Best practices for running cost-optimized Kubernetes applications on GKE

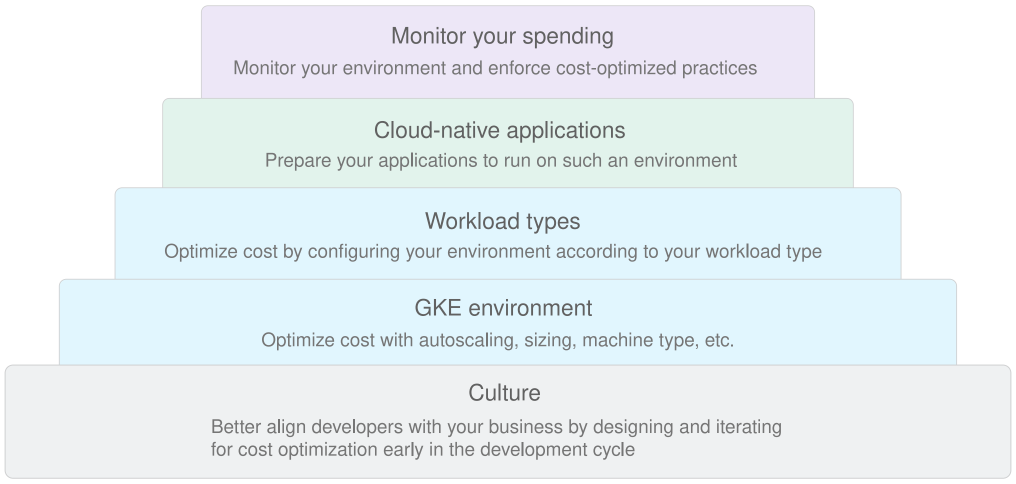
This document discusses Google Kubernetes Engine (GKE) (/kubernetes-engine) features and options, and the best practices for running cost-optimized applications on GKE to take advantage of the elasticity provided by Google Cloud. This document assumes that you are familiar with Kubernetes, Google Cloud, GKE, and autoscaling.

Introduction

As Kubernetes gains widespread adoption, a growing number of enterprises and platform-as-a-service (PaaS) and software-as-a-service (SaaS) providers are using multi-tenant Kubernetes clusters (/kubernetes-engine/docs/concepts/multitenancy-overview) for their workloads. This means that a single cluster might be running applications that belong to different teams, departments, customers, or environments. The multi-tenancy provided by Kubernetes lets companies manage a few large clusters, instead of multiple smaller ones, with benefits such as appropriate resource utilization, simplified management control, and reduced fragmentation.

Over time, some of these companies with fast-growing Kubernetes clusters start to experience a disproportionate increase in cost. This happens because traditional companies that embrace cloud-based solutions like Kubernetes don't have developers and operators with cloud expertise. This lack of cloud readiness leads to applications becoming unstable during autoscaling (for example, traffic volatility during a regular period of the day), sudden bursts, or spikes (such as TV commercials or peak scale events like Black Friday and Cyber Monday). In an attempt to "fix" the problem, these companies tend to over-provision their clusters the way they used to in a non-elastic environment. Over-provisioning results in considerably higher CPU and memory allocation than what applications use for most of the day.

This document provides best practices for running cost-optimized Kubernetes workloads on GKE. The following diagram outlines this approach.


Monitor your spending
Monitor your environment and enforce cost-optimized practices
Cloud-native applications
Prepare your applications to run on such an environment
Workload types
Optimize cost by configuring your environment according to your workload type
GKE environment
Optimize cost with autoscaling, sizing, machine type, etc.
Culture
Better align developers with your business by designing and iterating for cost optimization early in the development cycle

The foundation of building cost-optimized applications is spreading the cost-saving culture across teams. Beyond moving cost discussions to the beginning of the development process, this approach forces you to better understand the environment that your applications are running in—in this context, the GKE environment.

In order to achieve low cost and application stability, you must correctly set or tune some features and configurations (such as autoscaling, machine types, and region selection). Another important consideration is your workload type because, depending on the workload type and your application's requirements, you must apply different configurations in order to further lower your costs. Finally, you must monitor your spending and create guardrails so that you can enforce best practices early in your development cycle.

The following table summarizes the challenges that GKE helps you solve. Although we encourage you to read the whole document, this table presents a map of what's covered.

Challenge	Action
I want to look at easy cost savings on GKE.	Select the appropriate region (#select_the_appropriate_region), sign up for committed-use discounts (#sign_up_for_committed_use_discounts), and use E2 machine types (#e2_machine_types).
I need to understand my GKE costs.	Observe your GKE clusters and watch for recommendations (#observe_your_gke_clusters_and_watch_for_recommendations), and enable GKE usage metering (#enable_gke_usage_metering)
I want to make the most out of GKE elasticity for my existing workloads.	Read Horizontal Pod Autoscaler (#horizontal_pod_autoscaler), Cluster Autoscaler (#cluster_autoscaler), and understand best practices for Autoscaler and over-provisioning (#autoscaler_and_over-provisioning).

Challenge	Action
I want to use the most efficient machine types.	Choose the right machine type (#choose_the_right_machine_type) for your workload.
Many nodes in my cluster are sitting idle.	Read best practices for Cluster Autoscaler (#cluster_autoscaler).
I need to improve cost savings in my batch jobs.	Read best practices for batch workloads (#batch_workloads).
I need to improve cost savings in my serving workloads.	Read best practices for serving workloads (#serving_workloads).
I don't know how to size my Pod resource requests.	Use Vertical Pod Autoscaler (VPA) (#vertical_pod_autoscaler), but pay attention to mixing Horizontal Pod Autoscaler (HPA) and VPA (#mixing_hpa_and_vpa) best practices.
My applications are unstable during autoscaling and maintenance activities.	Prepare cloud-based applications for Kubernetes (#prepare_cloud-based_applications_for_kubernetes), and understand how Metrics Server works and how to monitor it (#understand_how_metrics_server_works_and_how_to_monitor_it).
How do I make my developers pay attention to their applications' resource usage?	Spread the cost-saving culture (#spread_the_cost-saving_culture), consider using GKE Enterprise Policy Controller (#consider_using_anthos_policy_controller), design your CI/CD pipeline to enforce cost savings practices (#design_your_cicd_pipeline_to_enforce_cost-saving_practices), and use Kubernetes resource quotas (#use_kubernetes_resource_quotas).
What else should I consider to further reduce my ecosystem costs?	Review small development clusters (#review_small_development_clusters), review your logging and monitoring strategies (#review_your_logging_and_monitoring_strategies), and review inter-region egress traffic in regional and multi-zonal clusters (#review_inter-region_egress_traffic_in_regional_and_multi-zonal_clusters).

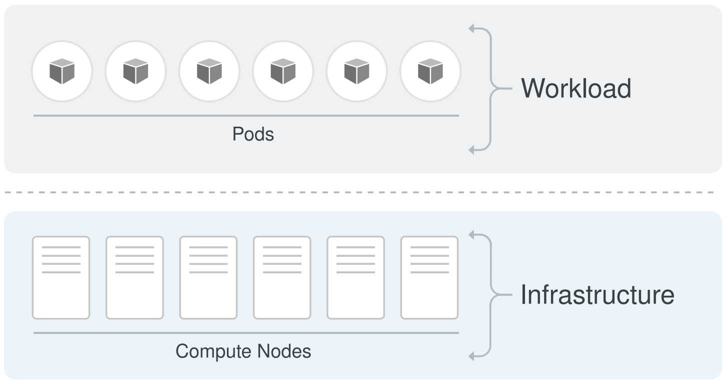
GKE cost-optimization features and options

Cost-optimized Kubernetes applications rely heavily on GKE autoscaling. To balance cost, reliability, and scaling performance on GKE, you must understand how autoscaling works and what options you have. This section discusses GKE autoscaling and other useful cost-optimized configurations for both serving and batch workloads.

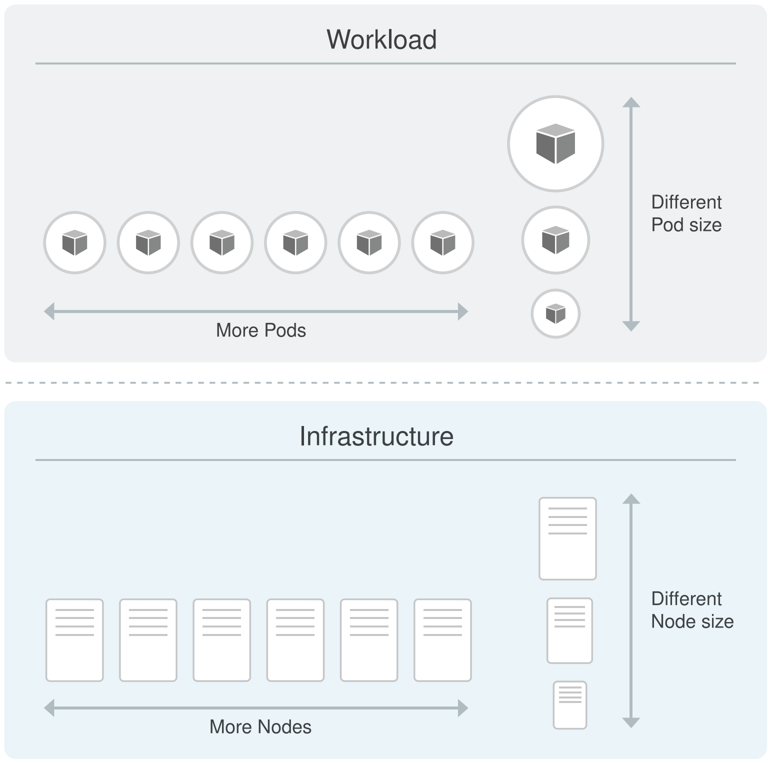
Fine-tune GKE autoscaling

Autoscaling is the strategy GKE uses to let Google Cloud customers pay only for what they need by minimizing infrastructure uptime. In other words, autoscaling saves costs by 1) making workloads and their underlying infrastructure start before demand increases, and 2) shutting them down when demand decreases.

