PLATFORM9

Ultimate guide to

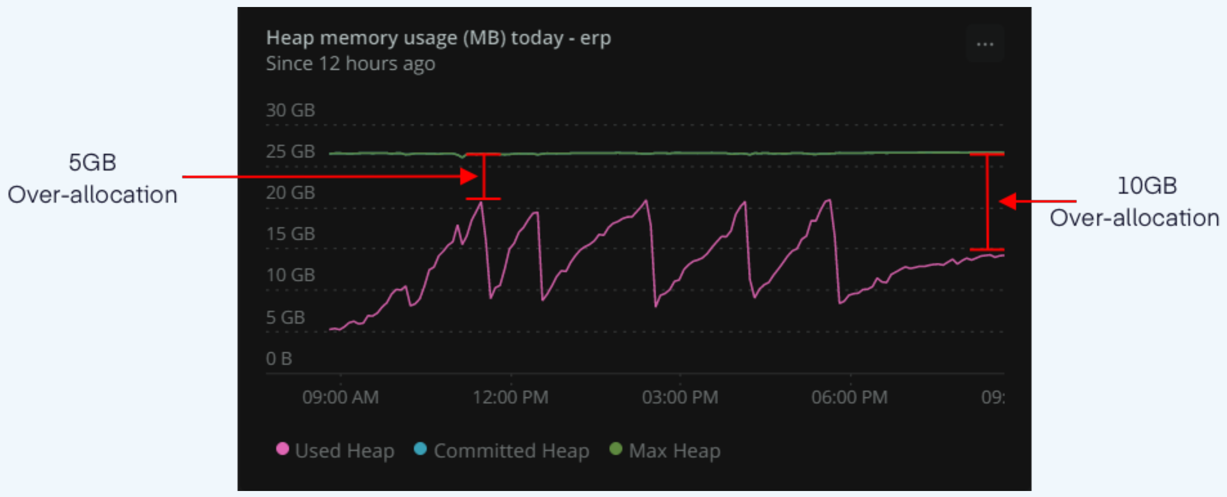
Tuning Java on Kubernetes

Cost Management Edition

Table of Contents

Introduction

Let’s get right to it. Java applications have always been deployed to production with a little extra: extra RAM, extra CPU—let’s call it overallocation of resources. Why does this happen so universally? At some point during the application‘s life cycle, including after the app is deployed in production, additional CPU and RAM were added to isolate or resolve an issue. Unfortunately, no one likely recalls why or even how much, but the end result is the same: Java applications end up with more computing resources than they require.



Java heap usage overtime showing 12hr period with 10GB change in heap over-allocation

What does this look like in practice? Java memory usage naturally fluctuates over time. In the real application example visualized below over a 12-hour period, actual usage stays at least 5GB less than what was allocated. In light of this example, what should the maximum memory allocation be? How much overallocation should exist? 5GB, 10GB? Why not 0GB?

Resource Overcommitment

Refers to the practice of allocating more computing resources than are physically available in the hope that not all resources will be needed simultaneously. This is a widely adopted technique by virtual machine hypervisors who time multiplex resources by dynamically reallocating them between resource consumers. To accomplish this, the hypervisor needs to be able to reclaim unused resources from consumers. Server consolidation, an industry wide trend over the last 20 years, is largely possible due to the hypervisor's resource overcommitment techniques.

Resource Overallocation

Occurs when more resources are allocated to a JVM than what it actually needs in order to ensure performance and availability. In the case of memory allocation, it is important to strike a balance between providing enough memory for optimal performance and not wasting resources by overallocating. Some overallocation is needed to absorb spikes in utilization. For example, zero GB memory overallocation would not be advisable as it could lead to performance issues and potential outages if a temporary spike in the system leads to response time spikes and reduced availability.

The Saga of Java, Infrastructure Trends and Overallocation

1995 – 2005: The Internet Era of Server Sprawl

Overallocation during the 2000s decade resulted in unused capacity on physical servers, causing additional power use, wasted data center space, and excess cooling costs. By the early 2000s, Java was starting to be run within on-premises virtual machines (VMs), and architects were discovering that they could use a popular feature in hypervisors like VMware, Openstack, and Hyper-V called resource overcommitment to reduce the cost impact of overallocation was reduced to near zero.

2006 – 2010: The P2V Era

By the 2010s, Java was primarily run within on-premises virtual machines (VMs), and the impact of overallocation was reduced to zero using hypervisors resource overcommitment. Java architects frequently performed P2V (physical to virtual) migrations so that another VM could use the „unused“ RAM. In the physical world, Java performance could be diagnosed in isolation as everything around it was not shared, sans the database and network. But diagnosis became much harder with server consolidation.

2010 – 2019: The VM Era

As the enterprise Java footprint grew, so did the demand for a deep understanding of Java performance in real-time. One key element of this period was 'dynamic bytecode instrumentation'. The -javaagent flag is a gold standard that still reigns king today.

Jump forward to 2019—most businesses were firmly in the cloud, and overallocation once again became financial waste. Not that many organizations were paying attention to this in 2019, but Java's overallocation problem in 2019 was the surprise budget miss that the CIO could never explain.

The advent of DevOps and the ability to investigate Java application performance in real-time shifted the focus from RAM allocation to the application's code. If you were savvy, you also considered how the Java application interacted with the underlying databases.

Little credence was given to those of us who delved deep into the full stack, looking into VM provisioning practices, contention, and storage performance. While critical and often the root cause of issues, this approach was not as exciting as diving into API latency in real-time.

2015 – 2024: The Cloud Instance Era

V2V2C (I added a ‚V‘ as the VM technically was re-virtualized in the process), the loved and dreaded virtual to cloud migration period didn't change much. Many companies felt that outsourcing entire datacenters and moving to the cloud would save money and improve performance due to the elastic access to resources.

Whereas the move to VMs had enabled overcommitment, those savings were lost in the move to the public cloud, which didn't offer control over the virtualization layer. Instead, public cloud offered elastic horizontal scale-up and scale-down, which made other deployment challenges simpler, thus leading to a public cloud industry boom. Thus, each new cloud instance brought the same amount of waste, as the JVM could still not be dynamically resized.

With infrastructure performance solved in the cloud, all focus is now centered inside the JVM. Byte-code instrumentation if you could afford it; if not, long live the log file that Splunk had made cool again. The drastic increase in waste that the JVM on the cloud experienced went unnoticed or ignored, possibly as a result of low interest rates and widespread economic growth.

Continuing to 2021, Java was making its next generational shift to containers—overallocation now reappeared as a problem in light of Kubernetes Pod requests and limits. Waste that was once mitigated using VMware was now back, and in a big way.

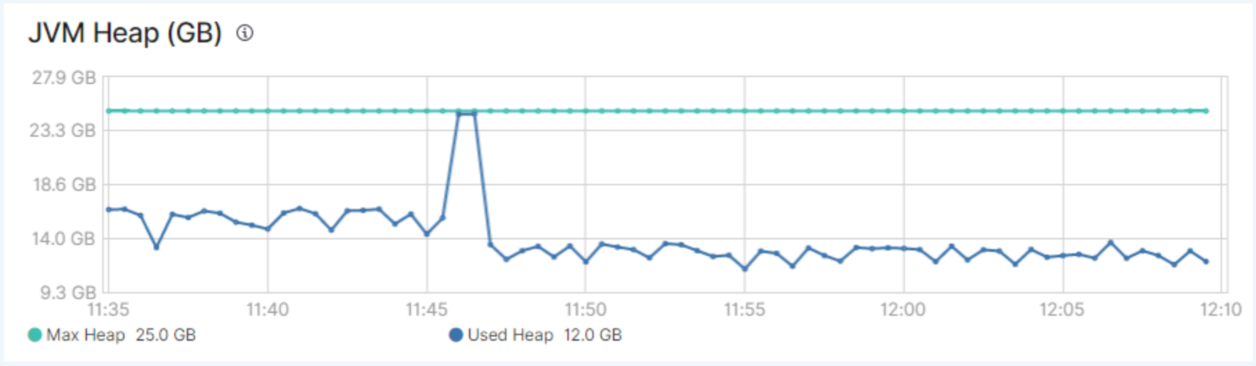
Java Microservice Era

Separate from the above JVM infrastructure hosting trends, another parallel trend has been splitting monoliths into microservices. Unfortunately, that makes the cost equations for Java even worse than ever before. Today, for each application, we have now split it into multiple components, each requiring a separate JVM deployed. This means that there are more JVMs, each of which is overallocated, thus multiplying the resource waste compared to the monolith strategy. Primarily, microservice architectures are deployed in containers in a Kubernetes environment.

Overall, Java resource overallocation has not really changed, but for a brief period in time, overallocation issues were hidden. In 2024, there are three ways a JVM might find itself running on Kubernetes:

1. **In the cloud, inside a container that runs on a virtual machine instance where you cannot control overcommitment and the waste manifests itself as unused Pod resources.**
2. **On-prem, inside a container that runs on a virtual machine where you do control overcommitment and the waste is reduced to being negligible using overcommitment.**
3. **On-prem, inside a container on bare metal where overcommitment is not possible and the waste manifests itself as unused overallocated physical resources and low utilization.**

The progression to microservices, containers, and Kubernetes has increased over-allocation problems multiplicatively.



