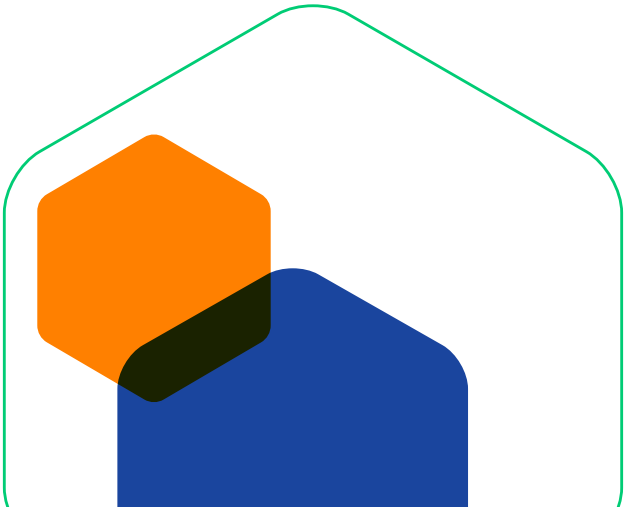
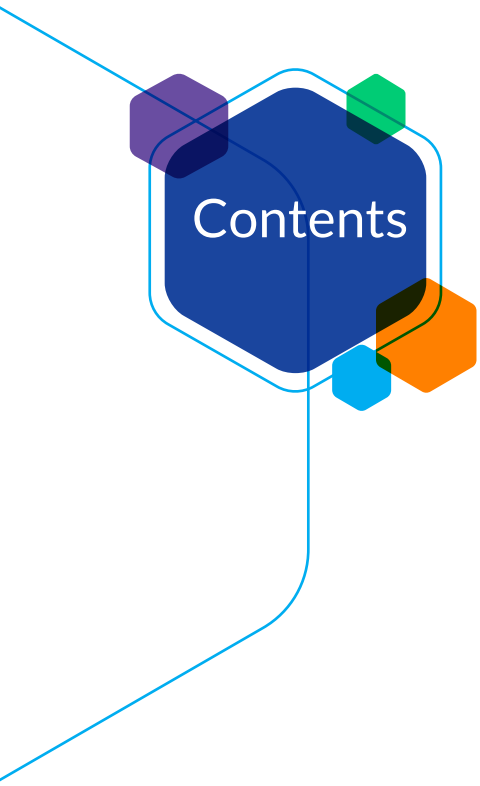
CloudBees.

How to Solve the Monolithic Jenkins Controller Problem





Introduction

Whether you've been working successfully on your continuous integration (CI) infrastructure for some time or are just starting to build it out, you have realized the power of CI and are enabling your teams to be successful by implementing, administering and sharing this powerful tool with the masses. Fast forward to 3:00 AM on a Sunday morning... your phone is ringing...and your CI solution is down. Now what?!

This is a preventable tragedy (in most cases) that befalls so many CI administrators. The number one reason we see this type of scenario is due to monolithic controllers (formerly called masters). But it's not that plain and simple. The road is paved with many red flags along the way that warranted a detour in your journey with CI. In this document, we aim to help you understand and identify those red flags to prevent you from creating these monolithic controllers and share success and failure stories from your peers along the way.

CloudBees brings well over a decade of field experience in the DevOps space, understanding the inner workings of Jenkins and most importantly, real world examplesthat date back to the inception of Jenkins. We know with certainty that monolithic controllers that house too many jobs and users aka "Jenkinsteins" do not belong in the success story category. The good news is that if you are dealing with a monolithic controller, we know how to help you. We have proven, documented methods of bringing you from what can be a painful experience into a modern, horizontally scaled, performant CI solution.

What is a Monolithic Controller?

We combined a decade of field data and found that any controller that houses over five thousand jobs has experienced, or will likely experience, performance and stability issues. Now, you might say this number is grossly underestimated, and in your case, it may be true. For example, five thousand "Hello World" jobs are going to perform completely differently than 5,000 complex pipelines with multiple stages, making different types of calls to binaries on the host. CloudBees has landed on the 5,000 number as a happy medium between the two examples. Coupled with field experience, data gathered from our performance team (and shared later in this document) shows that customers who experience performance and stability issues are likely running controllers that run more than 5,000 jobs.

