Scalability and Performance

Overview
Evaluating Resources for Work
Evaluating
Optimize Pod Performance with
Improving Kubernetes Stability
Disk Config

Overview

Use Scalability and performance to plan cluster scale, resource allocation, and performance-sensitive workload placement. This area collects scale evaluation, disk configuration, resource manager policies, and large-cluster stability guidance.

Need	Continue with
Evaluate workload cluster scale.	Evaluate workload clusters
Evaluate `global` cluster scale.	Evaluate global clusters
Configure CPU, memory, and topology manager policies.	Resource manager policies
Review large-cluster stability guidance.	Large clusters stability
Configure disk-related performance settings.	Disk configuration
Plan infra nodes or custom role nodes.	Infra Nodes

For validated scale limits, sizing formulas, or production hardening requirements, follow the current support matrix and topic-specific guidance.

Evaluating Resources for Workload Cluster

TOC

Recommended Control Plane Practices

Control Plane Node Sizing

Tested Cluster Maximums for Major Releases of ACP

Examples

Recommended Control Plane Practices

Use these practices to plan workload cluster control-plane resources in ACP.

INFO

The following requirements are based on the minimum ACP setup with Core packages installed. Actual requirements might be higher when Extensions are installed. Read the Extension-specific guidance for additional resource requirements.

Control Plane Node Sizing

Control plane sizing needs change with the cluster's node count, node types, and the number and kind of objects running. The recommendations below are derived from Cluster-density

tests that evaluate control plane behavior under load. Those tests deploy the following objects across a specified set of namespaces:

- 6 deployments, which pods only run the sleep process
- 6 services
- 6 ingresses pointing to the previous services
- 12 secrets containing 2048 random string characters
- 12 config maps containing 2048 random string characters

Number of worker nodes	Cluster-density (namespaces)	CPU cores	Memory (GB)
24	500	4	16
120	1000	8	32
254	4000	24	128

The data from the table above is based on a ACP environment installed on public-cloud virtual machines, using 16 vCPU and 128 GB RAM control-plane nodes and 8 vCPU and 32 GB RAM worker nodes.

On large, dense clusters with three control-plane nodes, taking one node offline—whether due to unexpected issues like power or network failures, infrastructure problems, or an intentional shutdown to save costs—forces the remaining two nodes to handle the extra work. When that occurs, CPU and memory usage on the surviving control-plane machines can rise significantly.
You will also see this pattern during upgrades, because control-plane nodes are typically cordoned, drained, and rebooted one after another while control-plane Operators are updated. That sequential maintenance concentrates demand on the nodes that remain active.
To reduce the risk of cascading failures, plan for sufficient headroom on control-plane servers: target an overall CPU and memory utilization of about 60% or less so the cluster can absorb transient load increases. If necessary, increase the control-plane CPU and RAM to prevent potential outages caused by resource exhaustion.

IMPORTANT NOTES

The node sizing varies depending on the number of nodes and object counts in the cluster. It also depends on whether the objects are actively being created on the cluster. During object creation, the control plane is more active in terms of resource usage compared to when the objects are in the **Running** phase.

Tested Cluster Maximums for Major Releases of ACP

WARNING

Overcommitting a node's physical resources can weaken the scheduling guarantees that Kubernetes relies on. Apply controls such as resource requests and limits, QoS classes, and node tuning to minimize memory swapping and other resource contention.

The figures come from Alauda test configuration, methodology, and tuning. Instead of presenting fixed caps, the following entries list the maximums observed for ACP under the tested conditions. Since the combinations of ACP version, control-plane workload, and network plugin is unlimited, these values are not guaranteed limits for all deployments and may not be simultaneously achievable across all dimensions.
Use them as guidance when planning deployments with similar characteristics.

INFO

When increasing or decreasing the number of your ACP clusters' nodes, it is recommended to:

- Spread nodes across all of the available zones when available, this contributes to higher availability.
- Scale up/down by no more than 25 to 50 nodes at once.

When scaling down large, densely populated clusters, the operation can take considerable time because workloads on the nodes slated for removal must be relocated or terminated before those nodes are shut down. This can be particularly lengthy when many resources must be processed at once.
If a large number of objects need eviction, the API client may begin to rate-limit requests. The

default client queries-per-second (QPS) and burst settings are **50** and **100**, respectively, and these values cannot be changed in ACP.