The following diagram illustrates this concept. In Kubernetes, your workloads are containerized applications that are running inside Pods (https://kubernetes.io/docs/concepts/workloads/pods/pod/), and the underlying infrastructure, which is composed of a set of Nodes, must provide enough computing capacity to run the workloads.



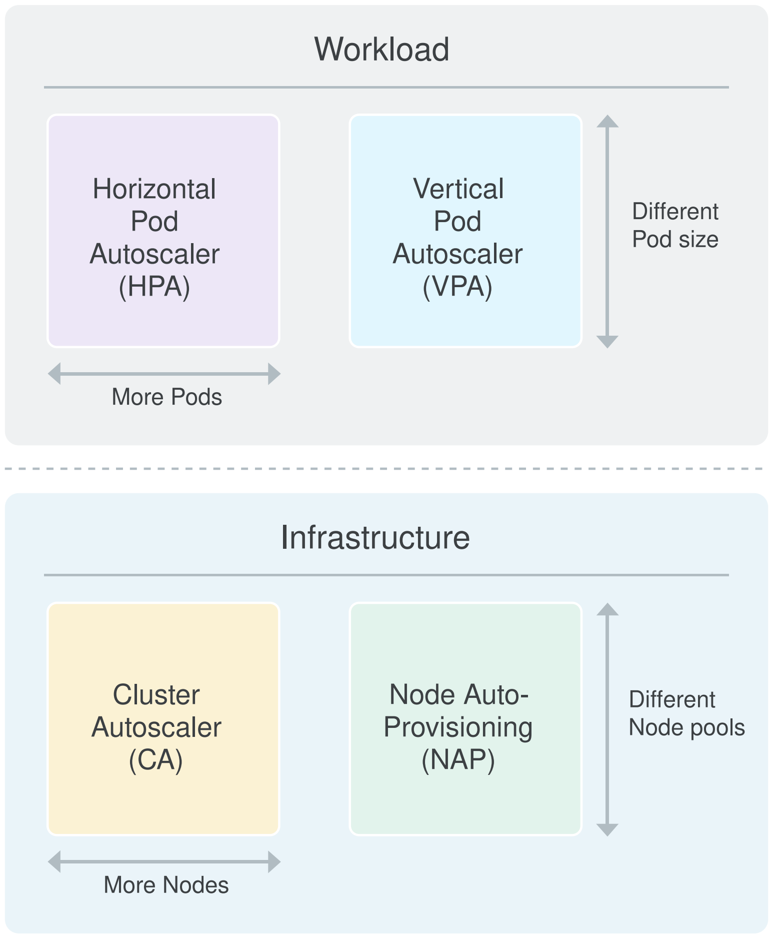
As the following diagram shows, this environment has four scalability dimensions. The workload and infrastructure can scale horizontally by adding and removing Pods or Nodes, and they can scale vertically by increasing and decreasing Pod or Node size.



GKE handles these autoscaling scenarios by using features like the following:

- Horizontal Pod Autoscaler (HPA) (#horizontal_pod_autoscaler), for adding and removing Pods based on utilization metrics.
- Vertical Pod Autoscaler (VPA) (#vertical_pod_autoscaler), for sizing your Pods.
- Cluster Autoscaler (#cluster_autoscaler), for adding and removing Nodes based on the scheduled workload.
- Node auto-provisioning (/kubernetes-engine/docs/how-to/node-auto-provisioning), for dynamically creating new node pools with nodes that match the needs of users' Pods.

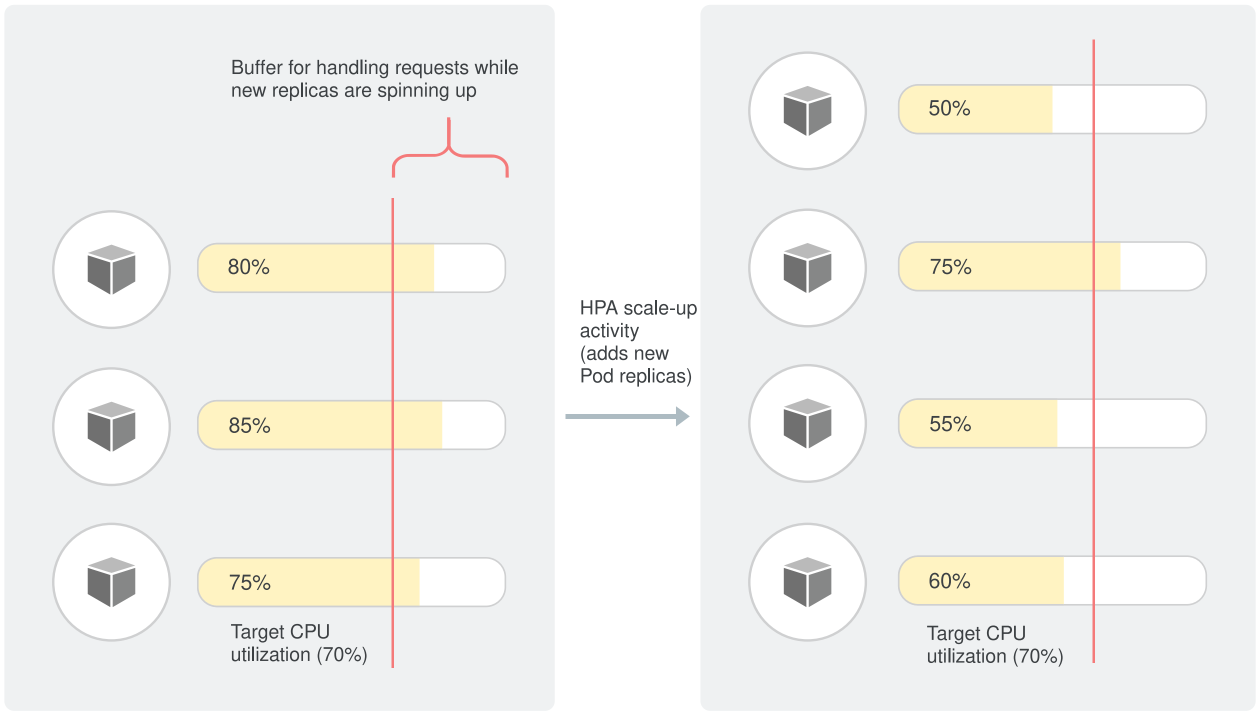
The following diagram illustrates these scenarios.



The remainder of this section discusses these GKE autoscaling capabilities in more detail and covers other useful cost-optimized configurations for both serving and batch workloads.

Horizontal Pod Autoscaler

Horizontal Pod Autoscaler (https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/) (HPA) is meant for scaling applications that are running in Pods based on metrics that express load. You can configure either CPU utilization or other custom metrics (for example, requests per second). In short, HPA adds and deletes Pods replicas, and it is best suited for stateless workers that can spin up quickly (#make_sure_your_container_is_as_lean_as_possible) to react to usage spikes, and shut down gracefully (#make_sure_your_applications_are_shutting_down_in_accordance_with_kubernetes_expectations) to avoid workload instability.


Buffer for handling requests while new replicas are spinning up
80%
85%
75%
Target CPU utilization (70%)
HPA scale-up activity (adds new Pod replicas)
50%
75%
55%
60%
Target CPU utilization (70%)

As the preceding image shows, HPA requires a target utilization threshold, expressed in percentage, which lets you customize when to automatically trigger scaling. In this example, the target CPU utilization is 70%. That means your workload has a 30% CPU buffer for handling requests while new replicas are spinning up. A small buffer prevents early scale-ups, but it can overload your application during spikes. However, a large buffer causes resource waste, increasing your costs. The exact target is application specific, and you must consider the buffer size to be enough for handling requests for two or three minutes during a spike. Even if you guarantee that your application can start up in a matter of seconds, this extra time is required when Cluster Autoscaler (#cluster_autoscaler) adds new nodes to your cluster or when Pods are throttled due to lack of resources.