The number one question Jenkins administrators have when broaching this subject is, "How many jobs can I run on my controller?" Unfortunately, there is not a simple answer due to the number of variables that are attached to the question. The pipeline complexity variable alone is enough to sway a guestimate into too many unknowns. The correct path however, is to use the five thousand jobs recommendation as a highwater mark to know when it is time to horizontally scale. Employing a monitoring solution to track macro metrics (CPU/RAM/DISK IO) and creating baselines will allow you to properly plan for scaling. Micro metrics such as garbage collection logs, object creation rate and thread counts can be analyzed and baselined so you understand the needs of your jobs and properly plan for additional controllers as needed in your cluster.

When too many users are on a single controller

In addition to job count, it is very typical that we see a high number of users on a single Jenkenstein. This situation typically happens because users are onboarded into Jenkins without a clear definition of roles or team association. A user sees the benefits that another user is seeing from Jenkins and asks to be onboarded, and they tell two friends and so on. As an administrator, this can be a difficult pain point to manage. We recommend a team-based approach when architecting your Jenkins cluster. Not only does it reduce cross-team friction when managing plugin versions and needs of agents, but it also limits the blast radius when something goes wrong. If Jenkins was only reliant on its core codebase, the likelihood of negative events could theoretically be limited thanks to extensive testing. However, Jenkins is a unique application that allows developers to inject their own code, which greatly increases the likelihood of human error. Thus, limiting the number of users or teams associated with a controller is considered a best practice.

Signs You Are on the Road to Running a Monolith Controller

1. You Experience Longer Garbage Collection Cycles

The solution to scaling has traditionally been to increase resources, but this vertical scaling methodology comes with a variety of challenges. You may see temporary relief, but ultimately other areas of the application or surrounding architecture are affected, albeit unintentionally, in a negative way. Most notably, administrators will tend to increase heap memory beyond the CloudBees recommendation of 16GB at most. By doing so, they are inviting longer running garbage collection cycles. As garbage collection cycles are inherently stop-the-world events, the aim is to limit the amount of time a garbage collection event takes place, ideally under one second as not to interrupt other operations such as HTTP requests. Horizontal scaling and reducing the footprint of the controller is the only way to properly plan and scale your CI architecture safely and in alignment with the direction of CI and DevOps as a whole.

2. You don't have a cleanup strategy

Another indicator of a monolithic controller is a large $JENKINS_HOME footprint. Jenkins does not rely on an external database, instead housing a multitude of configuration files inside of the $JENKINS_HOME defined storage location. In situations where a large number of users and jobs are in play within a monolithic environment, we also see a large number of abandoned jobs. These abandoned, sometimes multiple-years-old files are loaded into application memory, creating unnecessary additional overhead. Ensuring that you have a cleanup strategy and monitoring the growth of your $JENKINS_HOME location are both considered best practices in the ongoing prevention of creating a monolith.

3. Lack of Infrastructure / Growing too fast

Many times, especially in verticals such as financial services, we see Jenkins administrators struggle in acquiring the necessary infrastructure needed to accommodate growth in a timely manner. Understandably, there can be a lot of red tape involved that prevents them from spinning up additional virtual machines. Because of this, we often see monoliths form faster than usual. Then, Jenkins administrators are forced to maintain a delicate balancing act of appeasing their customers and keeping the controllers online. To remedy this, it is crucial to understand and plan for growth by segmenting controllers by team or business unit and setting limits to the number of jobs allowed on each controller, such as the five thousand job limit.

