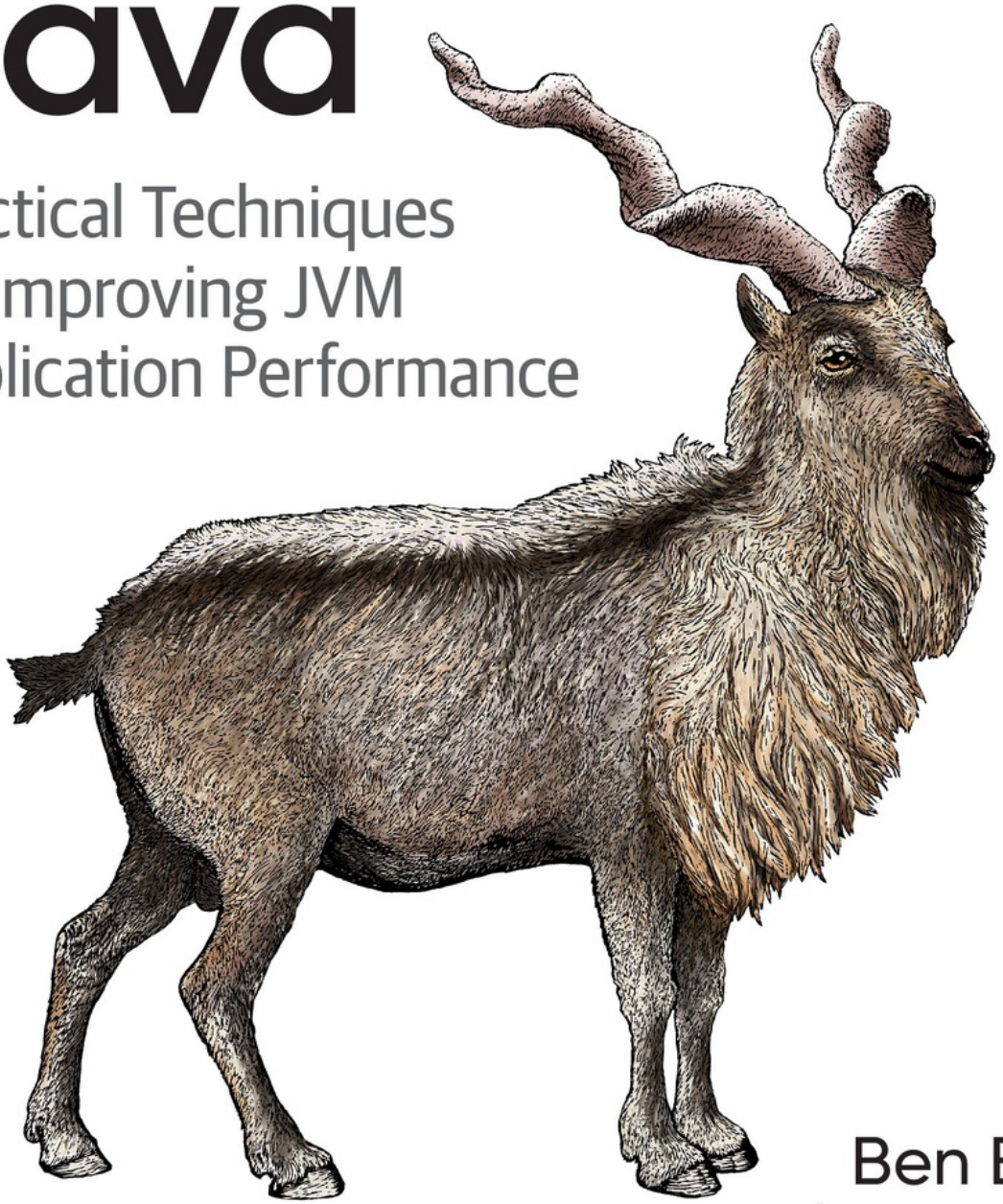
O'REILLY®

Second Edition

Optimizing Cloud Native Java

Practical Techniques for Improving JVM Application Performance



Ben Evans &
James Gough

Optimizing Cloud Native Java

SECOND EDITION

Practical Techniques for Improving JVM Application Performance

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Ben Evans and James Gough

Beijing • Boston • Farnham • Sebastopol • Tokyo

Optimizing Cloud Native Java

by Ben Evans and James Gough

Printed in the United States of America.

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

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

Acquisitions Editor: Brian Guerin

Development Editor: Rita Fernando

Production Editor: Ashley Stussy

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

October 2024: Second Edition

Revision History for the Early Release

- 2023-11-03: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098149345 for release details.

978-1-098-14928-4

[LSI]

Chapter 1. Optimization and Performance Defined

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 1st chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *rfernando@oreilly.com*.

Optimizing the performance of Java (or any other sort of code) is often seen as a Dark Art. There's a mystique about performance analysis—it's commonly viewed as a craft practiced by the "lone hacker, who is tortured and deep thinking" (one of Hollywood's favorite tropes about computers and the people who operate them). The image is one of a single individual who can see deeply into a system and come up with a magic solution that makes the system work faster.

This image is often coupled with the unfortunate (but all-too-common) situation where performance is a second-class concern of the software teams. This sets up a scenario where analysis is only done once the system is already in trouble, and so needs a performance "hero" to save it. The reality, however, is a little different.

The truth is that performance analysis is a weird blend of hard empiricism and squishy human psychology. What matters is, at one and the same time, the absolute numbers of observable metrics and how the end users and

stakeholders *feel* about them. The resolution of this apparent paradox is the subject of the rest of this book.

Since the publication of the First Edition, this situation has only sharpened. As more and more workloads move into the cloud, and as systems become ever-more complicated, the strange brew that combines very different factors has become even more important and prevalent. The "domain of concern" that an engineer who cares about performance needs to operate in has continued to broaden.

This is because production systems have become even more complicated. More of them now have aspects of distributed systems to consider in addition to the performance of individual application processes. As system architectures become larger and more complex, the number of engineers who must concern themselves with performance has also increased.

The new edition of this book responds to these changes in our industry by providing four things:

- A necessary deep-dive on the performance of application code running within a single-JVM
- A discussion of JVM internals
- Details of how the modern cloud stack interacts with Java / JVM applications
- A first look at the behavior of Java applications running on a cluster in a cloud environment

In this chapter, we will get going by setting the stage with some definitions and establishing a framework for *how* we talk about performance—starting with some problems and pitfalls that plague many discussions of Java performance.

Java Performance—The Wrong Way

For many years, one of the top three hits on Google for "Java performance

tuning” was an article from 1997–8, which had been ingested into the index very early in Google’s history. The page had presumably stayed close to the top because its initial ranking served to actively drive traffic to it, creating a feedback loop.

The page housed advice that was completely out of date, no longer true, and in many cases detrimental to applications. However, its favored position in the search engine results caused many, many developers to be exposed to terrible advice.

For example, very early versions of Java had terrible method dispatch performance. As a workaround, some Java developers advocated avoiding small methods and instead writing monolithic methods. Of course, over time, the performance of virtual dispatch greatly improved.

Not only that, but with modern JVM technologies (especially automatic managed inlining), virtual dispatch has now been eliminated at a large number—perhaps even the majority—of call sites. Code that followed the “lump everything into one method” advice is now at a substantial disadvantage, as it is very unfriendly to modern Just-in-Time (JIT) compilers.

There’s no way of knowing how much damage was done to the performance of applications that were subjected to the bad advice, but this case neatly demonstrates the dangers of not using a quantitative and verifiable approach to performance. It also provides yet another excellent example of why you shouldn’t believe everything you read on the internet.

> **NOTE**
>
> The execution speed of Java code is highly dynamic and fundamentally depends on the underlying Java Virtual Machine. An old piece of Java code may well execute faster on a more recent JVM, even without recompiling the Java source code.

As you might imagine, for this reason (and others we will discuss later) this book is not a cookbook of performance tips to apply to your code. Instead, we focus on a range of aspects that come together to produce good

performance engineering:

- Performance methodology within the overall software lifecycle
- Theory of testing as applied to performance
- Measurement, statistics, and tooling
- Analysis skills (both systems and data)
- Underlying technology and mechanisms

By bringing these aspects together, the intention is to help you build an understanding that can be broadly applied to whatever performance circumstances that you may face.

Later in the book, we will introduce some heuristics and code-level techniques for optimization, but these all come with caveats and tradeoffs that the developer should be aware of before using them.

> **TIP**
>
> Please do not skip ahead to those sections and start applying the techniques detailed without properly understanding the context in which the advice is given. All of these techniques are capable of doing more harm than good if you lack a proper understanding of how they should be applied.

In general, there are:

- No magic “go faster” switches for the JVM
- No “tips and tricks” to make Java run faster
- No secret algorithms that have been hidden from you

As we explore our subject, we will discuss these misconceptions in more detail, along with some other common mistakes that developers often make when approaching Java performance analysis and related issues.

Our “No Tips and Tricks” approach extends to our coverage of cloud

techniques. You will not find virtually any discussion of the vendor-specific techniques present on the cloud hyperscalars (AWS, Azure, GCP, OpenShift, and so on). This is for two main reasons:

- It would expand the scope of the book and make it unmanageably long
- It is impossible to stay current with such a large topic area

The progress made by teams working on those products would make any detailed information about them out-of-date by the time the book is published. So, instead, in the cloud chapters, we focus on fundamentals and patterns, which remain effective regardless of which hyperscalar your applications are deployed upon.

Still here? Good. Then let's talk about performance.

Java Performance Overview

To understand why Java performance is the way that it is, let's start by considering a classic quote from James Gosling, the creator of Java:

> *Java is a blue collar language. It's not PhD thesis material but a language for a job.*[1]
>
> —James Gosling

That is, Java has always been an extremely practical language. Its attitude to performance was initially that as long as the environment was *fast enough*, then raw performance could be sacrificed if developer productivity benefited. It was therefore not until relatively recently, with the increasing maturity and sophistication of JVMs such as HotSpot, that the Java environment became suitable for high-performance computing applications.

This practicality manifests itself in many ways in the Java platform, but one of the most obvious is the use of *managed subsystems*. The idea is that the developer gives up some aspects of low-level control in exchange for not having to worry about some of the details of the capability under management.

The most obvious example of this is, of course, memory management. The JVM provides automatic memory management in the form of a pluggable *garbage collection* subsystem (usually referred to as GC), so that memory does not have to be manually tracked by the programmer.

> **NOTE**
>
> Managed subsystems occur throughout the JVM and their existence introduces extra complexity into the runtime behavior of JVM applications.

As we will discuss in the next section, the complex runtime behavior of JVM applications requires us to treat our applications as experiments under test. This leads us to think about the statistics of observed measurements, and here we make an unfortunate discovery.

The observed performance measurements of JVM applications are very often not normally distributed. This means that elementary statistical techniques (especially *standard deviation* and *variance* for example) are ill-suited for handling results from JVM applications. This is because many basic statistics methods contain an implicit assumption about the normality of results distributions.

One way to understand this is that for JVM applications outliers can be very significant—for a low-latency trading application, for example. This means that sampling of measurements is also problematic, as it can easily miss the exact events that have the most importance.

Finally, a word of caution. It is very easy to be misled by Java performance measurements. The complexity of the environment means that it is very hard to isolate individual aspects of the system.

Measurement also has an overhead, and frequent sampling (or recording every result) can have an observable impact on the performance numbers being recorded. The nature of Java performance numbers requires a certain amount of statistical sophistication, and naive techniques frequently produce incorrect results when applied to Java/JVM applications.

These concerns also resonate into the domain of cloud native applications. Automatic management of applications has very much become part of the cloud native experience—especially with the rise of technologies such as Kubernetes. The need to balance the cost of collecting data with the need to collect enough to make conclusions is also an important architectural concern for cloud native apps—we will have more to say about that in Chapter 10.

Performance as an Experimental Science

Java/JVM software stacks are, like most modern software systems, very complex. In fact, due to the highly optimizing and adaptive nature of the JVM, production systems built on top of the JVM can have some subtle and intricate performance behavior. This complexity has been made possible by Moore's Law and the unprecedented growth in hardware capability that it represents.

> *The most amazing achievement of the computer software industry is its continuing cancellation of the steady and staggering gains made by the computer hardware industry.*
>
> —Henry Petroski (attr)

While some software systems have squandered the historical gains of the industry, the JVM represents something of an engineering triumph. Since its inception in the late 1990s the JVM has developed into a very high-performance, general-purpose execution environment that puts those gains to very good use.

The tradeoff, however, is that like any complex, high-performance system, the JVM requires a measure of skill and experience to get the absolute best out of it.

> *A measurement not clearly defined is worse than useless.*[2]
>
> —Eli Goldratt

JVM performance tuning is therefore a synthesis between technology, methodology, measurable quantities, and tools. Its aim is to effect measurable

outputs in a manner desired by the owners or users of a system. In other words, performance is an experimental science—it achieves a desired result by:

- Defining the desired outcome
- Measuring the existing system
- Determining what is to be done to achieve the requirement
- Undertaking an improvement exercise
- Retesting
- Determining whether the goal has been achieved

The process of defining and determining desired performance outcomes builds a set of quantitative objectives. It is important to establish what should be measured and record the objectives, which then form part of the project's artifacts and deliverables. From this, we can see that performance analysis is based upon defining, and then achieving, nonfunctional requirements.

This process is, as has been previewed, not one of reading chicken entrails or another divination method. Instead, we rely upon statistics and an appropriate handling (and interpretation) of results.

In this chapter, we discuss these techniques as they apply to a single JVM. In Chapter 2 we will introduce a primer on the basic statistical techniques that are required for accurate handling of data generated from a JVM performance analysis project. Later on, primarily in Chapter 10, we will discuss how these techniques generalize to a clustered application and give rise to the notion of Observability.

It is important to recognize that, for many real-world projects, a more sophisticated understanding of data and statistics will undoubtedly be required. You are therefore encouraged to view the statistical techniques found in this book as a starting point, rather than a definitive statement.

A Taxonomy for Performance

In this section, we introduce some basic observable quantities for performance analysis. These provide a vocabulary for performance analysis and will allow you to frame the objectives of a tuning project in quantitative terms. These objectives are the nonfunctional requirements that define performance goals. Note that these quantities are not necessarily directly available in all cases, and some may require some work to obtain from the raw numbers obtained from our system.

One common basic set of performance observables is:

- Throughput
- Latency
- Capacity
- Utilization
- Efficiency
- Scalability
- Degradation

We will briefly discuss each in turn. Note that for most performance projects, not every metric will be optimized simultaneously. The case of only a few metrics being improved in a single performance iteration is far more common, and this may be as many as can be tuned at once. In real-world projects, it may well be the case that optimizing one metric comes at the detriment of another metric or group of metrics.

Throughput

Throughput is a metric that represents the rate of work a system or subsystem can perform. This is usually expressed as number of units of work in some time period. For example, we might be interested in how many transactions per second a system can execute.

For the throughput number to be meaningful in a real performance exercise, it should include a description of the reference platform it was obtained on. For example, the hardware spec, OS, and software stack are all relevant to throughput, as is whether the system under test is a single server or a cluster. In addition, transactions (or units of work) should be the same between tests. Essentially, we should seek to ensure that the workload for throughput tests is kept consistent between runs.

Performance metrics are sometimes explained via metaphors that evoke plumbing. If we adopt this view point then, if a water pipe can produce 100 liters per second, then the volume produced in 1 second (100 liters) is the throughput. Note that this value is a function of the speed of the water and the cross-sectional area of the pipe.

Latency

To continue the metaphor of the previous section—latency is how long it takes a given liter to traverse the pipe. This is a function of both the length of the pipe and how quickly the water is moving through it. It is not, however, a function of the diameter of the pipe.

In software, latency is normally quoted as an end-to-end time—the time taken to process a single transaction and see a result. It is dependent on workload, so a common approach is to produce a graph showing latency as a function of increasing workload. We will see an example of this type of graph in “Reading Performance Graphs”.

Capacity

The capacity is the amount of work parallelism a system possesses—that is, the number of units of work (e.g., transactions) that can be simultaneously ongoing in the system.

Capacity is obviously related to throughput, and we should expect that as the concurrent load on a system increases, throughput (and latency) will be affected. For this reason, capacity is usually quoted as the processing available at a given value of latency or throughput.

Utilization

One of the most common performance analysis tasks is to achieve efficient use of a system's resources. Ideally, CPUs should be used for handling units of work, rather than being idle (or spending time handling OS or other housekeeping tasks).

Depending on the workload, there can be a huge difference between the utilization levels of different resources. For example, a computation-intensive workload (such as graphics processing or encryption) may be running at close to 100% CPU but only be using a small percentage of available memory.

As well as CPU, other resources types—such as network, memory, and (sometimes) the storage I/O subsystem—are becoming important resources to manage in cloud-native applications. For many applications, more memory than CPU is "wasted", and for many microservices network traffic has become the real bottleneck.

Efficiency

Dividing the throughput of a system by the utilized resources gives a measure of the overall efficiency of the system. Intuitively, this makes sense, as requiring more resources to produce the same throughput is one useful definition of being less efficient.

It is also possible, when one is dealing with larger systems, to use a form of cost accounting to measure efficiency. If solution A has a total cost of ownership (TCO) twice that of solution B for the same throughput then it is, clearly, half as efficient.

Scalability

The throughput or capacity of a system of course depends upon the resources available for processing. The scalability of a system or application can be defined in several ways—but one useful one is as the change in throughput as resources are added. The holy grail of system scalability is to have

throughput change exactly in step with resources.

Consider a system based on a cluster of servers. If the cluster is expanded, for example, by doubling in size, then what throughput can be achieved? If the new cluster can handle twice the volume of transactions, then the system is exhibiting “perfect linear scaling.” This is very difficult to achieve in practice, especially over a wide range of possible loads.

System scalability is dependent upon a number of factors, and is not normally a simple linear relationship. It is very common for a system to scale close to linearly for some range of resources, but then at higher loads to encounter some limitation that prevents perfect scaling.

Degradation

If we increase the load on a system, either by increasing the rate at which requests arrive or the size of the individual requests, then we may see a change in the observed latency and/or throughput.

Note that this change is dependent on utilization. If the system is underutilized, then there should be some slack before observables change, but if resources are fully utilized then we would expect to see throughput stop increasing, or latency increase. These changes are usually called the degradation of the system under additional load.

Correlations Between the Observables

The behavior of the various performance observables is usually connected in some manner. The details of this connection will depend upon whether the system is running at peak utility. For example, in general, the utilization will change as the load on a system increases. However, if the system is underutilized, then increasing load may not appreciably increase utilization. Conversely, if the system is already stressed, then the effect of increasing load may be felt in another observable.

As another example, scalability and degradation both represent the change in behavior of a system as more load is added. For scalability, as the load is

increased, so are available resources, and the central question is whether the system can make use of them. On the other hand, if load is added but additional resources are not provided, degradation of some performance observable (e.g., latency) is the expected outcome.

> **NOTE**
>
> In rare cases, additional load can cause counterintuitive results. For example, if the change in load causes some part of the system to switch to a more resource-intensive but higher-performance mode, then the overall effect can be to reduce latency, even though more requests are being received.

To take one example, in Chapter 6 we will discuss HotSpot's JIT compiler in detail. To be considered eligible for JIT compilation, a method has to be executed in interpreted mode "sufficiently frequently." So it is possible at low load to have key methods stuck in interpreted mode, but for those to become eligible for compilation at higher loads due to increased calling frequency on the methods. This causes later calls to the same method to run much, much faster than earlier executions.

Different workloads can have very different characteristics. For example, a trade on the financial markets, viewed end to end, may have an execution time (i.e., latency) of hours or even days. However, millions of them may be in progress at a major bank at any given time. Thus, the capacity of the system is very large, but the latency is also large.

However, let's consider only a single subsystem within the bank. The matching of a buyer and a seller (which is essentially the parties agreeing on a price) is known as *order matching*. This individual subsystem may have only hundreds of pending orders at any given time, but the latency from order acceptance to completed match may be as little as 1 millisecond (or even less in the case of "low-latency" trading).

In this section we have met the most frequently encountered performance observables. Occasionally slightly different definitions, or even different metrics, are used, but in most cases these will be the basic system numbers

that will normally be used to guide performance tuning, and act as a taxonomy for discussing the performance of systems of interest.

Reading Performance Graphs

To conclude this chapter, let's look at some common patterns of behavior that occur in performance tests. We will explore these by looking at graphs of real observables, and we will encounter many other examples of graphs of our data as we proceed.

The graph in Figure 1-1 shows sudden, unexpected degradation of performance (in this case, latency) under increasing load—commonly called a *performance elbow*.

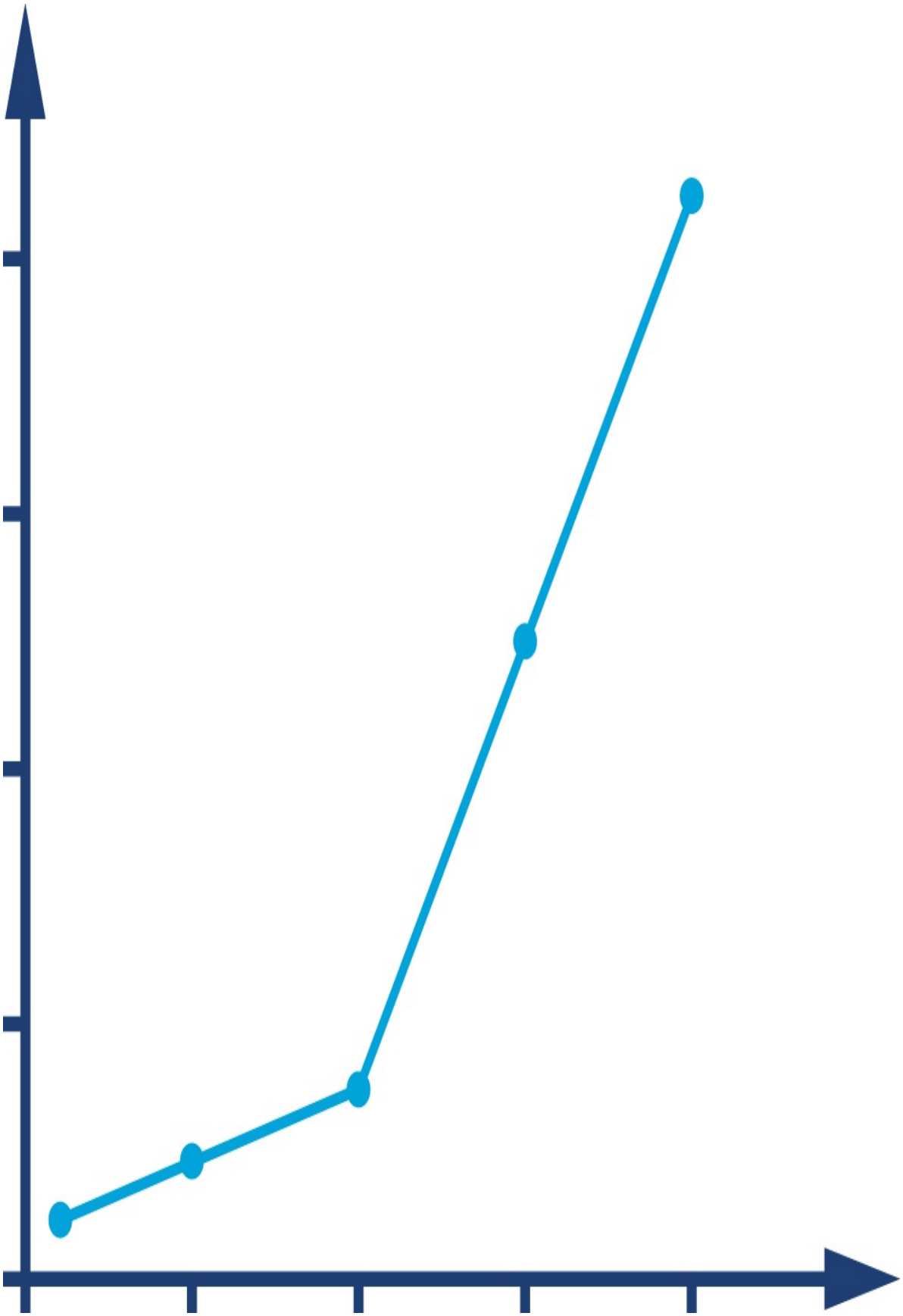


Figure 1-1. A performance elbow

By contrast, Figure 1-2 shows the much happier case of throughput scaling almost linearly as machines are added to a cluster. This is close to ideal behavior, and is only likely to be achieved in extremely favorable circumstances—e.g., scaling a stateless protocol with no need for session affinity with a single server.

