Title: Scaling Kubernetes with One-Node Clusters: A Paradigm Shift in Cloud-Native Orchestration

by Dinis Cruz and ChatGPT Pro Deep Research

Kubernetes has revolutionized container orchestration, providing a powerful abstraction for managing workloads across distributed infrastructure.

Traditionally, best practices dictate deploying Kubernetes in multi-node clusters, where a centralized control plane orchestrates workloads across worker nodes.

This model, while effective, introduces significant complexity—requiring intricate networking, control plane synchronization, and careful capacity management.

In this research, we challenge the conventional wisdom of Kubernetes scaling by proposing an alternative paradigm: **one-node clusters per instance**. Instead of scaling within a large, multi-node cluster, we scale by launching many small, self-contained Kubernetes clusters, each running on a single node.

This approach eliminates inter-node networking complexity, reduces maintenance overhead, and enhances failure isolation, all while retaining Kubernetes' powerful application deployment capabilities.

The one-node cluster model may seem counterintuitive, often met with skepticism from Kubernetes purists who argue that it contradicts traditional best practices. However, we demonstrate that, when combined with **cloud auto-scaling, global load balancing, and immutable infrastructure**, this approach is not only viable but also highly scalable, resilient, and operationally efficient.

This paper explores the business case for one-node clusters, technical implementation on AWS, comparisons with traditional Kubernetes architectures, multi-cloud portability, and best practices for overcoming potential challenges.

We make the case that, far from being an edge case, one-node clusters represent a forward-thinking evolution in Kubernetes scaling—simplifying operations while unlocking new levels of agility and resilience.

Introduction

Kubernetes has become the de facto platform for container orchestration, traditionally designed to manage multi-node clusters of machines. In typical setups, a cluster comprises multiple worker nodes overseen by a control plane, allowing workloads to be distributed for high availability. Over time, organizations have evolved their scaling approaches – from single large clusters to multi-cluster architectures – to meet growing demands for isolation and reliability (Multi-cluster Kubernetes: Benefits, Challenges and Tools) (Multi-cluster Kubernetes: Benefits, Challenges and Tools). Conventional wisdom suggests running a few sizable clusters, each with many nodes, to maximize resource sharing and simplify management. However, large multi-node clusters bring challenges: significant overhead for maintaining the control plane, complex networking (CNI plugins, overlay networks, etc.), and operational complexity in upgrades and scaling coordination.

Multi-node clusters introduce substantial maintenance effort. Each additional node must join the cluster, sync state, and participate in consensus (etcd) and scheduling. As clusters grow, the control plane must scale and etcd coordination can become a performance bottleneck (Architecting Kubernetes clusters — how many should you have?). Moreover, managing diverse workloads on a shared cluster complicates resource isolation and increases the blast radius of failures. For example, a misconfiguration or bug in one app could potentially impact others in the same cluster. Administrators often grapple with ensuring multi-tenancy security, network policies, and fair resource quotas in large clusters.

The need for an alternative scaling model is evident – one that prioritizes *simplicity, automation,* and *resilience* without sacrificing the benefits of Kubernetes. This paper proposes **one-node clusters per instance** as that alternative. In this model, each Kubernetes node is its own independent cluster, containing both control plane and workloads. By treating entire clusters as disposable units (much like treating servers as cattle rather than pets), we can automate cluster lifecycle with immutable infrastructure techniques. The result is a fleet of uniform single-node clusters that can be scaled out or replaced on demand. This design eliminates inter-node orchestration overhead and dramatically simplifies certain aspects of operations (networking is local to the node, no inter-node pod traffic routing needed (Kubernetes 101)). It also encapsulates failures – if one one-node cluster crashes, only that node's workloads are affected and other clusters continue serving traffic.

Importantly, Kubernetes practitioners often push back on single-node clusters due to the lack of built-in high availability and perceived wastefulness of running many control planes (Single node k8 cluster pros and cons : r/kubernetes) (Architecting Kubernetes clusters — how many should you have?). We address these concerns by distributing workloads across many one-node clusters behind load balancers, achieving resilience through redundancy. If one cluster (node) fails, its traffic is routed to

others, similar to how a multi-node cluster would reschedule pods on healthy nodes. Automation ensures new clusters can spin up quickly to replace failed ones. The following sections detail the business case for this model, its implementation (with a focus on AWS), comparisons with traditional clusters, multi-cloud applicability, and challenges to consider.

Business Case

Deploying one-node clusters per instance offers several business and operational advantages. This section examines the benefits in terms of maintenance overhead, cost efficiency, and fault tolerance, building the case that this unconventional model can solve real-world problems.

Reduced Operational Complexity: By design, a one-node cluster has no multi-node coordination – no need to manage cluster networking across hosts, no complex cluster autoscaler logic, and no etcd quorum concerns. Each cluster is simplistic: one kubelet, one API server, one scheduler running locally. This can dramatically streamline operations. Teams can manage each node like an immutable appliance. There is strong isolation between clusters since they share no resources (Architecting Kubernetes clusters — how many should you have?). Changes or issues in one cluster do not directly affect others, reducing the risk of cascading failures. Additionally, fewer engineers need access to each cluster, lowering the chance of human error per cluster (Architecting Kubernetes clusters — how many should you have?). In effect, the administrative domain is segmented into many small, independent pieces, which can simplify troubleshooting and reduce scope of impact.

Minimal Maintenance Overhead: Traditional Kubernetes clusters require regular maintenance – control plane upgrades, node patching, scaling adjustments – which become complex as clusters grow. In the one-node model, maintenance is minimized by automating cluster replacement instead of in-place changes. For example, upgrading Kubernetes version or OS is achieved by baking a new AMI and cycling out instances, rather than orchestrating a delicate in-place upgrade of a multi-node cluster. Each one-node cluster can be treated as ephemeral; if it drifts from the desired state or requires an update, automation simply terminates it and replaces it with a fresh instance. This *immutable infrastructure* approach cuts down on long-running snowflake servers and manual fixes. The operational burden shifts to maintaining the automation code and base images, which is often easier to standardize.

