profiling best practices 223
Conclusion 224
Key learnings 225

10. Performance Testing, Checklist to Best Practices 227
Introduction 227
Structure 227
Objectives 228
Performance validation checklist 228
System analysis *228*
Test plan *228*
Test execution *229*
Report *229*
Script development 230
Leverage functional test suite to create JMeter scripts 232
Workload model to scenario mapping 234
Environment preparation 236
Production vs. performance test environments 237
Production environment *237*
Performance test environment *238*
Test environment considerations *239*
Performance testing tools 239
The right tool for the job *239*
Mentionable performance testing tools *241*
Performance testing best practices 242
Conclusion 243
Key learnings 243

11. Test Data Management 245
Introduction 245
Structure 246

Objectives ... 246
Test data requirement definition ... 246
Test data characteristics classification ... 248
Garbage in, garbage out: About data quality ... 248
Additional data characteristics ... 248
Strategy to setup test data ... 249
Test data management ... 250
Relying on production data ... 252
Test data warehouse ... 252
Synthesized data ... 253
Test data clean-up and reuse ... 253
Service virtualization ... 255
Principle ... 255
Benefits ... 256
Automation ... 256
Large-scale tooling ... 257
Open options ... 257
Self developed scripts ... 258
Data security ... 261
Conclusion ... 263
Key learnings ... 263

12. Performance Benchmarking ... 265
Introduction ... 265
Structure ... 265
Objectives ... 266
Types of performance testing ... 266
Smoke testing ... 266
Different types of initial validation tests ... 266
Aspects of smoke tests ... 267
Examples ... 268
Single-user isolation testing ... 268
Load testing ... 270
Stress testing ... 275
Volume testing ... 277
Endurance testing ... 278

Readiness checklist 279
Scenario setup 282
Performance baselining and benchmarking 283
Benchmarks: Someone already did it! *283*
Baselines: Someone already did it! That was you! *285*
Continuous performance validation using Jenkins and JMeter 285
Continuous deliveries *285*
Jenkins FTW *286*
Continuous performance *286*
Pre-setup of JMeter and a web server *287*
Jenkins *289*
Creating and running a performance pipeline *291*
Conclusion 295
Key learnings 295

13. Golden Signals, KPI, Metrics, and Tools 297
Introduction 297
Structure 298
Objectives 298
Key performance indicators 298
Different metrics for different participants 300
Engineers *301*
Architects *301*
Business analysts *301*
Executives *302*
Infrastructure components and usage management 302
Application runtime vs. behavior metrics 303
Application runtime *303*
Behavior metrics *304*
Key differences *304*
White box and black box monitoring 305
Black box monitoring *305*
White box monitoring *306*
Good to know statistics for third-party hosted content 306
Performance monitoring with the open Elastic APM 307
ELK stack *308*

Additional notable tools *309*
Elastic APM *309*
Making it all work *310*
Conclusion 315
Key learnings 315

14. Performance Behavioral Correlation 317
Introduction 317
Structure 317
Objectives 318
Common scenarios and root causes 318
Root cause and root cause analysis *318*
Digging into common root cases with the fishbone diagram *319*
Analysis steps *321*
Common scenarios and root causes *322*
False positives and false negatives 323
The confusion matrix *323*
False positive *325*
False negative *325*
Correlation and suspect ranking 325
Correlation vs. causation *325*
Suspect ranking *327*
Example 1 Web application performance degradation *328*
Example 2 High error rates in API *328*
Example 3 Database performance issues *329*
Behavioral pattern analysis 329
Analytics types *329*
User behavior analysis *330*
User and entity behavior analytics *331*
Concluding actionable outcomes 332
Actionable analytics *332*
Example flow, from definitions to actions *332*
Trending analysis 335
Statistical methods *335*
Machine learning models *336*
Example flow *337*

Defect tracking and closure 344
Tracking *344*
Resolution and closure *345*
The definition of done *345*
Conclusion 346
Key learnings 346

15. Post-Production Management 347
Introduction 347
Structure 347
Objectives 348
Alerting and dashboarding 348
Dashboarding best practices and key features *349*
Alerting best practices and key features *350*
Learn from incidents 352
Fail fast, learn fast *352*
Learning from Incidents movement *353*
LFI vs. the traditional approach *354*
Continuous improvement journey 355
The Deming Cycle *356*
Kaizen *356*
Identifying key stakeholders 357
Defining and agreeing the level of ownership 359
RACI key roles *359*
Ownership *360*
Performance engineering culture across teams 361
Embracing the culture *363*
Predictive analytics and projections 364
Reporting progress to key stakeholders 366
Conclusion 367
Points to remember 367

Index 369-382

Chapter 1
Introduction to Performance Engineering

Introduction

Many people in the IT world know about software engineering and the practices and roles that take part in building a software project. Many also know about performance and why it is important, although this aspect often gets sidelined and not properly addressed.