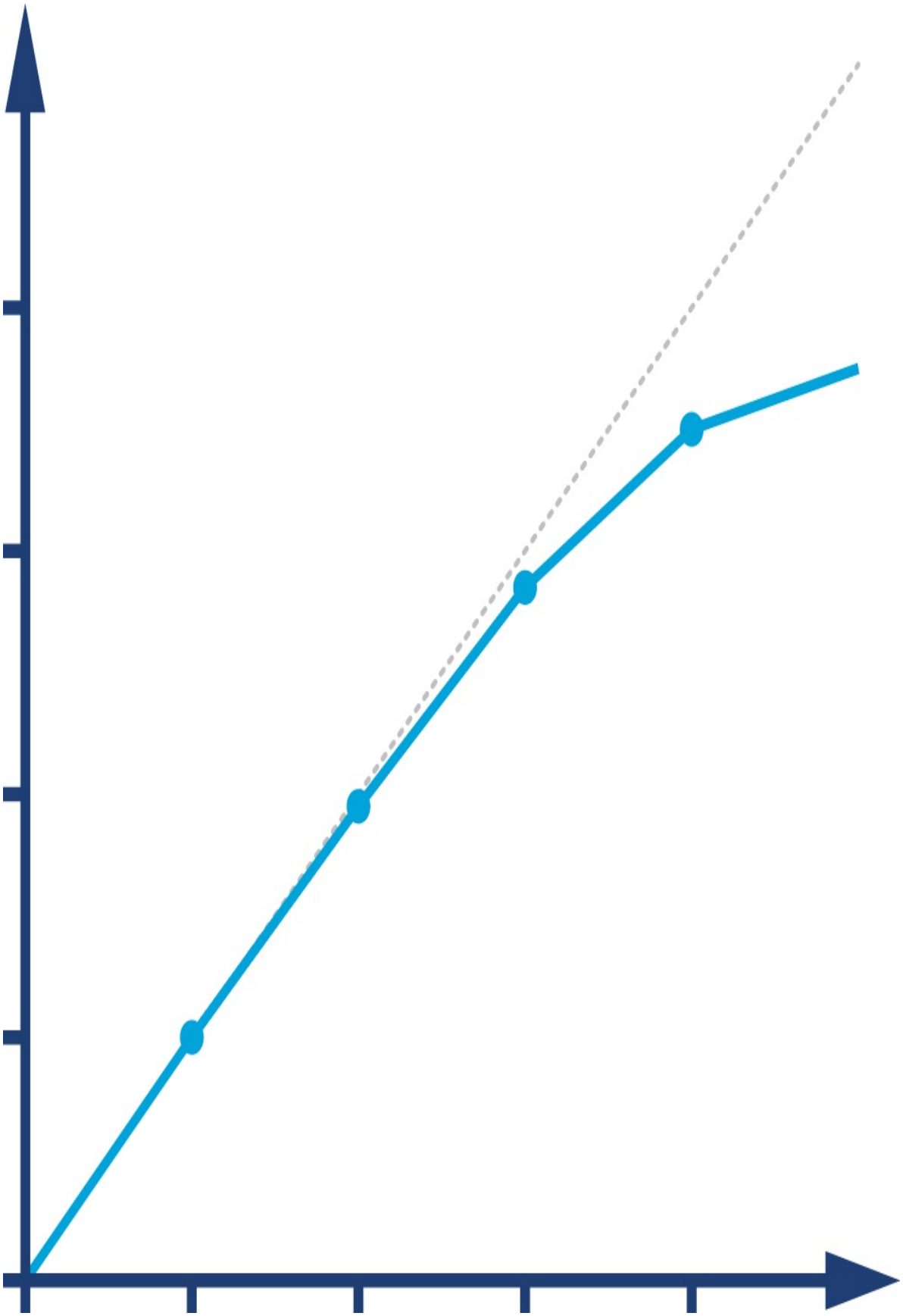


Figure 1-2. Near-linear scaling

In Chapter 13 we will meet Amdahl's Law, named for the famous computer scientist (and "father of the mainframe") Gene Amdahl of IBM. Figure 1-3 shows a graphical representation of his fundamental constraint on scalability; it shows the maximum possible speedup as a function of the number of processors devoted to the task.

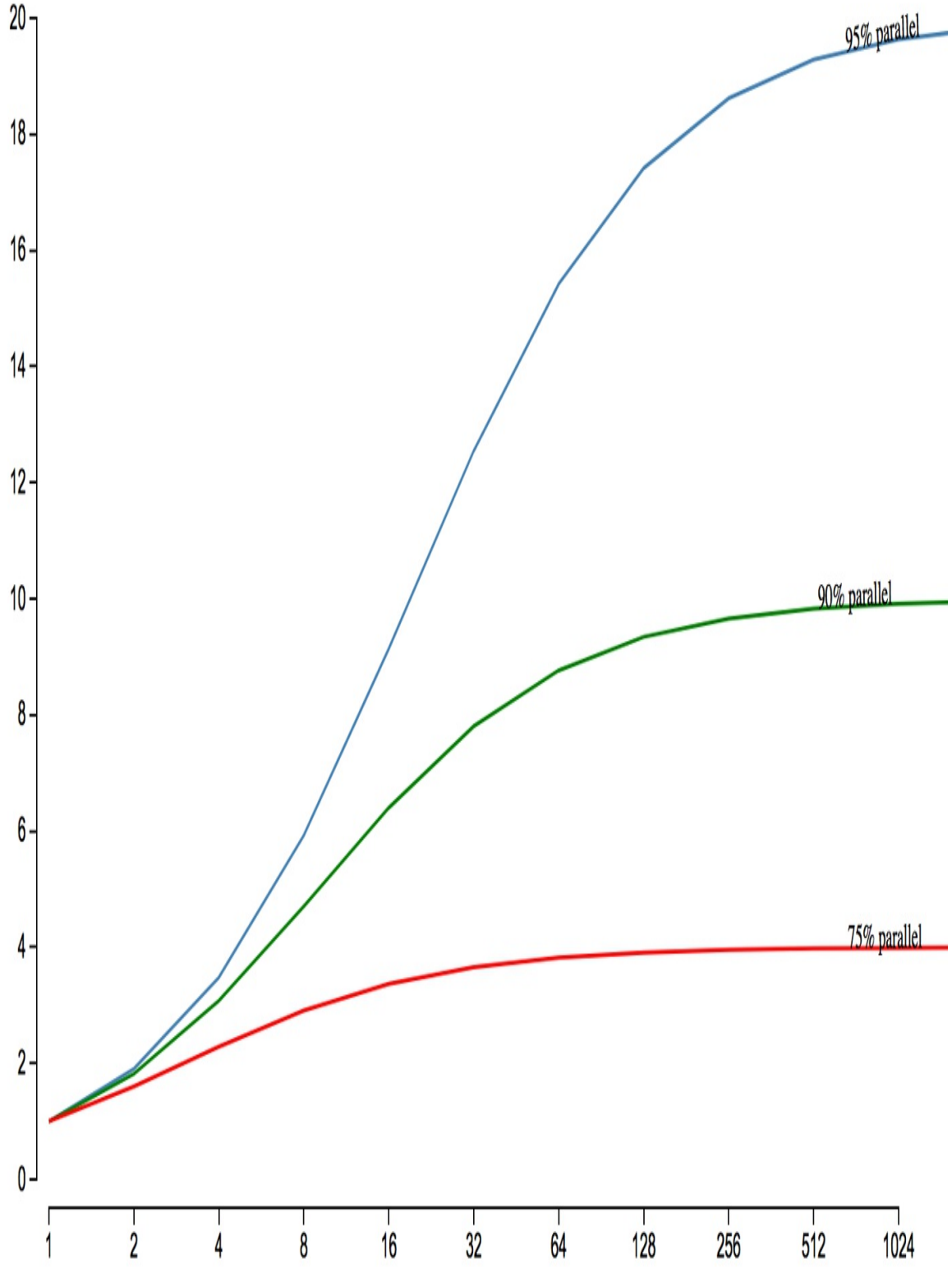

20
18
16
14
12
10
8
6
4
2
0
95% parallel
90% parallel
75% parallel
1
2
4
8
16
32
64
128
256
512
1024

Figure 1-3. Amdahl's Law

We display three cases: where the underlying task is 75%, 90%, and 95% parallelizable. This clearly shows that whenever the workload has any piece at all that must be performed serially, linear scalability is impossible, and there are strict limits on how much scalability can be achieved. This justifies the commentary around Figure 1-2—even in the best cases linear scalability is all but impossible to achieve.

The limits imposed by Amdahl's Law are surprisingly restrictive. Note in particular that the x-axis of the graph is logarithmic, and so even with an algorithm that is 95% parallelizable (and thus only 5% serial), 32 processors are needed for a factor-of-12 speedup. Even worse, no matter how many cores are used, the maximum speedup is only a factor of 20 for that algorithm. In practice, many algorithms are far more than 5% serial, and so have a more constrained maximum possible speedup.

Another common source of performance graphs in software systems is memory utilization. As we will see in Chapter 4, the underlying technology in the JVM's garbage collection subsystem naturally gives rise to a "sawtooth" pattern of memory used for healthy applications that aren't under stress. We can see an example in Figure 1-4 --which is a close-up of a screenshot from the Mission Control tool (JMC) provided by Eclipse Adoptium.

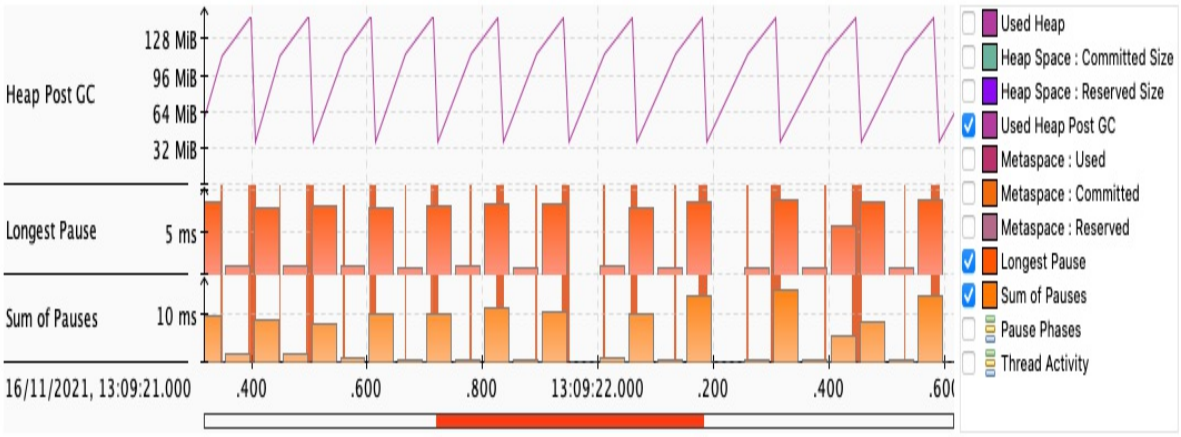


Figure 1-4. Healthy memory usage

One key performance metric for JVM is the allocation rate—effectively how

quickly it can create new objects (in bytes per second). We will have a great deal to say about this aspect of JVM performance in Chapter 4 and Chapter 5.

In Figure 1-5, we can see a zoomed-in view of allocation rate, also captured from JMC. This has been generated from a benchmark program that is deliberately stressing the JVM's memory subsystem—we have tried to make the JVM achieve 8GiB/s of allocation, but as we can see, this is beyond the capability of the hardware, and instead the maximum allocation rate of the system is between 4 and 5GiB/s.

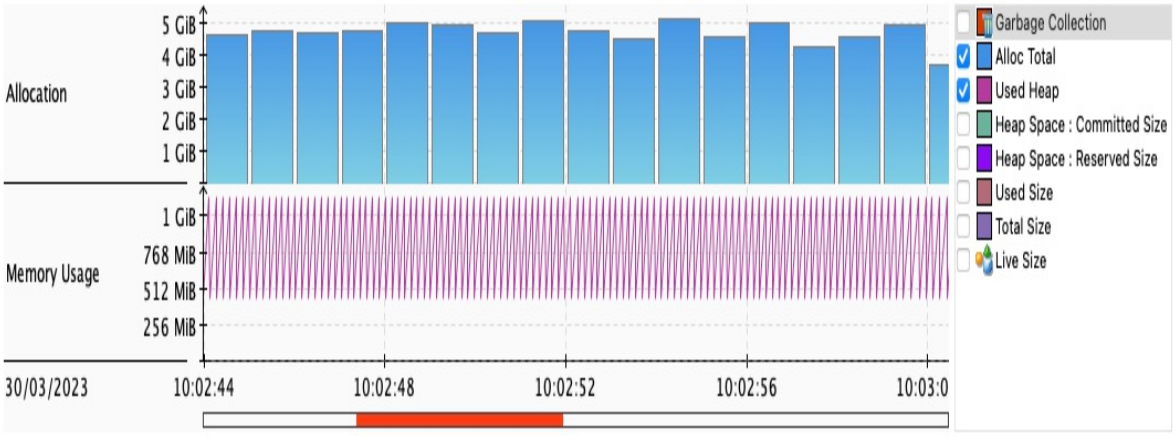


Figure 1-5. Sample problematic allocation rate

Note that tapped-out allocation is a different problem to the system having a resource leak. In that case, it is common for it to manifest in a manner like that shown in Figure 1-6, where an observable (in this case latency) slowly degrades as the load is ramped up, before hitting an inflection point where the system rapidly degrades.

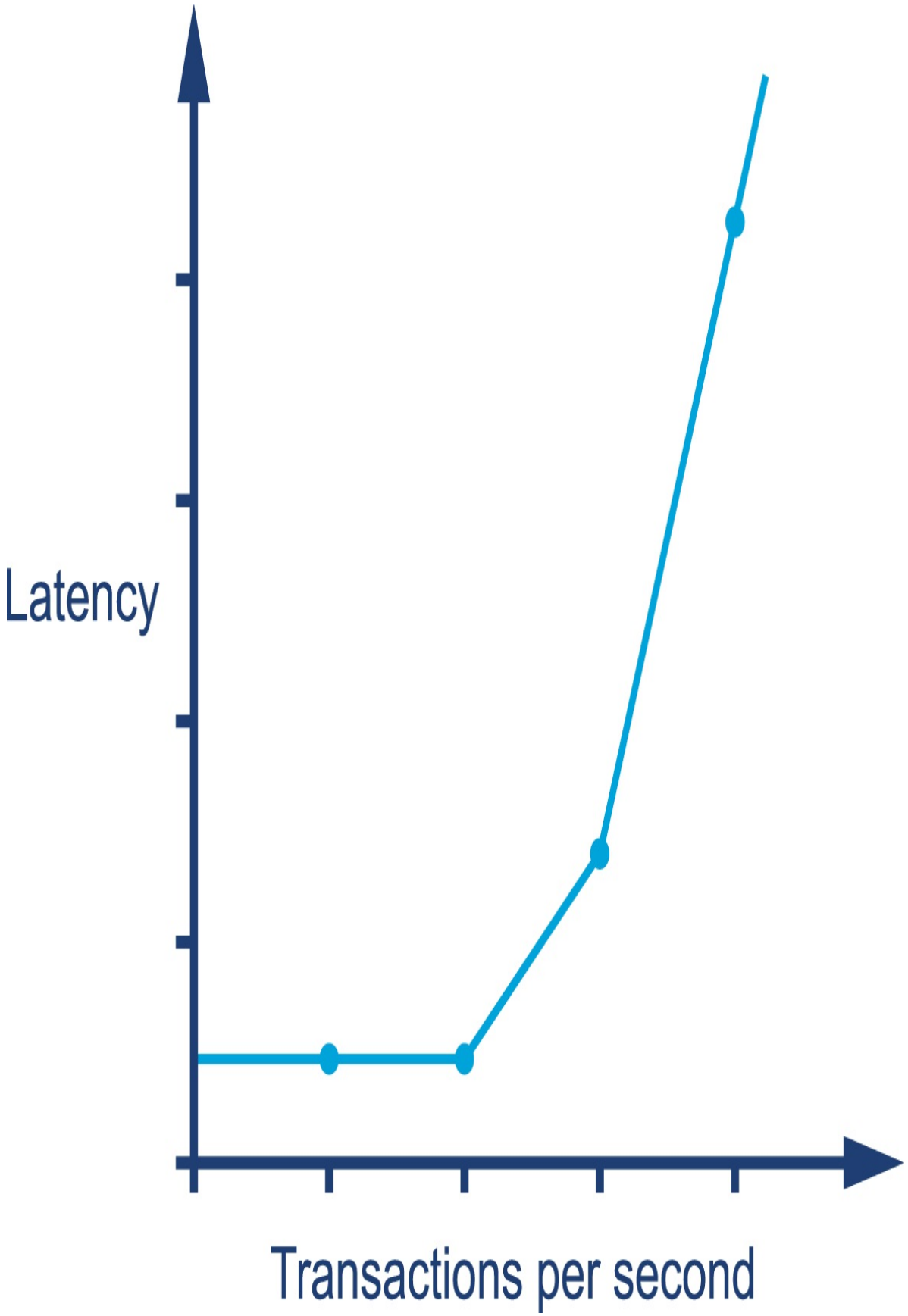

Latency
Transactions per second

Figure 1-6. Degrading latency under higher load

Let's move on to discuss some extra things to consider when working with cloud systems.

Performance in Cloud Systems

Modern cloud systems are nearly always distributed systems in that they are comprised of a cluster of nodes (JVM instances) interoperating via shared network resources. This means that, in addition to all the complexity of single-node systems, there is another level of complexity that must be addressed.

Operators of distributed systems have to think about things such as:

- How is work shared out among the nodes in the cluster?
- How do we roll out a new version of the software to the cluster (or new config)?
- What happens when a node leaves the cluster?
- What happens when a new node joins the cluster?
- What happens if the new node is misconfigured in some way?
- What happens if the new node behaves differently in some way, compared to the rest of the cluster?
- What happens if there is a problem with the code that controls the cluster itself?
- What happens if there is a catastrophic failure of the entire cluster, or some infrastructure that it depends upon?
- What happens if a component in the infrastructure the cluster depends upon is a limited resource and becomes a bottleneck to scalability?

These concerns, which we will explore fully later in the book, have a major impact on how cloud systems behave. They affect the key performance

observables such as throughput, latency, efficiency and utilization.

Not only that, but there are two very important aspects—which differ from the single-JVM case—that may not be obvious at first sight to newcomers to cloud systems. First is that many possible impacts are caused by the internal behavior of a cluster, which may be opaque to the performance engineer.

We will discuss this in detail in Chapter 10 when we tackle the topic of Observability in modern systems, and how to implement solutions to this visibility problem.

The second is that the efficiency and utilization of how a service uses cloud providers has a direct effect on the cost of running that service. Inefficiencies and misconfigurations can show up in the cost base of a service in a far more direct way. In fact, this is one way to think about the rise of cloud.

In the old days, teams would often own actual physical servers in dedicated areas (usually called *cages*) in datacenters. Purchasing these servers represented *capital expenditure*, and the servers were tracked as an asset. When we use cloud providers, such as AWS or Azure, we are renting time on machines actually owned by companies such as Amazon or Microsoft. This is *operational expenditure*, and it is a cost (or liability). This shift means that the computational requirements of our systems are now much more open to scrutiny by the financial folks.

Overall, it is important to recognize that cloud systems fundamentally consist clusters of processes (in our case, JVMs) that dynamically change over time. The clusters can grow or shrink in size, but even if they do not, over time the participating processes will change. This stands in sharp contrast to traditional host-based systems where the processes forming a cluster are usually much more long-lived and belong to a known, and stable, collection of hosts.

Summary

In this chapter we have started to discuss what Java performance is and is not. We have introduced the fundamental topics of empirical science and

measurement, and the basic vocabulary and observables that a good performance exercise will use. We have introduced some common cases that are often seen within the results obtained from performance tests. Finally, we have introduced the very basics of the sorts of additional issues that can arise in cloud systems.

Let’s move on and begin discussing some major aspects of performance testing, as well as how to handle the numbers that are generated by those tests.

---

1 J. Gosling, “The feel of Java,” *Computer*, vol. 30, no. 6 (June 1997): 53-57

2 E. Goldratt and J. Cox, “The Goal,” (Gower Publishing, 1984)

Chapter 2. Performance Testing Methodology

Performance testing is undertaken for a variety of reasons. In this chapter we will introduce the different types of performance test that a team may wish to execute, and discuss some best practices for each subtype of testing.

Later in the chapter we will discuss statistics, and some very important human factors that, are often neglected when considering performance problems.

Types of Performance Test

Performance tests are frequently conducted for the wrong reasons, or conducted badly. The reasons for this vary widely, but are often rooted in a failure to understand the nature of performance analysis and a belief that "doing something is better than doing nothing." As we will see several times throughout the book, this belief is often a dangerous half-truth at best.

One of the more common mistakes is to speak generally of "performance

testing” without engaging with the specifics. In fact, there are many different types of large-scale performance tests that can be conducted on a system.

> **NOTE**
>
> Good performance tests are quantitative. They ask questions that produce a numeric answer that can be handled as an experimental output and subjected to statistical analysis.

The types of performance tests we will discuss in this book usually have independent (but somewhat overlapping) goals. It is therefore important to understand the quantitative questions you are trying to answer before deciding what type of testing should be carried out.

This doesn’t have to be that complex—simply writing down the questions the test is intended to answer can be enough. However, it is usual to consider why these tests are important for the application and confirming the reason with the application owner (or key customers).

Some of the most common test types, and an example question for each, are as follows:

Latency test

: What is the end-to-end transaction time?

Throughput test

: How many concurrent transactions can the current system capacity deal with?

Load test

: Can the system handle a specific load?

Stress test

: What is the breaking point of the system?

Endurance test

What performance anomalies are discovered when the system is run for an extended period?

Capacity planning test

Does the system scale as expected when additional resources are added?

Degradation

What happens when the system is partially failed?

Let's look in more detail at each of these test types in turn.

Latency Test

Latency is one of the most common types of performance test, because it is often a system observable of keen interest to management (and users): how long are our customers waiting for a transaction (or a page load)?

This can a double-edged sword, because the simplicity of the question (that a latency test seeks to answer) can cause teams to focus too much on latency. This, in turn, can cause the team to ignore the necessity of identifying quantitative questions for other types of performance tests.

> **NOTE**
>
> The goal of a latency tuning exercise is usually to directly improve the user experience, or to meet a service-level agreement.

However, even in the simplest of cases, a latency test has some subtleties that must be treated carefully. One of the most noticeable is that a simple mean (average) is not very useful as a measure of how well an application is reacting to requests. We will discuss this subject more fully in "Statistics for JVM Performance" and explore additional measures.