Cost Efficiency: At first glance, running many single-node clusters might seem cost-inefficient due to duplicated overhead. It's true that each cluster carries the fixed cost of a control plane. However, modern lightweight Kubernetes distributions and managed services can minimize this overhead. Solutions like *K3s* shrink the Kubernetes binary to <100 MB and use half the memory of upstream Kubernetes, making single-node clusters feasible even on small instances (K3s - Lightweight Kubernetes |

K3s). By eliminating auxiliary control plane nodes and consolidating control functions onto the worker, a one-node cluster avoids the need for dedicated master node instances. In cloud environments like AWS, using one-node clusters on EC2 means you pay only for the single instance's resources; there is no separate charge for a multi-node control plane (unlike, say, running 3 masters for HA in a traditional cluster). When using managed Kubernetes (like EKS or GKE), one could choose to use the smallest possible cluster size – even one worker node – per cluster; though managed control planes have a fee, it can be acceptable for the benefits gained.

Furthermore, this model can reduce waste by right-sizing each cluster to a specific workload or tenant. In a multi-tenant cluster, one often over-provisions capacity for peak loads of all services combined. With one-node clusters, you can scale out only the clusters that need more capacity. Unused clusters can even be shut down to zero. Overall utilization can improve if automation rapidly destroys underused clusters. Essentially, *horizontal scaling across clusters* replaces vertical scaling within a cluster. **Note:** There is a trade-off in resource efficiency – small clusters might not pack workloads as densely as a shared large cluster, potentially leaving more headroom on each node. In practice, the cost trade-off must be analyzed case by case. But for many applications and especially for microservices architectures with many small workloads, avoiding the overhead of large cluster management can offset the slight increase in per-node overhead.

Fault Tolerance and Resilience: One-node clusters inherently lack redundancy **within** themselves (if that node fails, the cluster's workloads go down) (Manage single-node clusters). However, by architecting with *redundancy across clusters*, we achieve high availability in a different way. Imagine an application that in a traditional setup might run 10 pods across 3 nodes in one cluster. In the one-node cluster model, you might run 10 one-node clusters each running 1 pod (or a few pods) of that application, all behind a load balancer. If any single cluster (instance) fails, its traffic is automatically routed to the remaining nine clusters. The impact is similar (losing one pod out of 10), but without needing a complex rescheduling mechanism – the infrastructure (Auto Scaling Group and load balancer) handles recovery by launching a new instance to replace the failed one. This results in *self-healing infrastructure*. The system can detect a downed instance and recover it in minutes, with minimal impact on uptime.

The resilience of this model also shines during updates. Blue-green or canary deployments can be done at the cluster level: spin up new one-node clusters with the updated software while the old ones still handle production load, then switch traffic gradually to the new clusters (for example, by adjusting load balancer target weights or DNS). This isolates any potential update issues to a subset of clusters. Rolling back is as simple as directing traffic back to the old cluster set. Traditional clusters can also do zero-downtime deploys with rolling updates, but cluster-per-instance provides an *extra layer of isolation* – a bad deployment might

crash one cluster's Kubernetes processes without affecting others, whereas in a multi-node cluster a bad pod could, in worst case, affect the entire cluster's health.

In summary, the business case for one-node clusters centers on **simplified operations**, **automation-friendly workflows**, **isolation for safety**, and **resilient, modular scaling**. Organizations with many small services or stringent uptime requirements may find this model attractive to reduce the cognitive load of Kubernetes management while still harnessing Kubernetes's core benefits (consistent API, deployment patterns, etc.). We next delve into how to implement this model on AWS, leveraging cloud capabilities to bring it to life.

Technical Implementation (AWS-Focused)

Implementing one-node-per-cluster architecture requires rethinking how we deploy Kubernetes. In an AWS environment, we can utilize EC2 auto-scaling, custom AMIs, and various AWS networking services to automate fleets of one-node clusters. This section outlines a reference implementation on AWS, covering the setup of single-node Kubernetes instances, scaling them with Auto Scaling groups, integrating load balancers (including global traffic management), and considerations for stateful components.

Immutable One-Node Cluster Instances

The foundation of this model is an **immutable Kubernetes node image**. We create a custom Amazon Machine Image (AMI) that has Kubernetes pre-installed and configured to run as a single-node cluster. There are a few approaches to achieve this:

- **Kubeadm or K3s Initialization:** One can use a user-data script or cloud-init to run `kubeadm init` on boot and set up a single-node cluster (tainting the master to allow workloads). However, to minimize boot time and external dependencies, a better approach is to pre-bake the cluster into the AMI. For example, using **K3s** is ideal – it's a lightweight, single-binary Kubernetes distribution that can run all control plane components within one process (K3s - Lightweight Kubernetes | K3s). We can install K3s on an EC2, configure it to launch on startup (as a systemd service), and verify that it forms a functional single-node cluster. This instance then serves as the golden image for all clusters. All necessary components (container runtime, CNI if needed, etc.) are included so that the instance is essentially a plug-and-play Kubernetes node.
- **Pre-loading Applications:** Depending on deployment strategy, we might also bake application containers or manifests into the image. One model is to include the Docker/OCI images of the primary application so that they are available locally on boot. Alternatively, the cluster could be configured to pull from a registry as usual. An even simpler approach is using

Kubernetes static pods (by placing pod YAML files in `/etc/kubernetes/manifests`) so that as soon as the single-node cluster comes up, it immediately starts the desired pods. This removes the need for an external deployment step – the cluster instantiates with the app running. However, static pods lack some of the management flexibility, so one could also have a daemon that applies standard Deployment manifests on startup.

- **Security and Config:** Each one-node cluster should come up with secure defaults – for instance, the kube-apiserver should listen only on localhost or a private interface (we generally don't want to expose each cluster's API externally). In AWS, the instances can be in private subnets, and we might not even need to interact with the Kubernetes API once it's running the app (all management done via AWS tools). Still, for debugging or advanced control, one could set up a bastion or use SSM Session Manager to run kubectl on the node. Certificates and tokens for the cluster's API are local to that instance.

Once our AMI is prepared, deploying a cluster is as simple as launching an EC2 instance with that AMI. This is where the **immutable infrastructure** principle shows its power: *identical images yielding identical clusters.* As CloudCaptain's Axel Fontaine noted, immutable AMIs make scaling trivially simple – whether you need 1 or 1000 instances, you just launch clones of the same image (It's Auto-Scaling time! - Blog - CloudCaptain • Immutable Infrastructure Made Easy). Each instance will self-configure as a Kubernetes node on boot, with no unique snowflake setup required. This uniformity lays the groundwork for effortless scaling and replacement.