Remedying a Monolithic Controller

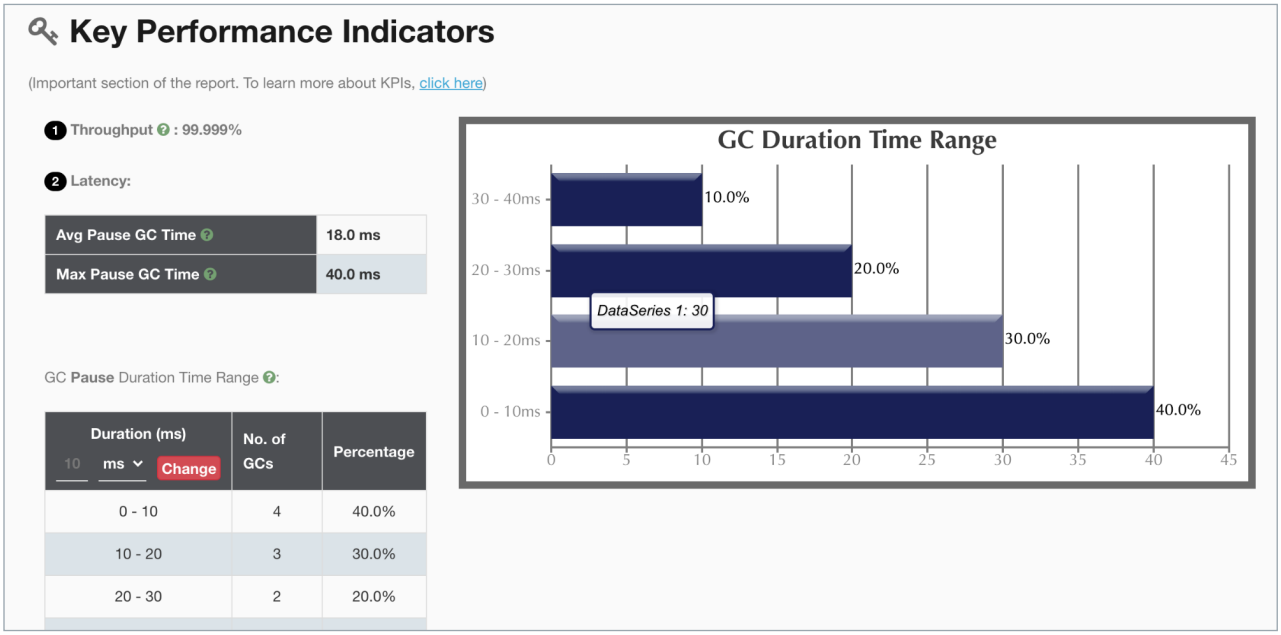
So, you are dealing with a monolithic controller and are experiencing pain because of it. The first step is to engage with CloudBees Support as we may be able to provide a temporary solution that can help you while we work together to develop a horizontal scaling plan.

Success in Scaling

Many years back, a large, multinational financial institution was experiencing performance issues with their controllers. Their users jumped at the chance to use Jenkins. The administrators at the time had no real plan in place to onboard these users outside of a first-come, first-serve basis. At the time, there were three controllers, split amongst three development teams, leading to about 50 users per controller.