Throughput Test

Throughput is probably the second most common quantity to be performance-tested. It can even be thought of as dual to latency, in some senses.

For example, when we are conducting a latency test, it is important to state (and control) the concurrent transactions count when producing a distribution of latency results. Similarly, when we are conducting a throughput test, we must make sure to keep an eye on latency and check that it is not blowing up to unacceptable values as we ramp up.

> **NOTE**
>
> The observed latency of a system should be stated at known and controlled throughput levels, and vice versa.

We determine the "maximum throughput" by noticing when the latency distribution suddenly changes—effectively a "breaking point" (also called an *inflection point*) of the system. The point of a stress test, as we will see in an upcoming section, is to locate such points and the load levels at which they occur.

A throughput test, on the other hand, is about measuring the observed maximum throughput before the system starts to degrade. Once again, these test types are discussed separately, but are rarely truly independent in practice.

Stress Test

One way to think about a stress test is as a way to determine how much spare headroom the system has. The test typically proceeds by placing the system into a steady state of transactions—that is, a specified throughput level (often the current peak). The test then ramps up the concurrent transactions slowly, until the system observables start to degrade.

The value just before the observables started to degrade determines the maximum throughput achieved in a stress test.

Load Test

A load test differs from a throughput test (or a stress test) in that it is usually framed as a binary test: “Can the system handle this projected load or not?” Load tests are sometimes conducted in advance of expected business events—for example, the onboarding of a new customer or market that is expected to drive greatly increased traffic to the application.

Other examples of possible events that could warrant performing this type of test include advertising campaigns, social media events, and “viral content.”

Endurance Test

Some problems manifest only over much longer periods of time (often measured in days). These include slow memory leaks, cache pollution, and memory fragmentation (especially for applications that may eventually suffer a GC concurrent mode failure; see Chapter 5 for more details).

To detect these types of issues, an endurance test (also known as a soak test) is the usual approach. These are run at average (or high) utilization, but within observed realistic loads for the system. During the test, resource levels are closely monitored to spot any breakdowns or exhaustions of resources.

This type of test is more common in low-latency systems, as it is very common that those systems will not be able to tolerate the length of a stop-the-world event caused by a full GC cycle (see Chapter 4 and subsequent chapters for more on stop-the-world events and related GC concepts).

Endurance tests are not performed as often as they perhaps should be, for the simple reason that they take a long time to run and can be very expensive—but there are no shortcuts. There is also the inherent difficulty of testing with realistic data or usage patterns over a long period. This can be one of the major reasons why teams end up “testing in production”.

This type of test is also not always applicable to microservice or other architectures where there may be a lot of code changes deployed in a short time.

Capacity Planning Test

Capacity planning tests bear many similarities to stress tests, but they are a distinct type of test. The role of a stress test is to find out what the current system will cope with, whereas a capacity planning test is more forward-looking and seeks to find out what load an upgraded system could handle.

For this reason, capacity planning tests are often carried out as part of a scheduled planning exercise, rather than in response to a specific event or threat.

Degradation Test

Once upon a time, rigorous failover and recovery testing was really only practiced in the most highly regulated and scrutinized environments (including banks and financial institutions). However, as applications have migrated to the cloud, clustered deployments (e.g. based on Kubernetes) have become more and more common. One primary consequence of this is that more and more developers now need to be aware of the possible failure modes of clustered applications.

> **NOTE**
>
> A full discussion of all aspects of resilience and fail-over testing is outside the scope of this book. In Chapter 15, we will discuss some of the simpler effects that can be seen in cloud systems when a cluster partially fails, or needs to recover.

In this section, the only type of resilience test we will discuss is the degradation test—this type of test is also known as a *partial failure* test.

The basic approach to this test is to see how the system behaves when a component or entire subsystem suddenly loses capacity while the system is running at simulated loads equivalent to usual production volumes. Examples could be application server clusters that suddenly lose members, or network bandwidth that suddenly drops.

Key observables during a degradation test include the transaction latency

distribution and throughput.

One particularly interesting subtype of partial failure test is known as the *Chaos Monkey*. This is named after a project at Netflix that was undertaken to verify the robustness of its infrastructure.

The idea behind Chaos Monkey is that in a truly resilient architecture, the failure of a single component should not be able to cause a cascading failure or have a meaningful impact on the overall system.

Chaos Monkey forces system operators to confront this possibility by randomly killing off live processes in the production environment.

In order to successfully implement Chaos Monkey–type systems, an organization must have very high levels of system hygiene, service design, and operational excellence. Nevertheless, it is an area of interest and aspiration for an increasing number of companies and teams.

Best Practices Primer

When deciding where to focus your effort in a performance tuning exercise, there are three golden rules that can provide useful guidance:

- Identify what you care about and figure out how to measure it.
- Optimize what matters, not what is easy to optimize.
- Play the big points first.

The second point has a converse, which is to remind yourself not to fall into the trap of attaching too much significance to whatever quantity you can easily measure. Not every observable is significant to a business, but it is sometimes tempting to report on an easy measure, rather than the right measure.

To the third point, it is also easy to fall into the trap of optimizing small things simply for the sake of optimizing.

Top-Down Performance

One of the aspects of Java performance that many engineers miss at first sight is that large-scale benchmarking of Java applications is usually much easier than trying to get accurate numbers for small sections of code.

This is such a widely misunderstood point, that to deliberately deemphasize it, we do not discuss *microbenchmarking* in the main book text at all. Instead, it is discussed in Appendix A --a placement that more accurately reflects the utility of the technique for the majority of applications.

> **NOTE**
>
> The approach of starting with the performance behavior of an entire application is usually called *top-down* performance.

To make the most of the top-down approach, a testing team needs a test environment, a clear understanding of what it needs to measure and optimize, and an understanding of how the performance exercise will fit into the overall software development lifecycle.

Creating a Test Environment

Setting up a test environment is one of the first tasks most performance testing teams will need to undertake. Wherever possible, this should be an exact duplicate of the production environment, in all aspects.

> **NOTE**
>
> Some teams may be in a position where they are forced to forgo testing environments and simply measure in production using modern deployment and Observability techniques. This is the subject of Chapter 10, but it is not recommended as an approach unless it’s necessary.

This includes not only application servers (which servers should have the

same number of CPUs, same version of the OS and Java runtime, etc.), but web servers, databases, message queues, and so on. Any services (e.g., third-party network services that are not easy to replicate, or do not have sufficient QA capacity to handle a production-equivalent load) will need to be mocked for a representative performance testing environment.

> **NOTE**
>
> Performance testing environments that are significantly different from the production deployments that they purport to represent are usually ineffective—they fail to produce results that have any usefulness or predictive power in the live environment.

For traditional (i.e., non-cloud-based) environments, a production-like performance testing environment is relatively straightforward to achieve in theory—the team simply buys as many machines as are in use in the production environment and then configures them in exactly the same way as production is configured.

Management is sometimes resistant to the additional infrastructure cost that this represents. This is almost always a false economy, but sadly many organizations fail to account correctly for the cost of outages. This can lead to a belief that the savings from not having an accurate performance testing environment are meaningful, as it fails to properly account for the risks introduced by having a QA environment that does not mirror production.

The advent of cloud technologies, has changed this picture. More dynamic approaches to infrastructure management are now widespread. This includes on-demand and autoscaling infrastructure, as well as approaches such as *immutable infrastructure*, also referred to as treating server infrastructure as "livestock, not pets".

In theory, these trends make the construction of a performance testing environment that looks like production easier. However, there are subtleties here. For example:

- Having a process that allows changes to be made in a test environment

first and then migrated to production

- Making sure that a test environment does not have some overlooked dependencies that depend upon production
- Ensuring that test environments have realistic authentication and authorization systems, not dummy components

Despite these concerns, the possibility of setting up a testing environment that can be turned off when not in use is a key advantage of cloud-based deployments. This can bring significant cost savings to the project, but it requires a proper process for starting up and shutting down the environment as scheduled.

Identifying Performance Requirements

The overall performance of a system is not solely determined by your application code. As we will discover throughout the rest of this book, the container, operating system, and hardware all have a role to play.

Therefore, the metrics that we will use to evaluate performance should not be thought about solely in terms of the code. Instead, we must consider systems as a whole and the observable quantities that are important to customers and management. These are usually referred to as performance *nonfunctional requirements* (NFRs), and are the key indicators that we want to optimize.

> **NOTE**
>
> In Chapter 7, we will meet a simple system model that describes in more detail how the interaction between OS, hardware, JVM and code impacts performance.

Some performance goals are obvious:

- Reduce 95% percentile transaction time by 100 ms.
- Improve system so that 5x throughput on existing hardware is possible.

- Improve average response time by 30%.

Others may be less apparent:

- Reduce resource cost to serve the average customer by 50%.
- Ensure system is still within 25% of response targets, even when application clusters are degraded by 50%.
- Reduce customer “drop-off” rate by 25% by removing 10 ms of latency.

An open discussion with the stakeholders as to exactly what should be measured and what goals are to be achieved is essential. Ideally, this discussion should form part of the first kick-off meeting for any performance exercise.

Performance Testing as Part of the SDLC

Some companies and teams prefer to think of performance testing as an occasional, one-off activity. However, more sophisticated teams tend to make ongoing performance tests, and in particular performance regression testing, an integral part of their software development lifecycle (SDLC).

This requires collaboration between developers and infrastructure teams to control which versions of code are present in the performance testing environment at any given time. It is also virtually impossible to implement without a dedicated testing environment.

Java-Specific Issues

Much of the science of performance analysis is applicable to any modern software system. However, the nature of the JVM is such that there are certain additional complications that the performance engineer should be aware of and consider carefully. These largely stem from the dynamic self-management capabilities of the JVM, such as the dynamic tuning of memory areas and JIT compilation.

For example, modern JVMs analyze which methods are being run to identify candidates for JIT compilation to optimized machine code. This means that if a method is not being JIT-compiled, then one of two things is true about the method:

- It is not being run frequently enough to warrant being compiled.
- The method is too large or complex to be analyzed for compilation.

The second condition is, by the way, much rarer than the first. In Chapter 6 we will discuss JIT compilation in detail, and show some simple techniques for ensuring that the important methods of applications are targeted for JIT compilation by the JVM.

Having discussed some of the most common best practices for performance, let’s now turn our attention to the pitfalls and antipatterns that teams can fall

prey to.

Causes of performance antipatterns

An antipattern is an undesired behavior of a software project or team that is observed across a large number of projects.[1] The frequency of occurrence leads to the conclusion (or suspicion) that some underlying factor is responsible for creating the unwanted behavior. Some antipatterns may at first sight seem to be justified, with their non-ideal aspects not immediately obvious. Others are the result of negative project practices slowly accreting over time.

A partial catalogue of antipatterns can be found in Appendix B—where an example of the first kind would be something like *Distracted By Shiny*, whereas *Tuning By Folklore* is an example of the second kind.

In some cases the behavior may be driven by social or team constraints, or by common misapplied management techniques, or by simple human (and developer) nature. By classifying and categorizing these unwanted features, we develop a *pattern language* for discussing them, and hopefully eliminating them from our projects.

Performance tuning should always be treated as a very objective process, with precise goals set early in the planning phase. This is easier said than done: when a team is under pressure or not operating under reasonable circumstances, this can simply fall by the wayside.

Many readers will have seen the situation where a new client is going live or a new feature is being launched, and an unexpected outage occurs—in user acceptance testing (UAT) if you are lucky, but often in production. The team is then left scrambling to find and fix what has caused the bottleneck. This usually means performance testing has not been carried out, or the team "ninja" made an assumption and has now disappeared (ninjas are good at this).

A team that works in this way will likely fall victim to antipatterns more often than a team that follows good performance testing practices and has

open and reasoned conversations. As with many development issues, it is often the human elements, such as communication problems, rather than any technical aspect that leads to an application having problems.

One interesting possibility for classification was provided in a blog post by Carey Flichel called "Why Developers Keep Making Bad Technology Choices". The post specifically calls out five main reasons that cause developers to make bad choices. Let's look at each in turn.

Boredom

Most developers have experienced boredom in a role, and for some this doesn't have to last very long before they are seeking a new challenge or role either in the company or elsewhere. However, other opportunities may not be present in the organization, and moving somewhere else may not be possible.

It is likely many readers have come across a developer who is simply riding it out, perhaps even actively seeking an easier life. However, bored developers can harm a project in a number of ways.

For example, they might introduce code complexity that is not required, such as writing a sorting algorithm directly in code when a simple `Collections.sort()` would be sufficient. They might also express their boredom by looking to build components with technologies that are unknown or perhaps don't fit the use case just as an opportunity to use them—which leads us to the next section.

Résumé Padding

Occasionally the overuse of technology is not tied to boredom, but rather represents the developer exploiting an opportunity to boost their experience with a particular technology on their résumé (or CV).

In this scenario, the developer is making an active attempt to increase their potential salary and marketability as they're about to re-enter the job market. It's unlikely that many people would get away with this inside a well-functioning team, but it can still be the root of a choice that takes a project

down an unnecessary path.

The consequences of an unnecessary technology being added due to a developer's boredom or résumé padding can be far-reaching and very long-lived, lasting for many years after the original developer has left.

Social Pressure

Technical decisions are often at their worst when concerns are not voiced or discussed at the time choices are being made. This can manifest in a few ways; for example, perhaps a junior developer doesn't want to make a mistake in front of more senior members of their team, or perhaps a developer fears appearing to their peers as uninformed on a particular topic.

Another particularly toxic type of social pressure is for competitive teams, wanting to be seen as having high development velocity, to rush key decisions without fully exploring all the consequences.

Lack of Understanding

Developers may look to introduce new tools to help solve a problem because they are not aware of the full capability of their current tools. It is often tempting to turn to a new and exciting technology component because it is great at performing one specific task. However, introducing more technical complexity must be taken on balance with what the current tools can actually do.

For example, Hibernate is sometimes seen as the answer to simplifying translation between domain objects and databases. If there is only limited understanding of Hibernate on the team, developers can make assumptions about its suitability based on having seen it used in another project.

This lack of understanding can cause overcomplicated usage of Hibernate and unrecoverable production outages. By contrast, rewriting the entire data layer using simple JDBC calls allows the developer to stay on familiar territory.

One of the authors taught a Hibernate course that contained an attendee in

exactly this position; they were trying to learn enough Hibernate to see if the application could be recovered, but ended up having to rip out Hibernate over the course of a weekend—definitely not an enviable position.

Misunderstood/Nonexistent Problem

Developers may often use a technology to solve a particular issue where the problem space itself has not been adequately investigated. Without having measured performance values, it is almost impossible to understand the success of a particular solution. Often collating these performance metrics enables a better understanding of the problem.

To avoid antipatterns it is important to ensure that communication about technical issues is open to all participants in the team, and actively encouraged. Where things are unclear, gathering factual evidence and working on prototypes can help to steer team decisions. A technology may look attractive; however, if the prototype does not measure up then the team can make a more informed decision.

To see how these underlying causes can lead to a variety of performance antipatterns, interested readers should consult Appendix B.

Statistics for JVM Performance

If performance analysis is truly an experimental science, then we will inevitably find ourselves dealing with distributions of results data. Statisticians and scientists know that results that stem from the real world are virtually never represented by clean, stand-out signals. We must deal with the world as we find it, rather than the overidealized state in which we would like to find it.

> *In God we trust; all others must use data.*[2]
>
> —W. Edwards Deming (attr)

All measurements contain some amount of error. In the next section we'll describe the two main types of error that a Java developer may expect to

encounter when doing performance analysis.

Types of Error

There are two main sources of error that an engineer may encounter. These are:

Random error

> A measurement error or an unconnected factor affects results in an uncorrelated manner

Systematic error

> An unaccounted factor affects measurement of the observable in a correlated way

There are specific words associated with each type of error. For example, *accuracy* is used to describe the level of systematic error in a measurement; high accuracy corresponds to low systematic error. Similarly, *precision* is the term corresponding to random error; high precision is low random error.

The graphics in Figure 2-1 show the effect of these two types of error on a measurement. The extreme left image shows a clustering of shots (which represent our measurements) around the true result (the “center of the target”). These measurements have both high precision and high accuracy.

The second image has a systematic effect (miscalibrated sights perhaps?) that is causing all the shots to be off-target, so these measurements have high precision, but low accuracy. The third image shows shots basically on target but loosely clustered around the center, so low precision but high accuracy. The final image shows no clear pattern, as a result of having both low precision and low accuracy.

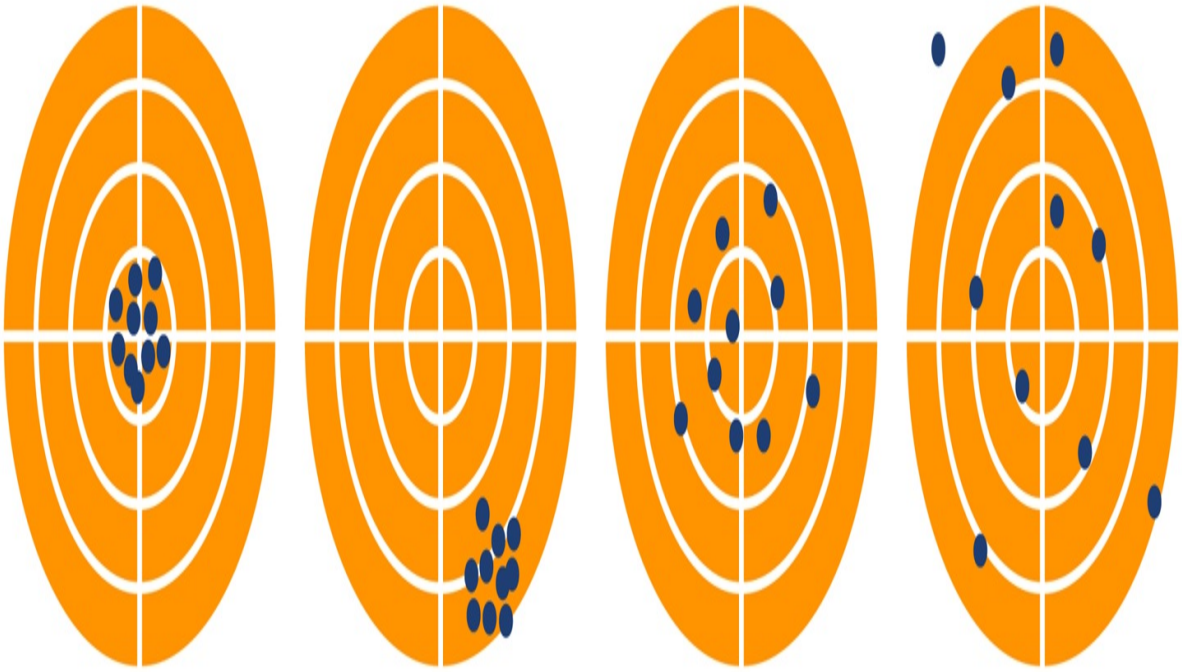


Figure 2-1. Different types of error

Let’s move on to explore these types of error in more detail, starting with random error.

Random error

Random errors are hopefully familiar to most people—they are a very well-trodden path. However, they still deserve a mention here, as any handling of observed or experimental data needs to contend with them to some level.

> **NOTE**
>
> The discussion assumes readers are familiar with basic statistical handling of normally distributed measurements (mean, mode, standard deviation, etc.); readers who aren’t should consult a basic textbook, such as *The Handbook of Biological Statistics*.[3]

Random errors are caused by unknown or unpredictable changes in the environment. In general scientific usage, these changes may occur in either the measuring instrument or the environment, but for software we assume that our measuring harness is reliable, and so the source of random error can

only be the operating environment.

Random error is usually considered to obey a Gaussian (aka normal) distribution. A couple of typical examples of Gaussian distributions are shown in Figure 2-2.

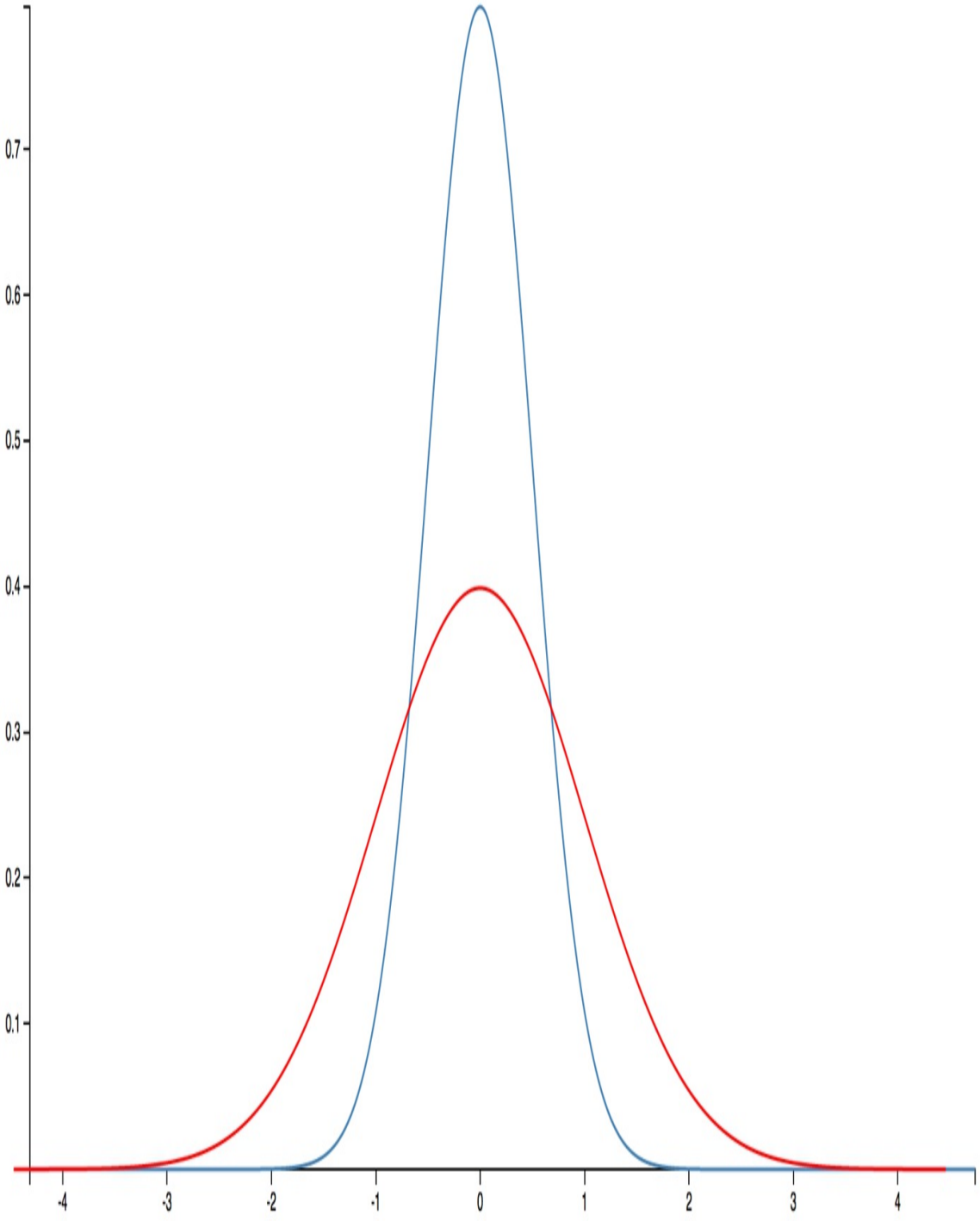


Figure 2-2. A Gaussian distribution (aka normal distribution or bell curve)

The distribution is a good model for the case where an error is equally likely to make a positive or negative contribution to an observable. However, as we will see in the section on non-normal statistics, the situation for JVM

measurements is a little more complicated.