Auto Scaling and Load Balancing

With the one-node cluster AMI ready, we leverage **EC2 Auto Scaling Groups (ASGs)** to manage the fleet. An Auto Scaling Group ensures that a specified number of instances are always running, and can adjust that number based on policies or metrics. The steps are as follows:

1. **Launch Template Configuration:** We create a Launch Template (or Launch Configuration) using the custom AMI, instance type (right-sized for the workload), security group, and user-data if needed (though ideally the AMI is fully self-contained). We might attach an IAM role if the pods or cluster need AWS API access (for example, to fetch from S3 or register with an ELB).
2. **Auto Scaling Group Setup:** Define an ASG that uses the above Launch Template. We can set a minimum, maximum, and desired capacity. For example, desired might start at 2 instances (for redundancy). The ASG is placed in appropriate subnets (multiple Availability Zones for high availability, so that instances go to different AZs). We also enable health checks – the ASG can use EC2 status checks and can be tied to load balancer health checks for more granular control (if an instance fails a load balancer health check, the ASG can terminate and replace it).

3. **Scaling Policies:** The ASG can scale out/in based on metrics. A common approach is to use a target tracking policy on CPU utilization. Since each instance runs a copy of our service, we can say, for example, keep average CPU around 50% – if load increases and CPU goes above target, ASG will launch more instances (clusters); if load drops, it will terminate some. AWS's integration allows scaling actions to happen relatively quickly (a new instance launch can take a minute or two to boot). We eliminate Kubernetes-specific scaling concerns (like the cluster autoscaler); instead, we rely on AWS CloudWatch metrics and ASG logic to add/remove entire cluster instances. This is conceptually simpler: we scale at the *infrastructure level* rather than the *pod level*. Each new instance automatically brings up the needed pods by virtue of the AMI's startup configuration.
4. **Elastic Load Balancer Integration:** To distribute traffic among the one-node clusters, we register them behind an **Elastic Load Balancer (ELB)**. In modern AWS terms, this would typically be an Application Load Balancer (ALB) for HTTP/HTTPS traffic or a Network Load Balancer for lower-level TCP/UDP needs. The ASG can be attached to an ALB Target Group so that instances register themselves on launch. The ALB performs health checks (e.g., hitting an HTTP endpoint on the service) to ensure each cluster instance is serving traffic. Only healthy instances receive traffic. This is how we achieve that if one instance's cluster fails (or the app crashes), the ALB will detect it and stop sending traffic to it, while the ASG will work to replace it. AWS notes that spreading workloads across instances ensures if one instance fails, others pick up the traffic (Scaling Kubernetes on AWS: Everything You Need to Know) (Scaling Kubernetes on AWS: Everything You Need to Know) – exactly the principle we apply here, just that each instance is a fully self-contained environment.
5. **DNS or API Gateway:** Clients will reach the service via the load balancer's endpoint (or a friendly DNS CNAME to it). For example, an ALB provides a domain name; we can map a custom URL (like `api.myservice.com`) to that. From the client perspective, it's a single endpoint, but behind it is a dynamic pool of one-node clusters scaling in or out.

The result is an *auto-healing, auto-scaling cluster-of-clusters*. We have effectively outsourced Kubernetes' horizontal pod scaling to the cloud provider's instance scaling. One might ask about scaling latency: adding a new pod in Kubernetes might take seconds, whereas launching a new EC2 instance might take a couple of minutes. This is a valid observation – scaling out is somewhat slower at the instance granularity. In practice, we can mitigate this by slightly over-provisioning or using smaller instance types such that launching 2-3 in advance is not overly costly. Additionally, AWS's scale-out policies can be proactive (based on request queue or latency) to trigger new instances before current ones max out.

Global Deployment and Caching

A powerful aspect of the one-node cluster model is how naturally it extends to multi-region or global deployments. Each cluster is independent and doesn't rely on a central control plane, so deploying clusters in multiple regions is straightforward. On AWS, we

can replicate the setup across regions to bring the service closer to users and provide regional redundancy.

Geo-Distributed Clusters: Suppose we want active-active presence in US East, US West, and Europe. We can create identical Auto Scaling Groups and ALBs in each region, each managing that region's one-node clusters. Then, to route users to the nearest cluster (and failover if a whole region goes down), we use global load balancing mechanisms. AWS Route 53 with latency-based routing is one option: it can resolve `api.myservice.com` to the ALB in the region that has the lowest latency for the client. Another option is AWS Global Accelerator, which provides a single anycast IP that fronts your application and routes to the nearest endpoints in AWS's network. AWS Global Accelerator continuously checks the health of each region's endpoint and will drop traffic to a failed region automatically (Operating a multi-regional stateless application using Amazon EKS | Containers) (Operating a multi-regional stateless application using Amazon EKS | Containers). This ensures that even a full region outage results in users being routed to another region seamlessly.

When combining multi-region clusters with a global traffic manager, we achieve an extremely resilient architecture. An AWS blog on multi-regional EKS notes that using AWS's global footprint improves availability and latency for applications (Operating a multi-regional stateless application using Amazon EKS | Containers) (Operating a multi-regional stateless application using Amazon EKS | Containers). Essentially, each region's one-node clusters ASG is an island that can operate on its own; losing one does not directly affect the others, and global routing ties them together.

Content Delivery and Caching: The inclusion of Amazon CloudFront in our design further boosts performance and offloads work from the clusters. CloudFront is a Content Delivery Network (CDN) that can sit in front of our regional load balancers as the public face of the service. By using CloudFront, we terminate user connections at the edge locations nearest to users, which reduces latency for the TLS handshake and initial request (Dynamic content delivery | Content Delivery Network (CDN), API Acceleration, Security | Amazon CloudFront) (Dynamic content delivery | Content Delivery Network (CDN), API Acceleration, Security | Amazon CloudFront). CloudFront can cache static content (like images, scripts, API responses that are cacheable) and even dynamically accelerate API calls by reusing connections to the origins. According to AWS, CloudFront helps improve performance and availability for dynamic web content by leveraging the AWS backbone network (Dynamic content delivery | Content Delivery Network (CDN), API Acceleration, Security | Amazon CloudFront). Slack, for instance, saw a large drop in global latency by fronting their API with CloudFront (Dynamic content delivery | Content Delivery Network (CDN), API Acceleration, Security | Amazon CloudFront).

