International Journal of Science and Research (IJSR)
ISSN: 2319-7064
Impact Factor 2024: 7.101

SRE-Driven Cost Optimization in Kubernetes Using Resource Profiling

Kiran Thankaraj

Sephora USA Inc

Abstract: ***In today's cloud-native era, managing the cost of Kubernetes is more business requirement than a tech issue. In this journal, we look at how Site Reliability Engineers (SREs) can face cost inefficiency head-on through resource profiling. Through the use of hands-on tools and production-grade workflows, SREs can identify what is over-provisioned, under-used, or mis-configured. We guide you through a hands-on, step-by-step method to optimizing Kubernetes workloads without compromising reliability. Along the way, we'll highlight key lessons, real challenges, and proven practices that empower SRE teams to make smarter, cost-conscious decisions.***

Keywords: Kubernetes, Site Reliability Engineering, Cost Optimization, Resource Profiling, Observability, Cloud-Native, Performance Tuning

1. Introduction

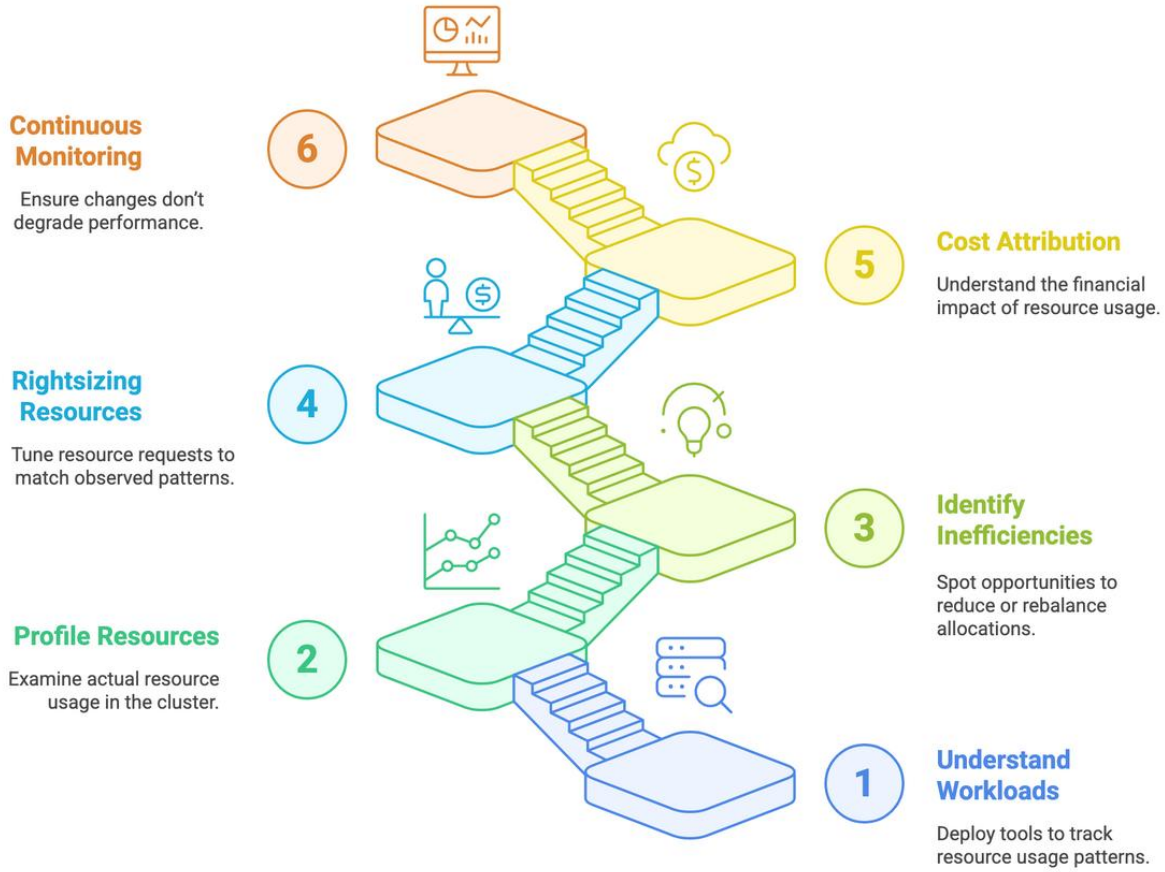
Kubernetes has quickly become the backbone of modern application infrastructure. Its flexibility, scalability, and rich ecosystem are game-changers—but they come with a price. Literally. As usage scales, so do cloud bills, and often, resources are over-provisioned "just to be safe." That's where SREs step in. SREs don't just keep systems reliable—they ensure they run efficiently. One powerful approach? Resource profiling. It's about knowing what your applications truly need to run well—and cutting the fat. This journal explores how SREs can lead the charge in cost optimization using the data and tools they already have.

Why This Matters for SREs?

As companies move more toward Kubernetes and cloud-native setups, the job of Site Reliability Engineers (SREs) is becoming more expansive. Sure, we're still laser-focused on keeping systems reliable, but these days, we're also being asked to think about cost efficiency. It's no longer enough for systems to just work well—they need to use resources in the smartest way possible to avoid unnecessary spending.