Systematic error

As an example of systematic error, consider a performance test running against a group of backend Java web services that send and receive JSON. This type of test is very common when it is problematic to directly use the application frontend for load testing.

Figure 2-3 was generated from the Apache JMeter load-generation tool. In it, there are actually two systematic effects at work. The first is the linear pattern observed in the topmost line (the outlier service), which represents slow exhaustion of some limited server resource.

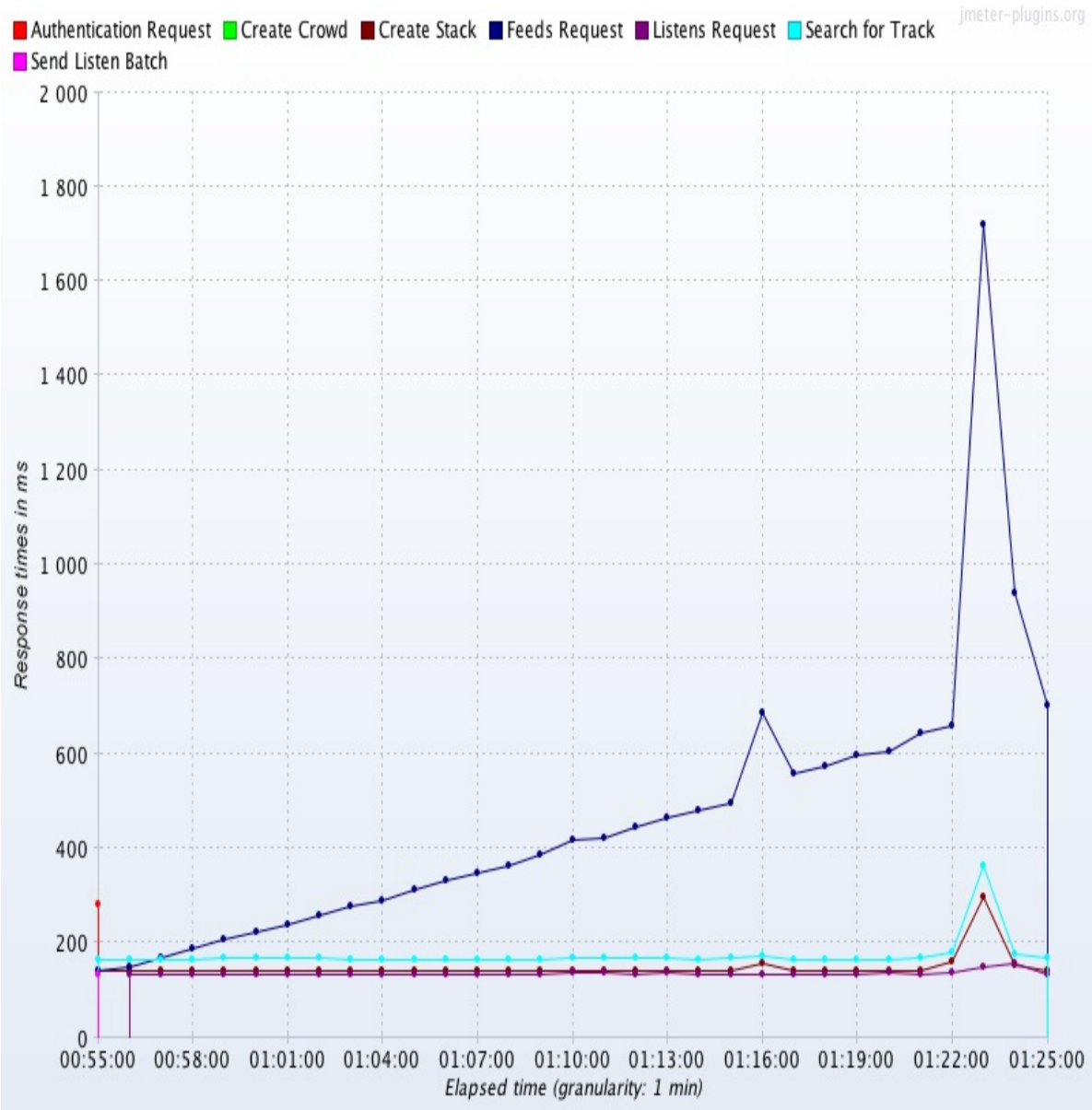


Figure 2-3. Systematic error

This type of pattern is often associated with a memory leak, or some other resource being used and not released by a thread during request handling, and represents a candidate for investigation—it looks like it could be a genuine problem.

> **NOTE**
>
> Further analysis would be needed to confirm the type of resource that was being affected; we can't just conclude that it's a memory leak.

The second effect that should be noticed is the consistency of the majority of the other services at around the 180 ms level. This is suspicious, as the services are doing very different amounts of work in response to a request. So why are the results so consistent?

The answer is that while the services under test are located in London, this load test was conducted from Mumbai, India. The observed response time includes the irreducible round-trip network latency from Mumbai to London. This is in the range 120–150 ms, and so accounts for the vast majority of the observed time for the services other than the outlier.

This large, systematic effect is drowning out the differences in the actual response time (as the services are actually responding in much less than 120 ms). This is an example of a systematic error that does not represent a problem with our application.

Instead, this error stems from a problem in our test setup, and so the good news is that this artifact completely disappeared (as expected) when the test was rerun from London.

To finish off this section, let's take a quick look at a notorious problem that frequently accompanies systematic error—the spurious correlation.

Spurious correlation

One of the most famous aphorisms about statistics is "correlation does not imply causation" —-that is, just because two variables appear to behave similarly does not imply that there is an underlying connection between them.

In the most extreme examples, if a practitioner looks hard enough, then a correlation can be found between entirely unrelated measurements. For example, in Figure 2-4 we can see that consumption of chicken in the US is well correlated with total import of crude oil.[4]

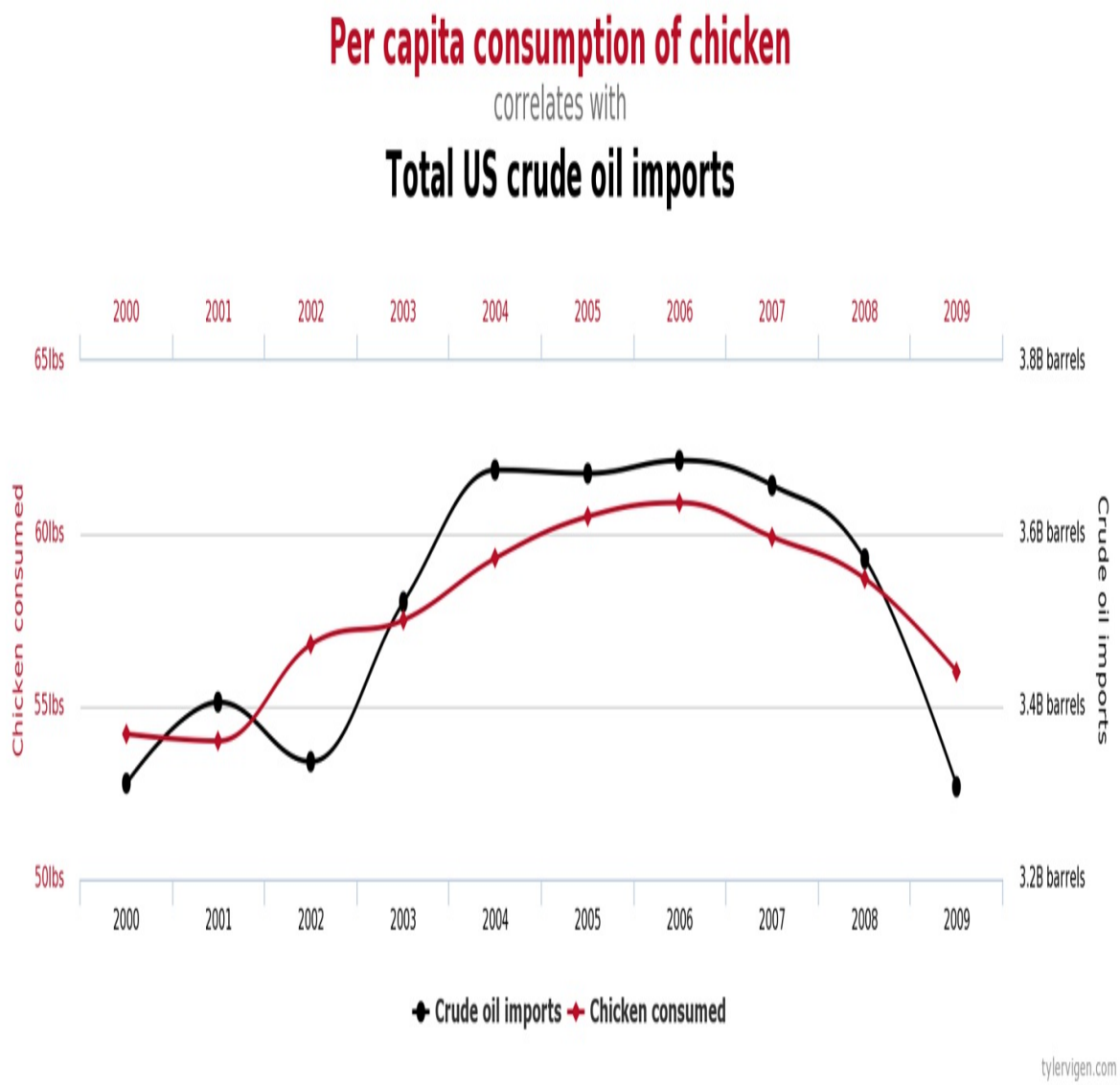


Figure 2-4. A completely spurious correlation (Vigen)

These numbers are clearly not causally related; there is no factor that drives both the import of crude oil and the eating of chicken. However, it isn't the absurd and ridiculous correlations that the practitioner needs to be wary of.

In Figure 2-5, we see the revenue generated by video arcades correlated to the number of computer science PhDs awarded. It isn't too much of a stretch to imagine a sociological study that claimed a link between these observables, perhaps arguing that "stressed doctoral students were finding relaxation with a few hours of video games." These types of claim are depressingly common, despite no such common factor actually existing.

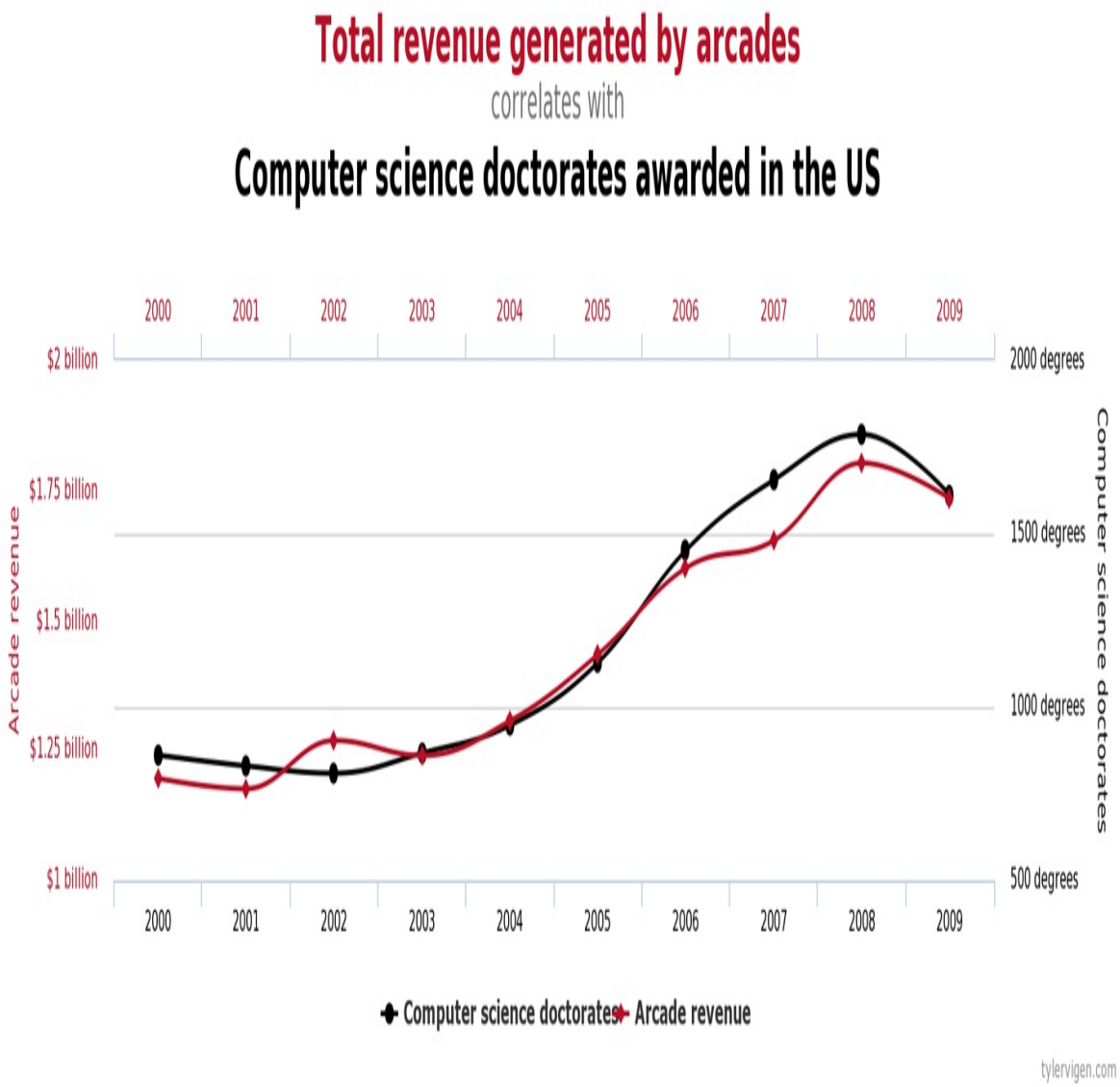


Figure 2-5. A less spurious correlation? (Vigen)

In the realm of the JVM and performance analysis, we need to be especially careful not to attribute a causal relationship between measurements based solely on correlation and that the connection "seems plausible."

> *The first principle is that you must not fool yourself—and you are the easiest person to fool.*[5]
>
> —Richard Feynman

We've met some examples of sources of error and mentioned the notorious bear traps of spurious correlation and fooling oneself, so let's move on to discuss an aspect of JVM performance measurement that requires some

special care and attention to detail.

Non-Normal Statistics

Statistics based on the normal distribution do not require much mathematical sophistication. For this reason, the standard approach to statistics that is typically taught at pre-college or undergraduate level focuses heavily on the analysis of normally distributed data.

Students are taught to calculate the mean and the standard deviation (or variance), and sometimes higher moments, such as skew and kurtosis. However, these techniques have a serious drawback, in that the results can easily become distorted if the distribution has even relatively few far-flung outlying points.

> **NOTE**
>
> In Java performance, the outliers represent slow transactions and unhappy customers. We need to pay special attention to these points, and avoid techniques that dilute the importance of outliers.

To consider it from another viewpoint: unless a large number of customers are already complaining, it is unlikely that improving the average response time is a useful performance goal. For sure, doing so will improve the experience for everyone, but it is far more usual for a few disgruntled customers to be the cause of a latency tuning exercise. This implies that the outlier events are likely to be of more interest than the experience of the majority who are receiving satisfactory service.

In Figure 2-6 we can see a more realistic curve for the likely distribution of method (or transaction) times. It is clearly not a normal distribution.

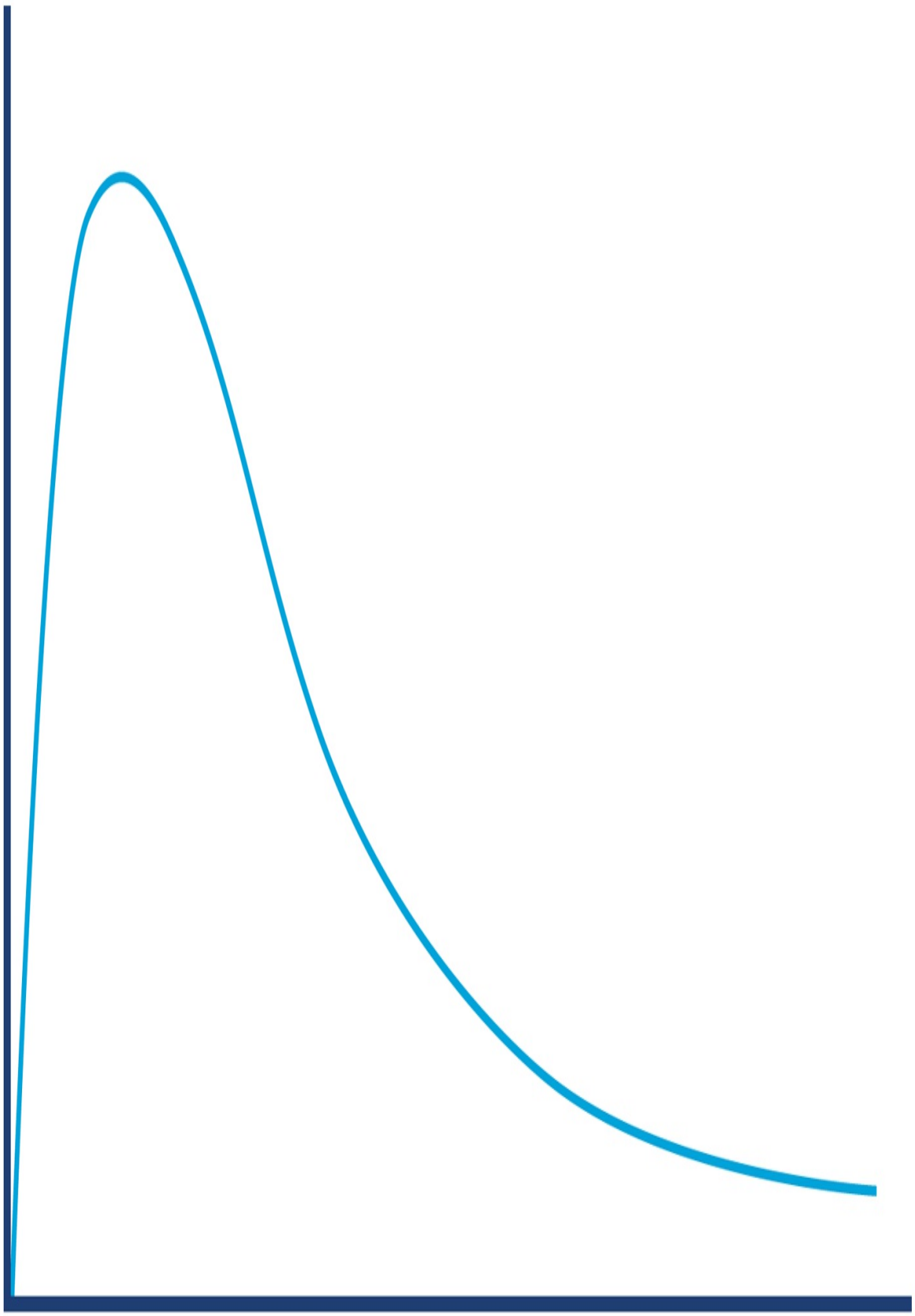


Figure 2-6. A more realistic view of the distribution of transaction times

The shape of the distribution in Figure 2-6 shows something that we know intuitively about the JVM: it has "hot paths" where all the relevant code is already JIT-compiled, there are no GC cycles, and so on. These represent a best-case scenario (albeit a common one); there simply are no calls that are "a bit faster" due to random effects.

This violates a fundamental assumption of Gaussian statistics and forces us to consider distributions that are non-normal.

> **NOTE**
>
> For distributions that are non-normal, many "basic rules" of normally distributed statistics are violated. In particular, standard deviation/variance and other higher moments are basically useless.

One technique that is very useful for handling the non-normal, "long-tail" distributions that the JVM produces is to use a modified scheme of percentiles. Remember that a distribution is a whole collection of points—a shape of data, and is not well-represented by a single number.

Instead of computing just the mean, which tries to express the whole distribution in a single result, we can use a sampling of the distribution at intervals. When used for normally distributed data, the samples are usually taken at regular intervals. However, a small adaptation allows the technique to be used more effectively for JVM statistics.

The modification is to use a sampling that takes into account the long-tail distribution by starting from the mean, then the 90th percentile, and then moving out logarithmically, as shown in the following method timing results. This means that we're sampling according to a pattern that better corresponds to the shape of the data:

```
50.0% level was 23 ns
90.0% level was 30 ns
99.0% level was 43 ns
```

```
99.9% level was 164 ns
99.99% level was 248 ns
99.999% level was 3,458 ns
99.9999% level was 17,463 ns
```

The samples show us that while the average time was 23 ns to execute a getter method, for 1 request in 1,000 the time was an order of magnitude worse, and for 1 request in 100,000 it was *two* orders of magnitude worse than average.

Long-tail distributions can also be referred to as *high dynamic range* distributions. The dynamic range of an observable is usually defined as the maximum recorded value divided by the minimum (assuming it's nonzero).

Logarithmic percentiles are a useful simple tool for understanding the long tail. However, for more sophisticated analysis, we can use a public domain library for handling datasets with high dynamic range. The library is called HdrHistogram and is available from GitHub. It was originally created by Gil Tene (Azul Systems), with additional work by Mike Barker and other contributors.

> **NOTE**
>
> A histogram is a way of summarizing data by using a finite set of ranges (called *buckets*) and displaying how often data falls into each bucket.

HdrHistogram is also available on Maven Central. At the time of writing, the current version is 2.1.12, and you can add it to your projects by adding this dependency stanza to *pom.xml*:

```
<dependency>
    <groupId>org.hdrhistogram</groupId>
    <artifactId>HdrHistogram</artifactId>
    <version>2.1.12</version>
</dependency>
```

Let's look at a simple example using HdrHistogram. This example takes in a

file of numbers and computes the HdrHistogram for the difference between successive results:

```
public class BenchmarkWithHdrHistogram {
    private static final long NORMALIZER = 1_000_000;

    private static final Histogram HISTOGRAM
            = new Histogram(TimeUnit.MINUTES.toMicros(1), 2);

    public static void main(String[] args) throws Exception {
        final List<String> values =
Files.readAllLines(Paths.get(args[0]));
        double last = 0;
        for (final String tVal : values) {
            double parsed = Double.parseDouble(tVal);
            double gcInterval = parsed - last;
            last = parsed;
            HISTOGRAM.recordValue((long)(gcInterval * NORMALIZER));
        }
        HISTOGRAM.outputPercentileDistribution(System.out, 1000.0);
    }
}
```

The output shows the times between successive garbage collections. As we'll see in Chapters 4 and 5, GC does do not occur at regular intervals, and understanding the distribution of how frequently it occurs could be useful. Here's what the histogram plotter produces for a sample GC log:

```
       Value     Percentile TotalCount 1/(1-Percentile)

       14.02 0.000000000000          1           1.00
     1245.18 0.100000000000         37           1.11
     1949.70 0.200000000000         82           1.25
     1966.08 0.300000000000        126           1.43
     1982.46 0.400000000000        157           1.67

...

    28180.48 0.996484375000        368         284.44
    28180.48 0.996875000000        368         320.00
    28180.48 0.997265625000        368         365.71
    36438.02 0.997656250000        369         426.67
    36438.02 1.000000000000        369
#[Mean    =      2715.12, StdDeviation   =      2875.87]
```

```
#[Max     =     36438.02, Total count    =          369]
#[Buckets =           19, SubBuckets     =          256]
```

The raw output of the formatter is rather hard to analyze, but fortunately, the HdrHistogram project includes an online formatter that can be used to generate visual histograms from the raw output.

For this example, it produces output like that shown in Figure 2-7.