In practice, we would set CloudFront's origin as our load balancer (or perhaps separate cache behaviors by path). For purely API traffic, caching may be limited (as responses might be unique per request), but CloudFront can still offload TLS and shield the

origin from traffic bursts (through its distributed caching mechanism – even if TTL is 0, it can reduce repeated simultaneous requests). Additionally, CloudFront offers edge-level security (WAF, DDoS protection via AWS Shield) which adds to the resilience of our setup.

Architecture Summary: At this point, our AWS architecture for one-node clusters looks like this:

- **CloudFront CDN (optional):** Receives user requests; if content is cached, serves from edge; otherwise forwards to regional origin.
- **Route 53 / Global Accelerator:** Directs requests to the appropriate regional ALB based on latency or as a failover (if using active-passive).
- **Regional ALB (per region):** Distributes traffic to all healthy one-node cluster instances in that region. Performs health checks.
- **Auto Scaling Group (per region):** Maintains the fleet of one-node cluster EC2 instances, scaling as needed. Each instance is identical via the immutable AMI.
- **One-Node Kubernetes Instances:** Each running application pods and necessary services. They may connect to regional databases or other services as needed.
- **Data layer (shared):** Likely a managed database or storage service accessible from all clusters (discussed next).

This setup maximizes scalability (we can always add more regions or more instances), resilience (no single cluster or region failure breaks the app globally (Using latency-based routing with Amazon CloudFront for a multi-Region active-active architecture | Networking & Content Delivery)), and simplicity in the sense that each building block (cluster) is uniform and replaceable.

Handling Stateful Applications and Databases

Stateful components require special consideration in any stateless horizontal scaling architecture, and one-node clusters are no exception. If our application is completely stateless (e.g., serving computations or transient data only), we can freely scale clusters without worrying about data consistency. But most real applications need state: databases, file storage, user sessions, etc.

Externalizing State: The recommended approach is to externalize stateful services to managed cloud services or dedicated stateful clusters. For example, instead of running a database inside each one-node cluster (which would create many isolated pockets of data), one can use AWS RDS or Aurora for a shared database. All the one-node clusters then connect to that database

endpoint. This way, the database is highly available (multi-AZ, perhaps) and remains consistent. AWS offers global database options like *Aurora Global Database* or *DynamoDB Global Tables* for multi-region replication (Operating a multi-regional stateless application using Amazon EKS | Containers). If multi-region active-active is required for the database, those services can replicate data across regions so that clusters in each region talk to a local instance of the database with data sync happening behind the scenes (Operating a multi-regional stateless application using Amazon EKS | Containers).

Managed caches (ElastiCache for Redis, DynamoDB, etc.) can handle session state or frequently accessed data, accessible to all clusters. For file storage, S3 or EFS (Elastic File System) can provide shared persistent storage (e.g., if user uploads need to be accessible from all instances, store them in S3 rather than on local disk).

Stateful Workloads on One-Node Clusters: If one must run a stateful component on the cluster itself (say, a small database or a service with local disk state), it complicates matters. You would not get the benefit of that state if the instance dies, unless you implement a replication at the application level. For example, you could run a replicated database with one replica per one-node cluster, but then effectively you're reinventing distributed databases – not ideal. Generally, *avoid running single-node databases per cluster for production.* It's better to run a separate small multi-node (or single but backed up) database cluster that all app clusters use, or use cloud services as mentioned.

If a particular microservice is stateful and can't easily be scaled as multiple identical instances (for example, it might be a leader that coordinates something), it might be better not to break it into many one-node clusters. That service could run in one cluster with proper HA configuration (if needed, that cluster could actually be a multi-node just for that service or use EBS volumes and restart on failure). However, this is an edge case; core services like databases are typically handled outside of Kubernetes in many architectures even with traditional clusters.

Persistent Data on Clusters: In scenarios where each cluster might handle a portion of data (sharding by user, etc.), careful design is needed. You could designate each one-node cluster to serve a partition of your dataset. This is not the typical use case for one-node clusters model, which is more geared towards identical replicas for scaling. Still, if partitioning is used, a routing layer would send users to the cluster responsible for their data partition. This starts to resemble running multiple independent installations of an app (like many independent SaaS deployments), which might be a deliberate choice for multi-tenancy. In such a case, each one-node cluster *is* a full environment for a subset of customers. This provides extreme tenant isolation, but coordination of data between clusters is then not needed (tenants are separate). The trade-off is uneven load distribution if some tenants are much heavier.

In summary, for stateful needs our recommendations are: use cloud-managed databases or shared state layers whenever possible, design the application to be stateless at the service level, and if stateful components must exist, consider dedicating special clusters or external services for them rather than mixing into the ephemeral one-node clusters. By doing so, the one-node clusters remain disposable and easy to scale or replace without data loss concerns.

Comparison with Traditional Multi-Node Clusters

To evaluate the merits of the one-node-per-instance model, it's important to compare it against the conventional multi-node cluster approach across several dimensions: complexity, deployment speed, resource utilization, failure handling, and resilience. Below we outline these comparisons:

- **Infrastructure Complexity:** A traditional multi-node Kubernetes cluster centralizes control but introduces internal complexity. You must manage the control plane (often multiple masters for HA), etcd clustering, and networking between nodes. One-node clusters eliminate internal cluster networking issues entirely – for example, "pod networking doesn't even need CNI plugins or overlay networks" when you have a single-node cluster (Kubernetes 101). The one-node model externalizes complexity to the cloud provider (load balancers, etc.) while simplifying each Kubernetes environment. However, **management complexity** shifts to handling many clusters instead of one. In a single large cluster, you manage one API endpoint and one set of RBAC policies. In a one-node cluster fleet, there are multiple API endpoints (which might be a non-issue if we seldom directly use them) and multiple Kubernetes configs. This can be mitigated with automation and treating clusters as throwaway, but it's a trade-off: **simplicity within each cluster vs. multiplicity of clusters**. Administering dozens or hundreds of clusters *manually* would be impractical, so automation and possibly cluster management tooling (like Kubernetes Cluster Federation or cluster APIs) become important.
- **Resource Utilization & Overhead:** Multi-node clusters are more efficient at sharing resources. Workloads can be bin-packed on nodes up to their capacity, and a single control plane amortizes its cost over many nodes. With one-node clusters, each instance reserves some CPU/RAM for its control plane components (API server, controller, etcd). If you have 100 one-node clusters, that is 100× the control plane overhead. This *inefficient resource usage* is a known downside of many small clusters (Architecting Kubernetes clusters — how many should you have?) (Architecting Kubernetes clusters — how many should you have?). There is also potential duplication of system pods (each cluster might run its own CoreDNS, ingress controller, etc., unless those are disabled or not needed for the app). On the other hand, in a large multi-node cluster, you often still have system overhead (DNS, CNI) on each node plus control plane instances. The difference is mainly that etcd and controllers are duplicated in the one-node case. Our approach minimizes this by using lightweight K3s (which can run etcd or SQLite as a

