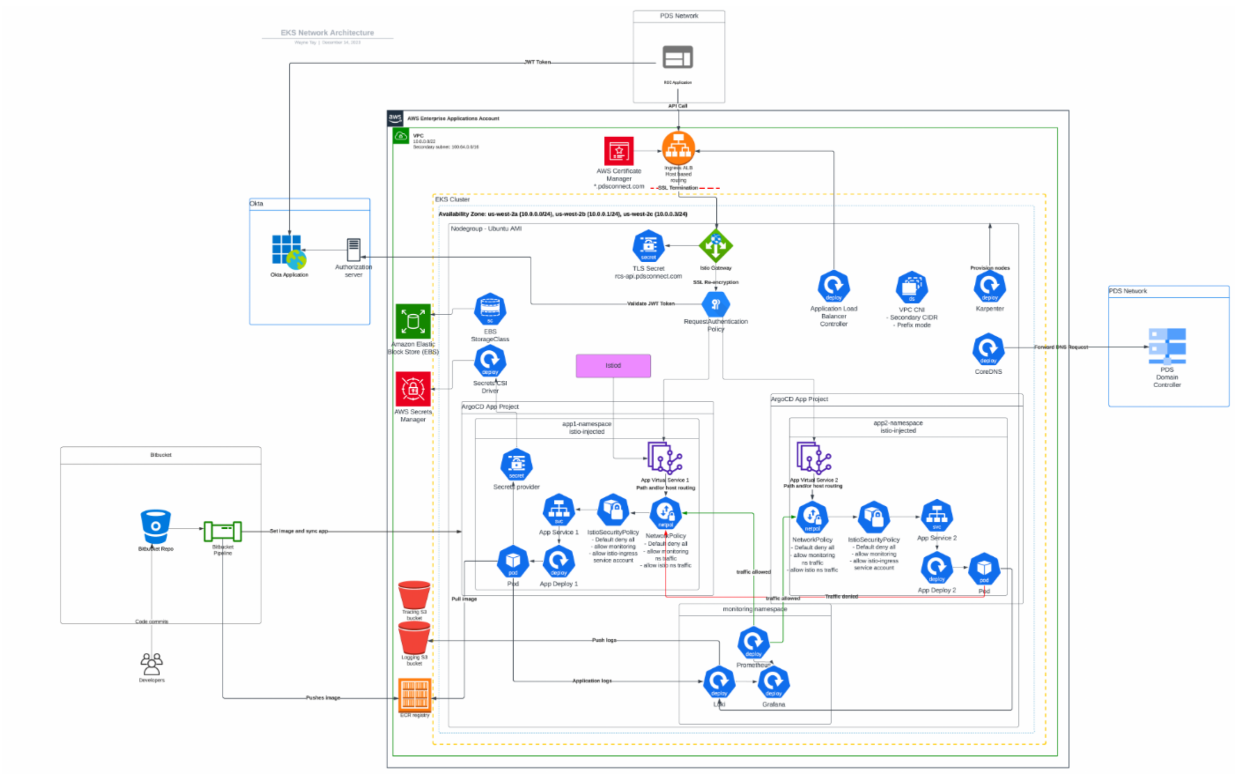
EKS - Cluster Optimization

The EKS Network Architecture document was provided by the customer, describes a Kubernetes-based infrastructure on AWS.



1. Current State Architecture

Key components and how they interact:

1. EKS Cluster: The central component is the Amazon EKS cluster, which manages Kubernetes worker nodes and pods.
2. Node Groups: The architecture utilizes Ubuntu AMI-based node groups, which are sets of EC2 instances that run the application containers.
3. VPC and Subnets: The EKS cluster is deployed within an AWS VPC (Virtual Private Cloud) with a primary CIDR of 10.0.0.0/22 and a secondary subnet of 100.64.0.0/16. Availability Zones are specified as us-west-2a, 2b, and 2c.
4. Load Balancers and Routing: An Ingress Application Load Balancer (ALB) is used with host-based routing and SSL termination. There's an Istio Gateway and SSL re-encryption for secure communication.
5. AWS Certificate Manager: Used for managing SSL certificates (*.pdsconnect.com).
6. Storage and Secrets Management: Amazon Elastic Block Store (EBS) is used for persistent storage, with a defined EBS StorageClass. AWS Secrets Manager and Secrets CSI Driver are used for managing sensitive data.
7. Networking and Security Policies: Istio is used for service mesh capabilities, along with Network Policies for controlling traffic and IstioSecurityPolicies for additional security layers.
8. CI/CD and Development Workflow: Bitbucket is used for source code management, with a pipeline for continuous

integration and deployment. Developers push changes to Bitbucket, triggering automated deployments.