The following are best practices for enabling HPA in your application:

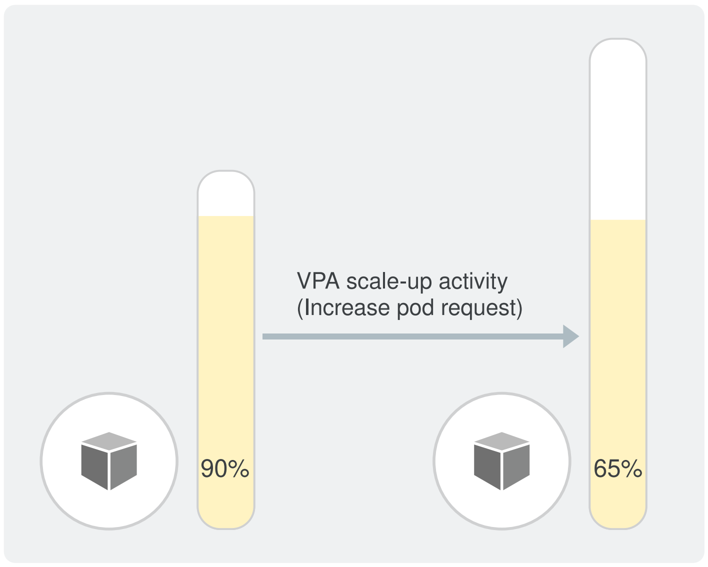
- Size your application correctly by setting appropriate resource requests and limits. (#set_appropriate_resource_requests_and_limits)
- Set your target utilization to reserve a buffer that can handle requests during a spike.
- Make sure your application starts as quickly as possible (#make_sure_your_container_is_as_lean_as_possible) and shuts down according to Kubernetes expectations (#make_sure_your_applications_are_shutting_down_in_accordance_with_kubernetes_expectations).

- Set meaningful readiness and liveness probes (#set_meaningful_readiness_and_liveness_probes_for_your_application).
- Make sure that your Metrics Server is always up and running (#understand_how_metrics_server_works_and_how_to_monitor_it).
- Inform clients of your application that they must consider implementing exponential retries (#consider_using_retries_with_exponential_backoff) for handling transient issues.

For more information, see Configuring a Horizontal Pod Autoscaler (/kubernetes-engine/docs/how-to/horizontal-pod-autoscaling).

Vertical Pod Autoscaler

Unlike HPA, which adds and deletes Pod replicas for rapidly reacting to usage spikes, Vertical Pod Autoscaler (/kubernetes-engine/docs/concepts/verticalpodautoscaler) (VPA) observes Pods over time and gradually finds the optimal CPU and memory resources required by the Pods. Setting the right resources is important for stability and cost efficiency. If your Pod resources are too small, your application can either be throttled or it can fail due to out-of-memory errors. If your resources are too large, you have waste and, therefore, larger bills. VPA is meant for stateless and stateful workloads not handled by HPA or when you don't know the proper Pod resource requests.



As the preceding image shows, VPA detects that the Pod is consistently running at its limits and recreates the Pod with larger resources. The opposite also happens when the Pod is consistently underutilized—a scale-down is triggered.

VPA can work in three different modes:

- **Off**. In this mode, also known as *recommendation mode*, VPA does not apply any change to your Pod. The recommendations are calculated and can be inspected in the VPA object.
- **Initial**: VPA assigns resource requests only at Pod creation and never changes them later.
- **Auto**: VPA updates CPU and memory requests during the life of a Pod. That means, the Pod is deleted, CPU and memory are adjusted, and then a new Pod is started.

If you plan to use VPA, the best practice is to start with the **Off** mode for pulling VPA recommendations. Make sure it's running for 24 hours, ideally one week or more, before pulling recommendations. Then, only when you feel confident, consider switching to either **Initial** or **Auto** mode.

Follow these best practices for enabling VPA, either in **Initial** or **Auto** mode, in your application:

- Don't use VPA either **Initial** or **Auto** mode if you need to handle sudden spikes in traffic. Use HPA (#horizontal_pod_autoscaler) instead.
- Make sure your application can grow vertically (#make_sure_your_application_can_grow_vertically_and_horizontally).
- Set minimum and maximum container sizes in the VPA objects to avoid the autoscaler making significant changes when your application is not receiving traffic.
- Don't make abrupt changes, such as dropping the Pod's replicas from 30 to 5 all at once. This kind of change requires a new deployment, new label set, and new VPA object.
- Make sure your application starts as quickly as possible (#make_sure_your_container_is_as_lean_as_possible) and shuts down according to Kubernetes expectations (#make_sure_your_applications_are_shutting_down_in_accordance_with_kubernetes_expectations).
- Set meaningful readiness and liveness probes (#set_meaningful_readiness_and_liveness_probes_for_your_application).
- Make sure that your Metrics Server is always up and running (#understand_how_metrics_server_works_and_how_to_monitor_it).
- Inform clients of your application that they must consider implementing exponential retries (#consider_using_retries_with_exponential_backoff) for handling transient issues.
- Consider using node auto-provisioning along with VPA (/kubernetes-engine/docs/concepts/verticalpodautoscaler#vertical_pod_autoscaling_in_auto_mode) so that if a Pod gets large enough to fit into existing machine types, Cluster Autoscaler provisions larger machines to fit the new Pod.

Whether you are considering using **Auto** mode, make sure you also follow these practices:

- Make sure your application can be restarted while receiving traffic.
- Add Pod Disruption Budget (PDB) (#add-pod_disruption_budget-to-your-application) to control how many Pods can be taken down at the same time.

For more information, see Configuring Vertical Pod Autoscaling (/kubernetes-engine/docs/how-to/vertical-pod-autoscaling).

Mixing HPA and VPA

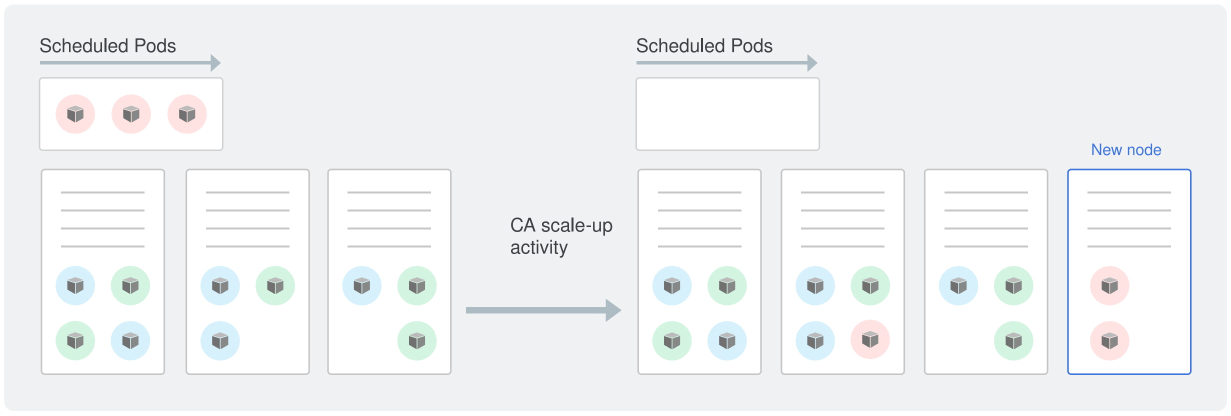
The official recommendation (/kubernetes-engine/docs/concepts/verticalpodautoscaler#limitations_for_vertical_pod_autoscaling) is that you must not mix VPA and HPA on either CPU or memory. However, you can mix them safely when using recommendation mode in VPA or custom metrics in HPA—for example, requests per second. When mixing VPA with HPA, make sure your deployments are receiving enough traffic—meaning, they are consistently running above the HPA min-replicas. This lets VPA understand your Pod's resource needs.

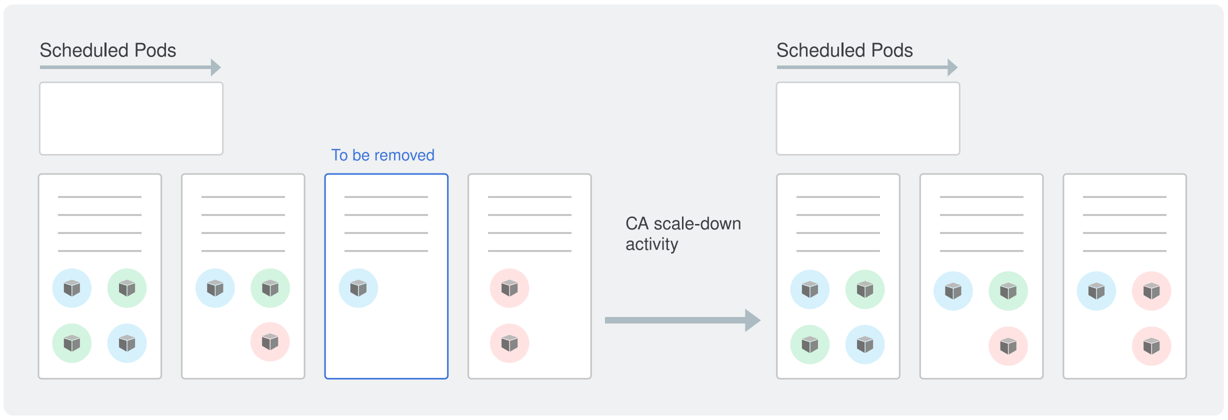
For more information about VPA limitations, see Limitations for Vertical Pod autoscaling (/kubernetes-engine/docs/concepts/verticalpodautoscaler).