tiny local datastore). In cloud-managed scenarios, the cloud might handle control plane off-node (e.g., EKS control plane isn't on your node), but if we use that, then each cluster's control plane is a paid managed service. Cost-wise, running many EKS clusters would incur multiple control plane fees (Architecting Kubernetes clusters — how many should you have?). For cost-sensitive cases, self-managed with K3s on the instance might be better.

- **Deployment Speed & Flexibility:** One area the one-node model excels is *deployment speed for new clusters or versions*. In a traditional cluster, deploying a new version of an application means doing a rolling update of pods, which depends on scheduling and cluster capacity. Deploying a new version of Kubernetes itself means upgrading the cluster, which is complex and must be coordinated (control plane then nodes). In contrast, with one-node clusters, deploying a new version of the *entire stack* (app plus Kubernetes version) can be as simple as launching new instances with an updated AMI. The new clusters come up in parallel while old ones continue to run. This is effectively a blue-green deployment at the cluster level. As soon as the new clusters are healthy, traffic can be switched over (for instance, swap the target group in the ALB or update DNS). The old clusters can then be terminated. This **immutable rollout** approach reduces downtime and avoids in-place upgrades, which aligns with best practices in immutable infrastructure (similar to how one might replace VMs in an autoscaling group with new ones during deployment) (Immutable Kubernetes clusters : r/kubernetes) (Immutable Kubernetes clusters : r/kubernetes). Pod-level deployment in Kubernetes is already good, but cluster-level deployment ensures even underlying changes (like OS or Kubernetes itself) are applied cleanly. Flexibility is also enhanced because each cluster could potentially run a different version or configuration if needed (for example, canary clusters with a new config while others run stable config) without affecting a whole cluster's worth of workloads.
- **High Availability and Failure Recovery:** Traditional clusters achieve HA by running multiple replicas of pods and multiple nodes. If a node fails, the pods are rescheduled on another node by the scheduler. This process can be quick, but not instantaneous; it depends on failure detection and re-spinning containers, which might take tens of seconds to a few minutes. In the one-node cluster setup, if an instance fails, the AWS load balancer detects it usually within seconds (depending on health check interval) and stops sending traffic there. There is *no rescheduling of pods* – the pods on that node are just gone. But since other clusters had their own pods serving the same service, the traffic is instantly served by the remaining ones. The lost capacity triggers the ASG to launch a new instance, which might take a couple of minutes to join back. In terms of user experience, both models aim for no downtime beyond perhaps a brief capacity reduction. One difference: if you had stateful pods with persistent volume in a multi-node cluster, they could potentially reschedule and attach their volumes to a new node (if using PVCs and storage classes). In a one-node cluster, a local state would be lost with the instance. We've addressed this by offloading state externally. So for stateless services, both models provide seamless failover. For control plane failures, a multi-node cluster with a single control plane (no HA masters) could become completely inoperable if the

master node fails. Best practice is to run multiple masters in HA mode to avoid that. In one-node clusters, the control plane is single and not HA, *but* the failure of a control plane is equivalent to the loss of that entire cluster (which is handled by redundancy in others). Thus the system as a whole remains available. The one-node model embraces failure as normal: treat the cluster as expendable. Traditional cluster tries to be resilient internally to keep the cluster alive.

- **Operational Maintenance:** In steady state, a multi-node cluster is one unit to monitor (one set of logs/metrics). A one-node cluster environment means multiple units. This can complicate **observability** – you need aggregate monitoring. However, modern monitoring tools can aggregate across many sources easily. You might run an agent on each instance sending metrics to a central system (CloudWatch, Datadog, etc.). Logging can be centralized via CloudWatch Logs or an ELK stack that ingests logs from all nodes. It's an extra setup step but solvable. Cluster maintenance tasks like upgrades or config changes in one-node model translate to updating the AMI or bootstrap and rolling out new instances, which can actually be less work than carefully cordoning and upgrading nodes one by one in a live cluster. From a **DevOps workflow** perspective, one-node clusters fit well with infrastructure-as-code: changes to cluster configuration are done by building a new image (or adjusting user-data scripts), which is version-controlled. In contrast, making changes in a persistent cluster might involve imperative operations or config maps that drift over time.

In summary, **traditional multi-node clusters** excel at efficient resource utilization and central management, but at the cost of intricate internal complexity and potentially slower cluster-wide changes. **One-node clusters** invert that: they bring clarity and simplicity to each unit and make global changes easy by brute-force replacement, but require careful automation to handle the multitude of units and accept some resource overhead. Neither is strictly "better" in all cases; the choice depends on priorities. For maximum simplicity and fault isolation, one-node clusters shine. For maximum efficiency and centralized control, a well-tuned multi-node cluster might be preferable. It's also possible to adopt a hybrid: e.g., cluster-per-service but each cluster has a couple of nodes for that service; however, here we focus on the extreme end of one node per cluster.