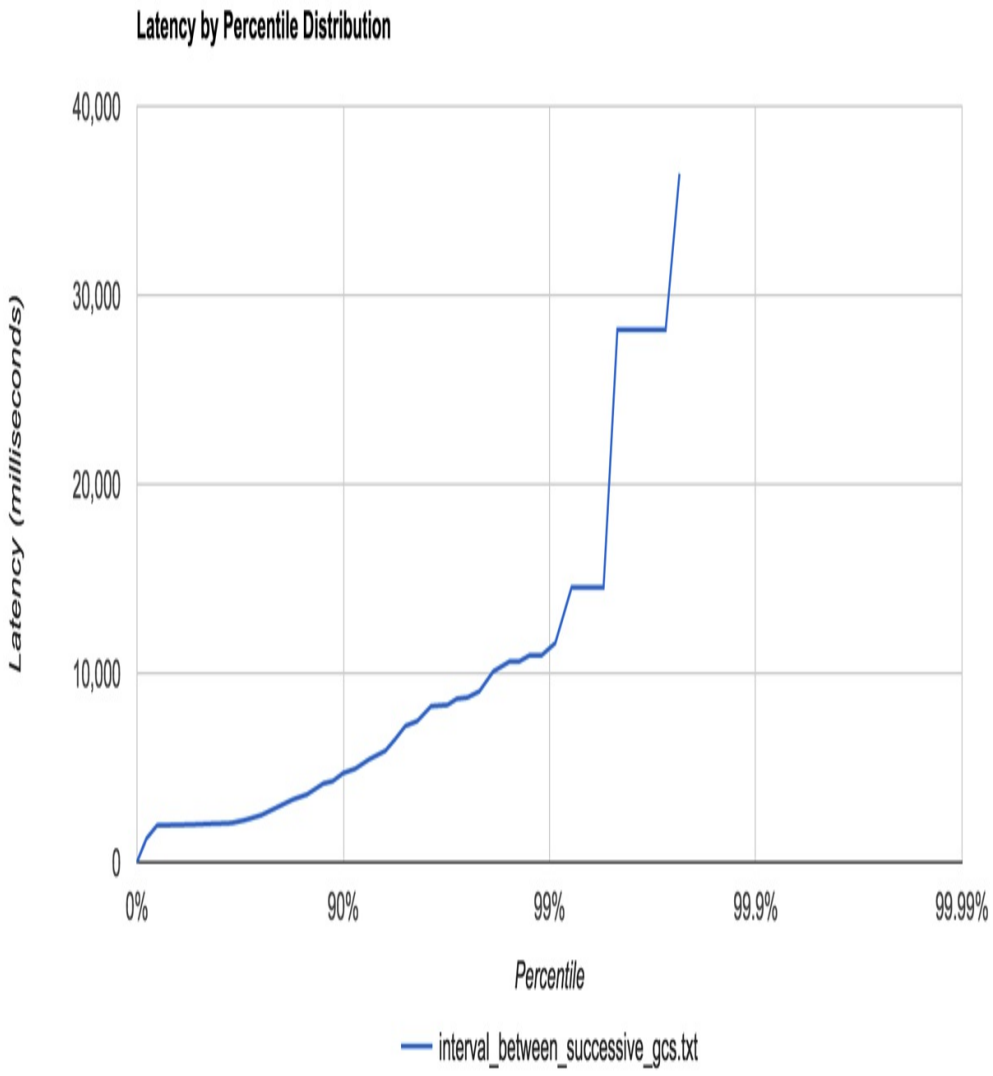


Figure 2-7. Example HdrHistogram visualization

For many observables that we wish to measure in Java performance tuning,

the statistics are often highly non-normal, and HdrHistogram can be a very useful tool in helping to understand and visualize the shape of the data.

Interpretation of Statistics

Empirical data and observed results do not exist in a vacuum, and it is quite common that one of the hardest jobs lies in interpreting the results that we obtain from measuring our applications.

> *No matter what the problem is, it's always a people problem.*
>
> —Gerald Weinberg (attr)

In Figure 2-8 we show an example memory allocation rate for a real Java application. This example is for a reasonably well-performing application.

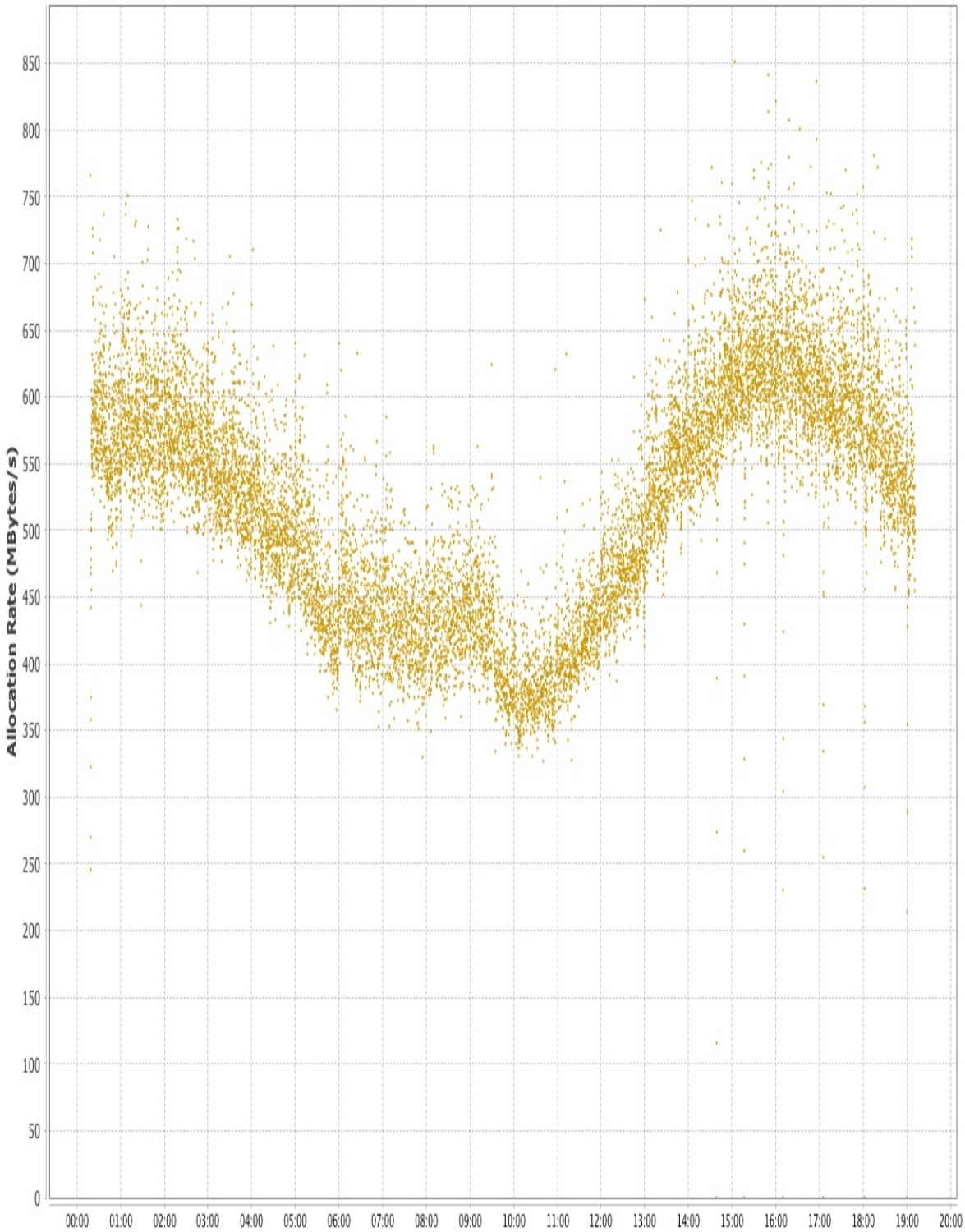


Figure 2-8. Example allocation rate

The interpretation of the allocation data is relatively straightforward, as there is a clear signal present. Over the time period covered (almost a day),

allocation rates were basically stable between 350 and 700 MB per second. There is a downward trend starting approximately 5 hours after the JVM started up, and a clear minimum between 9 and 10 hours, after which the allocation rate starts to rise again.

These types of trends in observables are very common, as the allocation rate will usually reflect the amount of work an application is actually doing, and this will vary widely depending on the time of day. However, when we are interpreting real observables, the picture can rapidly become more complicated.

This can lead to what is sometimes called the "Hat/Elephant" problem, after a passage in *The Little Prince* by Antoine de Saint-Exupéry. In the book, the narrator describes drawing, at age six, a picture of a boa constrictor that has eaten an elephant. However, as the view is external, the picture just resembles (at least to the ignorant eyes of the adults in the story) a slightly shapeless hat.

The metaphor stands as an admonition to the reader to have some imagination and to think more deeply about what you are really seeing, rather than just accepting a shallow explanation at face value.

The problem, as applied to software, is illustrated by Figure 2-9. All we can initially see is a complex histogram of HTTP request-response times. However, just like the narrator of the book, if we can imagine or analyze a bit more, we can see that the complex picture is actually made up of several fairly simple pieces.

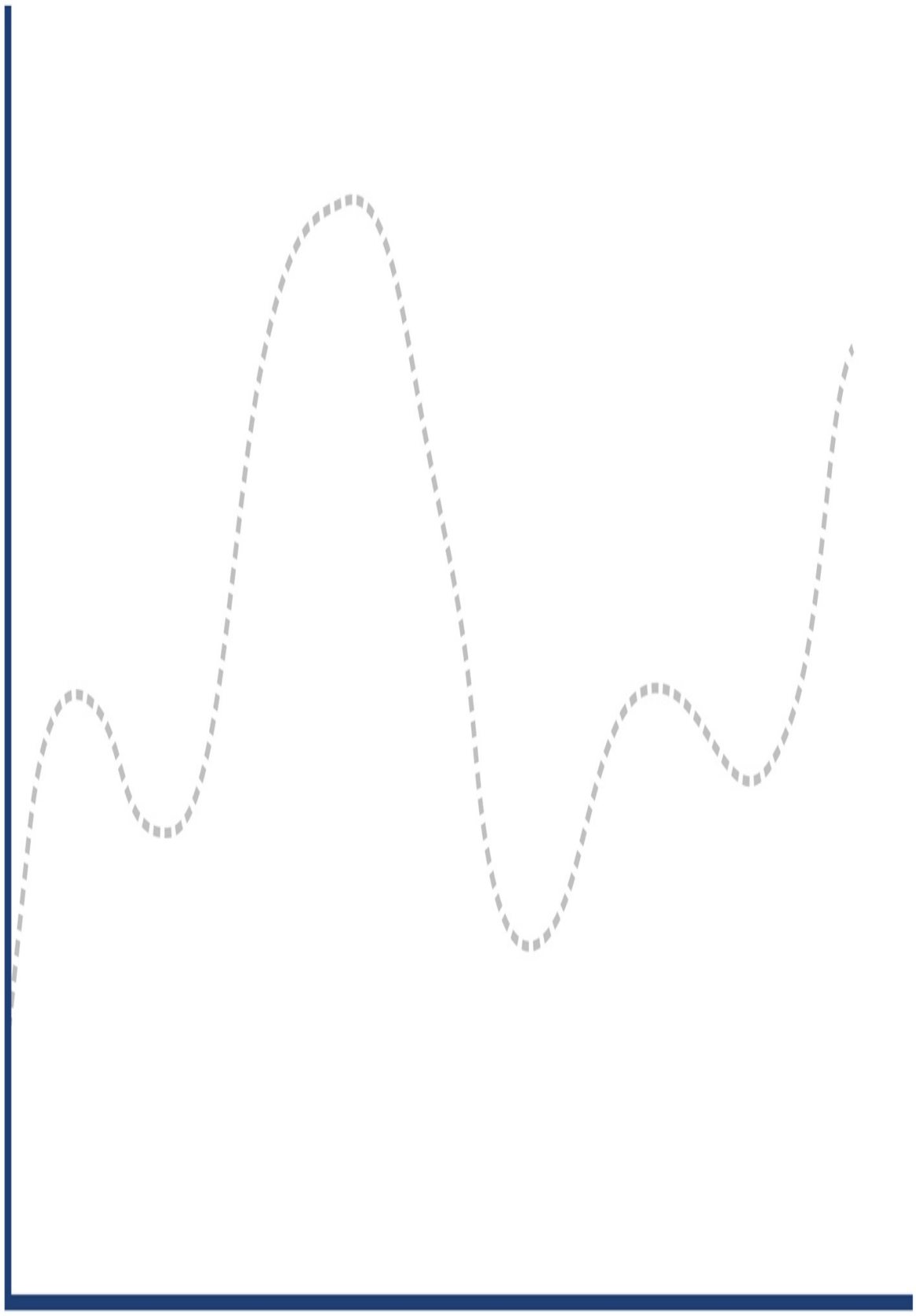


Figure 2-9. Hat, or elephant eaten by a boa?

The key to decoding the response histogram is to realize that “web application responses” is a very general category, including successful requests (so-called 2xx responses), client errors (4xx, including the infamous 404 error), and server errors (5xx, especially 500 Internal Server Error).

Each type of response has a different characteristic distribution for response times. If a client makes a request for a URL that has no mapping (a 404), then the web server can immediately reply with a response. This means that the histogram for only client error responses looks more like Figure 2-10.

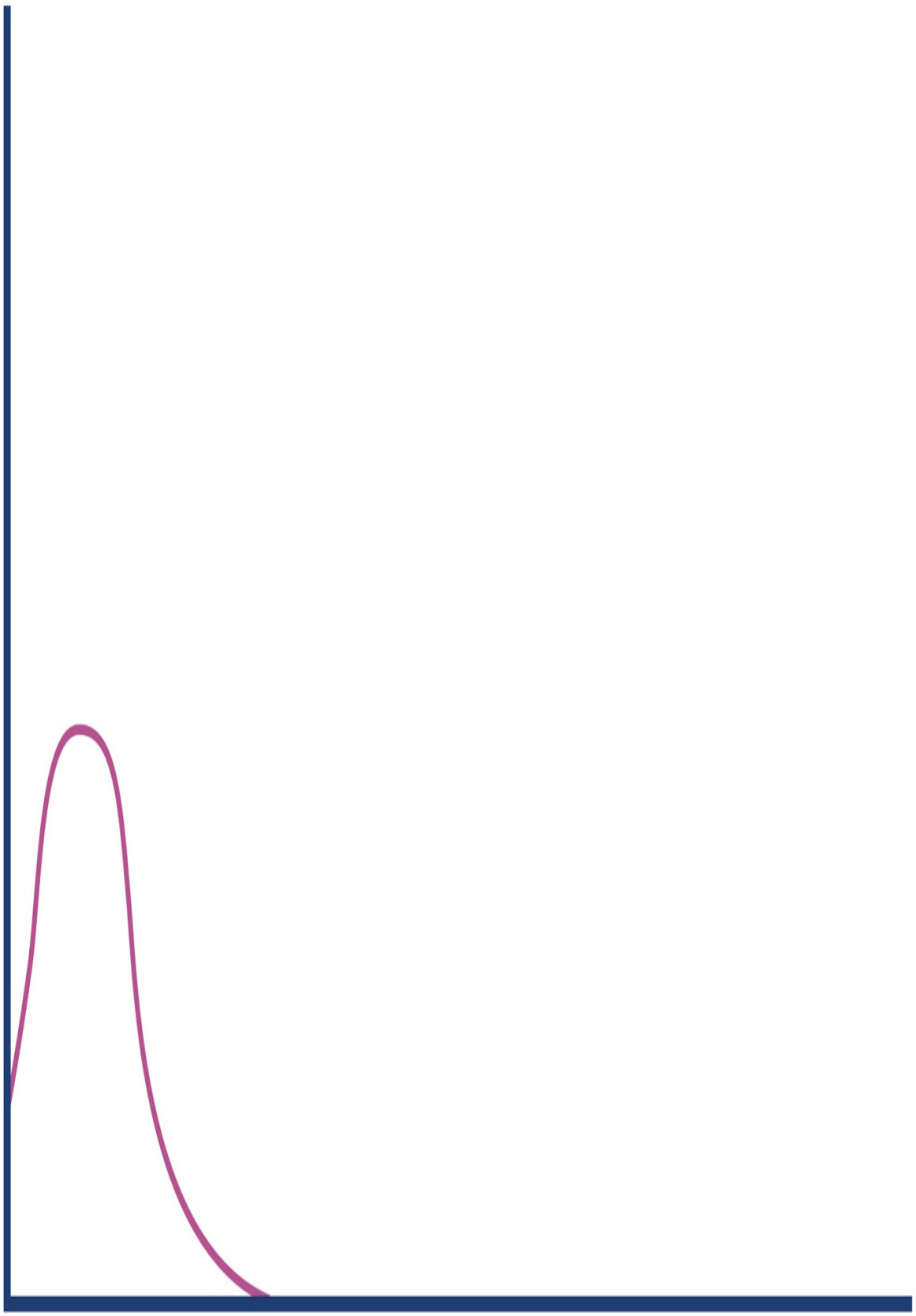


Figure 2-10. Client errors

By contrast, server errors often occur after a large amount of processing time has been expended (for example, due to backend resources being under stress or timing out). So, the histogram for server error responses might look like Figure 2-11.

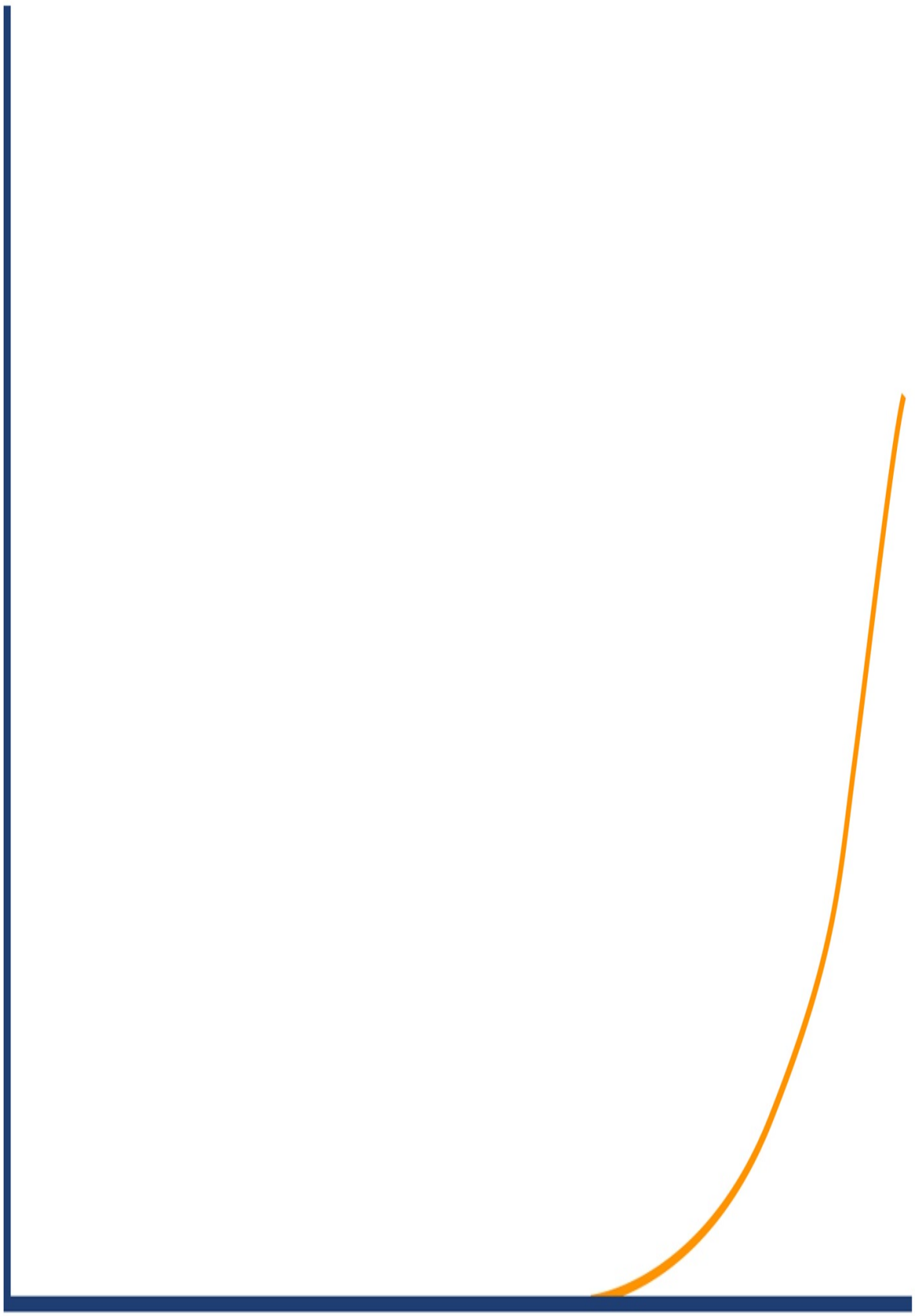


Figure 2-11. Server errors

The successful requests will have a long-tail distribution, but in reality we may expect the response distribution to be “multimodal” and have several local maxima. An example is shown in Figure 2-12, and represents the possibility that there could be two common execution paths through the application with quite different response times.

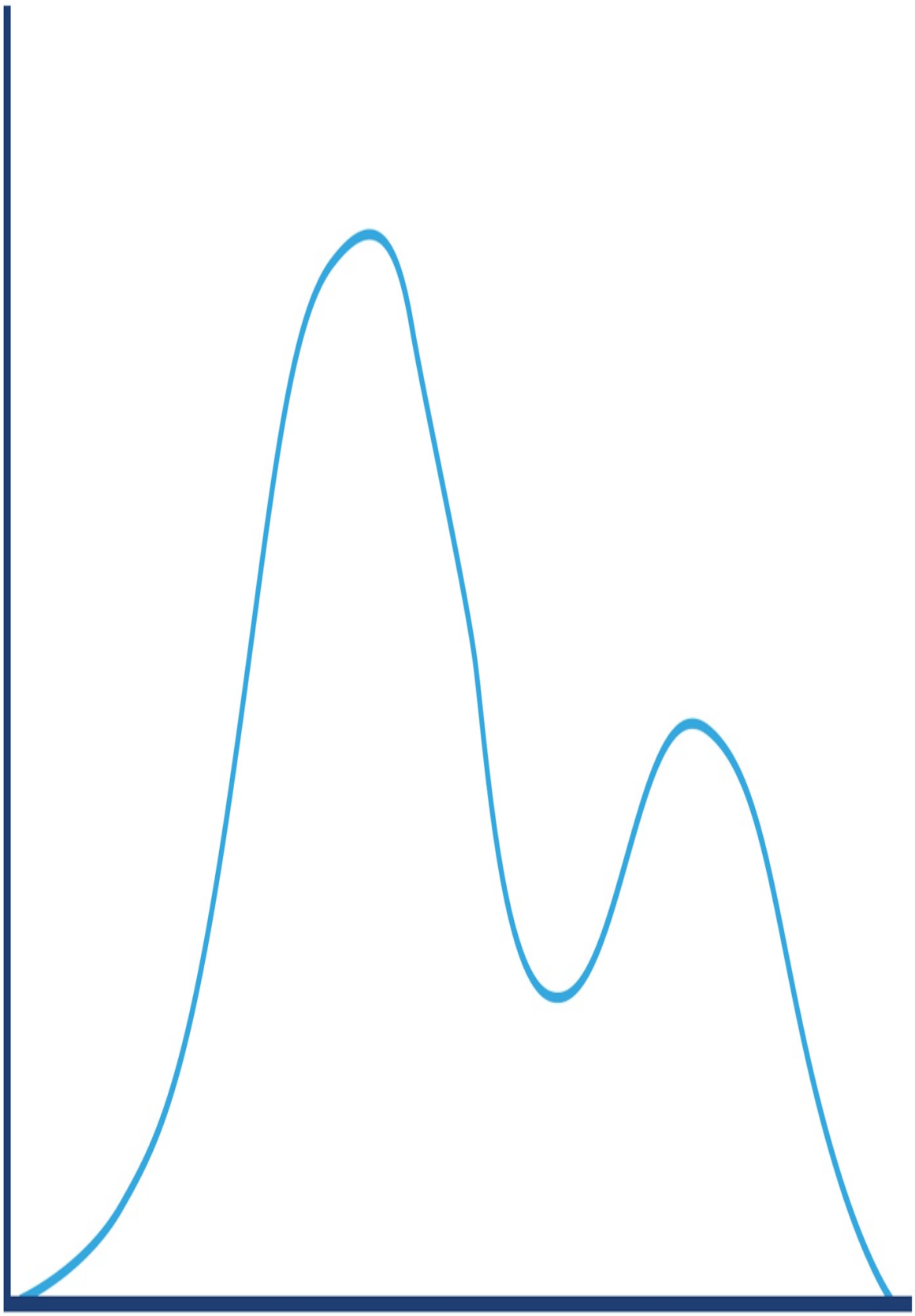


Figure 2-12. Successful requests

Combining these different types of responses into a single graph results in the structure shown in Figure 2-13. We have rederived our original “hat” shape from the separate histograms.