However, not many know about performance engineering as a thing, all the more so as a field with specialized professionals and structured methodologies.

Performance engineering, nevertheless, is a wide field with many faces, which touches on many aspects and concepts. Some of which are project management, system design, software engineering practices, hardware architecture, testing and automation, and others. This chapter introduces the tip of the iceberg of performance engineering before going deeper into explanations.

Structure

This chapter will cover the following topics:

- The story of performance engineering
- Modern principles of software engineering

Objectives

In this chapter, we will get acquainted with the concept of performance engineering in the light of software engineering. We will briefly review modern concepts and practices of software development and project management. We will understand what performance engineering is, why it is important, what its challenges, objectives, and risks (as well as the risks of neglecting it) are, and what we can benefit from it.

The story of performance engineering

Putting first things first, let us first understand what exactly we are talking about here, what performance engineering is.

About performance engineering

Intuitively speaking, proper **engineering** (i.e., designing, outlining, implementing, testing, and delivering) of software, considering its **performance**. That is, how well it executes under various conditions, mostly in terms of (but not limited to) speed and efficiency.

In contrast to classic methodologies of software engineering, with their patterns and rules, where it is clear who (the developer) does what (writes the code), performance engineering is a broad set of processes and techniques that are to be applied during the entire software development lifecycle. It is rooted in processes, people, and technologies, and has an impact on the optimization of an application's performance prior to product deployment, as well as following up on it afterwards.

Alongside a software's feasible, operational, user-facing interactive **functional** features, its buttons, menus, and interactive and responsive behavior, there is often a list of **non-functional** requirements: supporting features that are transparent to the user but are required to keep the app and its data stable, reliable, durable, and secure.

Performance is a non-functional feature, which is sometimes included in the list of requirements, and sometimes omitted. Concerns such as how much time a page should take to load, how quickly data should be retrieved, what should be the throughput of records per second, how it should handle multiple users and heavy load, etc.

As a small side note from my own personal experience on industrial software projects over more than 27 years, some software applications, like those that deal with finance and regulations, come with hard, specific performance requirements, as processed data needs to be delivered at a specific rate, at specific times. Some systems require the data delivery to be instantaneous (or at least pseudo-instantaneous). Those are intrinsic performance features that come pre-baked in the product's list of requirements, as the integrity and regulatory nature of the system rely on it. Of course, in many cases, performance is *not the most important* aspect of the software. Other properties, such as compatibility, functionality, maintainability, modularity, profitability, and usability, are at the base of any application,

as they are its selling points. But performance is sometimes compared to a currency, with which we can *trade* other properties of the application. For example, we can sacrifice performance in favor of making the code more readable, or sacrifice performance in order to make sure our program is secure, etc.

Speaking of currencies, putting efforts into improving performance does translate directly to money, as it may take more time and manpower to build highly performant code, as well as compatible tests and monitoring, and integrate the whole chunk of work into the development process.

On the flip side, in many cases, a software project is born from a functional idea, and not a regulatory need. Thus, the focus of the development process is to make the dream and vision come to life. This leans mostly on *what* we want the app to do, rather than *how, how well*, or *how fast*. Building software in this mindset is incredibly common, and performance comes last on the list of features, if at all. Considering performance as a feature means utilizing efforts and resources, which, in many cases, is transparent to the developers, too. They would rather put their work into tangible, functional fruits, and sometimes think of performance as an extra that may not be worth their time, or do not think about it at all. Project managers await visible features they can show and talk about, and many people in the development circle would consider working on performance features a needless hassle, until they have to.

Life is short; time is money. On the other side of the application sits a user, staring at progress bars (which are made to comfort them that everything is fine: things are happening, work is being done), waiting for data to be submitted or retrieved, or worse: staring at a blank, unresponsive page. While performance is not at the top of the list of many application developers, it is right in front of the users' eyes, and, depending on the complexity of the application, it is very noticeable.

Given this common progression, it is not uncommon practice to start noticing and considering performance when the project already stands, and not while planning and designing it. This can be metaphorically compared to making fundamental construction changes to a house while planning and building it, vs. a renovation, after it is already built and standing. Doing this in advance can save a lot of trouble and extra work and produce a smoother, holistically better-integrated result.

Hence, considering performance engineering as part of the initial requirements and integrating performance practices and relevant quality tests into the software development lifecycle may be beneficial, even if performance is not one of the topmost considered priorities for the project, especially if future scale and growth are desired.