Maximum type	tested maximum
Number of nodes	500 [1]
Number of pods [2]	100,000
Number of pods per node	250
Number of namespaces [3]	10,000
Number of pods per namespace [4]	25,000
Number of secrets	40,000
Number of config maps	40,000
Number of services	10,000
Number of services per namespace	5,000
Number of back-ends per service	5,000
Number of deployments per namespace [4]	2,000
Number of custom resource definitions (CRD)	1,024 [5]

1. ACP supports clusters with more than 500 nodes; the number here is the recommended maximum. If your use case needs larger clusters, contact technical support.
2. The pod numbers shown are counts from the test environment. Actual pod capacity will vary according to each application's memory, CPU, and storage requirements.
3. With many active projects, etcd performance can degrade if the keyspace grows too large and surpasses its space quota. Regular etcd maintenance—such as defragmentation—is recommended to reclaim storage and avoid performance issues.
4. Several control loops iterate over all objects in a namespace in response to state changes. A very large number of objects of a single type within one namespace makes those loops

expensive and can slow processing. The stated limit assumes the system has sufficient CPU, memory, and disk to meet application needs.

5. The test was run on a 29-server cluster (3 control-plane nodes, 2 infra nodes, and 24 worker nodes) with 500 namespaces. ACP enforces a cap of 1,024 total custom resource definitions (CRDs), including those provided by ACP, CRDs added by integrated products, and user-created CRDs. Creating more than 1,024 CRDs may cause **kubectl** requests to become throttled.

Examples

In one test scenario, 500 worker nodes, each with 8 vCPUs and 32 GB of memory, were tested with ACP 4.1, the Kube-OVN network plugin, and the following workload objects:

- 200 namespaces, in addition to the defaults
- 60 pods per node; 30 server and 30 client pods (30k total)
- 15 services/ns backed by the server pods (3k total)
- 20 secrets/ns (4k total)
- 10 config maps/ns (2k total)
- 6 network policies/ns, including deny-all, allow-from ingress and intra-namespace rules

The following factors are known to affect cluster workload scaling and should be factored into scale planning. For additional guidance, contact technical support.

- Number of pods per node
- Number of containers per pod
- Type of probes used (for example, liveness/readiness, exec/http)
- Number of network policies
- Number of projects, or namespaces
- Number of services/endpoints and type
- Number of shards
- Number of secrets
- Number of config maps

- Rate of API calls, which is an estimation of how quickly things change in the cluster configuration.
  - Prometheus query for pod creation requests per second over 5 minute windows: `sum(irate(apiserver_request_count{resource="pods",verb="POST"}[5m]))`
  - Prometheus query for all API requests per second over 5 minute windows: `sum(irate(apiserver_request_count{}[5m]))`
- Cluster node resource consumption of CPU
- Cluster node resource consumption of memory

Evaluating Resources for Global Cluster

TOC

- Overview
- Node Sizing
  - Baseline Production Sizing
  - Vertical Scaling Guidelines
  - Horizontal Scaling Guidelines
  - Resource Validation and Monitoring
- Summary

Overview

Use these resource evaluation practices when planning the `global` cluster for a Multi-Cluster ACP deployment.

Proper node sizing helps the `global` cluster manage registered clusters, handle synchronization traffic, and process user API and Web Console requests within the performance expectations of your environment.

Node Sizing

The `global` cluster is responsible for:

- Maintaining cluster registration and metadata.
- Handling inbound API requests from the Web Console and CLI.
- Coordinating synchronization and heartbeat messages with connected clusters.
- Managing internal controllers and resource reconciliation loops.

Because the `global` cluster handles both management operations and data aggregation from all connected clusters, plan resource allocation according to the expected scale and workload intensity.

Baseline Production Sizing

The production-scale sizing depends primarily on:

- Number of **managed clusters**
- Frequency of **synchronization cycles**
- **Concurrent API request rate** (from users or automation)
- Volume of **streaming requests**
- Number of **plugins installed**

The following table provides reference configurations validated through internal performance testing.