Cluster Autoscaler

Cluster Autoscaler (/kubernetes-engine/docs/concepts/cluster-autoscaler) (CA) automatically resizes the underlying computer infrastructure. CA provides nodes for Pods that don't have a place to run in the cluster and removes under-utilized nodes. CA is optimized for the cost of infrastructure. In other words, if there are two or more node types in the cluster, CA chooses the least expensive one that fits the given demand.

Unlike HPA and VPA, CA doesn't depend on load metrics. Instead, it's based on scheduling simulation and declared Pod requests. It's a best practice to enable CA whenever you are using either HPA or VPA. This practice ensures that if your Pod autoscalers determine that you need more capacity, your underlying infrastructure grows accordingly.





As these diagrams show, CA automatically adds and removes compute capacity to handle traffic spikes and save you money when your customers are sleeping. It is a best practice to define Pod Disruption Budget (#add-pod_disruption_budget-to-your-application) (PDB) for all your applications. It is particularly important at the CA scale-down phase when PDB controls the number of replicas that can be taken down at one time.

Certain Pods cannot be restarted by any autoscaler (https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md#what-types-of-pods-can-prevent-ca-from-removing-a-node) when they cause some temporary disruption, so the node they run on can't be deleted. For example, system Pods (such as `metrics-server` and `kube-dns`), and Pods using local storage won't be restarted. However, you can change this behavior by defining PDBs (https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md#how-to-set-pdbs-to-enable-ca-to-move-kube-system-pods) for these system Pods and by setting `"cluster-autoscaler.kubernetes.io/safe-to-evict": "true"` (https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md#what-types-of-pods-can-prevent-ca-from-removing-a-node) annotation for Pods using local storage that are safe for the autoscaler to restart. Moreover, consider running long-lived Pods that can't be restarted on a separate node pool (/kubernetes-engine/docs/concepts/node-pools), so they don't block scale-down of other nodes. Finally, learn how to analyze CA events in the logs (/kubernetes-engine/docs/how-to/cluster-autoscaler-visibility) to understand why a particular scaling activity didn't happen as expected.

If your workloads are resilient to nodes restarting inadvertently and to capacity losses (#prepare_your_environment_to_fit_your_workload_type), you can save more money by creating a cluster or node pool with preemptible VMs (/kubernetes-engine/docs/how-to/preemptible-vms#create). For CA to work as expected, Pod resource requests need to be large enough for the Pod to function normally (#set_appropriate_resource_requests_and_limits). If resource requests are too small, nodes might not have enough resources and your Pods might crash or have troubles during runtime.

The following is a summary of the best practices for enabling Cluster Autoscaler in your cluster:

- Use either HPA or VPA to autoscale your workloads.
- Make sure you are following the best practices described in the chosen Pod autoscaler.
- Size your application correctly by setting appropriate resource requests and limits (#set_appropriate_resource_requests_and_limits) or use VPA.
- Define a PDB for your applications.
- Define PDB for system Pods that might block your scale-down. For example, `kube-dns`. To avoid temporary disruption in your cluster, don't set PDB for system Pods that have only 1 replica (such as `metrics-server`).
- Run short-lived Pods and Pods that can be restarted in separate node pools, so that long-lived Pods don't block their scale-down.
- Avoid over-provisioning by configuring idle nodes in your cluster. For that, you must know your minimum capacity—for many companies it's during the night—and set the minimum number of nodes in your node pools to support that capacity.
- If you need extra capacity to handle requests during spikes, use pause Pods, which are discussed in Autoscaler and over-provisioning (#autoscaler_and_over-provisioning).

For more information, see Autoscaling a cluster (/kubernetes-engine/docs/how-to/cluster-autoscaler).

Node auto-provisioning

Node auto-provisioning (/kubernetes-engine/docs/how-to/node-auto-provisioning) (NAP) is a mechanism of Cluster Autoscaler that automatically adds new node pools (/kubernetes-engine/docs/concepts/node-pools) in addition to managing their size on the user's behalf.

Without node auto-provisioning, GKE considers starting new nodes only from the set of user-created node pools. With node auto-provisioning, GKE can create and delete new node pools automatically.

Node auto-provisioning tends to reduce resource waste by dynamically creating node pools that best fit with the scheduled workloads. However, the autoscale latency can be slightly higher when new node pools need to be created. If your workloads are resilient to nodes restarting inadvertently and to capacity losses (#prepare_your_environment_to_fit_your_workload_type), you can further lower costs by configuring a preemptible VM's toleration in your Pod (/kubernetes-engine/docs/how-to/node-auto-provisioning#support_for_preemptible_vms).

The following are best practices for enabling node auto-provisioning:

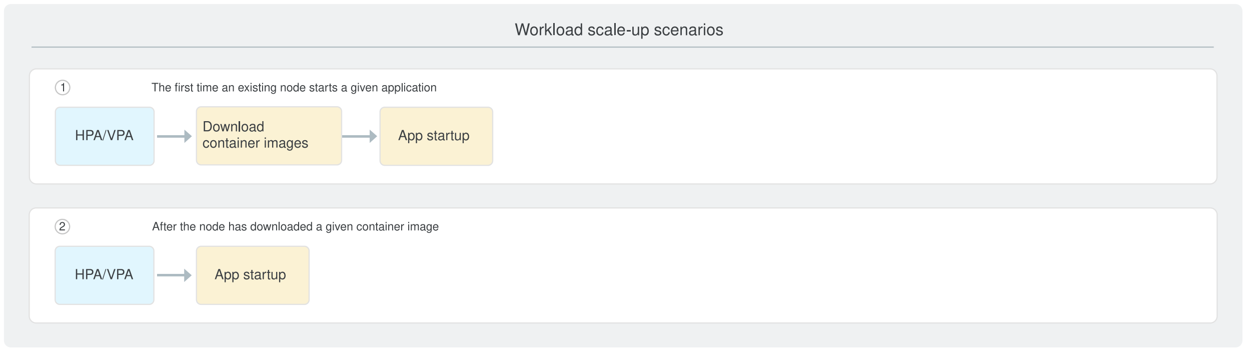
- Follow all the best practice of Cluster Autoscaler.
- Set minimum and maximum resources sizes to avoid NAP making significant changes in your cluster when your application is not receiving traffic.
- When using Horizontal Pod Autoscaler for serving workloads, consider reserving a slightly larger target utilization buffer because NAP might increase autoscaling latency in some cases.

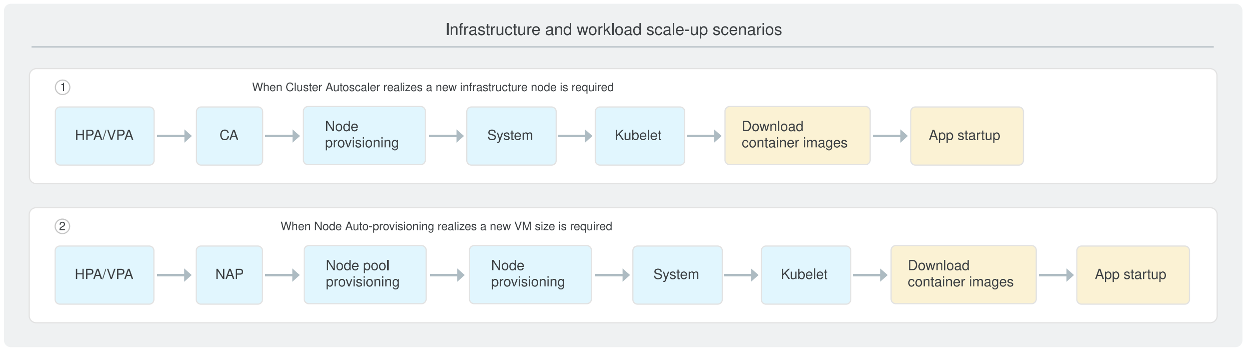
For more information, see Using node auto-provisioning (/kubernetes-engine/docs/how-to/node-auto-provisioning) and Unsupported features (/kubernetes-engine/docs/how-to/node-auto-provisioning#unsupported_features).

Autoscaler and over-provisioning

In order to control your costs, we strongly recommend that you enable autoscaler according to the previous sections. No one configuration fits all possible scenarios, so you must fine-tune the settings for your workload to ensure that autoscalers respond correctly to increases in traffic.

However, as noted in the Horizontal Pod Autoscaler (#horizontal_pod_autoscaler) section, scale-ups might take some time due to infrastructure provisioning. To visualize this difference in time and possible scale-up scenarios, consider the following image.


Workload scale-up scenarios
1 The first time an existing node starts a given application
HPA/VPA
Download container images
App startup
2 After the node has downloaded a given container image
HPA/VPA
App startup


Infrastructure and workload scale-up scenarios
1 When Cluster Autoscaler realizes a new infrastructure node is required
HPA/VPA
CA
Node provisioning
System
Kubelet
Download container images
App startup
2 When Node Auto-provisioning realizes a new VM size is required
HPA/VPA
NAP
Node pool provisioning
Node provisioning
System
Kubelet
Download container images
App startup

When your cluster has enough room for deploying new Pods, one of the **Workload scale-up scenarios** is triggered. Meaning, if an existing node never deployed your application, it must download its container images before starting the Pod (scenario 1). However, if the same node must start a new Pod replica of your application, the total scale-up time decreases because no image download is required (scenario 2).