One of the best tools we have as SREs is observability. This is where we finally get to start noticing inefficiencies. For example, sometimes services ask for way more CPU or memory than they even use. Yes, this may avoid failures, but it also leads to wasted resources and huge cloud bills. On the other hand, if a service lacks resources, then it can lead to slowdowns or even outages, which is not great. This is where resource profiling comes in. By looking more closely at how resources are being used, we can make better decisions about how to optimize. For instance, we can dynamically set CPU and memory limits according to actual usage, avoiding over- and under-provisioning. SREs are well-placed to do this because we sit at the intersection of infrastructure and the applications that execute on it.

2. Methodology: Step-by-Step SRE Workflow for Cost Optimization



3. Challenges in Resource Profiling & Optimization

- **Dynamic Workloads:** Application workloads vary over time due to traffic patterns, seasonality, or business events. Profiling based on a static snapshot often leads to inaccurate or short-lived optimizations.
- **Limited Usage Visibility:** Kubernetes doesn't natively provide granular resource usage data. Without the right observability stack, understanding real consumption versus allocation is difficult.
- **Performance vs. Efficiency Trade-offs:** Tuning for cost savings can impact system stability or response times. Striking the right balance between efficiency and reliability is a constant challenge.
- **Autoscaling Complexity:** Horizontal and vertical pod autoscalers rely on clean, timely metrics. Misconfigured thresholds or noisy signals can lead to unpredictable scaling behavior.
- **Multi-Tenancy Conflicts:** In shared environments, noisy neighbors and lack of resource isolation muddy profiling data. Ownership and accountability also become harder to enforce.
- **Fragmented Tooling:** Monitoring, cost analysis, and resource tuning often span multiple tools. Stitching insights together from disparate systems slows down decision-making.
- **Organizational Resistance:** Engineering teams often default to over-provisioning out of caution. Promoting resource-conscious development requires cultural change and education.