Java heap usage for 55 minutes showing 16GB change in memory usage

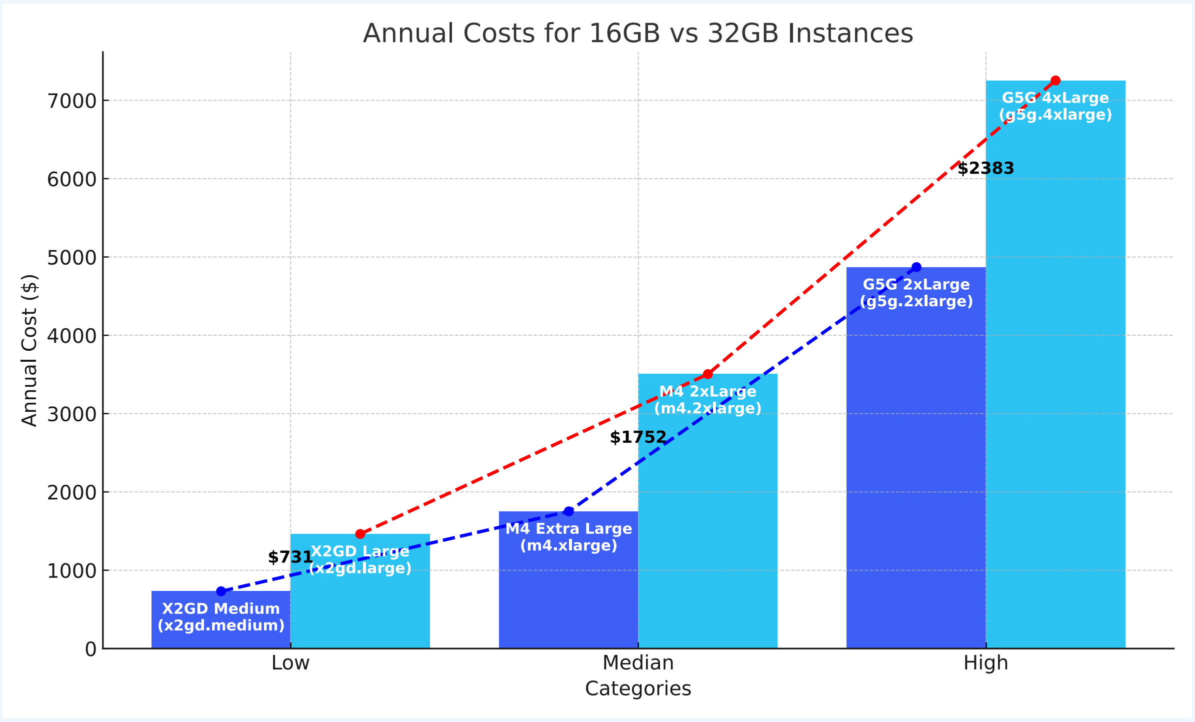
Java: The #1 Cause of Waste in Kubernetes

Empirical Analysis of High Cloud Costs of JVM Overallocation

Today, in 2024, overallocation results in large amounts of CPU and RAM that sit unused, idle, and wasted. Since each microservice becomes a separate JVM instance needing overallocation, the impact continues to multiply, often resulting in large budget waste. Ultimately, this is money spent that provides your organization zero ROI. Consuming budgets that could have been spent on revenue generating activities instead of padding the pockets of AWS, Google, and Azure.

A quick analysis of EC2 costs for US East 1 shows that running an instance with 32GB of RAM may cost anywhere from $1,463 to $7,253 per year. Compared to an EC2 VM with 16GB which ranges from $731 to $4,871 per year. It's easy to visualize that a Java application running with 32GB of allocated memory but only needing 16GB could easily be wasting $1,752 in the median case or $2,383 in the worst case every year.

Let's now scale up this example, and imagine that a single instance of this microservice can support 1,000 users. Since you have 40,000 users globally, you require 40 of these JVMs. Your worst case total waste now comes up to $2,383 x 40, or more than $95,000. But we also need to adjust for instances across development, testing, staging, and production, so we should add a minimum 2-3x factor on top. Let's be conservative and choose a 2.5x factor. Your waste increases to more than $225,000 per year. And this is just for a single microservice!



Understanding Why Java Causes Waste

Simply put, what used to work for Java on hypervisors does not work in containers or Kubernetes. This is caused by the design of Java whereby the status of “free” unused memory is opaque to everything outside of the JVM, operating system (OS) included. Once the JVM starts, the memory is considered used even if the Java application isn’t using it. This means no other application running in the Kubernetes cluster can use that memory.

Further complicating operations is the fact that tools and techniques for optimizing memory allocation of applications running on Kubernetes—from basics like the Vertical Pod Autoscaler to advanced features like in-place pod resizing—either don’t work at all or don’t work well with Java.

How memory allocation usually works

In many typical non-JVM applications on physical servers, VMs, or Kubernetes, memory is allocated and deallocated in real-time based on their needs. The operating system kernel is in charge of managing memory that an application has either not yet allocated or released. Unused memory is either assigned to other applications or used as cache for system-level activities like filesystem I/O. This allows applications with dynamic memory requirements to coexist even when their maximum memory requirements exceed what the system supports—if they don't all need their maximum memory at once.

How memory allocation works in Java

Java applications add a layer of abstraction to the ordinary operating system memory handling described above. Typically, at runtime, the Java Virtual Machine will be pre-allocated a large percentage of memory from the operating system. This memory may be referred to as the JVM Heap. The JVM then proceeds to manage memory itself, allocating to or reclaiming from the actual Java code running inside the JVM. The JVM subsumes control of fine-grained memory changes, which become invisible to the operating system.

The messy details - Java memory management is different

Java memory falls into 2 broad categories:

Heap Memory

- **Purpose:** This is where Java stores objects and class instances.
- **Management:** The heap is divided into generations (Young, Old, and Permanent) and each are cleaned or optimized by the process of garbage collection.

Stack Memory

- **Purpose:** This is used for method execution and local variables.
- **Management:** Each thread has its own stack, which grows and shrinks as methods are called and return.

As we have touched on, Java automatically handles memory allocation and deallocation—this feature reduces the risk of memory leaks and other memory-related issues and is in part why so many organizations adopted Java. The automated memory allocation implements a process known as garbage collection that runs in the background.

Garbage collection works by identifying and disposing of objects in the heap that are no longer needed, helping to maintain optimal memory usage. However, if left untuned, frequent garbage collection cycles degrade user experience and in the case of memory-intensive applications, can lead to higher CPU usage that further increases costs and waste.

Garbage collection, and why Java workloads are difficult to memory-manage

The host OS sees the entire amount of memory the JVM manages as in-use and therefore cannot manage it in any form, including SWAP. However, inside the JVM, memory may be almost completely unused, this is waste, and if the allocated memory is inflated, you have waste plus over-allocation. Memory may be marked as “unused” because it is either yet to be allocated (the JVM just started) or is used by unreferenced objects that need to be cleaned up by garbage collection.

Complicating the automated memory management features of Java is the fact that the JVM does not know which parts of its memory are used or unused. To identify the status of an object Java must trace an object reference, a process known as reachability. This is part of the garbage collection process and itself consumes CPU and RAM. Once this process completes unreferenced objects are marked “Unreachable”. These objects are deemed unused because there are no references pointing to them from any part of the running program.

As each step in the automated memory management consumes resources it is not a process that should be run frequently. Constant garbage collection results in an unresponsive application, and conversely, garbage collection that runs too infrequently can cause the JVM to crash entirely.

In all scenarios, once the JVM is allocated memory, it is very likely never returning to the OS, and therefore can never be used by another application. This is why it is important to ensure every Java application is only allocated exactly what it requires.

What is FinOps?

FinOps, short for Financial Operations, is a cloud financial management discipline that integrates finance, engineering, and business teams to optimize cloud spending and maximize business value. It popped up because cloud computing got super complex and traditional finance methods just weren’t cutting it anymore.

Java and FinOps

To understand overallocation and Java in the context of FinOps, we must first understand why Java is overallocated so often. When the language was first released, engineers rejoiced with the knowledge that they would never again need to manually manage memory, like they were forced to in languages like C or C++. Instead, Java would do it for them using a cleaning process known as garbage collection. But there's a catch: the application is momentarily paused when garbage collection runs, impacting user experience. The 'easy' solution? Provide more RAM so garbage collection happens less often, but it wasn't that simple. More RAM can lead to longer pauses because more garbage has to be collected, leading to further degradation in user experience.

Even today, teams often overallocate in an ill-guided attempt to ensure performance. Add just enough to avoid frequent cleaning events (garbage collections) and simultaneously avoid the long pauses associated with massive allocations of RAM.

As a FinOps practitioner attempting to reduce overallocation, you need to be aware that the process to tune garbage collection and reduce overallocation hasn't changed much in 20 years. The solution has always been to set the required CPU and RAM, test, refine, observe, and repeat. A process outlined in 2007 with ITIL 'Continual Service Improvement'.

Complicating matters is that with each generational change of technology and the persistent belief that "infrastructure" is the solution, companies worldwide have deferred addressing overallocation. By ignoring the issue and focusing on advancing infrastructure, companies running Java in the cloud, both with and without Kubernetes, now face performance and cost challenges. These challenges are intrinsically linked and not easily solved without significant investment in processes, tooling, and transformation.