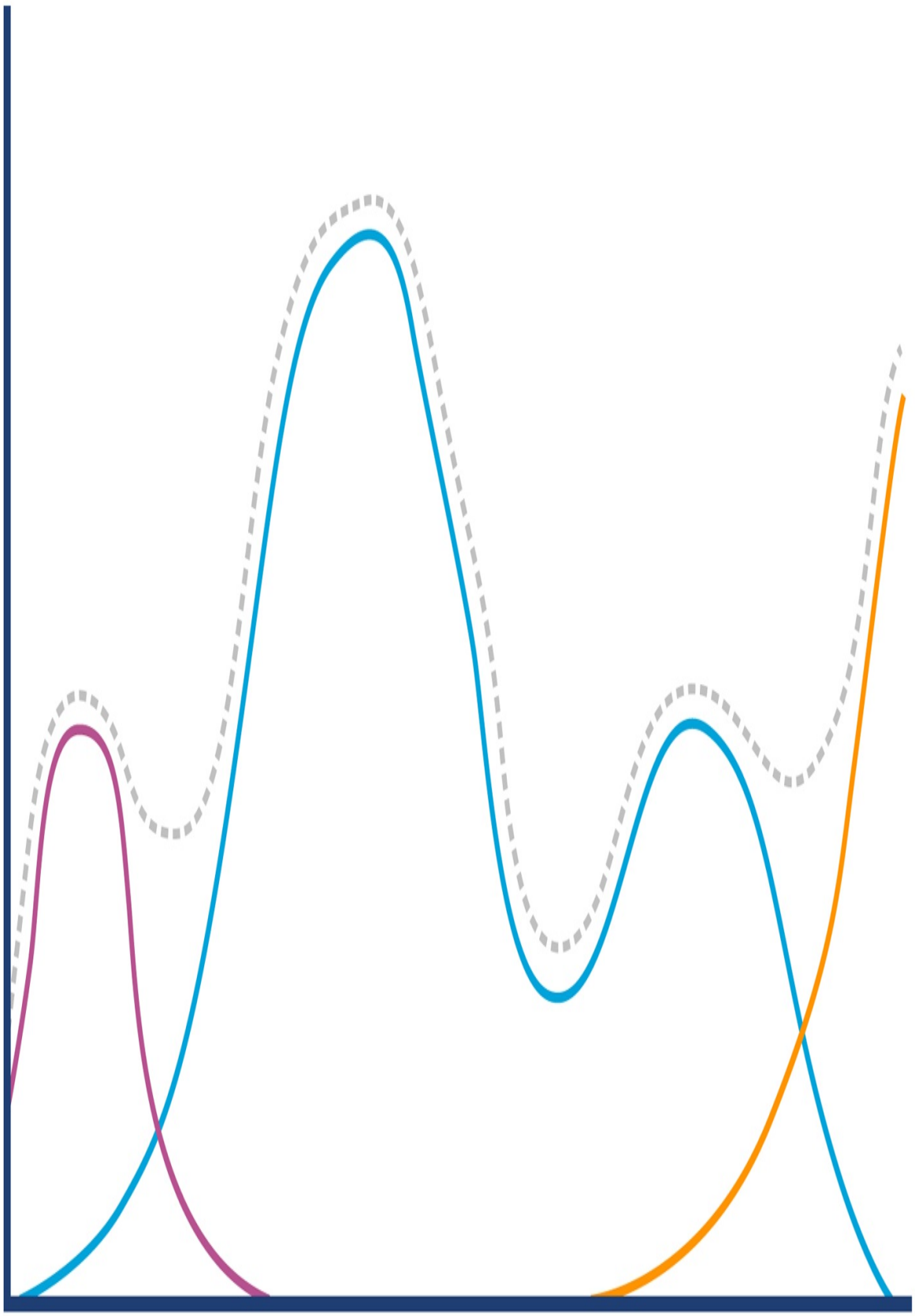


Figure 2-13. Hat or elephant revisited

The concept of breaking down a general observable into more meaningful sub-populations is a very useful one. It shows that we need to make sure that we understand our data and domain well enough before trying to infer conclusions from our results. We may well want to further break down our data into smaller sets; for example, the successful requests may have very different distributions for requests that are predominantly read, as opposed to requests that are updates or uploads.

The engineering team at PayPal have written extensively about their use of statistics and analysis; they have a blog that contains excellent resources. In particular, the piece "Statistics for Software" by Mahmoud Hashemi is a great introduction to their methodologies, and includes a version of the Hat/Elephant problem discussed earlier.

Also worth mentioning is the "Datasaurus Dozen" --a collection of datasets that have the same basic statistics but wildly different appearances.[6]

Cognitive Biases and Performance Testing

Humans can be bad at forming accurate opinions quickly—even when faced with a problem where they can draw upon past experiences and similar situations.

A cognitive bias is a psychological effect that causes the human brain to draw incorrect conclusions. It is especially problematic because the person exhibiting the bias is usually unaware of it and may believe they are being rational.

Many of the antipatterns we observe in performance analysis (such as those in Appendix B, which you might want to read in conjunction with this section) are caused, in whole or in part, by one or more cognitive biases that are in turn based on an unconscious assumptions.

For example, with the *Blame Donkey* antipattern, if a component has caused several recent outages the team may be biased to expect that same component

to be the cause of any new performance problem. Any data that's analyzed may be more likely to be considered credible if it confirms the idea that the Blame Donkey component is responsible.

The antipattern combines aspects of the biases known as confirmation bias and recency bias (a tendency to assume that whatever has been happening recently will keep happening).

> **NOTE**
>
> A single component in Java can behave differently from application to application depending on how it is optimized at runtime. In order to remove any pre-existing bias, it is important to look at the application as a whole.

Biases can be complementary or dual to each other. For example, some developers may be biased to assume that the problem is not software-related at all, and the cause must be the infrastructure the software is running on; this is common in the *Works for Me* antipattern, characterized by statements like "This worked fine in UAT, so there must be a problem with the production kit." The converse is to assume that every problem must be caused by software, because that's the part of the system the developer knows about and can directly affect.

Let's meet some of the most common biases that every performance engineer should look out for.

> *Knowing where the trap is—that's the first step in evading it.*[7]
>
> —Duke Leto Atreides I

By recognizing these biases in ourselves, and others, we increase the chance of being able to do sound performance analysis and solve the problems in our systems.

Reductionist Thinking

The reductionist thinking cognitive bias is based on an analytical approach

that presupposes that if you break a system into small enough pieces, you can understand it by understanding its constituent parts. Understanding each part means reducing the chance of incorrect assumptions being made.

The major problem with this view is simple to explain—in complex systems it just isn't true. Nontrivial software (or physical) systems almost always display emergent behavior, where the whole is greater than a simple summation of its parts would indicate.

Confirmation Bias

Confirmation bias can lead to significant problems when it comes to performance testing or attempting to look at an application subjectively. A confirmation bias is introduced, usually not intentionally, when a poor test set is selected or results from the test are not analyzed in a statistically sound way. Confirmation bias is quite hard to counter, because there are often strong motivational or emotional factors at play (such as someone in the team trying to prove a point).

Consider an antipattern such as *Distracted by Shiny*, where a team member is looking to bring in the latest and greatest NoSQL database. They run some tests against data that isn't like production data, because representing the full schema is too complicated for evaluation purposes.

They quickly prove that on a test set the NoSQL database produces superior access times on their local machine. The developer has already told everyone this would be the case, and on seeing the results they proceed with a full implementation. There are several antipatterns at work here, all leading to new unproved assumptions in the new library stack.

Fog of War (Action Bias)

The fog of war bias usually manifests itself during outages or situations where the system is not performing as expected and the team are under pressure. Some common causes include:

- Changes to infrastructure that the system runs on, perhaps without

notification or realizing there would be an impact

- Changes to libraries that the system is dependent on
- A strange bug or race condition the manifests itself, but only on busy days

In a well-maintained application with sufficient logging and monitoring, these should generate clear error messages that will lead the support team to the cause of the problem.

However, too many applications have not tested failure scenarios and lack appropriate logging. Under these circumstances even experienced engineers can fall into the trap of needing to feel that they're doing something to resolve the outage and mistaking motion for velocity—the "fog of war" descends.

At this time, many of the human elements discussed in this chapter can come into play if participants are not systematic about their approach to the problem.

For example, an antipattern such as *Blame Donkey* may shortcut a full investigation and lead the production team down a particular path of investigation—often missing the bigger picture. Similarly, the team may be tempted to break the system down into its constituent parts and look through the code at a low level without first establishing in which subsystem the problem truly resides.

Risk Bias

Humans are naturally risk averse and resistant to change. Mostly this is because people have seen examples of how change can cause things to go wrong—this leads them to attempt to avoid that risk. This can be incredibly frustrating when taking small, calculated risks could move the product forward. Much of this risk aversion arises from teams that are reluctant to make changes that might modify the performance profile of the application.

We can reduce this risk bias significantly by having a robust set of unit tests

and production regression tests. The performance regression tests are a great place to link in the system's non-functional requirements and ensure that the concerns the NFRs represent are reflected in the regression tests.

However, if either of these is not sufficiently trusted by the team, change becomes extremely difficult and the risk factor is not controlled. This bias often manifests in a failure to learn from application problems (including service outages) and implement appropriate mitigation.

Summary

When you are evaluating performance results, it is essential to handle the data in an appropriate manner and avoid falling into unscientific and subjective thinking. This includes avoiding the statistical pitfalls of relying upon Gaussian models when they are not appropriate.

In this chapter, we have met some different types of performance tests, testing best practices, and human problems that are native to performance analysis.

In the next chapter, we're going to move on to an overview of the JVM, introducing the basic subsystems, the lifecycle of a "classic" Java application and a first look at monitoring and tooling.

---

1 The term was popularized by the book *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, by William J. Brown, Raphael C. Malvo, Hays W. McCormick III, and Thomas J. Malbray (New York: Wiley, 1998).

2 M. Walton, *The Deming Management Method* (Mercury Books, 1989)

3 John H. McDonald, *Handbook of Biological Statistics*, 3rd ed. (Baltimore, MD: Sparky House Publishing, 2014).

4 The spurious correlations in this section come from Tyler Vigen's site and are reused here with permission under CC BY 4.0. If you enjoy them, there is a book with many more amusing examples available from his website.

5 R. Feynman and R. Leighton, "Surely You're Joking Mr Feynman" (W.W. Norton, 1985)

6 J. Matejka and G. Fitzmaurice, “Same stats, different graphs: Generating datasets with varied appearance and identical statistics through simulated annealing,” CHI 2017, Denver USA (2017)

7 F. Herbert, *Dune*, (Chilton Books 1965)

Chapter 3. Overview of the JVM

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *rfernando@oreilly.com*.

There is no doubt that Java is one of the largest technology platforms on the planet—the best available estimate is of over 10+ million developers working with Java.

The design of the Java system is *fully managed* --aspects such as garbage collection and execution optimization are controlled by the JVM on behalf of developers. The fact that Java is consciously aimed at mainstream developers, when combined with the fully-managed platform, leads to a situation in which many developers do not need to know about the low-level intricacies of the platform they work with on a daily basis. As a result, developers may not meet these internal aspects very frequently—but only when an issue such as a customer complaining about a performance problem arises.

For developers who are interested in performance, however, it is important to understand the basics of the JVM technology stack. Understanding JVM technology enables developers to write better software and provides the theoretical background required for investigating performance-related issues.

This chapter introduces how the JVM executes Java in order to provide a basis for deeper exploration of these topics later in the book. In particular,

Chapter 6 has an in-depth treatment of bytecode, which is complementary to the discussion here.

We suggest that you read through this chapter, but come back to it for a second pass after you have read Chapter 6.

Interpreting and Classloading

According to the specification that defines the Java Virtual Machine (usually called the VM Spec), the JVM is a stack-based interpreted machine. This means that rather than having registers (like a physical hardware CPU), it uses an execution stack of partial results and performs calculations by operating on the top value (or values) of that stack.

If you're not familiar with how interpreters work, then you can think of the basic behavior of the JVM interpreter as essentially "a `switch` inside a `while` loop". The interpreter processes each opcode of the program independently of the last, and uses the evaluation stack to hold the results of computations and as intermediate results.

> **NOTE**
>
> As we will see when we delve into the internals of the Oracle/OpenJDK VM (HotSpot), the situation for real production-grade Java interpreters is more complex, but *switch-inside-while* using a stack interpreter is an acceptable mental model for the moment.

When we launch our application using the `java HelloWorld` command, the operating system starts the virtual machine process (the `java` binary). This sets up the Java virtual environment and initializes the interpreter that will actually execute the user code in the `HelloWorld.class` file.

The entry point into the application will be the `main()` method of `HelloWorld.class`. In order to hand over control to this class, it must be loaded by the virtual machine before execution can begin.

To achieve this, the Java classloading mechanism is used. When a new Java

process is initializing, a chain of classloaders is used. The initial loader is known as the Bootstrap classloader (historically also known as the "primordial classloader") and it loads classes in the core Java runtime. The main point of the Bootstrap classloader is to get a minimal set of classes (which includes essentials such as `java.lang.Object`, `Class`, and `Classloader`) loaded to allow other classloaders to bring up the rest of the system.

At this point it is also instructive to discuss a little bit of how the Java modules system (sometimes referred to as JPMS) has somewhat changed the picture of application startup. First of all, from Java 9 onwards, all JVMs are modular—there is no "compatibility" or "classic" mode that restores the Java 8 monolithic JVM runtime.

This means that during startup a module graph is always constructed—even if the application itself is non-modular. This must be a Directed Acyclic Graph (DAG), and it is a fatal error if the application's module metadata attempts to construct a module graph that contains a cycle.

The module graph has various advantages, including:

- Only required modules are loaded
- Inter-module metadata can be confirmed to be good at startup time

The module graph has a main module, which is where the entrypoint class lives. If the application has not yet been fully modularized, then it will have both a modulepath and a classpath, and the application code may be in the `UNNAMED` module.

> **NOTE**
>
> Full details of the modules system are outside the scope of this book. An expanded treatment can be found in *Java in a Nutshell (8th Edition)* by Benjamin J. Evans, Jason Clark and David Flanagan (O'Reilly) or a more in-depth reference, such as *Java 9 Modularity* by Sander Mak and Paul Bakker (O'Reilly).

In practice, the work of the Bootstrap classloader involves loading `java.base` and some other supporting modules (including some perhaps-surprising entries—e.g. `java.security.sasl` and `java.datatransfer`)

Java models classloaders as objects within its own runtime and type system, so there needs to be some way to bring an initial set of classes into existence. Otherwise, there would be a circularity problem in defining what a classloader is.

The Bootstrap classloader does not verify the classes it loads (largely to improve startup performance), and it relies on the boot classpath being secure. Anything loaded by the bootstrap classloader is granted full security permissions and so this group of modules is kept as restricted as possible.

> **NOTE**
>
> Legacy versions of Java up to and including 8 used a monolithic runtime, and the Bootstrap classloader loaded the contents of `rt.jar`.

The rest of the base system (i.e. the equivalent of the rest of the old `rt.jar` used in version 8 and earlier) is loaded by the *platform classloader*, and is available via the method `ClassLoader::getPlatformClassLoader`. It has the Bootstrap classloader as its parent, as the old Extension classloader has been removed.

In the new modular implementations of Java, far less code is required to bootstrap a Java process and accordingly, as much JDK code (now represented as modules) as possible has been moved out of the scope of the bootstrap loader and into the platform loader instead.

Finally, the Application classloader is created; it is responsible for loading user classes from the defined classpath. Some texts unfortunately refer to this as the “System” classloader. This term should be avoided, for the simple reason that it doesn’t load the system classes (the Bootstrap and Platform classloaders do). The Application classloader is encountered extremely

frequently, and it has the Platform loader as its parent.

Java loads in dependencies on new classes when they are first encountered during the execution of the program. If a classloader fails to find a class, the behavior is usually to delegate the lookup to the parent. If the chain of lookups reaches the Bootstrap classloader and it isn't found, a `ClassNotFoundException` will be thrown. It is important that developers use a build process that effectively compiles with the exact same classpath that will be used in production, as this helps to mitigate this potential issue.

Normally, Java only loads a class once and a `Class` object is created to represent the class in the runtime environment. However, it is important to realize that under some circumstances the same class can be loaded twice by different classloaders. As a result, a class in the system is identified by the classloader used to load it as well as the fully qualified class name (which includes the package name).

> **NOTE**
>
> Some execution contexts, such as application servers (e.g. Tomcat or JBoss EAP) display this behavior when multiple tenant applications are present in the server.

It is also the case that some tools (e.g. Java agents) can potentially reload and retransform classes as part of bytecode weaving—and such tools are often used in monitoring and Observability.

Executing Bytecode

It is important to appreciate that Java source code goes through a significant number of transformations before execution. The first is the compilation step using the Java compiler `javac`, often invoked as part of a larger build process.

The job of `javac` is to convert Java code into *.class* files that contain

bytecode. It achieves this by doing a fairly straightforward translation of the Java source code, as shown in Figure 3-1. Very few optimizations are done during compilation by `javac`, and the resulting bytecode is still quite readable and recognizable as Java code when viewed in a disassembly tool, such as the standard `javap`.

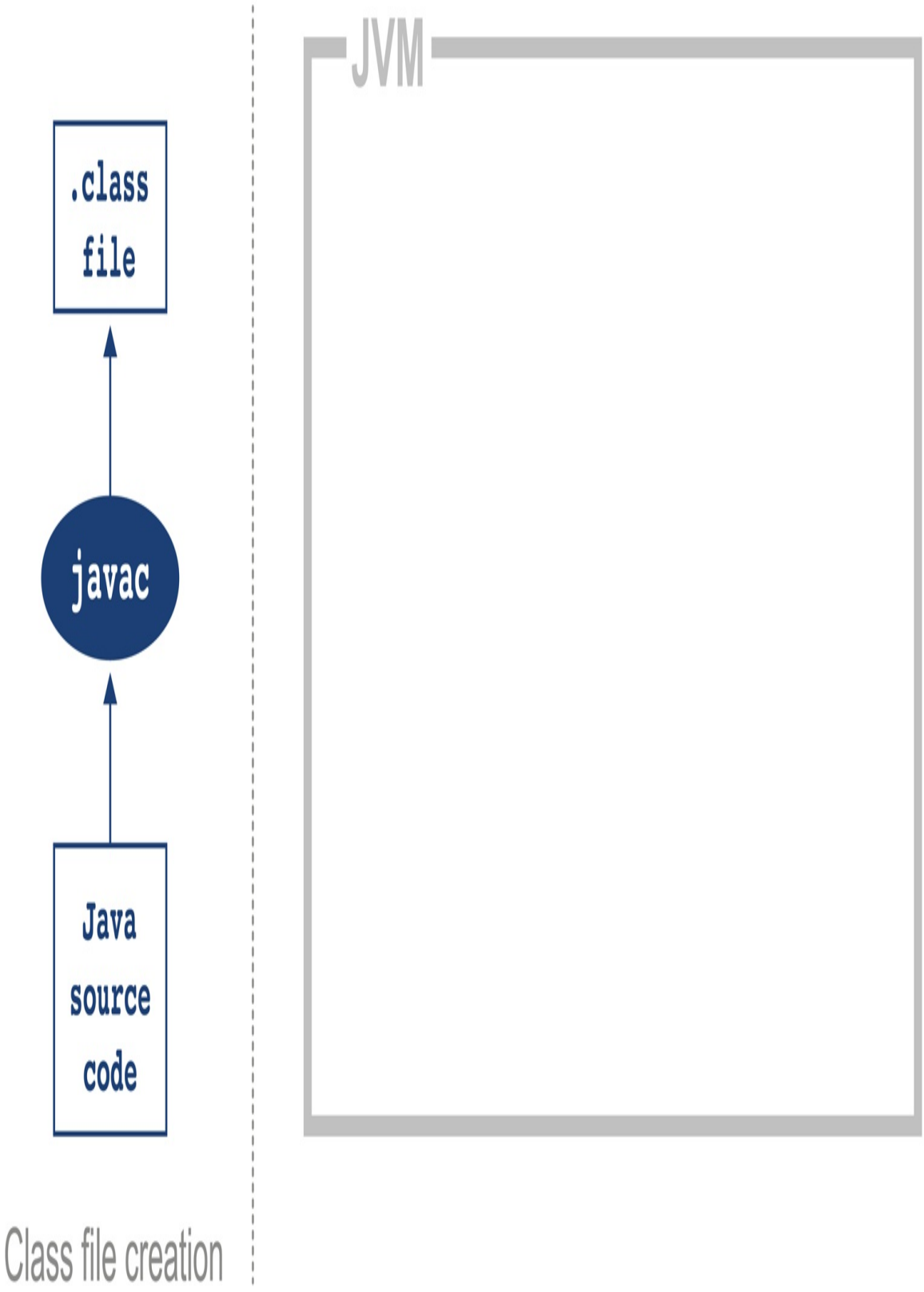


Figure 3-1. Java class file compilation

Bytecode is an intermediate representation that is not tied to a specific machine architecture. Decoupling from the machine architecture provides portability, meaning already developed (or compiled) software can run on any platform supported by the JVM and provides an abstraction from the Java language. This provides our first important insight into the way the JVM executes code.

> **NOTE**
>
> The Java language and the Java Virtual Machine are now to a degree independent, and so the J in JVM is potentially a little misleading, as the JVM can execute any JVM language that can produce a valid class file. In fact, Figure 3-1 could just as easily show the Kotlin compiler `kotlinc` generating bytecode for execution on the JVM.

Regardless of the source code compiler used, the resulting class file has a very well-defined structure specified by the VM spec (Table 3-1). Any class that is loaded by the JVM will be verified to conform to the expected format before being allowed to run.

Table 3-1. Anatomy of a class file

Component	Description
Magic number	`0xCAFEBABE`
Version of class file format	The minor and major versions of the class file
Constant pool	The pool of constants for the class
Access flags	Whether the class is abstract, static, and so on
This class	The name of the current class

Superclass	The name of the superclass
Interfaces	Any interfaces in the class
Fields	Any fields in the class
Methods	Any methods in the class
Attributes	Any attributes of the class (e.g., name of the source file, etc.)