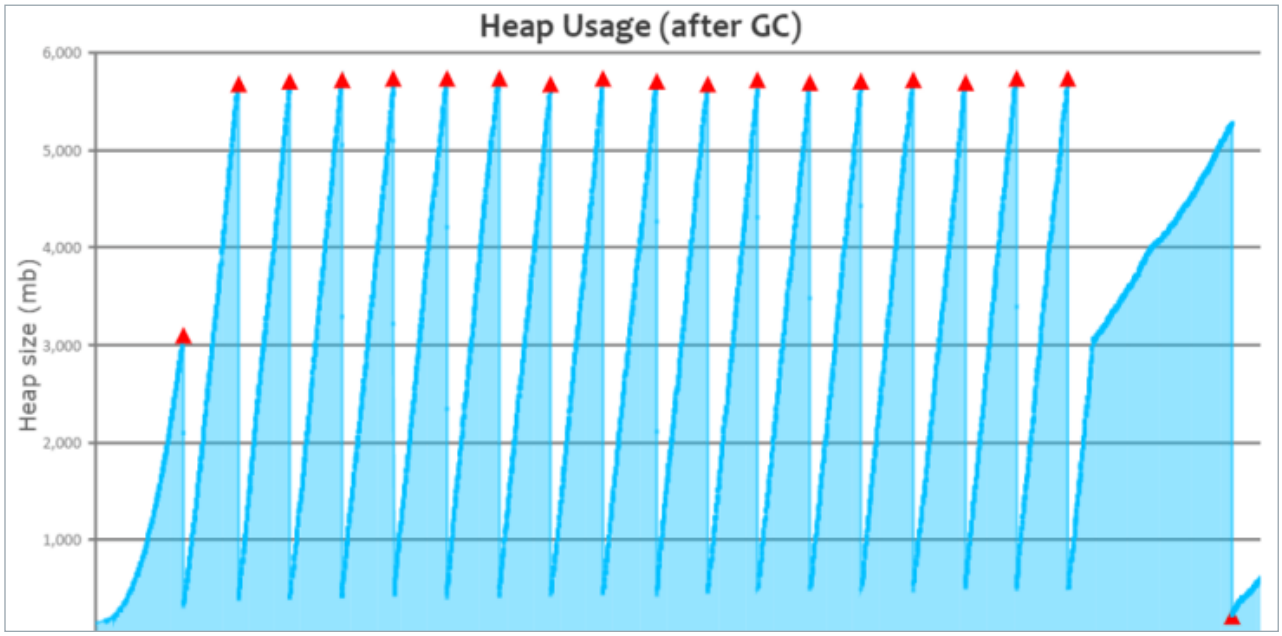
A review of garbage collection logs during this period reveals healthy KPI's, where we are focused on application throughput (the amount of time the application is not executing GC cycles) and max pause GC time, which in a healthy instance we expect to be greater than 99 percent and less than one second respectively.

Figure 1



During GC log analysis (Figures 1, 2 and 3), we looked for a sawtooth pattern, which is indicative of a healthy garbage collection cycle. and confirmed this was occurring.