Java for the CFO—A Primer

If you find yourself in the position where you need to explain to your finance team why Java will always have some percentage of memory waste, try using the analogy below:

Imagine Java memory management as the financial operations of a company, with you as the CFO overseeing everything. Here's how it breaks down:

Memory Allocation: The Budget Plan

When a Java program runs, it needs memory to store data and perform tasks, similar to how a company needs funds to operate. To run an application Java allocates memory in two main areas:

- **Heap Memory:** Think of this as your long-term investments. It's used for storing objects that need to exist for a longer period and like financial forecasting to manage uncertainties risk is managed by allocation more funds to a budget. Java manages Heap Memory through Garbage Collection (more on this below), which is like periodically reviewing and selling off investments that are no longer needed.
- **Stack Memory:** This is your short-term cash flow, used for temporary data like method calls and local variables. Each time a method is called, a new block of memory is allocated on the stack, and it's freed up when the method finishes, similar to managing short-term expenses.

Garbage Collection: The Automated Audit

Garbage Collection is an automatic process that identifies and removes objects that are no longer in use, freeing up memory. Think of it as an automated audit that ensures no resources are being wasted. This process helps keep the system efficient and prevents memory from being clogged with unnecessary data. But like audits, if done too often, performance degrades. It's a tradeoff.

Memory Leaks: Financial Leaks

Just as a CFO needs to be vigilant about financial leaks, Java developers need to watch out for memory leaks. These occur when memory that is no longer needed is not released, leading to inefficient use of memory and potentially slowing down or crashing the application. It's like having funds tied up in unproductive investments. Often, if a leak is suspected the Java application is given more memory than required to help hold off Garbage Collection (the audit) or total application failure.

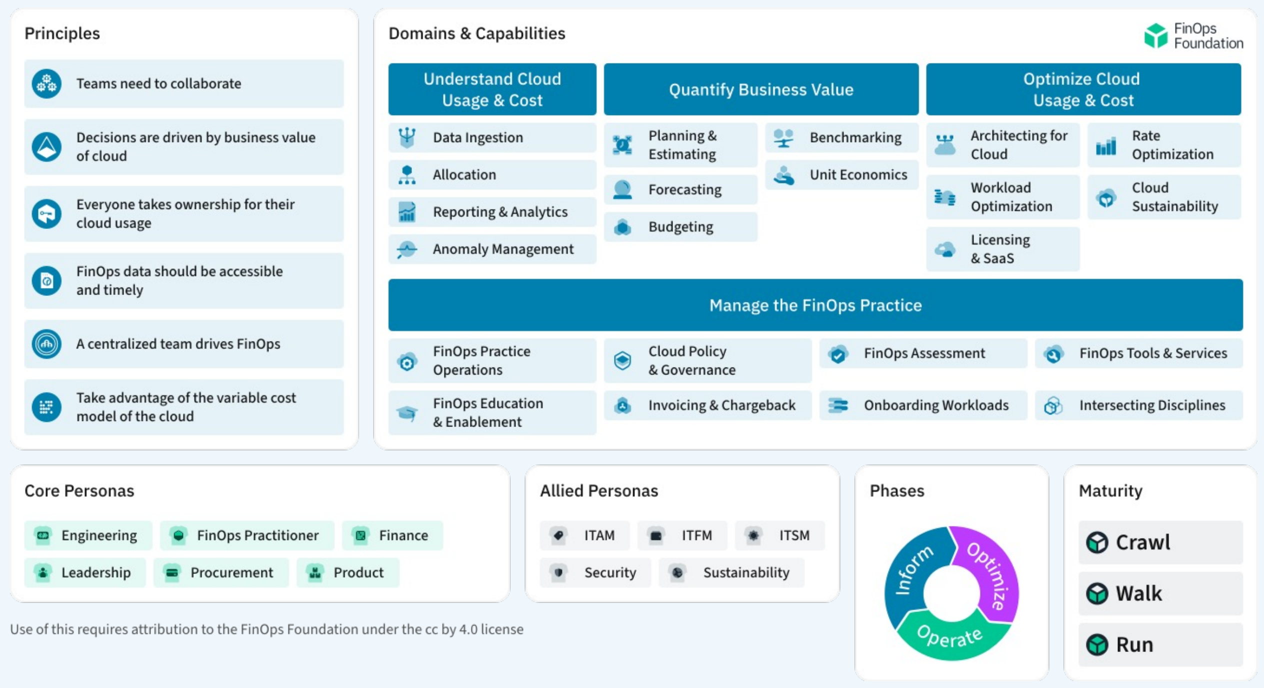
Efficient Memory Management: Financial Health

By managing memory efficiently, Java ensures that applications run smoothly, and resources are not wasted. This is akin to effective financial management ensuring a company's financial health and stability. If Java is allocated too much budget, like most of your business teams, it will take it, report back that it is being used, and only release it if forced and under much duress.

What to do as a FinOps Practitioner

Let’s imagine you’re responsible for reducing your EKS (Elastic Kubernetes Service) spend by 30%. You engage the Platform Operations or DevOps team and analyze the usage of CPU and RAM across all your clusters. Immediately you identify your financial reconciliation platform as a target; For every $1 spent on RAM, only 25% is used. You organize a meeting with Ops and Engineering to start exploring options to improve your unit economics.

Your focus should be within the Optimize Cloud Usage & Cost domain of the FinOps Framework.



FinOps Framework by the FinOps Foundation

There are three capabilities within FinOps Optimize Cloud Usage & Cost domain that will help:

Rate Optimization

With a rate optimization initiative, you could adopt one of the following:

- **Move the application onto Reserved Instances with low negotiated rates but a commitment period.**
- **Move the application onto Spot Instances and manage the constant churn of instances and the associated outages when the Spot Instance is removed.**
- **Migrate out of cloud to on-prem with lower costs.**

Architecting for Cloud

You can request the application architecture be reviewed and work through the process of identifying its current state, how it got their and if it conforms to practices such as the 12-factor application framework.

> Note, in some scenarios legacy java applications may be monolithic and require a multi-year project to refactor and rearchitect. Only after this investment will you see reductions in resource utilization.

Workload Optimization

Dive headfirst into workload optimization to uncover how you got to the place you are today, and what is used to determine resource allocations.

- **Try asking the question, 'why is so much of the compute we are paying for not being used?"**

A Workload Optimization initiative will likely be the most effective long term and permanent solution. It may contain elements of an architectural review, however, it should be more focused on how the Java applications RAM and CPU requirements were determined, and when they were last reviewed. The steps outlined in this guide provide a framework for permanently implementing workload optimization as an organizational practice.

Myths about Java overallocation

One of the first rebuttals often raised by engineering teams when they're asked to review Java application resource allocations is "performance and user experience will be degraded if we reduce the RAM allocation." This response may be true, but at the same time it cannot be blindly accepted.

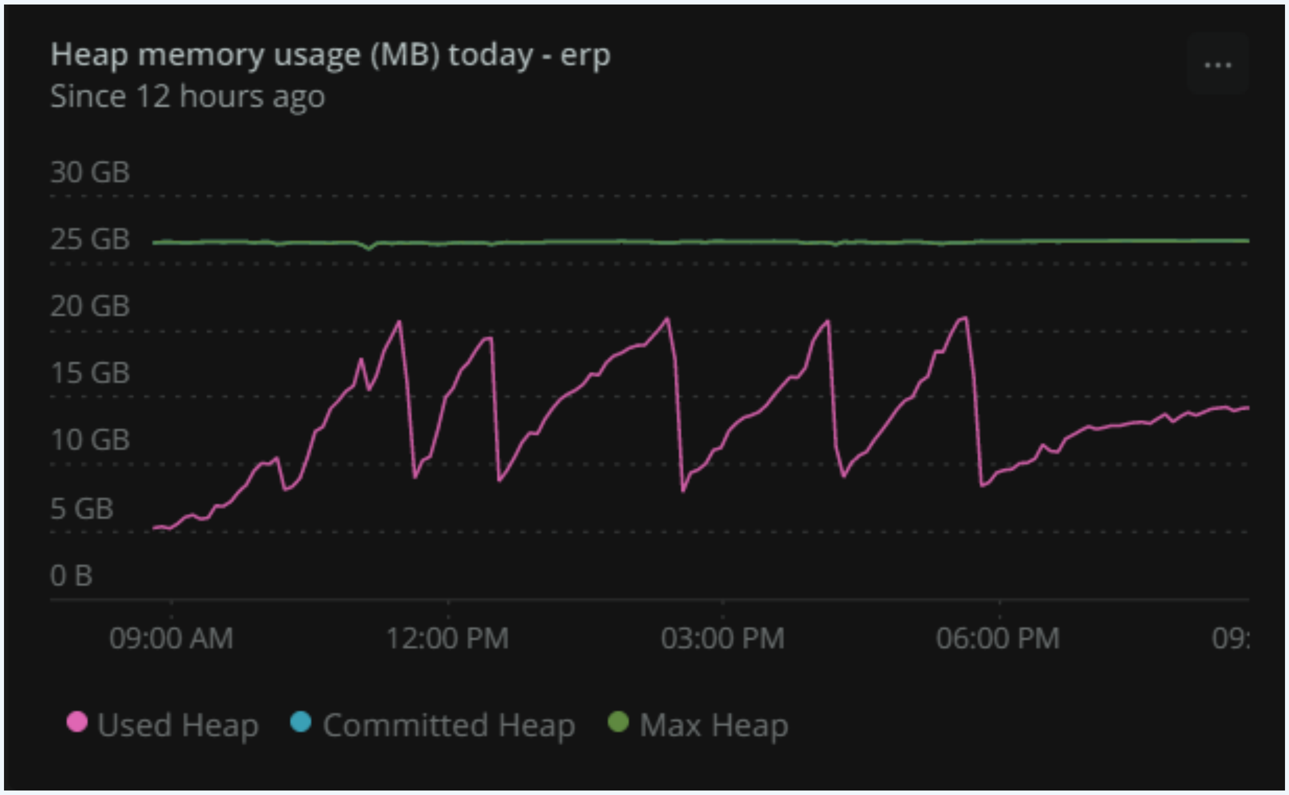
Myth: Over-allocation of memory = superior user experience