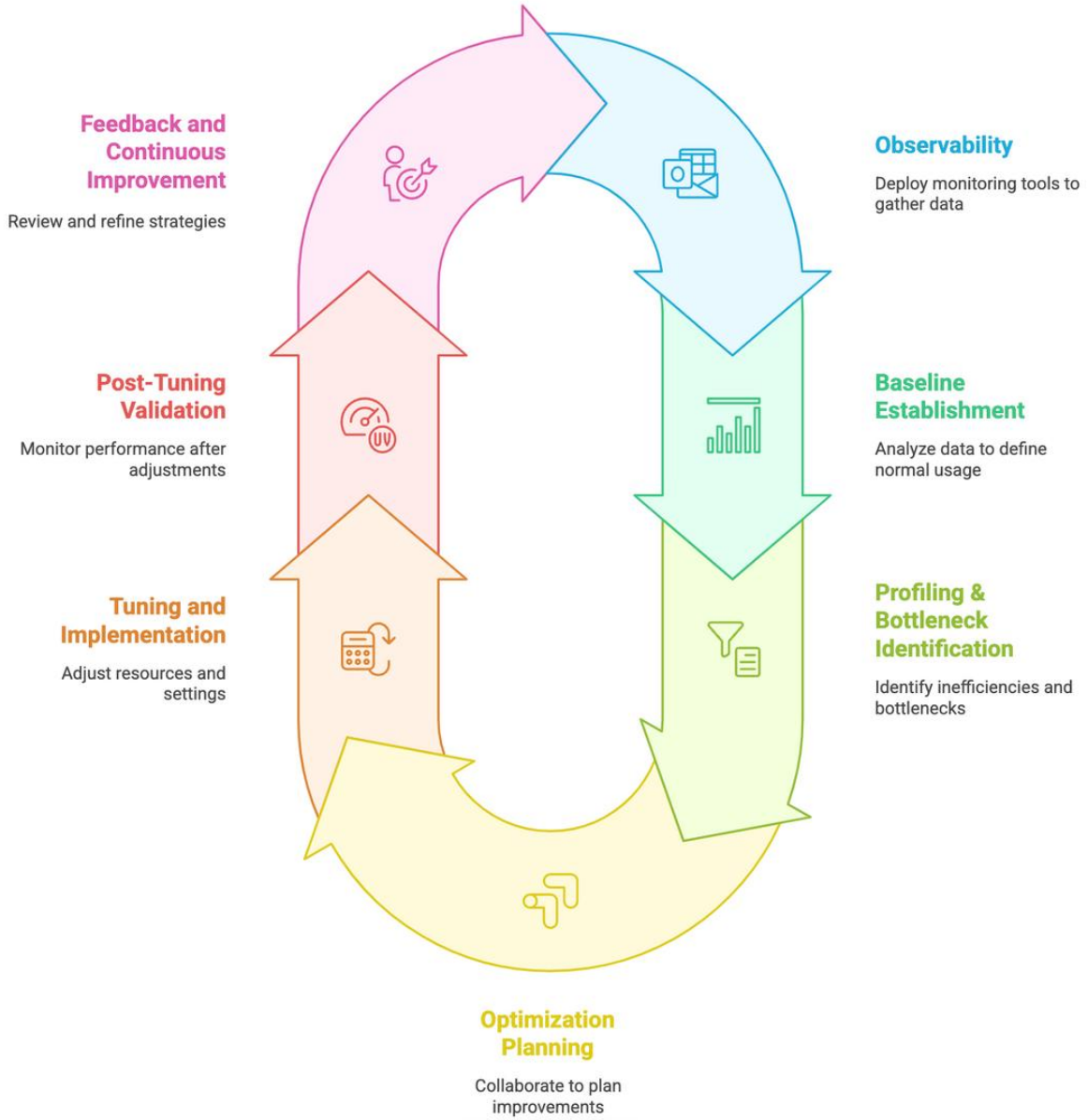
4. Best Practices

- **Establish Baseline Metrics:** Keep an eye on how your services utilize CPU, memory, and other critical resources over time. This is a good reference point and enables you to spot anything unusual early on.
- **Use Profiling in Staging First:** It's a less risky environment to test out your profiling methods before you start making changes in production. It's a better place to figure out what works—and what doesn't—without risking user dissatisfaction.
- **Set Realistic Requests & Limits:** Avoid guessing and the default settings—use what your metrics are already indicating. This prevents both waste from over-provisioning and slowdowns from under-provisioning.
- **Implement Fine-Grained Autoscaling:** Enabling autoscaling and then bail is not enough—ensure that it's calibrated for your true workloads. Accurate autoscaling is money-saver and keeps things running smoothly when demand surges.
- **Leverage Cost Visualization Tools:** Utilize tools such as Kubecost or your cloud provider's billing dashboards to illustrate where the spend is occurring. When teams can see the numbers, they'll care—and take action.
- **Run Regular Optimization Reviews:** What was good last quarter might not be good this quarter, so check your resource settings frequently. A daily or monthly check-in can prevent bigger problems (and bills) later on.
- **Promote a Resource-Conscious Culture:** Get developers to understand that resources aren't "free" and over-provisioning expenses accumulate. When everyone

has skin in the cost story, smarter decisions get made up front.

- **Automate Where Possible:** Use policy tools or scripts to enforce resource limits and avoid surprises. This keeps your environment clean and saves your team from chasing configuration drift.

5. Illustration: Profiling & Optimization Lifecycle



1) **Start with Observation:** The very first thing to do is to observe simply. We rely on the monitoring tooling—Prometheus, Grafana, possibly Datadog or New Relic—to offer us a proper comprehension of what's going on underneath the hood. We are not guessing; we are looking at how services behave at different times during the day, traffic loads, and deploys.
2) **Understand the Baseline:** After a couple weeks of watching, trends begin to emerge. You have an idea of what is "normal"—how much CPU something typically uses at its busiest times, or how much memory it holds after a job completes. This is your baseline, and all else is built upon it.
3) **Spot the Inefficiencies**: As soon as you have sufficient context, inefficiencies are in the forefront. You have pods that are allocating much more CPU than they ever will need, or spiked-up, crashed jobs due to memory constraints which are too tight. It is detective work—it's either occasionally because of a bad limit or legacy config that no one ever questioned.
4) **Bring in the Dev Team**: This section is important: don't do this by yourself. Sharing these results with the development team takes profiling from a cost-saving exercise and makes it a collaborative effort. Maybe they were over-provisioning because of one bad incident six months earlier. Sharing real utilization facts, you can make smarter decisions together.
5) **Adjust Resources Carefully**: Now comes the tuning phase. You tweak CPU and memory requests, realign autoscaler targets, maybe even introduce vertical pod autoscaling to some pickier services. Roll out changes piecemeal—canary or blue/green if that makes a difference—always checking to make sure nothing gets harmed.
6) **Validate, Don't Assume**: Just because you've "optimized" something doesn't necessarily mean it's

optimal. After-the-fact monitoring is valuable. Did latency balloon? Are there retries through? Are you writing what you're supposed to? Sometimes, proper settings aren't apparent until they've been deployed in production for a few weeks.

7) **Keep the Loop Going**: This is not a one-off activity. Business needs shift. New features release. Traffic behavior shifts. Profiling and tuning become a drumbeat—something you return to during incident reviews, capacity planning, or simply your monthly optimization check-ins.

6. Conclusion

Kubernetes gives us tremendous power—but with great power comes great responsibility (and often, a great cloud bill). By taking a structured, data-driven approach to resource profiling, SREs can cut waste, improve performance, and help their organizations scale more sustainably. This isn't just about saving money—it's about building smarter systems. With the right mindset and tools, SREs can lead the way to a more efficient, resilient future.

References

[1] Kubecost. https://kubecost.com
[2] Goldilocks by Fairwinds. https://github.com/FairwindsOps/goldilocks
[3] Vertical Pod Autoscaler. https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler
[4] ArgoCD. https://argo-cd.readthedocs.io
[5] Prometheus. https://prometheus.io