Extending to Other Cloud Providers & On-Prem

While our discussion has focused on AWS, the one-node cluster model can be applied to other environments with suitable adaptations. Each cloud provider offers analogous features (VM images, auto-scaling, load balancing) that can enable this pattern. Here's how the approach translates:

- **Google Cloud Platform (GCP):** GCP's equivalent would use Instance Templates and Managed Instance Groups (MIGs) for auto-scaling one-node clusters on GCE (Google Compute Engine) VMs. One could create a custom VM image with a single-

node Kubernetes (for example, using the COS OS with a preloaded Kubernetes). GCP also has *Google Kubernetes Engine (GKE)*; one could create many single-node GKE clusters, but this might be heavy due to multiple control planes. Instead, using raw VMs with K3s might be more economical. For load balancing, GCP's **Cloud Load Balancing** can serve a similar role to ALB. Notably, GCP's HTTP(S) Load Balancer is global by default and can route traffic to instances across regions. This means a single GCP load balancer can balance between one-node clusters in say, US and Europe, using Google's global network (Cross-region load balancing + routing on Google Container Engine). This simplifies global setup (no need for separate DNS latency routing; Google's LB can do cross-region balancing out of the box for HTTP). Cloud CDN can be enabled on the load balancer for caching similar to CloudFront. The principles of using MIG health checks and auto-healing remain the same. GCP's Preemptible VMs could even be leveraged for cost savings (since clusters are ephemeral, you might tolerate occasional VM termination, though it adds complexity in ensuring a new one boots immediately).

- **Microsoft Azure:** Azure provides Virtual Machine Scale Sets (VMSS) which are analogous to AWS ASGs. We'd create a custom Azure VM image (using Azure Image Builder or similar) with our one-node Kubernetes cluster configuration. Then a scale set can auto-scale instances based on metrics, and Azure Load Balancer or Application Gateway can distribute traffic. Azure Front Door or Traffic Manager can be used for global routing (Front Door acts like a global HTTP load balancer with caching, similar to CloudFront+Route53 combo). Azure has AKS (Azure Kubernetes Service), but again running many AKS clusters (each with one node) might not be cost-effective due to overhead of many managed control planes. A key consideration on Azure is to ensure the VM image includes cloud-init or script extensions to bring up K3s and that the networking (Azure VNet) allows the load balancer to probe the instances. Azure also supports custom autoscale rules via metrics (like CPU, queue length, etc.) to trigger scale set changes. The one-node model aligns with Azure's *VM as unit of scale* approach in VMSS.
- **Kubernetes Distributions on VMware or Bare Metal:** On-premises, one-node clusters can be achieved with virtualization or even on bare-metal servers using lightweight K8s distros. VMware vSphere, for example, could clone a VM template that has K3s installed. Tools like VMware Auto Deploy or orchestration through vCenter REST APIs could automatically clone and manage the desired number of instances. One could use NSX for providing a virtual load balancer or simply use an F5 or HAProxy appliance as an external load balancer to distribute traffic across nodes. Without a cloud's native autoscaler, an on-prem deployment might use a custom script or controller to monitor load and trigger VM creation via the virtualization API. The concept of an **edge deployment** is also relevant: each edge location might have a one-node cluster (K3s is often cited for edge computing due to its small footprint (K3s - Lightweight Kubernetes | K3s)). For global on-prem or hybrid setups, DNS round-robin or commercial global traffic managers (like Cisco GSS, etc.) can direct users to the nearest site.

- **Other Cloud Providers:** The pattern can extend to any cloud with compute instances and load balancing: Oracle Cloud, IBM Cloud, DigitalOcean, etc. DigitalOcean, for instance, has a simpler environment – one could deploy 1-VM Kubernetes clusters using something like k3s on Droplets and use DO's load balancer. DigitalOcean's Kubernetes service (DOKS) even explicitly notes that one-node clusters are possible, though they caution about performance and resiliency (How to Create Kubernetes Clusters | DigitalOcean Documentation). Our model would mitigate resiliency by using multiple one-node clusters.

Key Considerations for Different Providers:

- *Auto-scaling triggers:* Ensure that the provider offers metrics-based autoscaling for instances. AWS, GCP, Azure all do. If not, an external process might be needed to watch metrics and call the API to scale (which is doable with scripts or tools like Terraform Cloud or custom operators).
- *Image building:* Investing in a good automated pipeline for building the VM images (AMI, GCP Image, Azure Managed Image) is important. Packer by HashiCorp is a common tool to script image creation for multiple clouds.
- *Networking and Firewalls:* Each one-node cluster might need certain ports open (e.g., the app port, maybe 443 for HTTPS if terminating SSL on the node, or just 80/443 on the LB with node running plain HTTP). Cloud firewalls/security groups should be set accordingly. Also, if using internal load balancers, the clusters should be in the same network or VPC.
- *Cloud provider limits:* Be mindful of limits like maximum instances per region, load balancer target limits, etc. Many one-node clusters could mean many endpoints – ensure the chosen LB can handle that (most are fine with hundreds of targets).
- *Managed Kubernetes option:* Some providers have the notion of virtual nodes or on-demand clusters. For example, Google's Anthos or Azure Arc could manage multiple clusters in a fleet, though these are more hybrid management planes. There's also the CNCF Cluster API project which can create and manage multiple clusters (even ephemeral) via a Kubernetes-style API. Such tools can help manage many clusters uniformly across clouds.

In essence, the cloud-agnostic recipe is: **custom image + instance auto-scaling + load balancer + optional global routing**. Each provider has its flavor of those components. The one-node cluster model's viability does not hinge on any AWS-specific service, so it is quite portable. On-premises, lacking native autoscaling, may require more custom tooling, but it's feasible especially with virtualization and modern on-prem orchestrators (OpenStack Heat could be another approach in an open cloud environment, to spin up VMs on demand based on metrics).

Challenges & Considerations

No approach is without trade-offs. Before embracing one-node Kubernetes clusters, engineers should be aware of potential challenges and edge cases. We discuss these pitfalls and considerations, and outline best practices to address them.

1. Control Plane Overhead & Limits: As noted, running a large number of Kubernetes clusters multiplies the control plane overhead. In extreme scaling scenarios, this could lead to waste or even hitting limits (for example, you wouldn't run thousands of one-node clusters without significant resources). Additionally, if using managed K8s for each, the cost can stack up (EKS, for instance, charges a fixed fee per cluster). *Mitigation:* use lightweight Kubernetes distros (like K3s or MicroK8s) to keep per-cluster resource usage minimal. Profile how much CPU/memory the control plane uses for your workload – if it's small (e.g., K3s can run control plane in ~512MB RAM or less when idle), you can likely afford it. Also, ensure your instance size accounts for control plane overhead plus workload; don't size it exactly for the app load ignoring Kubernetes' needs. In terms of limits, if you foresee hundreds of clusters, you'll want automation to manage them (consider using Kubernetes Cluster API or similar tools to treat clusters as declarative objects, so creation and destruction is handled systematically).