This is the biggest myth of them all. The truth is, allocating a JVM more memory does not guarantee better performance. Allocating more RAM to a JVM can, in fact, increase the time taken for garbage collection to run. This is because larger heap sizes can lead to longer garbage collection pause times, particularly during major garbage collection events. When this occurs, user experience is impacted significantly.

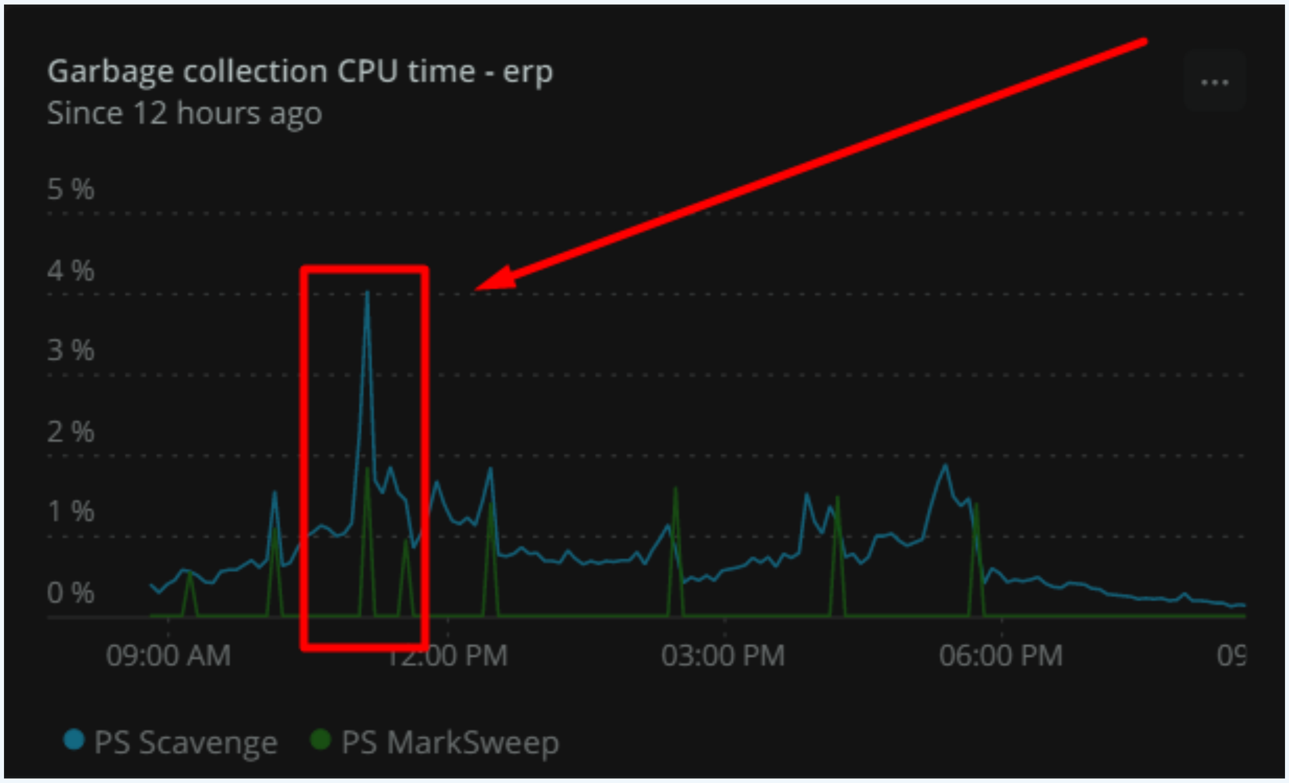
- **Increased Pause Times:** With larger heap sizes, the garbage collector has more memory to scan and manage, which can result in longer „stop-the-world" pauses where the application is halted to perform garbage collection. These pauses can affect application performance, especially if the application has latency requirements.
- **Garbage Collection Frequency:** While a larger heap can reduce the frequency of garbage collection events, when they do occur, they may take longer to complete due to the increased amount of memory that needs to be processed.
- **Tuning Considerations:** It is important to monitor and tune the garbage collector to optimize performance. The right balance between heap size and garbage collection performance depends on the specific needs and behavior of the application.

Overall, while more RAM can reduce the frequency of garbage collection, it can also increase the duration of each collection event, potentially impacting application performance.

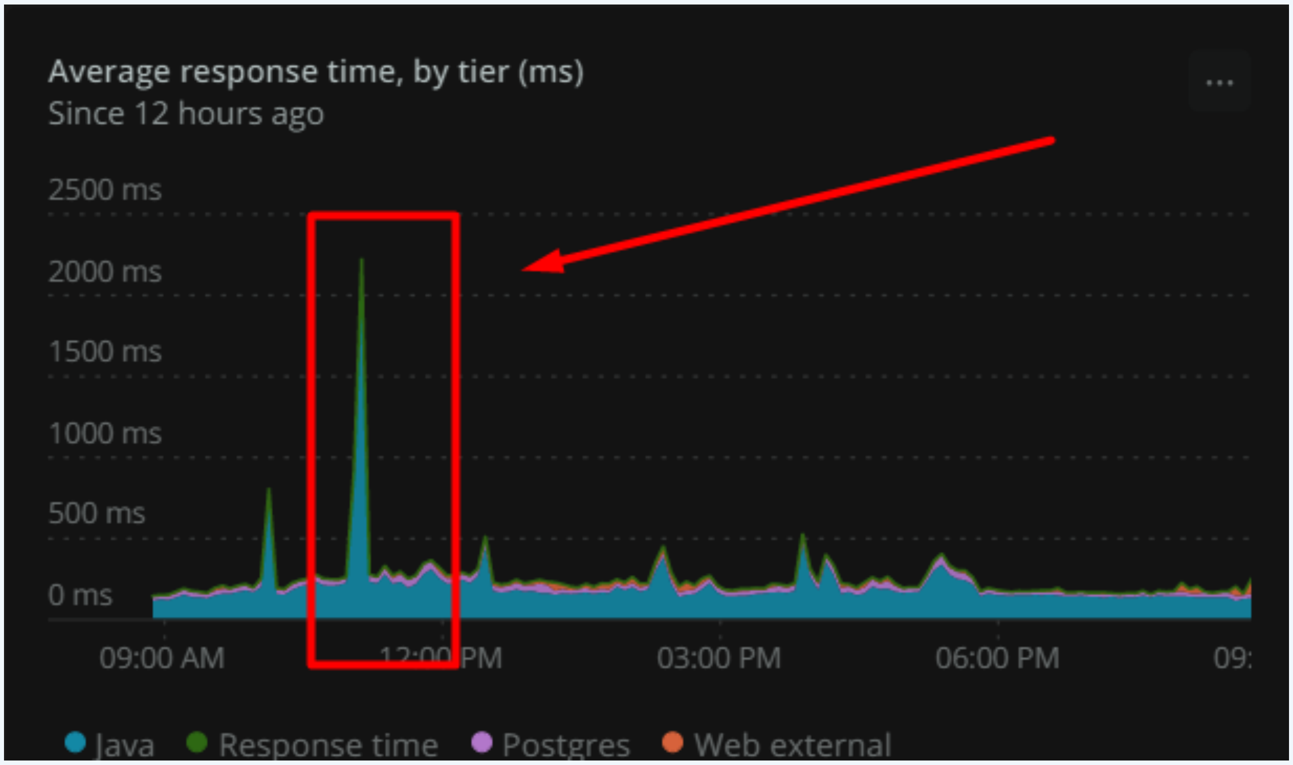
Take the example in the sequence of images below. The first image depicts typical management of heap. The second image highlights a massive spike in CPU utilization during a garbage collection event around 12pm. The third image captures an associated spike in response times, a massive 2 second delay in.



Java Heap Usage over a 12 Hour Period



CPU Time for Garbage Collection over a 12 hour Period



Application Response Times over a 12 hour Period

What to do to protect user experience

As stated below, the only way to protect user experience is to effectively monitor user experience. There are multiple tools to achieve this, however the most effective means are a combination of instrumentation, real user monitoring and a robust performance and load testing practice. Data from production and QA must be reviewed and monitored for deviations and closely reviewed in relation to any changes in a JVMs memory allocation and garbage collection configuration.