When your cluster doesn't have enough room for deploying new Pods, one of the **Infrastructure and Workload scale-up scenarios** is triggered. This means that Cluster Autoscaler must provision new nodes and start the required software before approaching your application (scenario 1). If you use node auto-provisioning, depending on the workload scheduled, new node pools might be required. In this situation, the total scale-up time increases because Cluster Autoscaler has to provision nodes and node pools (scenario 2).

For scenarios where new infrastructure is required, don't squeeze your cluster too much—meaning, you must over-provision but only for reserving the necessary buffer to handle the expected peak requests during scale-ups.

There are two main strategies for this kind of over-provisioning:

- **Fine-tune the HPA utilization target**. The following equation is a simple and safe way to find a good CPU target:

  (1 - *buff*)/(1 + *perc*)

- *buff* is a safety buffer that you can set to avoid reaching 100% CPU. This variable is useful because reaching 100% CPU means that the latency of request processing is much higher than usual.
- *perc* is the percentage of traffic growth you expect in two or three minutes.

For example, if you expect a growth of 30% in your requests and you want to avoid reaching 100% of CPU by defining a 10% safety buffer, your formula would look like this:

(1 - **0.1**)/(1 + **0.3**) = 0.69

★ **Note:** If you have large nodes and small applications whose traffic is consistently below what is requested, then you can get more aggressive (by reducing or removing the *buff* variable). You can be more aggressive because there are likely some free CPU cycles on the machine, so in some cases, it's safe for Pods to pass 100% utilization.

- **Configure pause Pods**. There is no way to configure Cluster Autoscaler to spin up nodes upfront. Instead, you can set an HPA utilization target to provide a buffer to help handle spikes in load. However, if you expect large bursts, setting a small HPA utilization target might not be enough or might become too expensive.

  An alternative solution for this problem is to use pause Pods (https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md#how-can-i-configure-overprovisioning-with-cluster-autoscaler)
  . Pause Pods are low-priority deployments that do nothing but reserve room in your cluster. Whenever a high-priority Pod is scheduled, pause Pods get evicted and the high-priority Pod immediately takes their place. The evicted pause Pods are then rescheduled, and if there is no room in the cluster, Cluster Autoscaler spins up new nodes for fitting them. It's a best practice to have only a single pause Pod per node. For example, if you are using 4 CPU nodes, configure the pause Pods' CPU request with around 3200m.

Choose the right machine type

Beyond autoscaling, other configurations can help you run cost-optimized kubernetes applications on GKE. This section discusses choosing the right machine type.

Preemptible VMs

Preemptible VMs (/kubernetes-engine/docs/how-to/preemptible-vms) (PVMs) are Compute Engine VM instances that last a maximum of 24 hours and provide no availability guarantees. PVMs are up to 80% cheaper (/compute/all-pricing) than standard Compute Engine VMs, but we recommend that you use them with caution on GKE clusters. PVMs on GKE are best suited for running batch or fault-tolerant jobs that are less sensitive to the ephemeral, non-guaranteed nature of PVMs. Stateful and serving workloads must not use PVMs unless you prepare your system and architecture to handle PVMs' constraints.

Whatever the workload type, you must pay attention to the following constraints:

- Pod Disruption Budget might not be respected because preemptible nodes can shut down inadvertently.
- There is no guarantee that your Pods will shut down gracefully (#make_sure_your_applications_are_shutting_down_in_accordance_with_kubernetes_expectations) once node preemption ignores the Pod grace period (https://kubernetes.io/docs/concepts/workloads/pods/pod/#termination-of-pods).
- It might take several minutes for GKE to detect that the node was preempted and that the Pods are no longer running, which delays rescheduling the Pods to a new node.

In order to mitigate these constraints, you can deploy in your cluster a community Node Termination Event Handler (https://github.com/GoogleCloudPlatform/k8s-node-termination-handler) project (important: this is not an official Google project) that provides an adapter for translating Compute Engine node termination events to graceful Pod terminations in Kubernetes. This community project does not reliably solve all the PVMs' constraints once Pod Disruption Budgets can still be disrespected. Therefore, pods can take a little longer to be rescheduled.

Finally, PVMs have no guaranteed availability, meaning that they can stock out easily in some regions. To overcome this limitation, we recommend that you set a backup node pool without PVMs. Cluster Autoscaler gives preference to PVMs because it is optimized for infrastructure cost.

For more information, see Running preemptible VMs on GKE (/kubernetes-engine/docs/how-to/preemptible-vms) and Run web applications on GKE using cost-optimized Spot VMs (/kubernetes-engine/docs/archive/run-web-applications-on-gke-using-cost-optimized-spot-vms-and-traffic-director)
.

E2 machine types

E2 machine types (/blog/products/compute/google-compute-engine-gets-new-e2-vm-machine-types) (E2 VMs) are cost-optimized VMs that offer you 31% savings compared to N1 machine types (/compute/docs/machine-types#n1_machine_types). E2 VMs are suitable for a broad range of workloads, including web servers, microservices, business-critical applications, small-to-medium sized databases, and development environments.

For more information about E2 VMs and how they compare with other Google Cloud machine types, see Performance-driven dynamic resource management in E2 VMs (/blog/products/compute/understanding-dynamic-resource-management-in-e2-vms) and Machine types (/compute/docs/machine-types).

Select the appropriate region

When cost is a constraint, where you run your GKE clusters matters. Due to many factors, cost varies per computing region (/compute/all-pricing). So make sure you are running your workload in the least expensive option but where latency doesn't affect your customer. If your workload requires copying data from one region to another—for example, to run a batch job—you must also consider the cost of moving this data.

For more information on how to choose the right region, see Best practices for Compute Engine regions selection (/solutions/best-practices-compute-engine-region-selection).

Sign up for committed-use discounts

If you intend to stay with Google Cloud for a few years, we strongly recommend that you purchase committed-use discounts (/compute/docs/instances/signing-up-committed-use-discounts) in return for deeply discounted prices for VM usage. When you sign a committed-use contract, you purchase compute resources at a discounted price (up to 70% discount) in return for committing to paying for those resources for one year or three years. If you are unsure about how much resource to commit, look at your minimum computing usage—for example, during nighttime—and commit the payment for that amount.

For more information about committed-use prices for different machine types, see VM instances pricing (/compute/all-pricing).

Review small development clusters

For small development clusters where you need to minimize costs, consider using Autopilot (/kubernetes-engine/docs/concepts/autopilot-overview) clusters. With clusters in this mode of operation, you aren't charged for system Pods, operating system costs, or unscheduled workloads.

Review your logging and monitoring strategies

If you use Cloud Logging (/logging?hl=pt-br) and Cloud Monitoring (/monitoring) to provide observability into your applications and infrastructure, you are paying only for what you use. However, the more your infrastructure and applications log, and the longer you keep those logs, the more you pay for them. Similarly, the more external and custom metrics you have, the higher your costs. Review your logging and monitoring strategies according to Cost optimization for Cloud Logging, Cloud Monitoring, and Application Performance Management (/solutions/stackdriver-cost-optimization).

Review inter-region egress traffic in regional and multi-zonal clusters

The types of available GKE clusters are single-zone (/kubernetes-engine/docs/concepts/types-of-clusters?hl=fa#single-zone_clusters), multi-zonal (/kubernetes-engine/docs/concepts/types-of-clusters?hl=fa#multi-zonal_clusters), and regional (/kubernetes-engine/docs/concepts/types-of-clusters?hl=fa#regional_clusters). Because of the high availability of nodes across zones, regional and multi-zonal clusters are well suited for production environments. However, you are charged by the egress traffic between zones (/vpc/network-pricing#general). For production environments, we recommend that you monitor the traffic load across zones and improve your APIs to minimize it. Also consider using inter-pod affinity and anti-affinity (https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#inter-pod-affinity-and-anti-affinity) configurations to colocate dependent Pods from different services in the same nodes or in the same availability zone to minimize costs and network latency between them. The suggested way to monitor this traffic is to enable GKE usage metering (#enable_gke_usage_metering) and its network egress agent (/kubernetes-engine/docs/how-to/cluster-usage-metering#enable-network-egress-metering), which is disabled by default.

For non-production environments, the best practice for cost saving is to deploy single-zone clusters.

Prepare your environment to fit your workload type

Enterprises have different cost and availability requirements. Their workloads can be divided into serving workloads, which must respond quickly to bursts or spikes, and batch workloads, which are concerned with eventual work to be done. Serving workloads require a small scale-up latency; batch workloads are more tolerant to latency. The different expectations for these workload types make choosing different cost-saving methods more flexible.

Batch workloads

Because batch workloads are concerned with eventual work, they allow for cost saving on GKE because the workloads are commonly tolerant to some latency at job startup time. This tolerance gives Cluster Autoscaler space to spin up new nodes only when jobs are scheduled and take them down when the jobs are finished.