Every class file starts with the magic number `0xCAFEBABE`, the first 4 bytes in hexadecimal serving to denote conformance to the class file format. The following 4 bytes represent the minor and major versions used to compile the class file, and these are checked to ensure that the version of the JVM is not of a lower version than the one used to compile the class file. The major and minor version are checked by the classloader to ensure compatibility; if these are not compatible an `UnsupportedClassVersionError` will be thrown at runtime, indicating the runtime is a lower version than the compiled class file.

> **NOTE**
>
> Magic numbers provide a way for Unix environments to identify the type of a file (whereas Windows will typically use the file extension). For this reason, they are difficult to change once decided upon. Unfortunately, this means that Java is stuck using the rather embarrassing and sexist `0xCAFEBABE` for the foreseeable future, although Java 9 introduced the magic number `0xCAFEDADA` for module files.

The constant pool holds constant values in code: for example, names of classes, interfaces, and fields. When the JVM executes code, the constant pool table is used to refer to values rather than having to rely on the precise

layout of memory structures at runtime.

Access flags are used to determine the modifiers applied to the class. The first part of the flag block identifies general properties, such as whether a class is public, followed by whether it is final and thus cannot be subclassed. The flags also determine whether the class file represents an interface or an abstract class. The final part of the flag block indicates whether the class file represents a synthetic class (not present in source code), an annotation type, or an enum.

The `this` class, superclass, and interface entries are indexes into the constant pool to identify the type hierarchy belonging to the class. Fields and methods define a signature-like structure, including the modifiers that apply to the field or method. A set of attributes is then used to represent structured items for more complicated and non-fixed-size structures. For example, methods make use of the `Code` attribute to represent the bytecode associated with that particular method.

Figure 3-2 provides a mnemonic for remembering the structure.

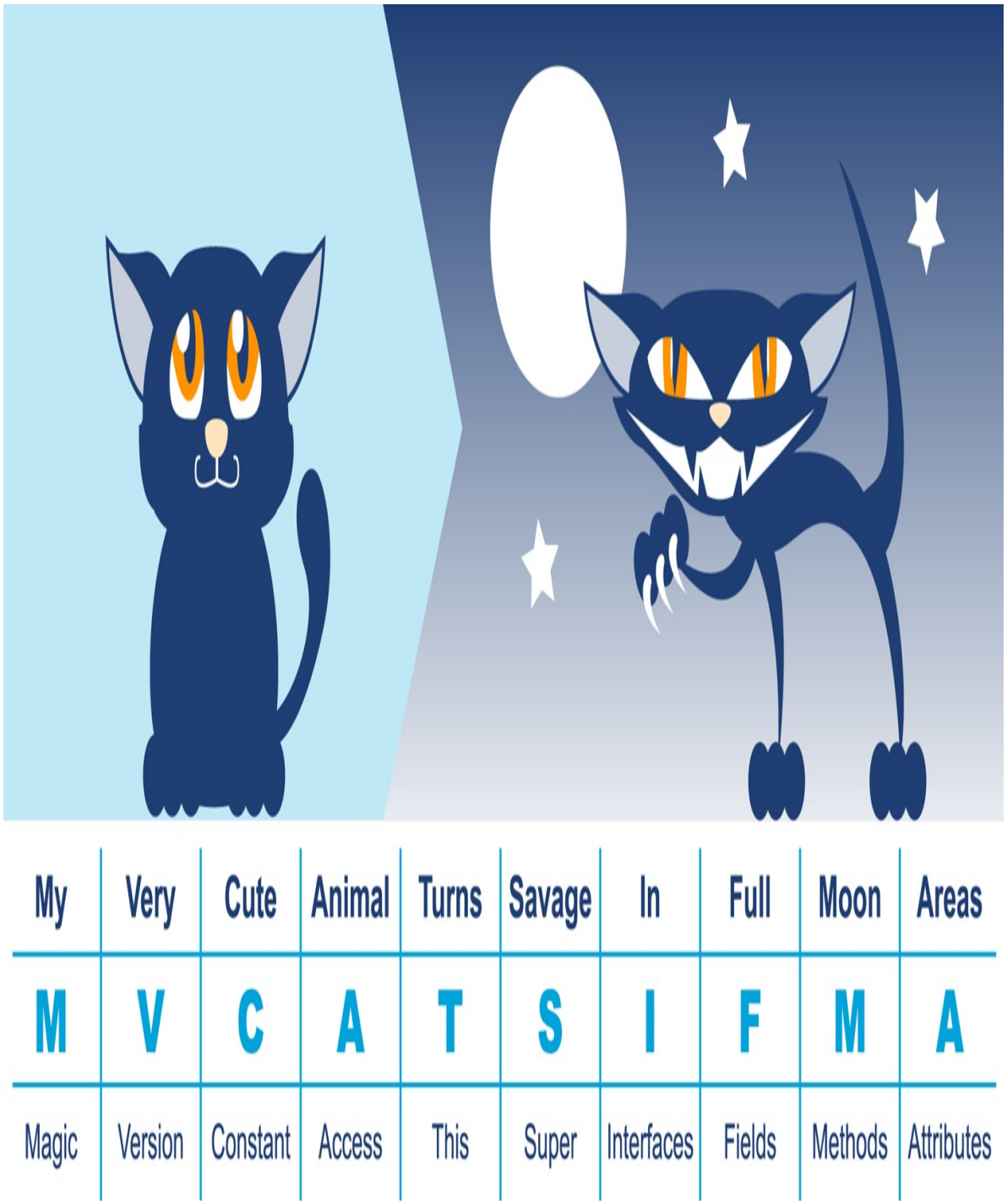


Figure 3-2. Mnemonic for class file structure

In this very simple code example, it is possible to observe the effect of running `javac`:

```
public class HelloWorld {
    public static void main(String[] args) {
```

```
        for (int i = 0; i < 10; i++) {
            System.out.println("Hello World");
        }
    }
}
```

Java ships with a class file disassembler called `javap`, allowing inspection of *.class* files. Taking the *HelloWorld* class file and running `javap -c HelloWorld` gives the following output:

```
public class HelloWorld {
  public HelloWorld();
    Code:
       0: aload_0
       1: invokespecial #1    // Method java/lang/Object."<init>":
()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: iconst_0
       1: istore_1
       2: iload_1
       3: bipush        10
       5: if_icmpge     22
       8: getstatic     #2    // Field java/lang/System.out ...
      11: ldc           #3    // String Hello World
      13: invokevirtual #4    // Method java/io/PrintStream.println
...
      16: iinc          1, 1
      19: goto          2
      22: return
}
```

This layout describes the bytecode for the file *HelloWorld.class*. For more detail `javap` also has a `-v` option that provides the full class file header information and constant pool details. The class file contains two methods, although only the single `main()` method was supplied in the source file; this is the result of `javac` automatically adding a default constructor to the class.

The first instruction executed in the constructor is `aload_0`, which places the `this` reference onto the first position in the stack. The

`invokespecial` command is then called, which invokes an instance method that has specific handling for calling superconstructors and creating objects. In the default constructor, the invoke matches the default constructor for `Object`, as an override was not supplied.

> **NOTE**
>
> Opcodes in the JVM are concise and represent the type, the operation, and the interaction between local variables, the constant pool, and the stack.

Moving on to the `main()` method, `iconst_0` pushes the integer constant `0` onto the evaluation stack. `istore_1` stores this constant value into the local variable at offset 1 (represented as `i` in the loop). Local variable offsets start at 0, but for instance methods, the 0th entry is always `this`. The variable at offset 1 is then loaded back onto the stack and the constant `10` is pushed for comparison using `if_icmpge` ("if integer compare greater or equal"). The test only succeeds if the current integer is >= 10.

For the first 10 iterations, this comparison test fails and so we continue to instruction 8. Here the static method from `System.out` is resolved, followed by the loading of the "Hello World" string from the constant pool. The next invoke, `invokevirtual`, invokes an instance method based on the class. The integer is then incremented and `goto` is called to loop back to instruction 2.

This process continues until the `if_icmpge` comparison eventually succeeds (when the loop variable is >= 10); on that iteration of the loop, control passes to instruction 22 and the method returns.

Introducing HotSpot

In April 1999 Sun introduced one of the biggest-ever changes (in terms of performance) to the dominant Java implementation. The HotSpot virtual machine is a key feature that has evolved to enable performance that is

comparable to (or better than) languages such as C and C++ (see Figure 3-3). To explain how this is possible, let’s delve a little deeper into the design of languages intended for application development.

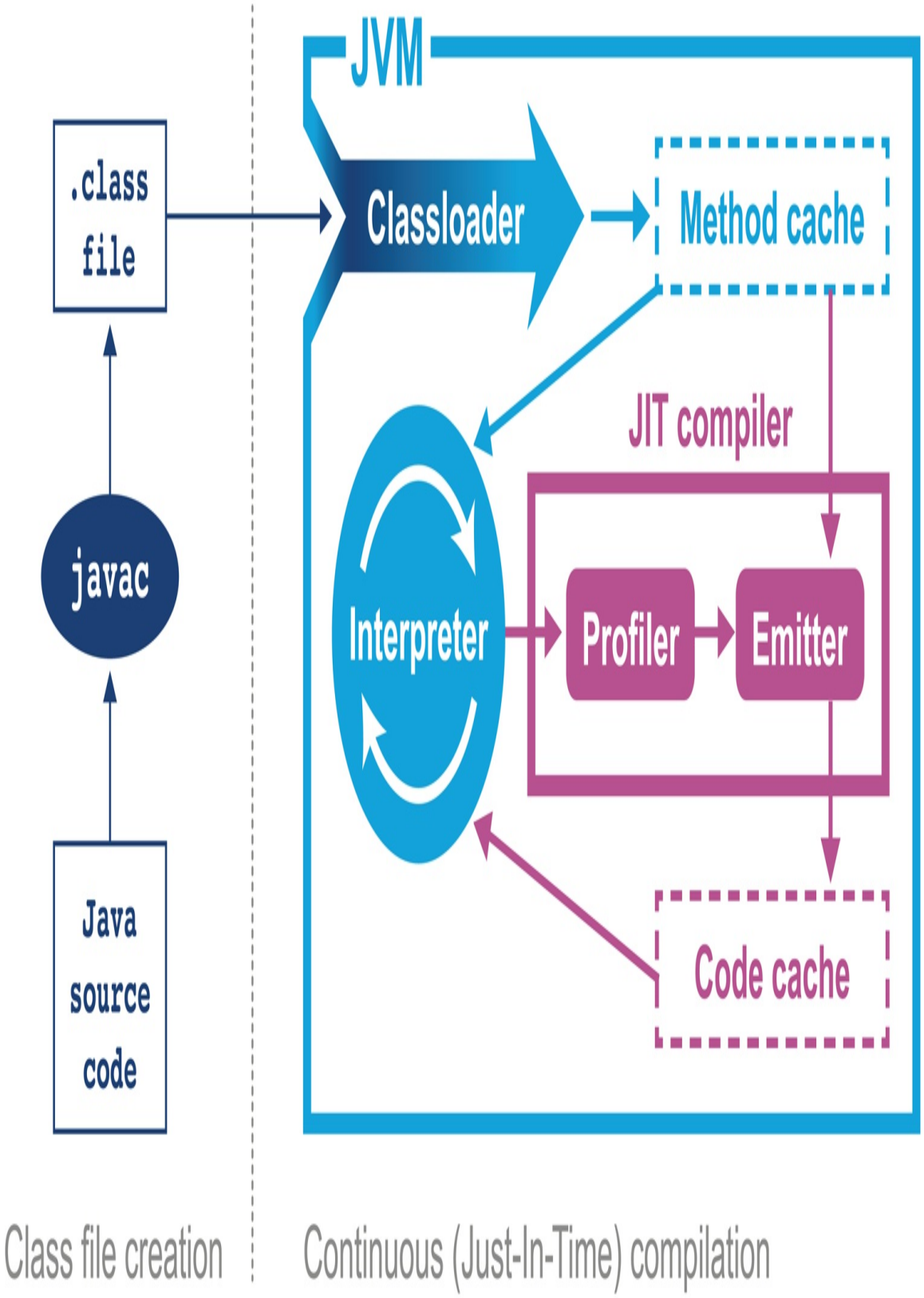


Figure 3-3. The HotSpot JVM

Language and platform design frequently involves making decisions and tradeoffs between desired capabilities. In this case, the division is between languages that stay "close to the metal" and rely on ideas such as "zero-cost abstractions," and languages that favor developer productivity and "getting things done" over strict low-level control.

> *In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better.*[1]
>
> —Bjarne Stroustrup

The zero-overhead principle sounds great in theory, but it requires all users of the language to deal with the low-level reality of how operating systems and computers actually work. This is a significant extra cognitive burden that is placed upon developers who may not care about raw performance as a primary goal.

Not only that, but it also requires the source code to be compiled to platform-specific machine code at build time—usually called *Ahead-of-Time* (AOT) compilation. This is because alternative execution models such as interpreters, virtual machines, and portability layers all are most definitely not zero-overhead.

The phrase "what you do use, you couldn't hand code any better" also has a sting in its tail. It implies a number of things, but most important for our purposes is that a developer is not able to produce better code than an automated system (such as a compiler).

Java has never subscribed to the zero-overhead abstraction philosophy. Instead, the approach taken by the HotSpot virtual machine is to analyze the runtime behavior of your program and intelligently apply optimizations where they will benefit performance the most. The goal of the HotSpot VM is to allow you to write idiomatic Java and follow good design principles rather then contort your program to fit the VM.

Introducing Just-in-Time Compilation

Java programs begin their execution in the bytecode interpreter, where instructions are performed on a virtualized stack machine. This abstraction from the CPU gives the benefit of class file portability, but to get maximum performance your program must make optimal use of its native features.

HotSpot achieves this by compiling units of your program from interpreted bytecode into native code, which then executes directly, without requiring the overhead of the abstractions of the interpreter. The units of compilation in the HotSpot VM are the method and the loop. This is known as *Just-in-Time* (JIT) compilation.

JIT compilation works by monitoring the application while it is running in interpreted mode and observing the parts of code that are most frequently executed. During this analysis process, programmatic trace information is captured that allows for more sophisticated optimization. Once execution of a particular method passes a threshold, the profiler will look to compile and optimize that particular section of code.

There are many advantages to the JIT approach to compilation, but one of the main ones is that it bases compiler optimization decisions on trace information that is collected while methods are being interpreted. This information enables HotSpot to make more informed optimizations if the method is eligible for compilation.

> **NOTE**
>
> Some JIT compilers also have the capability to re-JIT if a better optimization becomes apparent later on during execution. This includes some of HotSpot's compilers.

Not only that, but HotSpot has had hundreds of engineering years (or more) of development attributed to it and new optimizations and benefits are added with almost every new release. This means that all Java applications benefit from the latest HotSpot performance optimizations in the VM without even needing to be recompiled.

> **TIP**
>
> After being translated from Java source to bytecode and now going through another step of (JIT) compilation, the code actually being executed has changed very significantly from the source code as written. This is a key insight, and it will drive our approach to dealing with performance-related investigations. JIT-compiled code executing on the JVM may well look nothing like the original Java source code.

The general picture is that languages like C++ (and the up-and-coming Rust) tend to have more predictable performance, but at the cost of forcing a lot of low-level complexity onto the user.

Note also that "more predictable" does not necessarily mean "better." AOT compilers produce code that may have to run across a broad class of processors, and usually are not able to assume that specific processor features are available.

Environments that use profile-guided optimization (PGO), such as Java, have the potential to use runtime information in ways that are simply impossible to most AOT platforms. This can offer improvements to performance, such as dynamic inlining and optimizing away virtual calls. HotSpot can even detect the precise CPU type it is running on at VM startup, and can use this information to enable optimizations designed for specific processor features if available.

> **TIP**
>
> The technique of detecting precise processor capabilities is known as *JVM intrinsics*, and is not to be confused with the intrinsic locks introduced by the `synchronized` keyword.

A full discussion of PGO and JIT compilation can be found in Chapters 6 and 10.

The sophisticated approach that HotSpot takes is a great benefit to the majority of ordinary developers, but this tradeoff (to abandon zero-overhead abstractions) means that in the specific case of high-performance Java

applications, the developer must be very careful to avoid “common sense” reasoning and overly simplistic mental models of how Java applications actually execute.

> **NOTE**
>
> Once again, analyzing the performance of small sections of Java code (*microbenchmarks*) is usually much harder than analyzing entire applications, and is a very specialized task that the majority of developers should not undertake.

HotSpot’s compilation subsystem is one of the two most important subsystems that the virtual machine provides. The other is automatic memory management, which has been one of the major selling points of Java since the early years.

JVM Memory Management

In languages such as C, C++, and Objective-C the programmer is responsible for managing the allocation and release of memory. The benefits of managing memory and lifetime of objects yourself are more deterministic performance and the ability to tie resource lifetime to the creation and deletion of objects. However, these benefits come at a huge cost—for correctness, developers must be able to accurately account for memory.

Unfortunately, decades of practical experience showed that many developers have a poor understanding of idioms and patterns for memory management. Later versions of C++ and Objective-C have improved this using smart pointer idioms in the standard library. However, at the time Java was created poor memory management was a major cause of application errors. This led to concern among developers and managers about the amount of time spent dealing with language features rather than delivering value for the business.

Java looked to help resolve the problem by introducing automatically managed heap memory using a process known as *garbage collection* (GC). Simply put, garbage collection is a nondeterministic process that triggers to

recover and reuse no-longer-needed memory when the JVM requires more memory for allocation.

GC comes at a cost: when it runs, it traditionally *stopped the world*, which means while GC is in progress the application pauses. Usually these pause times are incredibly short, but as an application is put under pressure they can increase.

Having said that, the JVM's garbage collection is best-in-class, and is far more sophisticated than the introductory algorithm that is often taught in Computer Science undergraduate courses. For example, stopping the world is much less necessary and intrusive in modern algorithms, as we will see later.

Garbage collection is a major topic within Java performance optimization, so we will devote Chapters 4 and 5 to the details of Java GC.

Threading and the Java Memory Model

One of the major advances that Java brought in with its first version was built-in support for multithreaded programming. The Java platform allows the developer to create new threads of execution. For example, in Java 8 syntax:

```
Thread t = new Thread(() -> {System.out.println("Hello World!");});
t.start();
```

Not only that, but basically all production JVMs are multithreaded—and this means that all Java programs are inherently multithreaded, as they execute as part of a JVM process.

This fact produces additional, irreducible complexity in the behavior of Java programs, and makes the work of the performance analyst harder. However, it allows the JVM to take advantage of all available cores, which provides all sorts of performance benefits to the Java developer.

The relationship between Java's conception of a thread (an "application thread") and the operating system's view of a thread (a "platform thread") has

a slightly interesting history. In the very earliest days of the platform, there was a sharp distinction made between the two concepts and application threads were *remapped* or *multiplexed* onto a pool of platform threads—e.g. in the Solaris *M:N*, or the Linux *green threads* models.

However, this approach proved not to provide an acceptable performance profile and added needless complexity. As a result, in most mainstream JVM implementations, this model was replaced with a simpler one—each Java application thread corresponding precisely to a dedicated platform thread.

This is not the end of the story, however.

In the 20+ years since the "app thread == platform thread" transition, applications have grown and scaled massively—and so has the number of threads (or, more generally, *execution contexts*) that an application might want to create. This has led to the "thread bottleneck" problem, and solving it has been the focus of a major research project within OpenJDK (Project Loom).

The result is *virtual threads*, a new form of thread only available in Java 21+, which can be used efficiently for certain types of task—especially those performing network I/O.

Programmers must explicitly choose to create a thread as virtual—otherwise they are platform threads and retain the same behavior as before (so the semantics of all existing Java programs are preserved when run on a JVM with virtual thread capability).

> **NOTE**
>
> It is safe to assume that every platform thread (or any thread, before Java 21) is backed by a unique OS thread that is created when the `start()` method is called on the corresponding `Thread` object.

Virtual threads are Java's take on an idea that can be found in various other modern languages—for example, Go programmers may regard a Java virtual thread as being broadly similar to a goroutine. We will discuss virtual threads

in more detail in Chapter 14.

We should also briefly discuss Java's approach to handling data in a multithreaded program. It dates from the late 1990s and has these fundamental design principles:

- All threads in a Java process share a single, common garbage-collected heap.
- Any object created by one thread can be accessed by any other thread that has a reference to the object.
- Objects are mutable by default; that is, the values held in object fields can be changed unless the programmer explicitly uses the `final` keyword to mark them as immutable.

The Java Memory Model (JMM) is a formal model of memory that explains how different threads of execution see the changing values held in objects. That is, if threads A and B both have references to object `obj`, and thread A alters it, what happens to the value observed in thread B?

This seemingly simple question is actually more complicated than it seems, because the operating system scheduler (which we will meet in Chapter 7) can forcibly evict platform threads from CPU cores. This can lead to another thread starting to execute and accessing an object before the original thread had finished processing it, and potentially seeing the object in a prior or even invalid state.

The only defense the core of Java provides against this potential object damage during concurrent code execution is the mutual exclusion lock, and this can be very complex to use in real applications. Chapter 13 contains a detailed look at how the JMM works, and the practicalities of working with threads and locks.

Lifecycle of a traditional Java application

Earlier in the chapter we introduced Java program execution via classloading

and bytecode interpretation—but let's dive a little deeper into what actually happens when you type: `java HelloWorld`.

At a low level, standard Unix-like process execution occurs in order to set up the JVM process. The shell locates the JVM binary (e.g. possibly in `$JAVA_HOME/bin/java`) and starts a process corresponding to that binary, passing the arguments (including the entrypoint class name).

The newly started process analyzes the command line flags and prepares for VM initialization, which will be customized via the flags (for heap size, GC, etc). At this time the process probes the machine it is running on, and examines various system parameters, such as how many CPU cores the machine has; how much memory; what precise set of CPU instructions are available.

This very detailed information is used to customize and optimize how the JVM configures itself. For example, the JVM will use the number of cores to determine how many threads to use when garbage collection runs, and to size the *common pool* of threads.

One key early step is to reserve an area of userspace memory (from the C heap) equal to `Xmx` (or the default value) for the Java heap. Another vital step is to initialize a repository to store Java classes and associated metadata in (known as *Metaspace* in HotSpot).

Then the VM itself is created, usually via the function `JNI_CreateJavaVM`, on a new thread for HotSpot. The VM's own threads —such as the GC threads and the threads that perform JIT compilation—also need to be started up.

As discussed earlier, the bootstrapping classes are prepared and then initialized. The first bytecodes are run and first objects are created as soon as classes are loaded—e.g. in the class initializer (`static {}` blocks aka `clinit` methods) for the bootstrapping classes.

The significance of this is that the JVMs basic processes—such as JIT compilation and GC—are running from very early in the lifecycle of the application. As the VM starts up, there may be some GC and JIT activity

even before control reaches the entrypoint class. Once it does, then further classloading will happen as the application begins to execute and needs to run code from classes that are not present in the class metadata cache.

For most typical production applications, therefore, the startup phase is characterized by a spike in classloading, JIT and GC activity while the application reaches a steady state. Once this has occurred, the amount of JIT and classloading usually drops sharply because:

- The entire "world" of classes that the application needs has been loaded
- The set of methods that are called often have already been converted to machine code by the JIT compiler

However, it is important to recognize that "steady state" does not mean "zero change". It is perfectly normal for applications to experience further classloading and JIT activity—such as *deoptimization* and *reoptimization*. This can be caused when a rarely-executed code path is encountered and causes a new class to be loaded.

One other important special case of the startup-steady-state model is sometimes referred to as "2-phase classloading". This occurs in applications that use Spring, and other similar dependency injection techniques.