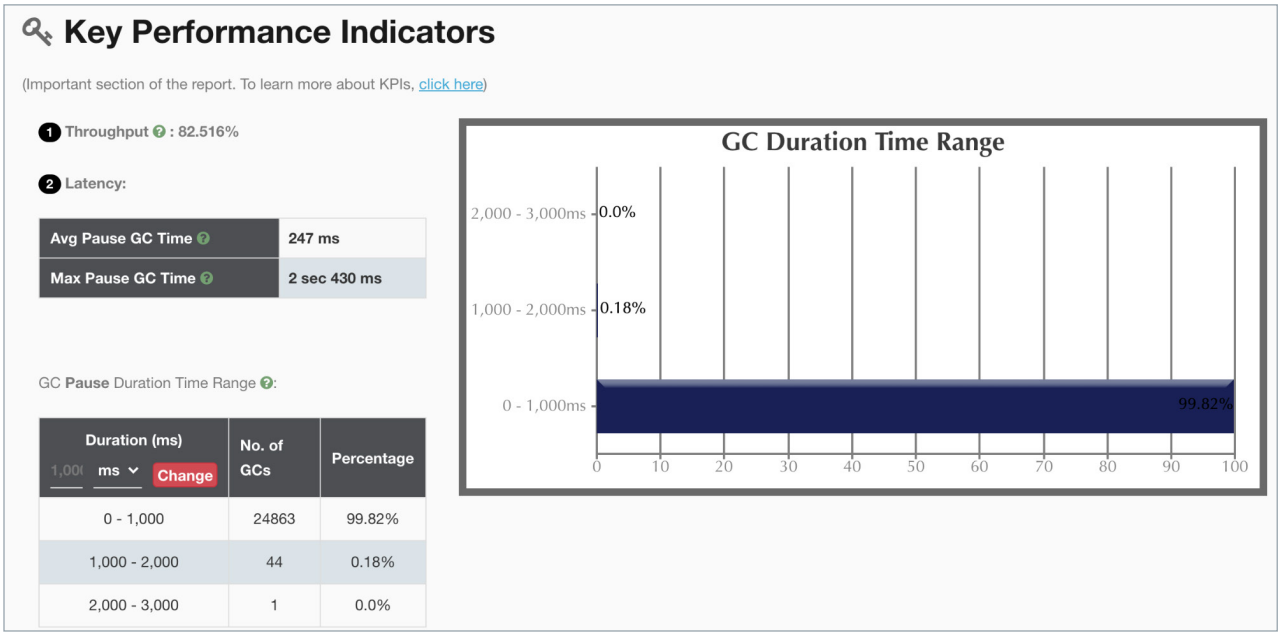
Figure 2



In the first few months, things were going swimmingly and the adoption rate of Jenkins was skyrocketing. Users were seeing clear benefits from the tool, and for the most part, things were manageable. Fast forward six months and adoption had multiplied exponentially. Administrators were fielding requests from users left and right about jobs breaking and plugin version mismatches. This put them in a very reactive mode of administration. “Who broke the build?” became the common theme of the day and even more often people were heard saying “Jenkins is down.”

During this time, a review of the garbage collection logs revealed that application throughput had gone down to 82 percent which means that the application was spending 18 percent of its time performing garbage collection. There was also an increased amount of long-running pauses greater one second, where the application would be temporarily perceived as unresponsive. (Garbage collection events are “stop the world” events and no other actions will take place until the garbage collection cycle is completed, including HTTP requests.)

Figure 3



This is every administrator's nightmare when it comes to being responsible for the success or failure of their CI solution. When we look at what happened in this journey, we saw that a monolithic architecture was at the root cause of their issues. In the course of several months, they were seeing an average job count per controller of about 25,000 and a user base that was essentially stepping on each other's toes when it came to managing their agents and plugin versions.

Another tell-tale sign of a monolithic controller that can be observed through garbage collection log analysis is a high object creation rate. Baselining and monitoring object creation rates can be used as a barometer for load. In this case, we saw an average creation rate of about 543MB per second, compared to six months earlier where we saw an average creation rate of about 36MB per second.

Figure 4: Before

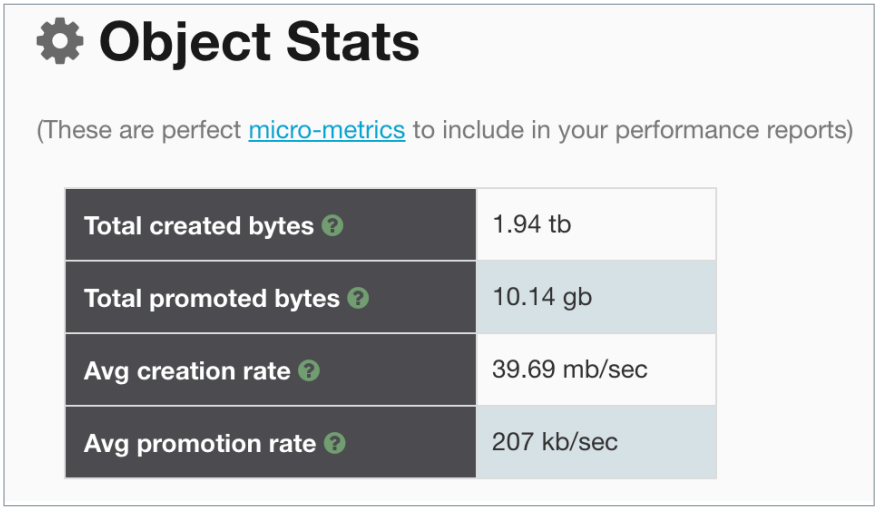
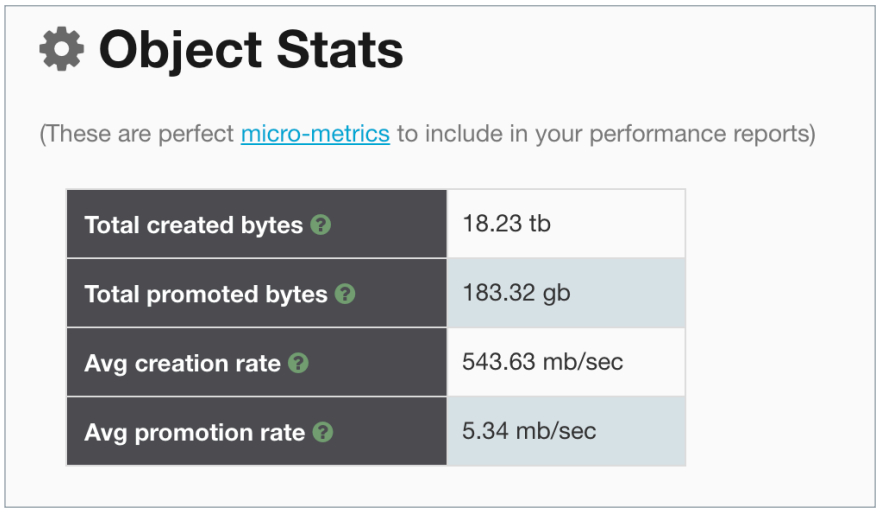