There are a few (albeit not too many) books, tutorials, and courses discussing the topic of performance engineering out there, and it is indeed a wide topic, to say the least. Spanning from hardware and processor architecture and utilization, through coding principles, patterns, and antipatterns, programming language-specific tricks and pitfalls, design of

services and cloud integration, various types of tests and benchmarks, tools for analysis and monitoring, and more.

This book will provide a modern view of performance engineering aimed at the current day's developers in mind. It will discuss common patterns, solutions, and methodologies. Performance is a theory with many faces, and hopefully, this book will put the perplexed software engineer in the right mindset, using simple, down-to-earth words.

Modern principles of software engineering

Software has existed since the beginning of digital computers. In the beginning (the 1940s), software was written with binary code, directly to the heart of the computer, which was a room-sized mainframe (the term **mainframe** comes from the large cabinet which was housing the computer, the main frame). Binary code is still used today, as this is the only language computer processors can understand, but it is written indirectly. Layers of compilation and interpretation separate the code the developer writes from what the processor eventually reads. Those layers create much more sophisticated and elegant ways to model our programs and build them in a more natural linguistic manner (mixed with mathematical logic), and programming languages, much like natural human languages, evolve and grow, become more elaborate, creating a universe of frameworks, methodologies, and practices of design and implementation. The computer processors have also grown immensely in capabilities, and smaller in size, to say the least.

In this segment, we will look a bit into this evolution and where it has brought us to today.

The modern era

The ever-evolving craft of software engineering has grown over the years into a wide variety of programming languages, frameworks, development and runtime environments, technologies, techniques, tools, and purposes.

Some languages are driven by object-oriented design; others are procedural. Some are strongly typed, while others are completely fluid. Some are interpreted, making them executable on multiple platforms, while others are compiled into binary executables for a specific operating system on a specific processor. Some would argue that declarative languages may not be programming languages at all, but in effect, they certainly are (just to add to the mix).

While the Assembly programming language has always been (and will probably remain) the closest wrapper representation to actual binary machine code, it too has evolved quite a lot over the years, as computer processors and their respective architectures have. Going through the specs of Assembly language keywords for x86 processors will reveal a long list of added instructions on every consequent generation.

Processors themselves have been keeping up nicely with Moore's law in the past decades. In 1965, *Gordon Moore* predicted that the number of transistors in an integrated circuit would double every 18-24 months (thereby increasing processing power exponentially). This prediction of growth, referred to as *Moore's law*, has been consistently confirmed over the years. If we compare the first microprocessor, Intel's 4004 from 1971 with 2,300 transistors and a clock speed of 750KHz, to today's latest technology of a TSMC's N3 processor, with the modern advanced 3-nanometer technology, with more than 314 million transistors, and clock speed of 3.16GHz, we see a growth of density of more than 136,000 times, and speed of more than 4,000 times. This rate of growth is incomparable to any other technology in any field.

It is not uncommon to argue that Moore's law has reached its end of potential, as modern processors are already pushing the limits of physics itself. The density of transistors is already literally bordering mere atomic scale, resulting in potential temperatures as hot as the surface of the sun (or much worse: the temperature of a slice of tomato inside a pressed grilled cheese sandwich!). In light of the physical limits of the processor itself, modern-day computers' CPUs are getting not only dense in themselves, but each modern CPU now packs multiple processors (cores) at once and uses additional architectural tricks, such as various data caches, in order to get as performant as possible.

Moore's law is well known and mentioned everywhere. Let us look at yet another, less-known law related to microprocessor's evolution: *Dennard scaling*, established in 1974 by *Robert Dennard*; in simple words, it states that as the transistors get smaller, their density on the chip grows, but the power consumption per surface area remains the same. This means that despite the greater (to say the least) processing horsepower, electric power consumption does not grow remotely as much.

Alongside all those evolutions, other shifts have taken place. Software projects have grown bigger in scale (with the potential to scale-grow indefinitely upon need). Data has grown tremendously, and the collection of big data has allowed the creation of enormous data models to be crunched by machine learning algorithms and artificial intelligence engines. Software still runs on the user's computer, but also inside their web browsers, on remote servers, in large cloud clusters of computers around the globe, inside databases, on mobile devices, on smart home appliances' microcontrollers, and in IoT. Software is not just for nerds who are hacking the local traffic lights system from their moms' basements, but for everyone, everywhere, all at once.

Interestingly enough, some companies, running large data centers, chose to place them in naturally cold locations, such as *Canada* (*CLUMEQ silo* in *Quebec*), *Scandinavia* (Google's Hamina data center in *Finland*, Facebook's data center near the *Arctic Circle* in *Lulea, Sweden*), to help the hardware cool while lowering energy costs.

Amid the rapid and ongoing advancements in both software and hardware, we have yet to even approach the early stages of quantum computing—an area that once seemed like a distant vision but is now steadily becoming a reality.