Scale Tier	Managed Clusters	Number of Nodes	CPU per Node	Memory per Node	N
Small	≤ 10	3	8 cores	16 GB	S sr er
Medium	≤ 50	3	16 cores	32 GB	D pr se
Large	≤ 100	3	24 cores	48 GB	S he

Scale Tier	Managed Clusters	Number of Nodes	CPU per Node	Memory per Node	N
					C a s
Extra Large	≤ 500	6	32 cores	64 GB	R h s d n

> **WARNING**
>
> These recommendations are general guidelines. Actual requirements depend on your cluster topology, user concurrency, and plugins installed.

Vertical Scaling Guidelines

When increasing load per node (for example, 2× more clusters or higher user concurrency), follow these adjustments:

Parameter	Scaling Recommendation
CPU	+50% for every 50 additional managed clusters
Memory	+50% for every 50 additional managed clusters

Horizontal Scaling Guidelines

When exceeding 100 managed clusters or encountering persistent API latency above 500 ms:

Add nodes to distribute request handling and controller workloads.

Resource Validation and Monitoring

After deployment, continuously monitor the following metrics to validate node sizing:

Metric	Recommended Range
Node CPU utilization	60–75% under peak load
Node Memory utilization	≤80% sustained
API request latency	P90 < 500ms
etcd commit latency	P99 < 50ms

Node CPU utilization

```
100 * (1 - avg by (instance)(rate(node_cpu_seconds_total{mode="idle"}[5m])))
```

Node Memory utilization

```
100 * (1 - (node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes))
```

API request latency

```
histogram_quantile(
  0.9,
  sum(rate(apiserver_request_duration_seconds_bucket{verb!="WATCH",resource!="events"}[5m])) by (le)
)
```

etcd commit latency

```
histogram_quantile(
  0.99,
  sum(rate(etcd_disk_backend_commit_duration_seconds_bucket[5m])) by
(le)
)
```

NOTE

If sustained resource usage consistently exceeds recommended thresholds, scale vertically (add CPU/memory) or horizontally (add nodes) before user-facing performance degradation occurs.

Summary

When sizing the `global` cluster:

1. Begin with **3 nodes × 16 cores × 32 GB** for moderate-scale deployments (≤50 clusters).
2. Scale **vertically** for higher request concurrency or heavy Web Console usage.
3. Scale **horizontally** beyond 100 clusters to maintain API responsiveness.
4. Re-evaluate sizing after every significant increase in managed cluster count or sync frequency.

Use these practices to keep resource planning aligned with the growth of your Multi-Cluster environment.

Optimize Pod Performance with Manager Policies

Kubernetes cluster administrators can enable and validate **CPU ManagerPolicy**, **Memory ManagerPolicy**, and **Topology ManagerPolicy** to align CPU pinning, NUMA affinity, and topology placement for latency-sensitive workloads.

TOC

- Scope and Prerequisites
- Quick Start: Sample Kubelet Config
- How to Calculate `reservedMemory`
- Applying the Configuration
  - Alauda OS
  - Non-Alauda OS
- Verification
- Key Policies and Behaviors
- Terminology

Scope and Prerequisites

Roles and Permissions

- Requires maintenance window access, `kubectl` admin privileges, and SSH access to nodes.

Workload Requirements

- To achieve dedicated CPU and NUMA affinity, Pods must run in **Guaranteed QoS** class: requests = limits and CPU specified in full cores (e.g., 2, 4).

Not Covered

- HugePages are out of scope. If you need HugePages support, contact your support team.