2. Operational Tooling for Fleet Management: Dealing with many clusters means monitoring, logging, and management can become complicated. **Monitoring:** It's not practical to check each cluster's dashboard manually. Instead, aggregate metrics at a higher level. For instance, run a monitoring agent on each node that pushes to a centralized Prometheus or CloudWatch. You might monitor overall error rates per cluster and trigger alerts if one cluster is behaving oddly (e.g., consistently high error rate might indicate it's unhealthy even if the LB hasn't kicked it out yet). **Logging:** Use centralized logging. Each instance can forward container logs to a central system (CloudWatch Logs, ELK, Splunk, etc.). Tag logs with the instance/cluster ID to know which cluster they came from. This way, you have one place to search logs across all clusters – crucial for debugging issues that might be isolated to one instance. **Upgrades and Configuration Drift:** With many clusters, you want to avoid having slightly different configurations on each. Rely on the immutable image to enforce uniformity. Implement an image build and release process (DevOps CI/CD pipeline) so that changes go into the image and then propagate to clusters via rebuild. If you need to update all running clusters (say a critical security patch in the base OS), you might script a rolling replacement: gradually increase ASG sizes with new image and decrease old ones. This can be automated with blue-green flipping as previously described.

3. Networking and Service Discovery Between Clusters: In a microservices architecture, different services need to communicate. If each service is deployed as its own set of one-node clusters, how do they talk to each other? In a monolithic cluster, services use internal cluster DNS (service names) to communicate, and Kubernetes handles service discovery and load balancing. In the multi-cluster approach, a service in cluster A must call service B which runs in separate clusters. Solutions include: using the public (or internal) endpoint of service B's load balancer. For instance, configure service A to call

`http://service-b.mycompany.com` which is a DNS that goes to B's load balancer (could be an internal load balancer if within same VPC). Essentially you treat inter-service calls like external calls via well-known URLs or a service mesh that spans clusters. Technologies like **service mesh** can in theory connect multiple clusters (e.g., using Istio's multi-mesh federation or Linkerd with gateways) but that adds complexity we aimed to remove. A simpler method is to use cloud DNS service discovery (AWS Cloud Map or just hardcoded DNS hostnames for each service's ALB). The downside is slightly more network hop (out and back in through a LB), but within a cloud region that overhead is small (and possibly all internal traffic). Another approach is to colocate tightly coupled services on the same cluster instance if feasible to minimize cross-cluster calls. *Consideration:* The architecture should be designed with this in mind – either embrace that services talk over the network as if they are external (twelve-factor style), or use a consolidation strategy for chatty services.

4. Scenarios Where Traditional Clusters Win: There are cases where the overhead of multi-cluster outweighs its benefits:

- **Very large workloads with shared data or in-memory state:** If you have a workload that requires a lot of in-memory state that is not easily partitioned, one big cluster might handle it by scaling pods on a huge node or using specialized daemon sets. Many one-node clusters might not help because that state can't be fragmented without a distributed system. For example, a Spark big data job or certain HPC workloads prefer one cluster with many nodes working in tandem.
- **Tight Pod-to-Pod latency requirements:** If two components communicate extremely frequently with low-latency requirements, keeping them in one cluster (possibly on the same node or same AZ) might be better. Cross-cluster communication will be a bit higher latency.
- **Resource bin-packing for cost optimization:** If your goal is to maximize utilization, a scheduler that can fill up nodes with various pods is beneficial. Many small clusters could lead to fragmentation (one cluster might have unused CPU that another cluster could have used, but they don't share). Overprovisioning buffer has to be set per cluster rather than once globally, potentially increasing total overhead. Traditional clusters with a cluster autoscaler can achieve high utilization by flexibly scheduling pods.
- **Simpler use-case / small scale:** If you only have a handful of services and low load, the complexity of even setting up automation for multi-cluster might not be worth it. A single cluster with a few nodes might be perfectly fine and easier for a small team to manage.

5. Performance Bottlenecks: In the one-node cluster model, each instance is a performance island. The max throughput of a single instance is capped by that instance's hardware. In a multi-node cluster, a single pod's throughput could be increased by vertical scaling or by spreading load across multiple pods on multiple nodes. In one-node, if you need more throughput for one

service, you add another cluster (another instance). This scales linearly, but there could be edge cases like connection handling – e.g., if you have a stateful long-lived connection that a user must consistently hit the same node (like WebSocket without sharing state), you might need a session stickiness approach at the LB. While ALBs do support sticky sessions, you lose some failover transparency (if a node goes down, that session breaks). This is similar in any horizontally scaled scenario though. To mitigate performance issues: choose appropriate instance types for each cluster to handle baseline load, and use auto-scaling to react to spikes. Also leverage CloudFront caching to reduce repeated work. *Testing is critical* – load test the one-node setup to ensure it behaves as expected at scale (especially test scenarios of rapid scale-out, and failover, to tune health checks and scaling policies).

6. Cultural/Knowledge Hurdles: This model is still unconventional, and teams may be hesitant to adopt it. Kubernetes expertise often revolves around multi-node clusters, so engineers might need time to get comfortable treating clusters as cattle. There could be pushback such as *"Why use Kubernetes at all if each instance is independent?"* (Immutable Kubernetes clusters : r/kubernetes). It's important to articulate that Kubernetes still provides a consistent application management and packaging format, and the ecosystem (Helm charts, etc.) can still be used, just applied to one-node at a time. The method essentially leverages Kubernetes for what it's good at (running containers reliably) while sidestepping some complexity (multi-node coordination). Over time, as tooling improves and more success stories emerge, this approach could gain wider acceptance in the community.