Figure 5: After



When the administrators approached CloudBees Support, we immediately engaged with them to construct a plan of action. The first thing we suggested was to lock down roles via role-based access control (RBAC) and ensure only team leaders had privileges to make changes. This, coupled with implementing a change management strategy, prevented users from unknowingly making changes that affected other teams. We also suggested best practices to add to their change management process, utilizing tools like the Audit Trail Plugin and Job Config History Plugin to bring some order to chaos. In addition, we worked with the large financial institution to ensure they were following our documented JVM Best Practices.

The next step was to calculate total controller requirements. With three controllers, housing around 500 Jenkins users, 20 teams and about 25,000 jobs each, it was decided to make the ask to the infrastructure team immediately for an additional 25 hosts. The goal was that each team would receive their own instance and have a few extra hosts for forecasted team growth. The easiest solution here is migrating to a modern architecture in the cloud, however in this situation, due to a lack of cloud infrastructure, we were forced to continue scaling a traditional platform.

In the interim, while the hosts were being provisioned over a six month period, the administrators continued to feel the pain of having so many users and jobs on a single controller. The controllers had to be restarted daily due to the sheer amount of jobs that were being processed, leading to long running garbage collection cycles and end-users complaining of not being able to access the UI. By vertically scaling and maxing out resources to the hosts, coupled with tuning the JVM per our best practices, we were able to lessen some of the pain, but it is not a long-term solution.

Once the new controllers were in place, we were able to follow the documentation on splitting controllers and migrated users off of the monoliths and into their own segmented team controllers.

The benefits of moving to this horizontally scaled architecture were seen over the course of a six month engagement. Where once controllers were being restarted daily, the number of issues administrators were dealing with on a daily basis plummeted. Jenkins controllers were considered healthy, job counts were well under the 5,000 recommendation, and overall performance was at an all-time high.

The Future of CI and Horizontal Scaling

We typically see monolithic controllers in on-premise CI implementations. It's rare that CloudBees sees a case in CI Modern architectures where a customer is suffering from issues stemming from monolithic architecture unless they brought it with them from a previously on-premise environment. This is primarily due to the inherent horizontal and ephemeral nature of Kubernetes. Breaking out from monolith methodologies follows in the trend of containerization in software architecture and development.

:As we look to the future of CI, and with Kubernetes being the widely accepted vehicle for CI, if you aren't thinking about a path to CI with Kubernetes yet, you should be.

Learn More

- **Read the Case Study**
  Autodesk Builds Better Software Faster with CloudBees
- **Get More Tips on Managing and Scaling Jenkins**
- **Read the Whitepaper**
  The Definitive Guide to Modern Software Delivery

Get a Free Trial Today!

https://www.cloudbees.com/use-case/manage-scale-jenkins

CloudBees CI is built on top of Jenkins, an independent community project. Read more about Jenkins at: www.cloudbees.com/jenkins/about

0122v00

CloudBees, Inc.
4 North Second Street | Suite 1270
San Jose, CA 95113
United States
www.cloudbees.com
info@cloudbees.com