Quick Start: Sample Kubelet Config

The following kubelet configuration enables NUMA-aware scheduling. Adjust the values for your environment. How you apply it depends on the node operating system—see Applying the Configuration.

```
# ── CPU ManagerPolicy ──
cpuManagerPolicy: "static"              # Options: none | static
cpuManagerPolicyOptions:
  full-pcpus-only: "true"               # Recommended: allocate only full
cores
cpuManagerReconcilePeriod: "5s"
reservedSystemCPUs: ""                  # e.g. "0-1" if reserving specifi
c CPUs for the system

# ── Memory ManagerPolicy ──
memoryManagerPolicy: "Static"           # Options: none | Static
reservedMemory:
  - numaNode: 0
    limits:
      memory: "2048Mi"
  - numaNode: 1
    limits:
      memory: "2048Mi"

# ── Topology ManagerPolicy ──
topologyManagerPolicy: "single-numa-node"     # Options: none | best-effo
rt | restricted | single-numa-node
topologyManagerScope: "pod"                   # Options: container | pod
```

Notes:

- `full-pcpus-only: "true"` improves latency consistency.
- `topologyManagerScope: pod` ensures containers within the same Pod align to a common NUMA topology.
- `reservedMemory` must be calculated based on kubelet config and eviction thresholds (see next section).

How to Calculate `reservedMemory`

Formula:

```
R_total = kubeReserved(memory) + systemReserved(memory) + evictionHard(memory.available)
```

The sum of `reservedMemory` across all NUMA nodes must equal `R_total`.

Steps (for N NUMA nodes):

1. Calculate `R_total` (Mi).
2. Compute division and remainder:
   - base = floor(R_total / N)
   - rem = R_total − base × N
3. Assign values:
   - NUMA node 0 = base + rem
   - Remaining NUMA nodes = base

Example (2 NUMA nodes):

- kubeReserved=512Mi, systemReserved=512Mi, evictionHard=100Mi → R_total = 1124Mi
- base = 562, rem = 0

```yaml
reservedMemory:
- numaNode: 0
  limits:
    memory: "562Mi"
- numaNode: 1
  limits:
    memory: "562Mi"
```

Applying the Configuration

How you apply these kubelet settings depends on the node operating system.

Alauda OS

On nodes that run Alauda OS, CPU and Memory Manager policies are applied with assistance from Alauda technical support. Contact your Alauda support representative to enable these policies on these nodes.

Non-Alauda OS

On nodes that run a non-Alauda OS, apply the configuration on each node:

1. **Cordon and Drain**

```
kubectl cordon <node>
kubectl drain <node> --ignore-daemonsets --delete-emptydir-data
```

2. **Stop Kubelet and Clear State**

```
sudo systemctl stop kubelet
sudo rm -f /var/lib/kubelet/cpu_manager_state
sudo rm -f /var/lib/kubelet/memory_manager_state
```

3. **Restart Kubelet**

```
sudo systemctl daemon-reload
sudo systemctl start kubelet
```

4. **Reschedule Pods**

```
kubectl uncordon <node>
```

- For DaemonSets and system Pods, restart or delete Pods explicitly.

5. **Verify Recovery**

```
kubectl get nodes
kubectl get pods -A -o wide | grep <node>
```

Verification

CPU ManagerPolicy State

```
sudo cat /var/lib/kubelet/cpu_manager_state | jq .
```

Check:

- `.policyName` = `"static"`
- `.defaultCpuSet` lists non-dedicated CPUs
- `.entries` show container-to-CPU assignments

Memory ManagerPolicy State

```
sudo cat /var/lib/kubelet/memory_manager_state | jq .
```

Check:

- `.policyName` = `"Static"`
- Sum of reserved memory matches `R_total`
- Guaranteed Pods are assigned to NUMA nodes per `single-numa-node` policy

Key Policies and Behaviors

CPU ManagerPolicy

- Purpose: Allocate exclusive physical CPUs to Guaranteed Pods
- Config: `cpuManagerPolicy: static`, `full-pcpus-only: "true"`
- Behavior: Only applies to Guaranteed Pods; Burstable/BestEffort are unaffected

Memory ManagerPolicy

- Purpose: Reserve and align memory at NUMA node level
- Config: `memoryManagerPolicy: "Static"`, `reservedMemory`

- Behavior: Works best with Topology ManagerPolicy for alignment

Topology ManagerPolicy

- Purpose: Align CPU, memory, and device allocation on a single NUMA node
- Config: `topologyManagerPolicy: single-numa-node`, `topologyManagerScope: pod`
- Modes: best-effort, restricted, single-numa-node (strict)

Terminology

- **NUMA node**: Non-Uniform Memory Access domain
- **CPU pinning**: Binding containers to dedicated CPUs
- **NUMA affinity**: Preferring memory from the same NUMA node as CPU
- **Topology alignment**: Co-locating CPU, memory, and devices on one NUMA node
- **Guaranteed Pod**: requests = limits; CPU specified as full cores

Improving Kubernetes Stability for Large-Scale Clusters

Cluster operators and SREs can use these practices to reduce control-plane overload, improve reliability, and limit blast radius in large-scale Kubernetes clusters.