Best Practices for Implementation:

- **Automate Everything:** Manual intervention should be minimized. Use Infrastructure as Code (Terraform, CloudFormation, etc.) to define the ASGs, LBs, and related resources. Use CI/CD pipelines to build and test the Kubernetes node images. Implement health monitoring scripts or rely on cloud health checks for recycling instances.
- **Gradual Rollout:** If migrating from a traditional setup, introduce one-node clusters gradually, perhaps for a subset of less critical services, to build confidence. Monitor and gather data on performance and cost.
- **Consistency:** Keep configurations consistent across clusters using common templates. For example, use the same Kubernetes version and security settings on all (unless testing a new version). Inconsistencies can lead to unpredictable behavior.
- **Capacity Planning:** Even though scaling is automatic, plan your instance counts and sizes with headroom for failure. Ensure you always run at least 2 instances per service (so that one can fail and others carry load). Cross-AZ or cross-region

redundancy is vital since a one-node cluster is by itself not redundant (Manage single-node clusters). Essentially, treat N one-node clusters as a unit that together forms a reliable service.

- **Cleanup and Lifecycle:** Have a strategy for cleanup of decommissioned clusters. If an instance fails and is removed, any associated resources (like EBS volumes if used, or DNS entries) should be cleaned. Usually, AWS ASG handles most of that (terminating an instance will delete ephemeral storage, etc.). If clusters register themselves somewhere (not typical in this design), deregistration logic is needed.
- **Security:** Consider the security of having multiple clusters. It might actually improve isolation (one cluster can't directly affect another). Use separate IAM roles for each ASG if appropriate to limit blast radius of credentials. Also, patching the base image regularly is important since you won't be logging into each instance to patch – you rely on replacing instances to propagate patches.

In conclusion, while the one-node cluster model introduces some challenges – especially around managing scale – these can be addressed with thoughtful design and automation. The model isn't a one-size-fits-all; it should be applied in contexts where its advantages align with project goals (simplicity, resilience, etc.). For other contexts, a hybrid or traditional approach might be warranted. The key is understanding these trade-offs and making an informed decision.

Conclusion & Future Outlook

In this paper, we presented an alternative Kubernetes scaling paradigm: deploying one-node clusters per instance to achieve maximum scalability, resilience, and simplicity. Through our exploration, we demonstrated that contrary to initial skepticism, this model **can work and even excel** under the right conditions. By treating each node as an isolated cluster, organizations can gain strong fault tolerance (via many small failure domains), straightforward cluster lifecycle management (immutable replacements), and simplified networking (each cluster is self-contained). We showed how leveraging AWS capabilities like Auto Scaling groups, load balancers, and CloudFront can implement this architecture, and we compared it with traditional multi-node clusters to highlight its strengths and weaknesses.

Summary of Benefits: One-node clusters shine in scenarios requiring extreme uptime and ease of automation. They reduce the coordination complexity that often plagues larger clusters. Upgrades become simpler (flip to new cluster instances), and outages are graceful (other clusters carry on). The approach also aligns with modern DevOps practices of immutable infrastructure and microservices isolation. By deploying separate clusters, you inherently isolate workloads in terms of resources and security boundaries, which can be a boon for multi-tenant environments or compliance (no noisy neighbors, each tenant could even be on

their own Kubernetes cluster). Moreover, for edge computing, the only viable Kubernetes might be single-node at each edge – our model is effectively taking that pattern into the data center cloud and scaling it out horizontally.

Practical Applications: This model could be useful for organizations running large numbers of relatively small services (think of a SaaS with many independent customer deployments, or a microservices architecture where each service is modest in resource needs but must be highly available). It's also appealing for teams that want to simplify Kubernetes usage: instead of a platform team managing a complex cluster, much of the heavy lifting is handed to cloud automation, and developers get a consistent "cluster per service" environment. Startups concerned about over-engineering might find it paradoxically simpler to reason about one service per VM (with Kubernetes providing container management on that VM) than running a multi-service cluster.

Challenges Revisited: We acknowledged that this approach isn't without downsides – notably potential cost overhead and management of multiple clusters. However, the trajectory of cloud computing suggests these issues will lessen over time. Consider that as of now, projects are underway to make Kubernetes control planes more lightweight and multi-tenant themselves. For instance, virtual-kubelet or shared control planes could allow many logical clusters to run on a single control plane instance. In the future, one could envision a hosted service where you can create "sandboxed" one-node clusters at will, without incurring full overhead per cluster. If such technology matures, the one-node model would combine the best of both worlds (lightweight plus isolated).

Future Potential: The concept of **"clusters as cattle"** may play a significant role in cloud-native architectures. We already see trends in multi-cluster management and federation (Scaling Kubernetes on AWS: Everything You Need to Know) – tools are emerging to coordinate applications across clusters, treating clusters as just another deployable unit. Our one-node clusters could plug into such systems, benefitting from orchestration at a higher level. Another area of future development is serverless or on-demand clusters. AWS EKS has recently introduced features to spin up clusters more quickly and even scale them (EKS on Fargate, etc.). Perhaps one day you might not even need to manage the EC2 layer; you could request N one-node clusters from a service and pay per usage. That would truly unlock elastic cluster-per-instance scaling without worrying about control planes at all.

Encouragement for Experimentation: We conclude by encouraging the Kubernetes community and practitioners to experiment with this model. It may not yet be mainstream, but as we've argued, it holds significant promise for certain use cases. By challenging the conventional wisdom ("more nodes in one cluster is always better"), we can discover new ways to harness Kubernetes. Early adopters can report lessons learned, contributing data points on performance and cost. Over time, if the model

proves its worth, we might see more tooling and cloud support explicitly for it (for example, managed one-node cluster pools, or easier ways to do networking between many clusters).

In essence, Kubernetes is a flexible platform – it doesn't mandate how large or small a cluster must be. Running one-node clusters is absolutely within its capabilities (the official docs even state it's fine for testing, and we've pushed that boundary to production scenarios) (Kubernetes 101) (How to Create Kubernetes Clusters | DigitalOcean Documentation).

The **future outlook** is that as infrastructure becomes increasingly automated, the granularity of what we consider a "unit" can shrink.

In the past, an entire data center was the unit, then VMs, then containers. **Now we're proposing clusters themselves as units of scale**.

This approach dovetails with trends like multi-cloud deployments (where many clusters are deployed across providers) and the desire for robust failure isolation.

In conclusion, one-node Kubernetes clusters offer a compelling scaling model that flips traditional assumptions on their head. We have presented a comprehensive argument and framework for adopting this model, with evidence and reasoning to support its viability.

It's our hope that this sparks discussion and innovation in the Kubernetes ecosystem.

By thinking creatively about cluster architectures, we can continue to push the boundaries of what is possible in building scalable, resilient, and simple cloud-native systems.