Infrastructure first approach to solving performance and cost problems

Infrastructure teams like to take an infrastructure first approach to solving performance and cost problems. As this relates to Kubernetes and any application running therein, it will generally include the following:

- **Horizontal Pod Autoscaler (HPA):** One approach is to attempt to apply HPA. This will scale application processes out (from 1 to n). To leverage HPA your application must be architected to support horizontal scaling (more on this below).

- **In-Place Update (still Alpha):** Still in early development is a feature to support changing Pod Resources (CPU and RAM) in real time. Today some users are leveraging this with Java applications to spike CPU up drastically on Pod deployment to ensure a quick startup and then reduce it down for general operations.

    - Kubernetes 1.27: In-place Resource Resize for Kubernetes Pods (alpha)
    - Enhancement Issue in Git

- **Vertical Pod Autoscaler & Resource Usage Feedback Loop:** With only basic monitoring of CPU and RAM it is possible to implement a feedback loop that constantly modifies Pod Requests and Limits based on the average CPU and RAM usage over time. At a minimum, this approach requires a mature GitOps practice and capable CI/CD tooling. To avoid outages and degrading response times, feedback loops need to support seasonality, leverage granular data (1 minute or lower) and include critical data around user experience. It is worth noting that the feedback loop approach can be implemented manually. With either approach, manual or automated, a Feedback Loop is typically combined with bin-packing. However, until in-place resizing matures, this approach carries a significant risk of application availability disruption which needs to be measured and evaluated.

    - **Bin Packing:** In regards to Kubernetes and particularly Amazon Elastic Kubernetes Service (EKS), Bin-Packing refers to the efficient allocation and placement of containers (pods) across nodes to maximize resource utilization and minimize costs. Bin-Packing is simply putting as many Pods on a Node as possible and ensure the node in use is the most cost efficient and runs with minimal unused resources (CPU and RAM).

- **Spot Instances:** Spot Instances are a cost-effective way to run Kubernetes workloads by utilizing spare Amazon EC2 capacity at significantly reduced prices compared to On-Demand instances. Spot instances come with a 2-minute termination window for which any running application must be evacuated and started on a new node (read more about Spot Instances here). If your application takes longer than two minutes to terminate, Spot Instances may not be a good solution — Java applications often fall into this category.

The above tactics can be effective; however infrastructure can never solve application problems. Any infrastructure change either masks the issue or punts the problem further down the line to a future date.

Four Steps to Reducing Waste & Saving Money

Fortunately, there are proven strategies that can help remove over-allocation entirely and can be effectively combined with the tactics above.

Step 1: Performance & Load Testing

The shape of utilization is best determined through a combination of real-world production use and load testing. Load testing is a type of performance testing that simulates real-world user load on software applications, systems, or websites to evaluate their performance under varying conditions. This testing is crucial for identifying how a system behaves under normal and peak load conditions, ensuring it can handle the expected number of users or transactions without performance degradation.

Objectives of Load Testing

- **Scalability Evaluation:** Load testing assesses the system's ability to handle increasing user and transaction demands. It helps identify the point at which the system starts to perform poorly.
- **Capacity Planning:** It aids in determining the system's ability to accommodate future growth in users, transactions, and data volume, facilitating informed decisions regarding infrastructure upgrades.
- **Bottleneck Identification:** Load testing helps pinpoint performance bottlenecks within the application or infrastructure, which can be crucial for optimizing system performance.
- **Response Time Analysis:** It involves tracking and evaluating response times for critical transactions and user interactions to ensure the system maintains acceptable performance levels under load.
- **Memory Leak Detection:** Load testing can help identify memory leaks that might degrade system performance over time.

How does Load Testing Reduce the Cost of Java on Kubernetes

Load testing enables more accurate estimation for Kubernetes pod requests and limits to reduce or eliminate over-allocation. This will not mean there is zero waste. Waste will still exist as RAM usage over time will fluctuate based on user demand, aka “load”.

The feedback loop from production is important as it needs to influence each new release’s load test. Both for expected user counts and benchmark performance.

Another benefit for load testing is the ability to scale to zero, or at least scale down horizontally. 12-factor application principles would recommend applications scale-out by process, not scale-up in resource reservations, something Java cannot do. With load test data in-hand operations teams can develop effective scaling plans to ensure resource consumption matches demand.

Step 2: Profiling & Instrumentation

Core components of Observability are application profiling and instrumentation. These capabilities are the two most effective means of isolating and resolving poorly performing services and code.

- Java profiling involves analyzing the performance of a Java application to identify bottlenecks and optimize its efficiency. Profiling tools collect data on various aspects of an application‘s behavior, such as CPU usage, memory consumption, and execution times, to provide insights into performance issues.
- Instrumentation refers to the process of adding code to an application to monitor its execution. This can involve inserting probes or hooks into the application code to collect data on performance metrics such as execution time, memory usage, and method invocations.

Combined, instrumentation and profiling data can be used to not just tune performance, but also increase the efficiency of resource utilization.

Many organizations forgo instrumentation as the price appears high and teams struggle to articulate the ROI. This is further complicated by the fact that instrumentation and profiling must happen in every environment, at every stage; development, testing and production. Skipping any environment negates the performance and cost benefits almost entirety.

Skipping the instrumentation of applications in Development Environments

- Identifying performance issues in development is the cheapest location to resolve them. With the right implementation goals instrumentation in development can allow engineers to solve even the slightest uptick in resource usage or remove the most minimal in response time degradations.
- Ignoring development means at best QA catches an issue, opens a bug and a triage process begins. This of course relies on QA executing load testing.

Skipping the instrumentation of applications in Test Environments

- If development is skipped, normally testing is skipped too. In this scenario the value of load testing is drastically reduced, as instrumentation data may identify a single function that is disproportionally consuming resources.

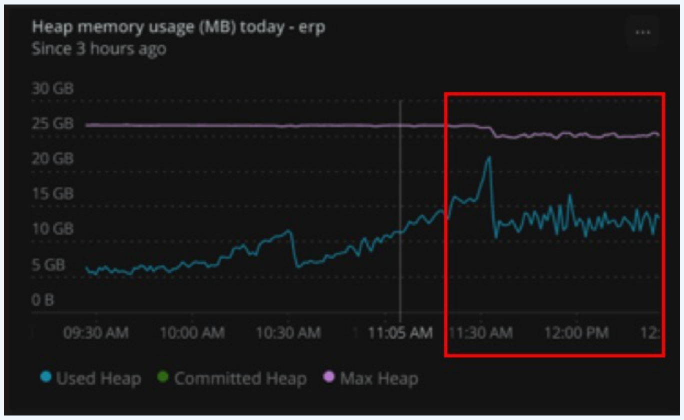
> Example: You have a microservice that is responsible for handling user login. Under a load of 300 simultaneous logins, which matches the normal 9am benchmark the service uses 300mb of RAM and the Kubernetes Pod request is set to 300mb. After enabling instrumentation, it’s identified that 200mb of RAM is used by the function that fetches users avatar image! 66% of the resource is consumed by a non-critical, quality of life feature.

- If instrumentation in Test environments is skipped, then all problems are pushed into production.

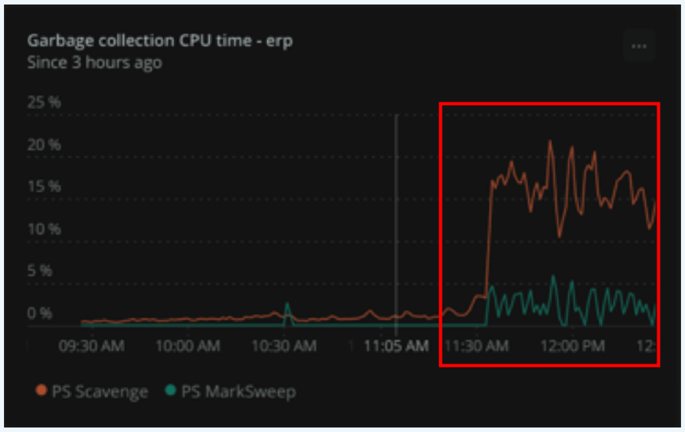
Skipping instrumentation in Production Environments

- Most organizations advance through a monitoring or observability strategy by starting with infrastructure basics (CPU, RAM), then onto ‘app’ by adding logs, and then dive into platform monitors such as database, Kubernetes — excluding or ignoring capabilities such as real user monitoring (RUM), profiling and instrumentation. Basic monitoring may suffice, but RUM, profiling and instrumentation greatly reduce mean time to identify and drastically improve mean time to resolution.
- Running in production without instrumentation can extend outages from hours into weeks.