TOC

Notes

Footnotes

Notes

- For network, storage, load-balancer, logging, and monitoring tuning, use the component-specific guidance for those capabilities.

WARNING

- Test configuration changes before production.
- Avoid a single huge cluster when risk is high; consider multi-cluster management to reduce blast radius.

Issue	Description	Optimization
etcd disk	etcd is highly sensitive to disk I/O in large clusters.	Use a dedicated NVMe SSD for the etcd data directory.
etcd DB size	DBs larger than ~8 GB significantly worsen latency, resource use and recovery time.	Keep the etcd DB ≤ 8 GB; remove unused objects and keep frequently-updated objects small (~≤100 KB).
etcd key churn	High read/write frequency on keys strains etcd.	Analyze etcd metrics to find and reduce hot keys.
Data size per resource type in etcd	Large totals per resource type make full-list operations expensive and can block controllers.	Keep each resource type's total data ≤ 800 MB; clean up unused Deployments/Services. [1]
API server LB bandwidth & connections	Load-balancer bandwidth or connection limits can cause nodes to become NotReady.	Monitor/provision the API-server load balancer in advance.
Services per namespace	Many Services cause large env var injection into Pods and slow startups.	Keep Services per namespace < 5,000, or set `enableServiceLinks: false`. [2]
Total Services in cluster	Excess Services increase kube-proxy rules and harm performance.	Keep total Services < 10,000 and garbage-collect unused ones. [1]
CoreDNS	Large Pod counts can degrade CoreDNS performance.	Run NodeLocal DNSCache (nodelocaldns).
Pod update rate	High update rates push Endpoint/EndpointSlice changes to all nodes and can cause storms.	Reduce Pod churn; for RollingUpdate set conservative `maxUnavailable` / `maxSurge`.
ServiceAccount token mounts	Each token Secret can create a watch; many watches burden the control plane.	For Pods that don't need API access, set `automountServiceAccountToken: false`. [3]

Issue	Description	Optimization
Object count/size	Accumulated ConfigMaps, Secrets, PVCs, etc. increase control-plane load.	Limit ReplicaSet history (`revisionHistoryLimit`) and use `ttlSecondsAfterFinished` for Jobs/CronJobs.
Pod requests/limits	Large gaps between requests and limits can trigger cascading failures on node loss.	Prefer setting requests equal to limits where feasible.
Controller restarts & monitoring	Controller or API server restarts cause re-lists that can overload the API server.	Monitor controllers, set appropriate resources to avoid restarts, and reduce unnecessary control-plane operations.

Footnotes

1. https://github.com/kubernetes/community/blob/master/sig-scalability/configs-and-limits/thresholds.md ↗ ↩ ↩[2]
2. https://kubernetes.io/docs/tutorials/services/connect-applications-service/#accessing-the-service ↗ ↩
3. https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/#opt-out-of-api-credential-automounting ↗ ↩

Disk Configuration

TOC

Storage Capacity
Recommended ETCD Practices
Validating the hardware for etcd
Benchmarking with fio

Storage Capacity

INFO

Mount the following partitions on dedicated disks or on LVM-provisioned logical volumes so they can be expanded later.

partition	Minimum size	Recommended size	Notes
/var/lib/etcd	10GB	20GB	A dedicated high-IO disk is recommended for hosting etcd data.

partition	Minimum size	Recommended size	Notes
/var/lib/containerd/	100GB	150GB	
/cpaas/	For control plane nodes of global cluster, at least 100GB; For other nodes, at least 40GB	200GB	Plan for additional space if you expect the infra node components which requires more space on /cpaas/. On Alauda OS nodes this partition is `/var/cpaas/`.
/	50GB	100GB, higher is better.	Ensure there is enough free disk space to keep utilization below 80%. If usage rises above this threshold, pods on the node may be evicted.
arbitrary location for downloading and unpacking the installer packages, extensions and so on.	20GB	250GB	Actual storage needs will vary depending on which extensions you plan to install. Plan for additional space if you expect to add more components or enable extra features later.