The first recommended practice is to separate batch workloads in different node pools (/kubernetes-engine/docs/concepts/node-pools) by using labels and selectors (https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/), and by using taints and tolerations (https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/). The rationale is the following:

- Cluster Autoscaler can delete empty nodes faster when it doesn't need to restart pods. As batch jobs finish, the cluster speeds up the scale-down process if the workload is running on dedicated nodes that are now empty. To further improve the speed of scale-downs, consider configuring CA's optimize-utilization profile (/kubernetes-engine/docs/concepts/cluster-autoscaler#autoscaling_profiles).
- Some Pods cannot be restarted, so they permanently block the scale-down of their nodes. These Pods, which include the system Pods, must run on different node pools so that they don't affect scale-down.

The second recommended practice is to use node auto-provisioning (#node-auto-provisioning) to automatically create dedicated node pools for jobs with a matching taint or toleration. This way, you can separate many different workloads without having to set up all those different node pools.

We recommend that you use preemptible VMs only if you run fault-tolerant jobs that are less sensitive to the ephemeral, non-guaranteed nature of preemptible VMs.

For more information about how to set up an environment that follows these practices, see the Optimizing resource usage in a multi-tenant GKE cluster using node auto-provisioning (/solutions/optimizing-resources-in-multi-tenant-gke-clusters-with-auto-provisioning) tutorial.

Serving workloads

Unlike batch workloads, serving workloads must respond as quickly as possible to bursts or spikes. These sudden increases in traffic might result from many factors, for example, TV commercials, peak-scale events like Black Friday, or breaking news. Your application must be prepared to handle them.

Problems in handling such spikes are commonly related to one or more of the following reasons:

- Applications not being ready to run on Kubernetes—for example, apps with large image sizes, slow startup times, or non-optimal Kubernetes configurations.
- Applications depending on infrastructure that takes time to be provisioned, like GPUs.
- Autoscalers and over-provisioning (#autoscaler_and_over-provisioning) not being appropriately set.

Prepare cloud-based Kubernetes applications

Some of the best practices in this section can save money by themselves. However, because most of these practices are intended to make your application work reliably with autoscalers, we strongly recommend that you implement them.

Understand your application capacity

When you plan for application capacity, know how many concurrent requests your application can handle, how much CPU and memory it requires, and how it responds under heavy load. Most teams don't know these capacities, so we recommend that you test how your application behaves under pressure. Try isolating a single application Pod replica with autoscaling off, and then execute the tests simulating a real usage load. This helps you understand your per-Pod capacity. We then recommend configuring your Cluster Autoscaler, resource requests and limits, and either HPA or VPA. Then stress your application again, but with more strength to simulate sudden bursts or spikes.

Ideally, to eliminate latency concerns, these tests must run from the same region or zone that the application is running on Google Cloud. You can use the tool of your choice for these tests, whether it's a homemade script or a more advanced performance tool, like Apache Benchmark (https://httpd.apache.org/docs/2.4/programs/ab.html), JMeter (https://jmeter.apache.org/), or Locust (https://locust.io/).

For an example of how you can perform your tests, see Distributed load testing using Google Kubernetes Engine (/solutions/distributed-load-testing-using-gke).

Make sure your application can grow vertically and horizontally

Ensure that your application can grow and shrink. This means you can choose to handle traffic increases either by adding more CPU and memory or adding more Pod replicas. This gives you the flexibility to experiment what fits your application better, whether that's a different autoscaler setup or a different node size. Unfortunately, some applications are single threaded or limited by a fixed number of workers or subprocesses that make this experiment impossible without a complete refactoring of their architecture.

Set appropriate resource requests and limits

By understanding your application capacity, you can determine what to configure in your container resources. Resources (https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/) in Kubernetes are mainly defined as CPU and memory (RAM). You configure CPU or memory as the amount required to run your application by using the request

spec.containers[].resources.requests.<cpu|memory>, and you configure the cap by using the request **spec.containers[].resources.limits.<cpu|memory>**.

When you've correctly set resource requests, Kubernetes scheduler can use them to decide which node to place your Pod on. This guarantees that Pods are being placed in nodes that can make them function normally, so you experience better stability and reduced resource waste. Moreover, defining resource limits helps ensure that these applications never use all available underlying infrastructure provided by computing nodes.