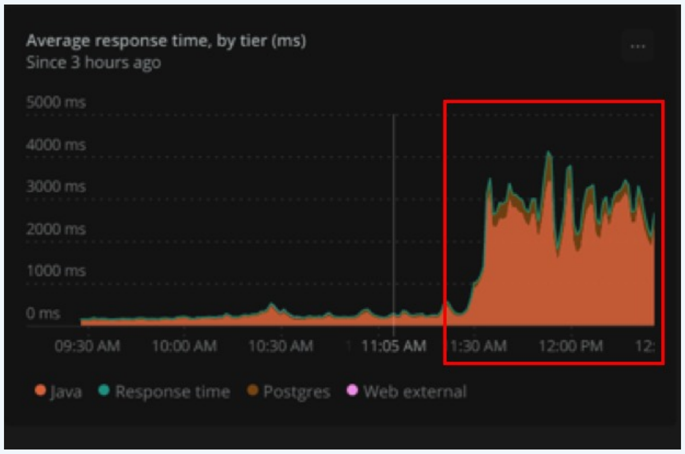
In a worst case scenario, your organization may release software that causes persistent slow response times. In the sequence of images below, users are impacted by a protracted period of 3 second response times that lasted for over 3 hours.



Java Heap with Constant Garbage Collection over a 3 hour Period



High CPU Usage Correlated too Constant Garbage Collection over a 3 hour Period



Poor Response Times Correlated to Constant Garbage Collection over a 3 hour Period

How does Profiling and Instrumentation Reduce the Cost of Java on Kubernetes

By catching issues before production and tuning the performance of specific functions organizations can ensure that the Pod Requests and Limits are not only accurate, but that the contained application is efficient and performant.

One of the most effective examples I have personally experienced was the use of instrumentation data to tune database queries that improved user experience and drastically reduced the load on the backend database. Ultimately, the reduction in usage by the Java application itself was minimal, but the database load reduction saved thousands per day.

Step 3: A Holistic Perspective - FinOps

Traditional technical problem solving necessitates a level of deep investigation, however, solving for cost and performance is orthogonal to this. Cost saving initiatives often start with a blown budget, and then a FinOps practitioner asks why. The first phase in FinOps is 'Inform' and focuses on visualization and allocation of spend. This phase starts with a holistic approach, and through the implementation of tools will help isolate the costliest aspects of an environment.

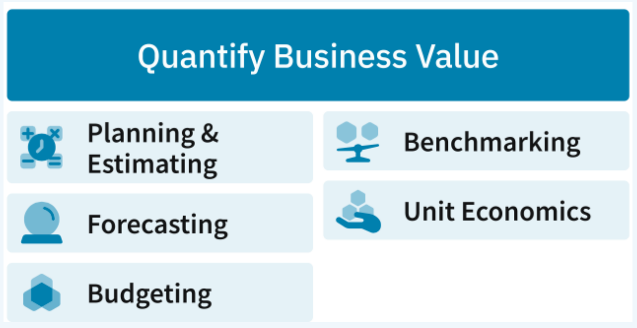
Isolating the costliest part of a Kubernetes cluster may seem like the most effective means for reducing costs, but the services that are driving load, both upstream and downstream, must be kept in context. For example, a payment service is tuned to reduce memory by half, a great savings, but the UX team adjust the checkout process and causes a sudden 3x increase in load. Ultimately the service has become more expensive and the quest for savings continues.

How does taking a Holistic Perspective Reduce the Cost of Java on Kubernetes?

A great approach to intrinsically linking your cloud spend to business value and achieving a holistic perspective is called Earned Value Management, a 40-year-old project management technique created by NASA and the US Military.

By focusing on the Business Value portion of the FinOps framework you will gain a contextual, near real-time view, into Budgeting, Forecasting and Benchmarking.

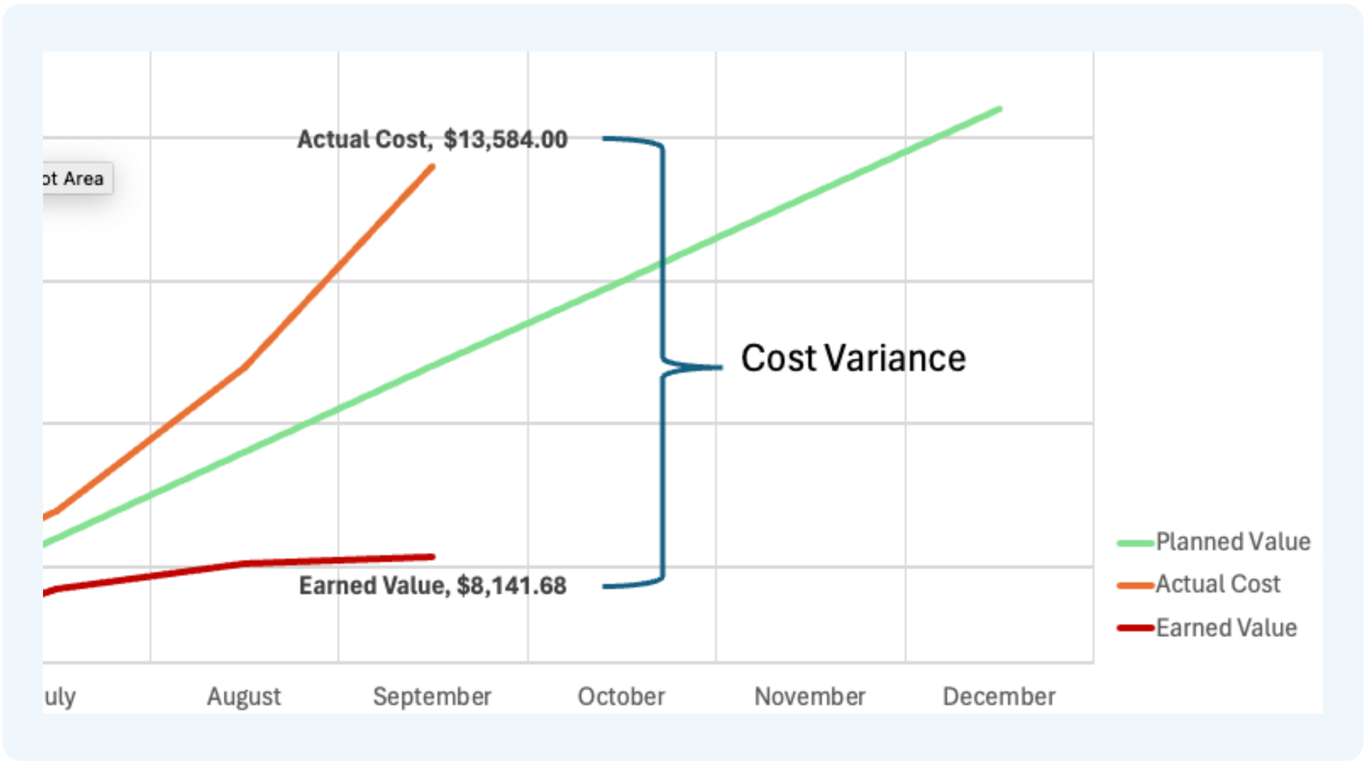
> Note: For Earned Value Management to work you must have defined unit economics for your applications.



The outcome of Earned Value Management is an immediate view for all teams, including engineering, into the cost of cloud and business outcomes of application services such as sales, support tickets, manufacturing, inference, deliveries. Pick what works for your business.

Earned Value Management enables a holistic perspective as it integrates outcomes into the cost management practice. You can read more about Earned Value Management in the blog Applying Earned Value Management to maximize ROI. By adopting Earned Value Managed you get views such as the chart below that highlight that a current budget miss is actually a colossal budget blowout. Earned Value Management generates a specific metric, 'Cost Variance' that calls out gains and losses based on your selected unit economics. This approach ensures your teams do not become hyper focused on just one aspect of tuning an application and simultaneously provides a view into business outcomes.

If we apply this to the payment service example, the team that cut the memory usage by half remains heroes, and the UX team needs to revisit their design. Ignoring this approach and operating without profiling or instrumentation would result in an unending quest to optimize the payment service.



Step 4: Apps on Kubernetes do not run like Apps on VMs

Static loads, low levels of resiliency, and monolithic multi-threaded applications—the world of VMs is not the world of Kubernetes and the approach to scale Kubernetes in a performant, cost efficient way is markedly different. To contextualize VMs vs Kubernetes it makes sense to discuss Kubernetes and applications by discussing the 12-Factor App methodology. Originally developed by Heroku, 12-factor outlines best practices for designing applications that are scalable, portable, and maintainable, which align well with the capabilities of Kubernetes as a container orchestration platform.

12-Factor Apps and Java

Process Model

The Twelve-Factor App methodology prefers the use of processes over threads for scaling applications. This model allows applications to handle diverse workloads by assigning different types of work to specific process types, such as web processes for HTTP requests and worker processes for background tasks.

Concurrency Management

While the Twelve-Factor App does not exclude internal concurrency management via threads, it emphasizes that the application must be able to span multiple processes across multiple physical machines. This means that while threads can be used within a process, the primary method for scaling should be through the process model.

Java Applications

In the context of Java applications, which typically manage concurrency internally via threads, the Twelve-Factor App suggests that these applications should still adhere to the process model for scaling. This involves using the Java Virtual Machine to manage threads internally but ensuring the application can scale by adding more processes rather than just increasing the number of threads within a single process.

12 Factor Apps, Kubernetes and the Impact on Waste in Java

Scaling on Kubernetes can be vertical (resizing a workload or compute node) or horizontal (adding additional workload replicas or nodes). For Java only CPU can be vertically scaled, and generally memory waste is more costly. As stated above, the more effective scaling method to adopt for resiliency and performance is horizontal, adding more instances of the same process. With respect to Java this would involve running multiple instances of the same Java application to handle load as it increases. The goal is to identify the lowest possible Java memory requirement to reduce the over-allocation to zero.