INFO

The partition layout above describes nodes that run a non-Alauda OS. On clusters whose nodes run Alauda OS, platform state is kept under `/var/cpaas` instead of `/cpaas`, and node disks are declared in the provider's machine configuration rather than partitioned manually. The capacity guidance still applies. For the disk declaration procedure of each provider, see Create Workload Clusters on Alauda OS.

Recommended ETCD Practices

Fast storage is essential for etcd to perform reliably. etcd depends on durable, low-latency disk operations to persist proposals to its write-ahead log (WAL).
If disk writes take too long, fsync delays can cause the member to miss heartbeats, fail to commit proposals promptly, and experience request timeouts or temporary leader changes. These issues can also slow the Kubernetes API and degrade overall cluster responsiveness. In conclusion, HDDs are a poor choice and are not recommended. If you must use HDDs for etcd, choose the fastest available (for example, 15,000 RPM).

INFO

The following hard drive practices provide optimal etcd performance:

- Prefer SSDs or NVMe as etcd drives. When write endurance and stability are priorities, consider server-grade single-level cell (SLC) SSDs. Avoid NAS, SAN, and HDDs.
  - Prefer drives with high write throughput to accelerate compaction and defragmentation.
  - Prefer drives with strong read bandwidth to reduce recovery time after failures.
  - Prefer drives with consistently low latency to ensure fast read and write operations.
- Avoid distributed block storage systems such as Ceph RADOS Block Device (RBD), Network File System (NFS), and other network-attached backends, because they introduce unpredictable latency.
- Keep etcd data on a dedicated drive or a dedicated logical volume.
  - Do not place I/O-sensitive (such as logging) or other intensive filesystem activity on control-plane hosts, or at least do not let them share the same underlying storage with etcd.
- Continuously benchmark with tools like `fio` and use the results to track performance as the cluster grows. For more information, see the disk benchmarking guide.

Validating the hardware for etcd

Specification	Minimum Requirement	Recommended	Notes
Sequential write IOPS	50	500 (higher is better)	Most cloud providers publish *concurrent* IOPS rather than *sequential* IOPS. The concurrent IOPS values are typically about 10× higher than sequential ones.
Disk bandwidth	10 MB/s	100 MB/s (higher is better)	Higher disk bandwidth allows faster data recovery when a failed member needs to catch up with the cluster.
Throughput (sequential 8 kB write with `fdatasync`)	50 writes per 10 ms	500 writes per 2 ms	Reflects sustained write throughput when data is flushed to disk after each write operation.

Benchmarking with fio

To measure actual sequential IOPS and throughput, use the disk benchmarking tool `fio`:

> **WARNING**
>
> Do not run these tests against any nodes of the ACP clusters.
>
> Instead, run the tests against a dedicated VM that has the same set up as the control plane nodes.

```
set -e
mkdir -p /var/lib/etcd/

echo "INFO: Running fio"
fio --rw=write --ioengine=sync --fdatasync=1 --directory=/var/lib/etcd --
size=100m --bs=8000 --name=etcd_perf --output-format=json --runtime=60 --
time_based=1 | tee /tmp/fio.out

# Scrape the fio output for p99 of fsync in ns
fsync=$(cat /tmp/fio.out | jq '.jobs[0].sync.lat_ns.percentile["99.00000
0"]')
iops=$(cat /tmp/fio.out | jq '.jobs[0].write.iops')
echo "INFO: 99th percentile of fsync is $fsync ns"

# Compare against the recommended value
if [[ $fsync -ge 10000000 ]]; then
    echo "WARN: IOPS is $iops, 99th percentile of the fsync is greater th
an the recommended value which is ${fsync} ns > 10 ms, faster disks are r
ecommended to host etcd for better performance"
else
    echo "INFO: IOPS is $iops, 99th percentile of the fsync is within the
recommended threshold: - 10 ms, the disk can be used to host etcd"
fi
```