A good practice for setting your container resources is to use the same amount of memory for requests and limits, and a larger or unbounded CPU limit. Take the following deployment as an example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wordpress
spec:
  replicas: 1
  selector:
    matchLabels:
      app: wp
  template:
    metadata:
      labels:
        app: wp
    spec:
      containers:
  - name: wp
    image: wordpress
    resources:
      requests:
        memory: "128Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
```

The reasoning for the preceding pattern is founded on how Kubernetes out-of-resource handling (https://kubernetes.io/docs/tasks/administer-cluster/out-of-resource/) works. Briefly, when computer resources are exhausted, nodes become unstable. To avoid this situation, **kubelet** (https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/) monitors and prevents total starvation of these resources by ranking the resource-hungry Pods (https://github.com/kubernetes/design-proposals-archive/blob/main/node/resource-qos.md#qos-classes).

When the CPU is contended, these Pods can be throttled down to its requests. However, because memory is an incompressible resource, when memory is exhausted, the Pod needs to be taken down. To avoid having Pods taken down—and consequently, destabilizing your environment—you must set requested memory to the memory limit.

You can also use VPA in recommendation mode to help you determine CPU and memory usage for a given application. Because VPA provides such recommendations based on your application usage, we recommend that you enable it in a production-like environment to face real traffic. VPA status then generates a report with the suggested resource requests and limits, which you can statically specify in your deployment manifest. If your application already defines HPA, see Mixing HPA and VPA (#mixing_hpa_and_vpa).

Make sure your container is as lean as possible

When you run applications in containers, it's important to follow some practices for building those containers (/solutions/best-practices-for-building-containers#build-the-smallest-image-possible). When running those containers on Kubernetes, some of these practices are even more important because your application can start and stop at any moment. This section focuses mainly on the following two practices:

- **Have the smallest image possible**. It's a best practice to have small images because every time Cluster Autoscaler provisions a new node for your cluster, the node must download the images that will run in that node. The smaller the image, the faster the node can download it.
- **Start the application as quickly as possible**. Some applications can take minutes to start because of class loading, caching, and so on. When a Pod requires a long startup, your customers' requests might fail while your application is booting.

Consider these two practices when designing your system, especially if you are expecting bursts or spikes. Having a small image and a fast startup helps you reduce scale-ups latency (#autoscaler_and_over-provisioning). Consequently, you can better handle traffic increases without worrying too much about instability. These practices work better with the autoscaling best practices discussed in GKE autoscaling (#fine-tune_gke_autoscaling).

For more information about how to build containers, see Best practices for building containers (/solutions/best-practices-for-building-containers#build-the-smallest-image-possible).

Add Pod Disruption Budget to your application

Pod Disruption Budget (https://kubernetes.io/docs/concepts/workloads/pods/disruptions/) (PDB) limits the number of Pods that can be taken down simultaneously from a voluntary disruption

(https://kubernetes.io/docs/concepts/workloads/pods/disruptions/#voluntary-and-involuntary-disruptions).
That means the defined disruption budget is respected at rollouts, node upgrades, and at any autoscaling activities. However, this budget can not be guaranteed when involuntary things happen, such as hardware failure, kernel panic, or someone deleting a VM by mistake.

When PDB is respected during the Cluster Autoscaler compacting phase, it's a best practice to define a Pod Disruption Budget for every application. This way you can control the minimum number of replicas required to support your load at any given time, including when CA is scaling down your cluster.

For more information, see Specifying a Disruption Budget for your Application (https://kubernetes.io/docs/tasks/run-application/configure-pdb/).

Set meaningful readiness and liveness probes for your application

Setting meaningful probes ensures your application receives traffic only when it is up and running and ready to accept traffic. GKE uses readiness probes (https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/#define-readiness-probes)
to determine when to add Pods to or remove Pods from load balancers. GKE uses liveness probes (https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/#define-a-liveness-command)
to determine when to restart your Pods.

The liveness probe is useful for telling Kubernetes that a given Pod is unable to make progress, for example, when a deadlock state is detected. The readiness probe is useful for telling Kubernetes that your application isn't ready to receive traffic, for example, while loading large cache data at startup.

To ensure the correct lifecycle of your application during scale-up activities, it's important to do the following:

- Define the readiness probe for all your containers.
- If your application depends on a cache to be loaded at startup, the readiness probe must say it's ready only after the cache is fully loaded.
- If your application can start serving right away, a good default probe implementation can be as simple as possible, for example, an HTTP endpoint returning a 200 status code.
- If you implement a more advanced probe, such as checking if the connection pool has available resources, make sure your error rate doesn't increase (#set_appropriate_resource_requests_and_limits) as compared to a simpler implementation.

- Never make any probe logic access other services. It can compromise the lifecycle of your Pod if these services don't respond promptly.

For more information, see Configure Liveness, Readiness and Startup Probes (https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/).

Make sure your applications are shutting down according to Kubernetes expectations

Autoscalers help you respond to spikes by spinning up new Pods and nodes, and by deleting them when the spikes finish. That means that to avoid errors while serving your Pods must be prepared for either a fast startup or a graceful shutdown.

Because Kubernetes asynchronously updates endpoints and load balancers, it's important to follow these best practices in order to ensure non-disruptive shutdowns:

- **Don't stop accepting new requests right after `SIGTERM`**. Your application must not stop immediately, but instead finish all requests that are in flight and still listen to incoming connections that arrive after the Pod termination begins. It might take a while for Kubernetes to update all kube-proxies (https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/) and load balancers. If your application terminates before these are updated, some requests might cause errors on the client side.
- **If your application doesn't follow the preceding practice, use the `preStop` hook** (https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/#hook-details). Most programs don't stop accepting requests right away. However, if you're using third-party code or are managing a system that you don't have control over, such as nginx (https://www.nginx.com/), the `preStop` hook is a good option for triggering a graceful shutdown without modifying the application. One common strategy is to execute, in the `preStop` hook, a sleep of a few seconds to postpone the `SIGTERM`. This gives Kubernetes extra time to finish the Pod deletion process, and reduces connection errors on the client side.
- **Handle SIGTERM for cleanups**. If your application must clean up or has an in-memory state that must be persisted before the process terminates, now is the time to do it. Different programming languages have different ways to catch this signal, so find the right way in your language.
- **Configure `terminationGracePeriodSeconds`** (https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes) **to fit your application needs**. Some applications need more than the default 30 seconds to finish. In this case, you must specify `terminationGracePeriodSeconds`. High values might

increase time for node upgrades or rollouts, for example. Low values might not allow enough time for Kubernetes to finish the Pod termination process. Either way, we recommend that you set your application's termination period to less than 10 minutes because Cluster Autoscaler honors it for 10 minutes only (/kubernetes-engine/docs/concepts/cluster-autoscaler#limitations).

- **If your application uses container-native load balancing** (#use_container-native_load_balancing_through_ingress)**, start failing your readiness probe when you receive a SIGTERM**. This action directly signals load balancers to stop forwarding new requests to the backend Pod. Depending on the race between health check configuration and endpoint programming, the backend Pod might be taken out of traffic earlier.

For more information, see Kubernetes best practices: terminating with grace (/blog/products/gcp/kubernetes-best-practices-terminating-with-grace).

Set up NodeLocal DNSCache

The GKE-managed DNS is implemented by `kube-dns` (/kubernetes-engine/docs/concepts/service-discovery), an add-on deployed in all GKE clusters. When you run DNS-hungry applications, the default `kube-dns-autoscaler` configuration, which adjusts the number of `kube-dns` replicas based on the number of nodes and cores in the cluster, might not be enough. In this scenario, DNS queries can either slow down or time out. To mitigate this problem, companies are accustomed to tuning the `kube-dns-autoscaler` ConfigMap (https://kubernetes.io/docs/tasks/administer-cluster/dns-horizontal-autoscaling/#tuning-autoscaling-parameters)
to increase the number of `kube-dns` replicas in their clusters. Although this strategy might work as expected, it increases the resource usage, and the total GKE cost.

Another cost-optimized and more scalable alternative is to configure the NodeLocal DNSCache (/kubernetes-engine/docs/how-to/nodelocal-dns-cache) in your cluster. NodeLocal DNSCache is an optional GKE add-on that improves DNS lookup latency, makes DNS lookup times more consistent, and reduces the number of DNS queries to `kube-dns` by running a DNS cache on each cluster node.

For more information, see Setting up NodeLocal DNSCache (/kubernetes-engine/docs/how-to/nodelocal-dns-cache).

Use container-native load balancing through Ingress

Container-native load balancing (/kubernetes-engine/docs/concepts/container-native-load-balancing) lets load balancers target Kubernetes Pods directly and to evenly distribute traffic to Pods by using a data model called network endpoint groups (NEGs) (/load-balancing/docs/negs). This approach improves network performance, increases visibility, enables advanced load-balancing features, and enables the use of Traffic Director (/traffic-director/docs), Google Cloud's fully managed traffic control plane for service mesh.

Because of these benefits, container-native load balancing is the recommended solution for load balancing through Ingress (/kubernetes-engine/docs/how-to/container-native-load-balancing). When NEGs are used with GKE Ingress, the Ingress controller facilitates the creation of all aspects of the L7 load balancer. This includes creating the virtual IP address, forwarding rules, health checks, firewall rules, and more.

Container-native load balancing becomes even more important when using Cluster Autoscaler. For non-NEG load balancers, during scale downs, load-balancing programming, and connection draining might not be fully completed before Cluster Autoscaler terminates the node instances. This might disrupt ongoing connections flowing through the node even when the backend Pods are not on the node.

Container-native load balancing is enabled by default for Services when all of the following conditions are true:

- The Services were created in GKE clusters 1.17.6-gke.7 and higher.
- If you are using VPC-native clusters.
- If you are not using a Shared VPC.
- If you are not using GKE Network Policy.

For more information, see Ingress GKE documentation (/kubernetes-engine/docs/concepts/ingress#container-native_load_balancing) and Using container-native load balancing (/kubernetes-engine/docs/how-to/container-native-load-balancing).

Consider using retries with exponential backoff

In microservices architectures running on Kubernetes, transient failures might occur for various reasons—for example:

- A large spike that triggered a still-working scale-up
- Network failures
- Connections dropped due to Pods not shutting down (#make_sure_your_applications_are_shutting_down_in_accordance_with_kubernetes_expectations)
- Preemptible VMs shutting down inadvertently (#prepare_your_environment_to_fit_your_workload_type)

- Applications reaching their rating limits

These issues are ephemeral, and you can mitigate them by calling the service again after a delay. However, to prevent overwhelming the destination service with requests, it's important that you execute these calls using an exponential backoff.

To facilitate such a retry pattern, many existing libraries implement the exponential retrial logic. You can use your library of choice or write your own code. If you use Istio (https://istio.io/latest/docs/concepts/what-is-istio/) or Anthos Service Mesh (/anthos/service-mesh) (ASM), you can opt for the proxy-level retry (https://istio.io/latest/docs/concepts/traffic-management/#retries) mechanism, which transparently executes retries on your behalf.

It's important to plan for your application to support service call retries, for example, to avoid inserting already-inserted information. Consider that a chain of retries might impact the latency of your final user, which might time-out if not correctly planned.