Zero Over-Allocation

Operating with near zero overallocation requires operational tooling, application features, and supporting practices that include:

- Application architectures that support horizontal scaling,
- QA and Testing teams that are able to execute very specific load tests, and
- Your organization must have implemented adequate performance monitoring as to identify any regressions in response times before they are identified in production and impacting users.

Combining these three elements makes it possible to identify when to scale an entire process vs. over-allocating memory to provide extra headroom that can support the increased load.

An adjacent benefit of this work is the often-overlooked capability of ‘scale-to-zero’ whereby when load hits zero, there are zero running processes. In this scenario, the Java application needs to be tuned to ensure near instant startup times to avoid negatively impacting users when they return to the product or service.

In conclusion, waste can be driven down using Kubernetes core features, but engineering and FinOps teams must work closely to continuously review and refine resource allocations and never lose sight of the bigger picture.

Special Mention

There is light on the horizon. For the first time in decades Java may become more system memory friendly for Java running on virtual machines. There is a very recent proposal to assist the OS in knowing which parts of memory inside the JVM are free by zeroing them out after garbage collection, which will make things like VM snapshots and restores faster since zeroed memory doesn’t need to actually be read or written – however, the prerequisite garbage-collect would be an expensive operation in many situations. In any case this is a long way from being merged and its impact on Kubernetes is still unknown.

How Elastic Machine Pool™ Eliminates Java Waste

As we mentioned at the beginning, the ability of hypervisors to overcommit physical resources drastically reduced the waste that the JVM caused when Java applications moved from physical servers to virtual machines. Unfortunately, Kubernetes has no such mechanism, and as clusters scale and more JVMs are deployed, each JVM's unused heap adds up. FinOps teams are impacted by ‚death-by-thousand cuts' and DevOps has no tools, tactics, or mechanisms to reduce Java's waste to zero.

Elastic Machine Pool (EMP) changes this by introducing overcommitment to Kubernetes. Using EMP clusters, specifically EKS clusters, can operate with overcommitted resources just as virtual machines did in the data center.

In respect to Java, the overallocated and unused portions of memory are dynamically detected and made available to other applications, not just Java. By removing Java memory waste, EMP is often able to reduce the operational cost of EKS by over 50%. Furthermore, the time investment to implement EMP is hours versus the weeks and months required to tune and rearchitect.

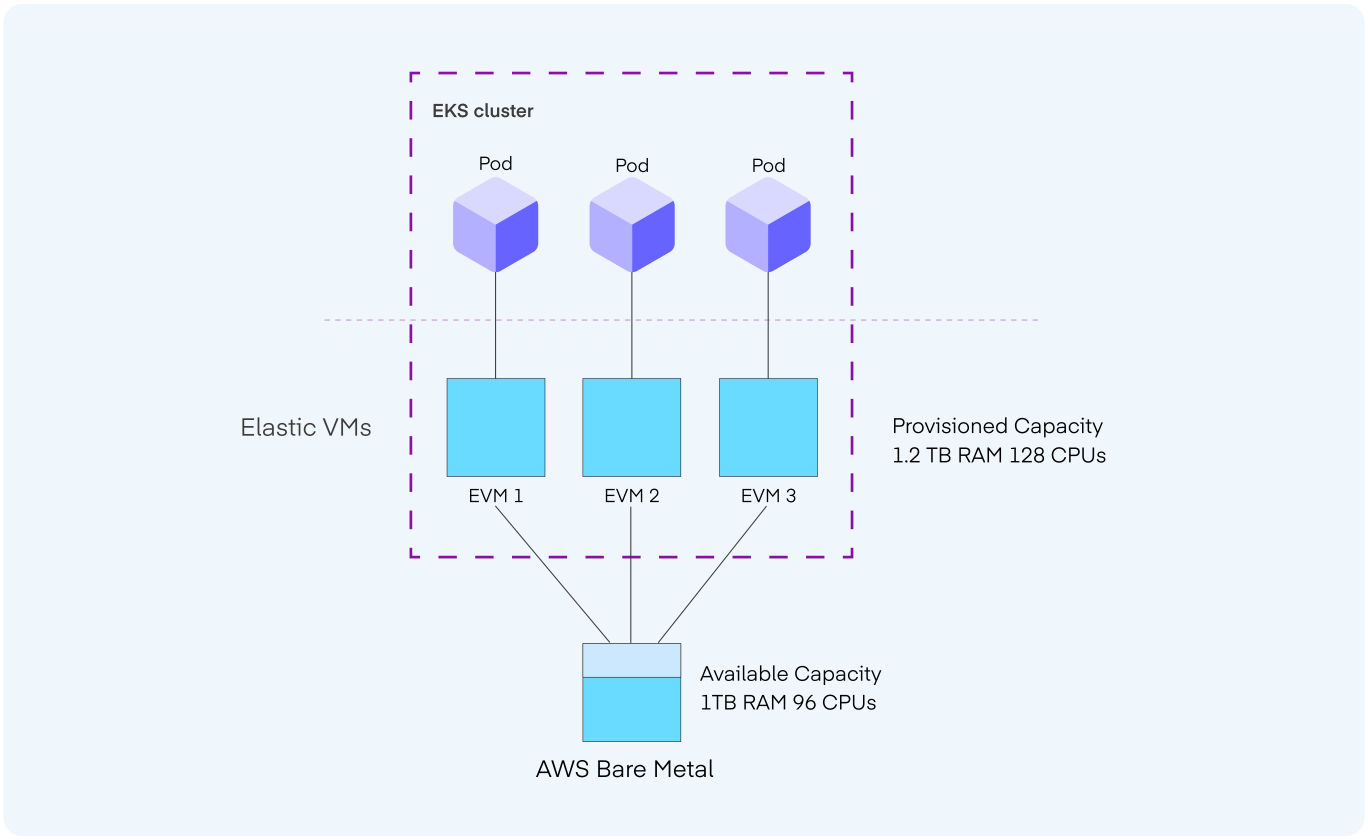
Overcommitment of Resources in Kubernetes to Remove Waste

As Kubernetes lacks a native mechanism to overcommit resources from EC2 instances, there is no means to utilize waste—memory that has been allocated to a JVM but is not in use. EMP allows DevOps teams and engineers to set resource overcommitment ratios for their EKS clusters. Once set, workloads can be scheduled to run on the nodes with overcommitted RAM and CPU.

EMP dynamically reallocates CPU and RAM that is allocated and unused by Pod-A to Pod-B. This reallocation is achieved by continuously evaluating resource requests and utilization across all workloads. When unused RAM or CPU is identified on a different node, workloads are live-migrated to leverage this prepaid capacity with zero pod disruption.

The only way to eliminate the waste that Java and its automated memory management cause is to change how Kubernetes can allocate resources.

Over-provisioning in EKS



- Around 2007, enterprises struggled with eliminating waste on physical servers – VMware addressed this by over-provisioning.
- Over-provisioning is a proven technology that automatically assigns more resources to virtual machines than available physical resources.
- EMP enables over-provisioning in EKS by replacing EC2 instances with Elastic VMs (EVMs) running on AWS Metal.
- EMP enables over-provisioning in EKS by replacing EC2 instances with Elastic VMs (EVMs) running on AWS Metal.
  EMP and EVMs run entirely within AWS and are fully compatible with EKS and other services like EBS, EFS, VPC etc

Java on Kubernetes Engineers Cheat sheet

Paths forward for running Java applications well in Kubernetes

Fortunately, there are a few techniques for running Java apps in Kubernetes that will help maximize your return on investment.

Investigate alternative garbage-collection options

Various alternative garbage collectors implement different collection processes and parameters to try to allow garbage collection to happen more often or more effectively without compromising critical dimensions of application performance. While an overview of alternative collectors is beyond the scope of this document, one of them will likely give you better behavior than the default – even enabling the G1 collector's memory-freeing behavior may be sufficient for your needs.

Tell the JVM in your containers how to manage memory better

Whether or not you use an alternative garbage-collector configuration, there are some knobs you can tweak to make the JVM manage memory more like you want it to:

- For cases where the JVM is trying to avoid long application pauses for garbage collection, but your application can actually tolerate those pauses well, you can use -XX:GCTimeRatio to make garbage collection run longer.
- Where the JVM is holding on to memory you would rather have it release, use -XX:MaxHeapFreeRatio to tell the JVM the maximum proportion of unused memory it should retain, releasing any excess back to the OS.

Make your application tolerant of more aggressive garbage-collection by making it more cloud-native

The main reason garbage collection has such a negative impact is that applications don't tolerate the pause it incurs well – depending on how garbage-collection happens, they might become unresponsive for relatively long periods. Hence, operators try not to enable aggressive garbage-collection. However, the issue of an unresponsive workload is not isolated to Java applications – any application can stall for various reasons, from I/O starvation on the node running the workload to something external that the application depends on (like a database) responding slowly.

In the cloud-native world there are well-understood methods for working around these issues – usually some variation of running multiple replicas and having data replication and failover for critical services, or caching data to avoid relatively long round-trips across a network link. It may require significant work to rearchitect things, but enabling your Java application (or things that depend on it) to handle a blip because of garbage-collection is likely to pay off in other congestion or failure scenarios.