9. Monitoring and Logging: The architecture includes Prometheus, Loki, and Grafana for monitoring, along with S3 buckets for tracing and logging.
10. Additional Components: ArgoCD for continuous delivery, CoreDNS for DNS management, and Karpenter for provisioning nodes dynamically.

2. Strategic Recommendations

1. Auto-Scaling: Ensure that Karpenter or the Horizontal Pod Autoscaler is properly configured for dynamic scaling based on load, which can optimize costs and performance.
2. Monitoring and Alerting: Strengthen monitoring and alerting mechanisms. Make sure that all critical metrics and logs are being monitored, and set up alerts for any anomalies.
3. Disaster Recovery: Implement a robust disaster recovery plan. This could include multi-region deployment, regular backups, and a clear rollback strategy.
4. Security Enhancements: Regularly audit security policies and IAM roles. Implement stricter network policies if necessary and ensure that all traffic is encrypted, both at rest and in transit.
5. Cost Optimization: Regularly review and optimize costs. This includes right-sizing resources, using spot instances where appropriate, and turning off unused resources.
6. Performance Tuning: Continuously monitor the performance and tune the configurations of the EKS cluster, databases, and other services for optimal performance.
7. CI/CD Improvements: Streamline and automate the CI/CD pipeline further to reduce deployment times and improve developer productivity.
8. Documentation and Training: Maintain comprehensive documentation of the architecture and ensure that the team is well-trained on Kubernetes and AWS best practices.
9. Use of Managed Services: Where possible, use AWS managed services to reduce the overhead of maintenance and management. *AWS managed prometheus and grafana*
10. Regular Audits: Conduct regular audits of the infrastructure for compliance, performance, and security.

3. App Monitoring & Logging Architecture

KEY COMPONENTS:

1. Grafana: Used for data visualization and analytics - central tool for observing the metrics and logs collected from various sources.
2. Prometheus: Used for monitoring and alerting.
3. Loki and Tempo: These are used for logging and tracing. Loki for structured logging, and Tempo integrates with Grafana for distributed tracing, using an S3 bucket for storage.
4. Grafana Agent and Promtail: They are likely used for collecting logs and metrics. The Grafana Agent might be collecting metrics, while Promtail focuses on log aggregation.
5. Microservices: The architecture probably involves various microservices emitting logs and metrics.
6. Kube-State-Metrics and Kubernetes API Server: These are standard Kubernetes components that provide metrics about the cluster state and are likely sources of monitoring data for Prometheus.
7. Storage Components: Uses EBS (Elastic Block Store) for persistent storage, likely for storing monitoring data and logs.

4. Optimization Recommendations for Monitoring

1. Refine Data Collection: Optimize data collection to gather only necessary metrics and logs to avoid data overload and increased costs.
2. Alerting and Anomaly Detection: Consider implementing anomaly detection for proactive issue identification.
3. Logging Levels and Retention: Set log retention policies to manage storage costs.
4. High Availability for Grafana and Prometheus: Ensure these services are highly available to avoid monitoring downtimes. Consider aws managed services.
5. Centralized Dashboard: Develop a centralized dashboard in Grafana that gives a holistic view of the entire infrastructure's health and performance.
6. Optimize Storage Usage: Regularly review and optimize the storage usage, especially for logs and metrics stored in EBS and S3, to control costs.
7. Tracing Enhancement: If you are using distributed tracing with Tempo, ensure it is effectively mapping the interactions between microservices for easier debugging and performance analysis.
8. Security and Access Control: Implement access controls and encryption for sensitive data in transit and at rest.
9. Integration with Incident Management: Integrate your monitoring setup with an *incident management system* for quicker response times.
10. Cost Management: Regularly review the costs associated with monitoring and logging, including storage and data transfer costs. Third party cost optimization tools?
11. Documentation and Training: Maintain up-to-date documentation for your monitoring architecture and ensure team members are trained in using and maintaining it.
12. Regular Auditing: Periodically audit the monitoring setup to ensure it aligns with your evolving application architecture and business needs.

5. Karpenter Fine tuning

How to optimize Karpenter taking action during spiky loads to avoid disruption across both stateful and stateless workloads. In this senario consider the following configurations for better control:

Pod Disruption Budgets (PDBs):

Define PDBs to limit the number of concurrent disruptions among stateful and stateless pods. This ensures that a minimum number of replicas are always running, thereby reducing disruptions.
Node Selector and Affinity Rules: Use node selector and affinity rules to ensure that certain pods (like those for databases) are scheduled on specific nodes that `aren't` frequently recycled by Karpenter.