Monitor your environment and enforce cost-optimized configurati and practices

In many medium and large enterprises, a centralized platform and infrastructure team is often responsible for creating, maintaining, and monitoring Kubernetes clusters for the entire company. This represents a strong need for having resource usage accountability and for making sure all teams are following the company's policies. This section addresses options for monitoring and enforcing cost-related practices.

Observe your GKE clusters and watch for recommendations

You can check the resource utilization in a Kubernetes cluster by examining the containers, Pods, and services, and the characteristics of the overall cluster. There are many ways you can perform this task, but the initial approach we recommend is observing your GKE clusters through the Monitoring Dashboard (/stackdriver/docs/solutions/kubernetes-engine/observing). This gives you time-series data of how your cluster is being used, letting you aggregate and span from infrastructure, workloads, and services.

Although this is a good starting point, Google Cloud provides other options—for example:

- In the Google Cloud console, on the **GKE Clusters** page, look at the **Notifications** column. If you have high resource waste in a cluster, the UI gives you a hint of the overall allocated versus requested information.

Go to GKE Cluster List (https://console.cloud.google.com/kubernetes/list)

- In the Google Cloud console, on the **Recommendations** page, look for **Cost savings** recommendation cards.

  Go to Recommendation Hub (https://console.cloud.google.com/home/recommendations)

For more details, see Observing your GKE clusters (/stackdriver/docs/solutions/kubernetes-engine/observing) and Getting started with Recommendation Hub (/recommender/docs/recommendation-hub/identify-configuration-problems).

Enable GKE usage metering

For a more flexible approach that lets you see approximate cost breakdowns, try GKE usage metering (/kubernetes-engine/docs/how-to/cluster-usage-metering). GKE usage metering lets you see your GKE clusters' usage profiles broken down by namespaces and labels. It tracks information about the resource requests and resource consumption of your cluster's workloads, such as CPU, GPU, TPU, memory, storage, and optionally network egress.

GKE usage metering helps you understand the overall cost structure of your GKE clusters, what team or application is spending the most, which environment or component caused a sudden spike in usage or costs, and which team is being wasteful. By comparing resource requests with actual utilization, you can understand which workloads are either under- or over-provisioned.

You can take advantage of the default Looker Studio templates, or go a step further and customize the dashboards according to your organizational needs. For more information about GKE usage metering and its prerequisites, see Understanding cluster resource usage (/kubernetes-engine/docs/how-to/cluster-usage-metering).

Understand how Metrics Server works and monitor it

Metrics Server is the source of the container resource metrics for GKE built-in autoscaling pipelines. Metrics Server retrieves metrics from kubelets (https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/) and exposes them through the Kubernetes Metrics API (https://github.com/kubernetes/metrics). HPA and VPA then use these metrics to determine when to trigger autoscaling.

For the health of GKE autoscaling, you must have a healthy Metrics Server. With GKE `metrics-server` deployment, a resizer *nanny* is installed, which makes the Metrics Server container grow vertically by adding or removing CPU and memory according to the cluster's node count. In-place update of Pods is still not supported in Kubernetes, which is why the nanny must restart the `metrics-server` Pod to apply the new required resources.

Although the restart happens quickly, the total latency for autoscalers to realize they must act can be slightly increased after a `metrics-server` resize. To avoid Metrics Server frequent restarts in fast-changing clusters, starting at GKE 1.15.11-gke.9, the nanny supports resize delays (https://github.com/kubernetes/autoscaler/issues/2597).

Follow these best practices when using Metric Server:

- Pick the GKE version that supports `metrics-server` resize delays. You can confirm it by checking whether the `metrics-server` deployment YAML file has the `scale-down-delay` configuration in the `metrics-server-nanny` container.
- Monitor `metrics-server` deployment. If Metrics Server is down, it means no autoscaling is working at all. You want your top-priority monitoring services to monitor this deployment.
- Follow the best practices discussed in GKE autoscaling (#fine-tune_gke_autoscaling).

Use Kubernetes Resource Quotas

In multi-tenant clusters, different teams commonly become responsible for applications deployed in different namespaces. For a centralized platform and infrastructure group, it's a concern that one team might use more resources than necessary. Starving all cluster's compute resources or even triggering too many scale-ups can increase your costs.

To address this concern, you must use resource quotas (https://kubernetes.io/docs/concepts/policy/resource-quotas/). Resource quotas manage the amount of resources used by objects in a namespace. You can set quotas in terms of compute (CPU and memory) and storage resources, or in terms of object counts. Resource quotas let you ensure that no tenant uses more than its assigned share of cluster resources.

For more information, see Configure Memory and CPU Quotas for a Namespace (https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/quota-memory-cpu-namespace/).

Consider using GKE Enterprise Policy Controller

GKE Enterprise Policy Controller (/anthos-config-management/docs/concepts/policy-controller) (APC) is a Kubernetes dynamic admission controller (https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/) that checks, audits, and enforces your clusters' compliance with policies related to security, regulations, or arbitrary business rules. Policy Controller uses constraints to enforce your clusters' compliance. For example, you can install in your cluster constraints for many of the best practices discussed in the Preparing your cloud-based Kubernetes application (#prepare_your_cloud-based_kubernetes_application) section. This way, deployments are rejected if they don't strictly adhere to your Kubernetes practices. Enforcing such rules helps to avoid unexpected cost spikes and reduces the chances of having workload instability during autoscaling.

For more information about how to enforce and write your own rules, see Creating constraints (/anthos-config-management/docs/how-to/creating-policy-controller-constraints) and Writing a constraint template (/anthos-config-management/docs/how-to/write-custom-constraint-templates). If you are not an GKE Enterprise (/anthos) customer, you can consider using Gatekeeper (https://github.com/open-policy-agent/gatekeeper), the open source software that APC is built on.

Design your CI/CD pipeline to enforce cost-saving practices

GKE Enterprise Policy Controller helps you avoid deploying noncompliant software in your GKE cluster. However, we recommend that you enforce such policy constraints early in your development cycle, whether in pre-commit checks, pull request (https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests) checks, delivery workflows, or any step that makes sense in your environment. This practice lets you find and fix misconfigurations quickly, and helps you understand what you need to pay attention to by creating guardrails.

Also consider using kpt functions (https://googlecontainertools.github.io/kpt/reference/fn/) in your CI/CD pipeline to validate whether your Kubernetes configuration files adhere to the constraints enforced by GKE Enterprise Policy Controller, and to estimate resource utilization or deployment cost. This way, you can stop the pipeline when a cost-related issue is detected. Or you can create a different deployment approval process for configurations that, for example, increase the number of replicas.

For more information, see Using Policy Controller in a CI pipeline (/anthos-config-management/docs/tutorials/app-policy-validation-ci-pipeline), and for a complete example of a delivery platform, see Modern CI/CD with GKE Enterprise (https://github.com/GoogleCloudPlatform/solutions-modern-cicd-anthos).

Spread the cost saving culture

Many organizations create abstractions and platforms to hide infrastructure complexity from you. This is a common practice in companies that are migrating their services from virtual machines to Kubernetes. Sometimes these companies let developers configure their own applications in production. However, it's not uncommon to see developers who have never touched a Kubernetes cluster.

The practices we recommend in this section don't mean that you should stop doing abstractions at all. Instead, they help you view your spending on Google Cloud and train your developers and operators on your infrastructure. You can do this by creating learning incentives and programs where you can use traditional or online classes, discussion groups, peer reviews, pair programming, CI/CD and cost-saving gamifications, and more. For example, in the Kubernetes world, it's important to understand the impact of a 3 Gb image application, a missing readiness probe, or an HPA misconfiguration.

Finally, as shown in Google's DORA research (/devops), culture capabilities are some of the main factors that drive better organizational performance, less rework, less burnout, and so on. Cost saving is no different. Giving your employees access to their spending aligns them more closely with business objectives and constraints.

Summary of best practices

The following table summarizes the best practices recommended in this document.

Topic	Task
GKE cost-optimization features and options	☐ Fine-tune GKE autoscaling (#fine-tune_gke_autoscaling) ☐ Choose the right machine type (#choose_the_right_machine_type) ☐ Select the appropriate region (#select_the_appropriate_region) ☐ Sign up for committed-use discounts (#sign_up_for_committed_use_discounts) ☐ Review small development clusters (#review_small_development_clusters) ☐ Review your logging and monitoring strategies (#review_your_logging_and_monitoring_strategies) ☐ Review inter-region egress traffic in regional and multi-zonal clusters (#review_inter-region_egress_traffic_in_regional_and_multi-zonal_clusters) ☐ Prepare your environment to fit your workload type (#prepare_your_environment_to_fit_your_workload_type)
Prepare your cloud-native Kubernetes applications	☐ Understand your application capacity (#understand_your_application_capacity) ☐ Make sure your application can grow both vertically and horizontally (#make_sure_your_application_can_grow_vertically_and_horizontally)

Topic	Task
	☐ Set appropriate resource requests and limits (#set_appropriate_resource_requests_and_limits ☐ Make sure your container is as lean as possible (#make_sure_your_container_is_as_lean_as_possible) ☐ Add Pod Disruption Budget to your application (#add-pod_disruption_budget-to-your-application) ☐ Set meaningful readiness and liveness probes for your application (#set_meaningful_readiness_and_liveness_probes_for_your_application) ☐ Make sure your applications are shutting down in accordance with Kubernetes expectations (#make_sure_your_applications_are_shutting_down_in_accordance_with_kubernetes_expectatio ☐ Setup NodeLocal DNSCache (#set_up_nodelocal_dnscache) ☐ Use container-native load balancing through Ingress (#use_container-native_load_balancing_through_ingress) ☐ Consider using retries with exponential backoff (#consider_using_retries_with_exponential_backoff)
Monitor your environment and enforce cost-optimized configurations and practices	☐ Observe your GKE clusters and watch for recommendations (#enable_gke_usage_metering) ☐ Enable GKE usage metering (#enable_gke_usage_metering) ☐ Understand how Metrics Server works and monitor it (#understand_how_metrics_server_works_and_how_to_monitor_it) ☐ Use Kubernetes Resource Quotas (#use_kubernetes_resource_quotas) ☐ Consider using GKE Enterprise Policy Controller (#spread_the_cost-saving_culture) ☐ Design your CI/CD pipeline to enforce cost-saving practices (#design_your_cicd_pipeline_to_enforce_cost-saving_practices)
Culture	☐ Spread the cost-saving culture (#spread_the_cost-saving_culture)

What's next

- Find more tips and best practices for optimizing costs at Cost optimization on Google Cloud for developers and operators (/solutions/cost-efficiency-on-google-cloud).
- For more details on how to lower costs on batch applications, see Optimizing resource usage in a multi-tenant GKE cluster using node auto-provisioning (/solutions/optimizing-resources-in-multi-tenant-gke-clusters-with-auto-provisioning).

- To learn how to save money at night or at other times when usage is lower, see the Reducing costs by scaling down GKE clusters during off-peak hours (/solutions/reducing-costs-by-scaling-down-gke-off-hours) tutorial.
- To learn more about using Spot VMs, see the Run web applications on GKE using cost-optimized Spot VMs (/kubernetes-engine/docs/archive/run-web-applications-on-gke-using-cost-optimized-spot-vms-and-traffic-director) tutorial.
- To understand how you can save money on logging and monitoring, take a look at Cost optimization for Cloud Logging, Cloud Monitoring, and Application Performance Management (/solutions/stackdriver-cost-optimization).
- For reducing costs in Google Cloud in general, see Understanding the principles of cost optimization (/resources/principles-of-cost-optimization-whitepaper).
- For a broader discussion of scalability, see Patterns for scalable and resilient apps (/solutions/scalable-and-resilient-apps).
- Explore reference architectures, diagrams, and best practices about Google Cloud. Take a look at our Cloud Architecture Center (/architecture).
- Learn more about Running a GKE application on Spot VMs with on-demand nodes as fallback (/blog/topics/developers-practitioners/running-gke-application-spot-nodes-demand-nodes-fallback).

Last updated 2024-01-31 UTC.