Whichever of these options you choose (or even something not covered above!), the big takeaway is that you have options that can help assist your other optimization tools in being more effective. Experiment and see what works for you!

Java Tuning Checklist

Use the checklist below to help avoid common pitfalls and mistakes when deploying Java on Kubernetes.

Java Version	**Notes**	**Status**
Use Java Version	JDK version 10+ have Docker container detection which enables better resource management Versions 11+ collect and use cgroups v1 data; Versions 17+ and 11u have cgroups v2 support;	[ ] Pending [ ] Complete [ ] Can't Do
Java Container Support		
-XX:+UseContainerSupport flag	The -XX:+UseContainerSupport flag is enabled by default in Java 8u191+ to allow the JVM to detect if it is running inside a container environment. When enabled, the JVM reads cgroup information like CPU quota, CPU period, memory limits etc. to optimize its configuration and resource usage accordingly. The JVM can automatically detect and optimize for these cgroup limits when running in Kubernetes with the UseContainerSupport flag.	[ ] Pending [ ] Complete [ ] Can't Do
XX:ActiveProcessorCount	Set the number of CPUs visible by Java with the flag to ~2x the number of your container CPU limit;	[ ] Pending [ ] Complete [ ] Can't Do
XX:PreferContainerQuotaForCPUCount	can explicitly set or prefer the cgroup CPU limits.	[ ] Pending [ ] Complete [ ] Can't Do
Memory management		
Setting appropriate JVM memory limits (-Xmx, -XX:MaxMetaspaceSize, etc.) is crucial to avoid OutOfMemoryErrors on Kubernetes.	The -XX:+UseContainerSupport flag is enabled by Calculate memory limits carefully, considering the heap, metaspace, code cache, and thread stack sizes.	[ ] Pending [ ] Complete [ ] Can't Do
requests.memory limits.memory	Set limits and requests using Kubernetes flags.	[ ] Pending [ ] Complete [ ] Can't Do

CPU Management		
requests.cpu limits.cpu	Even if the steady-state CPU usage is low, Java applications need more CPU for startup and peak loads. Consider using CPU requests equal to limits, or slightly lower limits to allow for bursting. Setting CPU limits too low can significantly impact Java application startup times.	[] Pending [] Complete [] Can't Do
Garbabe Collector		
Garbage collector choice:	Select an appropriate garbage collector (GC) based on your application's requirements (e.g., throughput, latency). Commonly used GCs include ParallelGC, G1GC, and Shenandoah (for low pause times). Avoid automatic switching to the SerialGC, which is optimized for single-CPU environments.	[] Pending [] Complete [] Can't Do
Lifecycle hooks		
PostStart and PreStop	Use Kubernetes lifecycle hooks like preStop to capture diagnostic information (thread dumps, heap dumps) before pod termination. This information can be invaluable for troubleshooting issues that caused the pod to become unhealthy.	[] Pending [] Complete [] Can't Do
Lifecycle hooks		
PostStart and PreStop	Use a minimal base OS image (e.g., Alpine, Alpaquita) to reduce the overall container size and resource footprint Smaller base images can improve startup times and reduce storage costs.	[] Pending [] Complete [] Can't Do

Flags for running Java on Kubernetes

With various Java versions in use, some of these approaches may not work universally.

Heap and garbage

Here are some common recommendations for tuning garbage collection on the client side

- **-XX:+UseG1GC**
    - Use the G1 Garbage Collector: The Garbage-First (G1) garbage collector, introduced in Java 7, is designed to provide good performance with minimal pause times, making it suitable for client-side applications. Enable it with the option.
- **-XX:+UseZGC (experimental feature in Java 11 onwards)**
    - Performs all expensive work concurrently without stopping application threads. Designed for applications requiring low latency (< 10ms pauses) and/or using very large heaps
- **-XX:+UseParallelGC**
    - Intended for applications with medium to large data sets running on multiprocessor systems. Uses multiple threads to perform garbage collection in parallel
    - A good approach for containers with 1GB and 2CPU- A good approach for containers with 1GB and 2CPU
- **-XX:+UseSerialGC**
    - Uses a single thread to perform all garbage collection work. Best suited for single processor machines and applications with small data sets (up to ~100MB)
- **-XX:MaxGCPauseMillis=<ms>**
    - Set Maximum Pause Time Goal: With the G1 collector, you can set a target for the maximum pause time using the option. For client applications, a value between 100-200ms is generally recommended.
- **-Xms and -Xmx**
    - Set the initial and maximum heap size options. Start with a reasonable value based on your application‘s memory requirements, and adjust it based on monitoring and profiling.

- **-XX:+UseStringDeduplication**
  - Enable String Deduplication to reduce the memory footprint of String objects, which can be beneficial for client applications with many String objects.
- **XX:ParallelGCThreads=<N>**
  - Parallel GC Threads and adjust the number of parallel GC threads based on the number of available CPU cores.
- **XX:ConcGCThreads=<N>**
  - Concurrent GC Threads can set the number of concurrent GC threads based on the available CPU resources.
- **Disable Explicit GC Calls**
  - Avoid calling System.gc() or Runtime.getRuntime().gc() as it can negatively impact performance and pause times.

UseContainerSupport Flag

The -XX:+UseContainerSupport flag is enabled by default in Java 8u191+ to allow the JVM to detect if it is running inside a container environment.

When enabled, the JVM reads cgroup information like CPU quota, CPU period, memory limits etc. to optimize its configuration and resource usage accordingly.

The JVM can automatically detect and optimize for these cgroup limits when running in Kubernetes with the UseContainerSupport flag.

Java has become increasingly container-aware starting with JDK 9 upgrade:

- JDK versions 10+ have better Docker container detection algorithm and allow for better resource configuration usage and more flexible adjustment of heap percentage with available RAM;
- Versions 11+ collect and use cgroups v1 data;
- Versions 17+ and 11u have cgroups v2 support;
- And so on.

The latest versions include numerous improvements to garbage collection, affecting the KPIs greatly. Therefore, upgrading the JDK is crucial for optimal Java performance, not just in Kubernetes.

Container-specific

These arguments are especially useful in the case of containerized applications, where they help to adjust the heap size based on the available container memory.

- **XX:ActiveProcessorCount**
  - Set the number of CPUs visible by Java with the flag -
  - to ~2x the number of your container CPU limit;
- **XX:PreferContainerQuotaForCPUCount**
  - can explicitly set or prefer the cgroup CPU limits.
- **-XX:MaxRAM**
  - Sets the max. amount of total memory used by the JVM
- **-XX:MaxRAMFraction**
  - Sets the RAM limit for JVM in fractions
- **-XX:MaxRAMPercentage**
  - Sets the RAM limit for JVM per cent

Kubernetes workload parameters

- Standard in all Kubernetes versions
- requests.cpu
- limits.cpu
- requests.memory
- Limits.memory

Kubernetes kubelet runtime flags

- 1.27 onwards (Alpha)
- Enable using: --feature-gates=InPlacePodVerticalScaling=true
  - resizePolicy
    1. resourceName: cpu
    2. restartPolicy: NotRequired
    3. resourceName: memory
    4. restartPolicy: RestartContainer

Appendix

References and sources

Below are reference used to write this guide along with articles for tuning Java. A big thanks to their authors for consolidating and diving into specific aspects of Java on Kubernetes along with the associated configurations.

1. Interesting Application Garbage Collection Patterns
2. JVM Kubernetes: Optimizing Kubernetes for Java Developers
3. Java 17 features: A comparison between versions 8 and 17. What has changed over the years?
4. 7 tips to optimize Java on Kubernetes
    - Resize CPU Limit To Speed Up Java Startup on Kubernetes
    - A Step-by-Step Guide to Java Garbage Collection Tuning
    - How the JVM uses and allocates memory
    - Java (JVM) Memory Model - Memory Management in Java
    - Garbage collection and heap expansion
    - How Garbage First Garbage Collector (G1GC) affected the performance of our back-end

Images sourced from

1. https://stackoverflow.com/questions/61079322/jvm-heap-usage-variation-is-this-graph-normal
2. https://discuss.elastic.co/t/heap-usage-issue/224554
3. https://stackoverflow.com/questions/67929735/major-gc-decreasing-performance
4. https://www.finops.org/framework/

Platform9 empowers enterprises with a faster, better, and more cost-effective way to go cloud native. Its fully automated container management and orchestration solution delivers cost control, resource reduction, and speed of application deployment. Its unique always-on assurance™ technology ensures 24/7 non-stop operations through remote monitoring, automated upgrades, and proactive problem resolution. Innovative enterprises like Juniper, Aeromexico, Mavenir, Rackspace and Cloudera achieve 4x faster time-to-market, up to 90% reduction in operational costs, and 99.9% uptime. Platform9 is an inclusive, globally distributed company backed by leading investors.

Follow us on:

Headquarters: 84W Santa Clara St Suite 800, San Jose, CA 95113.
India office: 7th Floor, Smartworks M Agile Building, Pan Card Club Road, Baner Pune, 411045 Maharashtra, India.

Phone:	Website:	Email:
+1 650-898-7369	https://platform9.com	info@platform9.com