In this case, the core framework classes are loaded first. After that, the framework examines the main application code and config to determine a graph of objects that need to be instantiated to activate the application. This triggers a second phase of classloading where the application code and its other dependencies are loaded.

The case of GC behavior is a little bit different. In an application which is not suffering any particular performance problem, the pattern of GC is also likely to change when the steady state is reached—but GC events will still occur. This is because in any Java application, objects are created, live for some time and then are automatically collected—this is the entire point of automatic memory management. However, the pattern of steady state GC may well look very different to that of the startup phase.

The overall impression that you should be building up from this description is one of a highly dynamic runtime. Applications that are deployed on it display the runtime characteristics of a well-defined startup phase, followed by a steady state where minimal change occurs.

This is the standard mental model for the behavior of Java applications, and has been for as long as Java has had JIT compilation, etc. However, it does have certain drawbacks—the major one being that execution time can be slower while the application transitions into steady state (often called "JVM warmup").

This transition time can easily run into the 10s of seconds after application start. For long-running applications this is not usually a problem—a process that is running continuously for hours (or days or weeks) receives far more benefit from the JIT compiled code than the one-off effort expended to create it at startup.

In the cloud native world, however, processes may be much shorter-lived. This raises the question: whether the amortized cost of Java startup and JIT is actually worth it, and if not, what could be done to make Java applications start up faster?

In turn, this has fueled interest in new operational and deployment modes for Java—including AOT compilation (but not limited to it, as we will see). The community has adopted the term *dynamic VM mode* for the traditional lifecycle we have just discussed. We will have a good amount to say about the emerging alternatives to it throughout the rest of the book.

Monitoring and Tooling for the JVM

The JVM is a mature execution platform, and it provides a number of technology alternatives for instrumentation, monitoring, and Observability of running applications. The main technologies available for these types of tools for JVM applications are:

- Java Management Extensions (JMX)

- Java agents
- The JVM Tool Interface (JVMTI)
- The Serviceability Agent (SA)

JMX is a general-purpose technology for controlling and monitoring JVMs and the applications running on them. It provides the ability to change parameters and call methods in a general way from a client application. A full treatment of how this is implemented is, unfortunately, outside the scope of this book. However, JMX (and its associated network transport, *remote method invocation* or RMI) is a fundamental aspect of the management capabilities of the JVM.

A Java agent is a tooling component, written in Java (hence the name), that makes use of the interfaces in `java.lang.instrument` to modify the bytecode of methods as classes are loaded. The modification of bytecode allows instrumentation logic, such as method timing or distributed tracing (see Chapter 10 for more details), to be added to any application, even one that has not been written with any support for those concerns.

This is an extremely powerful technique, and installing an agent changes the standard application lifecycle that we met in the last section. To install an agent, it must be packaged as a JAR and provided via a startup flag to the JVM:

```
-javaagent:<path-to-agent-jar>=<options>
```

The agent JAR must contain a manifest file, `META-INF/MANIFEST.MF`, and it must include the attribute `Premain-Class`.

This attribute contains the name of the agent class, which must implement a public static `premain()` method that acts as the registration hook for the Java agent. This method will run on the main application thread *before* the `main()` method of the application (hence the name). Note that the premain method must exit, or the main application will not start.

Bytecode transformation is the usual intent of an agent, and this is done by

creating and registering bytecode transformers—objects that implement the `ClassFileTransformer` interface. However, a Java agent is just Java code, and so it can do anything that any other Java program can, i.e. it can contain arbitrary code to execute. This flexibility means that, for example, an agent can start additional threads that can persist for the entire life of the application, and collect data for sending out of the application and into an external monitoring system.

> **NOTE**
>
> We will have a little more to say about JMX and agents in Chapter 11 where we discuss their use in cloud Observability tools.

If the Java instrumentation API is not sufficient, then the JVMTI may be used instead. This is a native interface of the JVM, so agents that make use of it must be written in a native compiled language—essentially, C or C++. It can be thought of as a communication interface that allows a native agent to monitor and be informed of events by the JVM. To install a native agent, provide a slightly different flag:

```
-agentlib:<agent-lib-name>=<options>
```

or:

```
-agentpath:<path-to-agent>=<options>
```

The requirement that JVMTI agents be written in native code means that these agents can be more difficult to write and debug. Programming errors in JVMTI agents can damage running applications and even crash the JVM.

Therefore, where possible, it is usually preferable to write a Java agent over JVMTI code. Agents are much easier to write, but some information is not available through the Java API, and to access that data JVMTI may be the only possibility available.

The final approach is the Serviceability Agent. This is a set of APIs and tools that can expose both Java objects and HotSpot data structures.

The SA does not require any code to be run in the target VM. Instead, the HotSpot SA uses primitives like symbol lookup and reading of process memory to implement debugging capability. The SA has the ability to debug live Java processes as well as core files (also called *crash dump files*).

VisualVM

The JDK ships with a number of useful additional tools along with the well-known binaries such as `javac` and `java`.

One tool that is often overlooked is VisualVM, which is a graphical tool based on the NetBeans platform. VisualVM used to ship as part of the JDK but has been moved out of the main distribution, so developers will have to download the binary separately from the VisualVM website. After downloading, you will have to ensure that the `visualvm` binary is added to your path or you may get an obselete version from an old Java version.

> **TIP**
>
> `jvisualvm` is a replacement for the now obsolete `jconsole` tool from earlier Java versions. If you are still using `jconsole`, you should move to VisualVM (there is a compatibility plug-in to allow `jconsole` plug-ins to run inside VisualVM).

When VisualVM is started for the first time it will calibrate the machine it is running on, so there should be no other applications running that might affect the performance calibration. After calibration, VisualVM will finish starting up and show a splash screen. The most familiar view of VisualVM is the Monitor screen, which is similar to that shown in Figure 3-4.

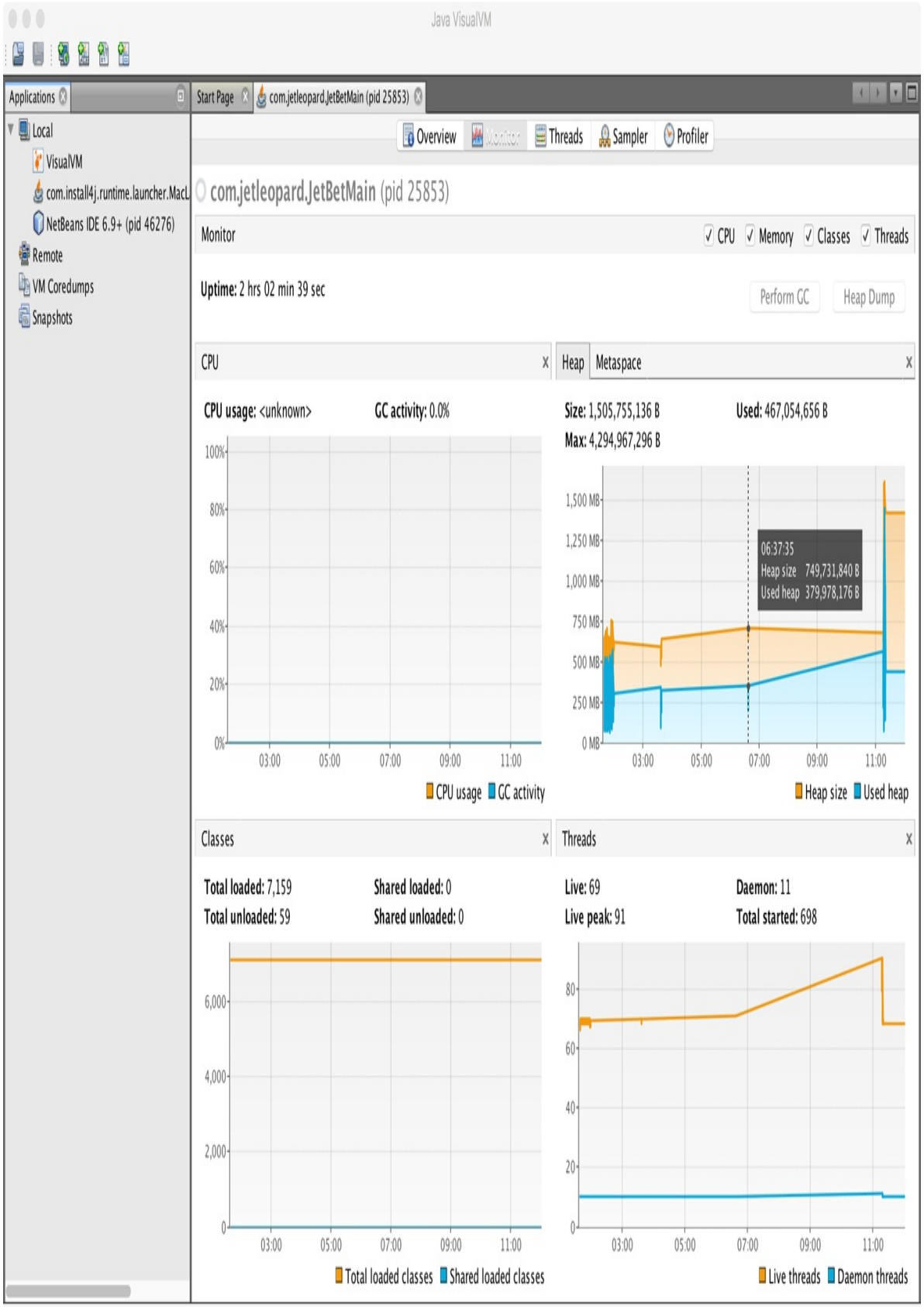

Java VisualVM
Applications
Start Page
com.jetleopard.JetBetMain (pid 25853)
Local
VisualVM
com.install4j.runtime.launcher.MacL
NetBeans IDE 6.9+ (pid 46276)
Remote
VM Coredumps
Snapshots
Overview
Monitor
Threads
Sampler
Profiler
com.jetleopard.JetBetMain (pid 25853)
Monitor
CPU
Memory
Classes
Threads
Uptime: 2 hrs 02 min 39 sec
Perform GC
Heap Dump
CPU
CPU usage: <unknown>
GC activity: 0.0%
100%
80%
60%
40%
20%
0%
03:00
05:00
07:00
09:00
11:00
CPU usage
GC activity
Heap
Metaspace
Size: 1,505,755,136 B
Used: 467,054,656 B
Max: 4,294,967,296 B
1,500 MB
1,250 MB
1,000 MB
750 MB
500 MB
250 MB
0 MB
06:37:35
Heap size 749,731,840 B
Used heap 379,978,176 B
Heap size
Used heap
Classes
Total loaded: 7,159
Shared loaded: 0
Total unloaded: 59
Shared unloaded: 0
6,000
4,000
2,000
0
Total loaded classes
Shared loaded classes
Threads
Live: 69
Daemon: 11
Live peak: 91
Total started: 698
80
60
40
20
0
Live threads
Daemon threads

Figure 3-4. VisualVM Monitor screen

VisualVM is used for live monitoring of a running process, and it uses the JVM's *attach mechanism*. This works slightly differently depending on whether the process is local or remote.

Local processes are fairly straightforward. VisualVM lists them down the lefthand side of the screen. Double-clicking on one of them causes it to appear as a new tab in the righthand pane.

To connect to a remote process, the remote side must accept inbound connections (over JMX). For standard Java processes, this means `jstatd` must be running on the remote host (see the manual page for `jstatd` for more details).

> **NOTE**
>
> Many application servers and execution containers provide an equivalent capability to `jstatd` directly in the server. Such processes do not need a separate `jstatd` process so long as they are capable of port-forwarding JMX and RMI traffic.

To connect to a remote process, enter the hostname and a display name that will be used on the tab. The default port to connect to is 1099, but this can be changed easily.

Out of the box, VisualVM presents the user with five tabs:

Overview

Provides a summary of information about your Java process. This includes the full flags that were passed in and all system properties. It also displays the exact Java version executing.

Monitor

This is the tab that is the most similar to the legacy `jconsole` view. It shows high-level telemetry for the JVM, including CPU and heap usage. It also shows the number of classes loaded and unloaded, and an

overview of the numbers of threads running.

Threads

Each thread in the running application is displayed with a timeline. This includes both application threads and VM threads. The state of each thread can be seen, with a small amount of history. Thread dumps can also be generated if needed.

Sampler and Profiler

In these views, simplified sampling of CPU and memory utilization can be accessed. This will be discussed more fully in Chapter 11.

The plug-in architecture of VisualVM allows additional tools to be easily added to the core platform to augment the core functionality. These include plug-ins that allow interaction with JMX consoles and bridging to legacy JConsole, and a very useful garbage collection plug-in, VisualGC.

Java implementations, distributions and releases

In this section we will briefly discuss the landscape of Java implementations and distributions, as well as the Java release cycle.

This is an area that changes a lot over time—so this description is correct at time of writing only. Since then, for example, vendors may have entered (or exited) the business of making a Java distribution or the release cycle may have changed. Caveat lector!

Many developers may only be familiar with the Java binaries produced by Oracle (Oracle JDK). However, as of 2023, we have quite a complex landscape, and it's important to understand the basic components of what makes up "Java".

First, there's the source code that will be built into a binary. The source code required to build a Java implementation comes in two parts:

- Virtual machine source code
- Class library source code

The OpenJDK project, which can be found at the OpenJDK website is the project to develop the open source reference implementation of Java—which is licensed under the GNU Public License version 2, with Classpath Exemption (GPLv2+CE).[2] The project is led and supported by Oracle—who provide a majority of the engineers who work on the OpenJDK codebase.

The critical point to understand about OpenJDK is that it provides *source code only*. This is true both for the VM (HotSpot) and for the class libraries.

The combination of HotSpot and the OpenJDK class libraries forms the basis of the vast majority of Java distributions used in today's production environments (including Oracle's). However, there are several other Java VMs that we will meet—and discuss briefly in this book—including Eclipse OpenJ9 and GraalVM. These VMs can also be combined with the OpenJDK class libraries to produce a complete Java implementation.

However, source code, by itself, is not all that useful to developers—it needs to be built into a binary distribution, tested and optionally certified.

This is somewhat similar to the situation with Linux—the source code exists and is freely available, but in practice virtually no-one except those folks developing the next version work directly with source. Instead, developers consume a binary Linux distribution.

In the Java world there are a number of vendors who make distributions available, just as there are for Linux. Let's meet the vendors and take a quick look at their various offerings.

Choosing a distribution

Developers and architects should consider carefully their choice of JVM vendor. Some large organizations—notably Twitter (as of 2022) and Alibaba—even choose to maintain their own private (or semi-public) builds of OpenJDK, although the engineering effort required for this is beyond the

reach of many companies.

With this in mind, the main factors that organizations typically care about are:

1. Do I have to pay money to use this in production?
2. How can I get any bugs I discover fixed?
3. How do I get security patches?

To take these in turn:

A binary that has been built from OpenJDK source (which is GPLv2+CE-licensed) is free to use in production. This includes all binaries from Eclipse Adoptium, Red Hat, Amazon, and Microsoft; as well as binaries from lesser-known suppliers such as BellSoft. Some, but not all, of Oracle's binaries also fall into this category.

Next up, to get a bug fixed in OpenJDK the discoverer may do one of two things: either buy a support contract and get the vendor to fix it; or ask an OpenJDK author to file a bug against the OpenJDK repo and then hope that (or ask nicely) someone fixes it for you. Or there's always the inevitable third option that all open-source software provides—fix it yourself and then submit a patch.

The final point—about security updates—is slightly more subtle. First off, note that almost all changes to Java start off as commits to a public OpenJDK repository on GitHub. The exception to this is certain security fixes that have not yet been publicly disclosed.

When a fix is released and made public, there is a process by which the patch flows back into the various OpenJDK repos. The vendors will then be able to take that source code fix and build and release a binary which contains it. However, there are some subtleties to this process, which is one reason why most Java shops prefer to remain on a long-term support (or LTS) version—we will have more to say about this in the section about Java versions.

Now that we've discussed the main criteria for choosing a distribution, let's

meet some of the main offerings that are available:

Oracle

Oracle's Java (Oracle JDK) is perhaps the most widely known implementation. It is essentially the OpenJDK codebase, relicensed under Oracle's proprietary licenses with a few extremely minor differences (such as the inclusion of some additional components that are not available under an open-source license). Oracle achieves this by having all contributors to OpenJDK sign a license agreement that permits dual licensing of their contribution to both the GPLv2+CE of OpenJDK and Oracle's proprietary license.[3]

Eclipse Adoptium

This community-led project started life as AdoptOpenJDK, changing name when it transitioned into the Eclipse Foundation. The Members of the Adoptium project (from companies such as Red Hat, Google, Microsoft, and Azul) consist mostly of build and test engineers, rather than development engineers (who implement new features and fix bugs). This is by design—many of Adoptium's member companies also make major contributions to upstream OpenJDK development, but do so under their own company names, rather than Adoptium. The Adoptium project takes the OpenJDK source and builds fully-tested binaries on multiple platforms. As a community project, Adoptium does not offer paid support, although member companies may choose to do so—for example Red Hat does for some operating systems.

Red Hat

Red Hat is the longest-standing non-Oracle producer of Java binaries—as well as the second-largest contributor to OpenJDK (behind Oracle). They produce builds and provide support for their operating systems—RHEL and Fedora—and Windows (for historical reasons). Red Hat also releases freely-available container images based on their Universal Base Image (UBI) Linux system.

Amazon Corretto

Corretto is Amazon's distribution of OpenJDK, and it is intended to run primarily on AWS cloud infrastructure. Amazon also provides builds for Mac, Windows and Linux in order to provide a consistent developer experience, and to encourage developers to use their builds across all environments.

Microsoft OpenJDK

Microsoft has been producing binaries since May 2021 (OpenJDK 11.0.11) for Mac, Windows and Linux. Just as for AWS, Microsoft's distribution is largely intended to provide an easy on-ramp for developers who will be deploying on their Azure cloud infrastructure.

Azul Systems

Zulu is a free OpenJDK implementation provided by Azul Systems—who also offer paid support for their OpenJDK binaries. Azul also offer a high-performance proprietary JVM called "Azul Platform Prime" (previously known as Zing). Prime is not an OpenJDK distribution.

GraalVM

GraalVM is a relatively new addition to this list. Originally a research project at Oracle Labs, it has graduated to a fully productionized Java implementation (and much more besides). GraalVM can operate in dynamic VM mode and includes an OpenJDK-based runtime—augmented with a JIT compiler that is written in Java. However, GraalVM is also capable of *native compilation* of Java—essentially AOT compilation. We will have more to say on this subject later in the book.

OpenJ9

OpenJ9 started life as IBM's proprietary JVM (when it was just called J9) but was open-sourced in 2017 partway through its life (just like HotSpot). It is now built on top of an Eclipse open runtime project (OMR). It is

fully compliant with Java certification. IBM Semeru Runtimes are zero-cost runtimes built with the OpenJDK class libraries and the Eclipse OpenJ9 JVM (which is Eclipse-licensed).

Android

Google's Android project is sometimes thought of as being "based on Java." However, the picture is actually a little more complicated. Android uses a cross compiler to convert class files to a different (*.dex*) file format. These `.dex` files are then executed by the Android Runtime (ART), which is not a JVM. In fact, Google now recommends the Kotlin language over Java for developing Android apps. As this technology stack is so far from the other examples, we won't consider Android any further in this book.

Note that this list is not intended to be comprehensive—there are other distributions available as well.

The vast majority of the rest of this book focuses on the technology found in HotSpot. This means the material applies equally to Oracle's Java and the distributions provided by Adoptium, Red Hat, Amazon, Microsoft, Azul Zulu, and all other OpenJDK-derived JVMs.

We also include some material related to Eclipse Open J9 . This is intended to provide an awareness of alternatives rather than a definitive guide. Some readers may wish to explore these technologies more deeply, and they are encouraged to proceed by setting performance goals, and then measuring and comparing, in the usual manner.

Finally, before we discuss the Java release cycle, a word about the performance characteristics of the various OpenJDK distributions.

Teams occasionally ask questions about performance—sometimes because they mistakenly believe that certain distributions include different JIT or GC components that are not available in other OpenJDK-based distributions.

So let's clear that up right now: All the OpenJDK distributions build from the same source, and there should be *no* systematic performance-related

differences between the various HotSpot-based implementations, when comparing like-for-like versions and build flag configurations.

> **NOTE**
>
> Some vendors choose very specific build flag combinations that are highly specific to their cloud environments, and some research indicates that these combinations *may* help for some subset of workloads, but this is far from clear-cut.

Once in a while, social media excitedly reports that significant performances differences have been found between some of the distributions. However, carrying out such tests in a sufficiently controlled environment is notoriously difficult—so any results should be treated with healthy skepticism unless they can be independently verified as statistically rigorous.

The Java release cycle

We can now complete the picture by briefly discussing the Java release cycle.

New feature development happens in the open—at a collection of GitHub repositories. Small to medium features and bug fixes are accepted as pull requests directly against the main branch in the main OpenJDK repository.[4] Larger features and major projects are frequently developed in forked repos and then migrated into mainline when ready.

Every 6 months, a new release of Java is cut from whatever is in main. Features that "miss the train" must wait for the next release—the 6-month cadence and strict timescale has been maintained since September 2017. These releases are known as "feature releases", and they are run by Oracle, in their role as stewards of Java.

Oracle ceases to work on any given feature release as soon as the next feature release appears. However, an OpenJDK member of suitable standing and capability can offer to continue running the release after Oracle steps down. To date, this has only happened for certain releases—in practice Java 8, 11, 17 and 21, which are known as *update releases*.

The significance of these releases is that they match Oracle's Long-Term Support release concept. Technically, this is purely a construct of Oracle's sales process—whereby Oracle customers who do not want to upgrade Java every 6 months have certain stable versions that Oracle will support them on.

In practice, the Java ecosystem has overwhelmingly rejected the official Oracle dogma of "upgrade your JDK every 6 months"--project teams and engineering managers simply have no appetite for it. Instead, teams upgrade from one LTS version to the next, and the update release projects (8u, 11u, 17u and 21u) remain active, delivering security patches and a small number of bug fixes and backports. Oracle and the community work together to keep all these maintained code streams secure.

This is the final piece we need to answer the question of how to pick a Java distribution. If you want a zero-cost Java distribution that receives security patches and has a non-zero chance of security (and possibly bug) fixes, select your choice of OpenJDK vendor and stick to the LTS versions. Any of: Adoptium, Red Hat, Amazon, Microsoft and Azul is a fine choice—and so are some of the others. Depending on how and where you're deploying your software (e.g. applications deployin in AWS may prefer Amazon's Corretto distribution) you may have a reason to pick one of those over the others.

For a more in-depth guide to the various options and some of the licensing complexities, you can consult Java Is Still Free This document was written by the Java Champions, an independent body of Java experts and leaders.

Summary

In this chapter we have taken a quick tour through the overall anatomy of the JVM, including: compilation of byte code, interpretation, JIT compilation to native code, memory management, threading, the lifecycle of a Java process monitoring, and finally, how Java is built and distributed.

It has only been possible to touch on some of the most important subjects, and virtually every topic mentioned here has a rich, full story behind it that will reward further investigation.

In Chapter 4 we will begin our journey into garbage collection, starting with the basic concepts of mark-and-sweep and diving into the specifics, including some of the internal details of how HotSpot implements GC.

---

1 B. Stroustrup, “Abstraction and the C++ Machine Model,” *Lecture Notes in Computer Science*, vol. 3605 (Springer 2005)

2 *https://openjdk.org/legal/gplv2+ce.xhtml*

3 The latter has changed multiple times, so linking to the currently latest version might not be helpful—it could be out-of-date by the time you read this.

4 *https://github.com/openjdk/jdk*