Taints and Tolerations:

Apply taints to nodes meant for stateful workloads (like databases) to prevent stateless workloads from being scheduled on them, and use tolerations for stateful pods to be scheduled on these nodes.

Separate Node Groups:

Create `separate node groups` for stateful and stateless workloads. This way, Karpenter's actions on one group won't impact the other.

Karpenter Configuration:

Review and adjust Karpenter's provisioning and scaling policies. Fine-tune them to be more sensitive to the needs of stateful workloads, such as delaying scale-down actions.

Monitor and Alerting:

Implement robust monitoring and alerting systems to quickly identify and respond to potential disruptions caused by scaling actions.
These configurations help balance the dynamic scaling capabilities of Karpenter with the stability required for stateful and stateless workloads, ensuring minimal disruption during varying load conditions.

IMPLEMENTATION STEP

To implement the steps for better control during spiky loads with Karpenter in your EKS environment, follow these detailed actions:

Pod Disruption Budgets (PDBs):

- Define a PDB for each deployment, especially for stateful applications like databases.
- In the PDB manifest, specify `minAvailable` or `maxUnavailable` to control the minimum number of pods that should remain operational during voluntary disruptions.

Node Selector and Affinity Rules:

- Use `nodeSelector` in your pod specifications to assign pods to specific nodes.
- For more granular control, use `affinity` and `antiAffinity` rules to attract or repel pods from certain nodes.

Taints and Tolerations:

- Apply taints to nodes dedicated to specific workloads (e.g., stateful sets) using `kubectl taint nodes`.
- Add corresponding tolerations in the pod spec of the pods that should be scheduled on these nodes.

Separate Node Groups:

- Create distinct node groups in EKS for stateful and stateless workloads.
- Ensure these node groups have *different* scaling policies appropriate for their workloads.

Karpenter Configuration:

- Review Karpenter's scaling policies and adjust them to be more considerate of stateful workloads.
- Configure scaling thresholds and cooldown periods to minimize unnecessary scale-in and scale-out actions.

Monitoring and Alerting:

- Set up monitoring using tools like Amazon CloudWatch.

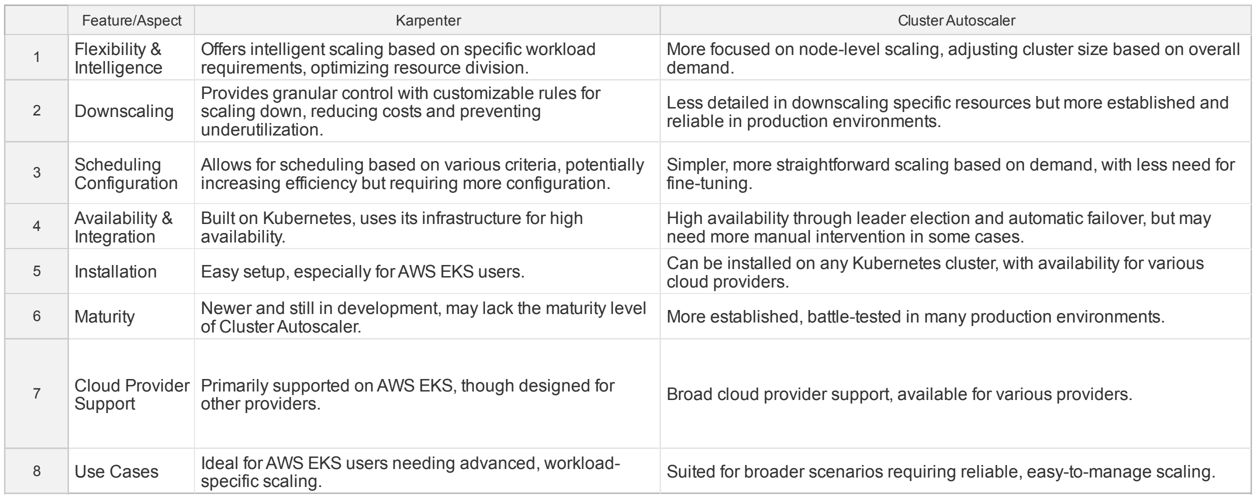
    - Create alerts for metrics that indicate potential scaling issues or disruptions to your workloads.

By following these steps, you can achieve a balance between efficient scaling and maintaining the stability and availability of both stateful and stateless workloads in your EKS environment.

6. Karpenter vs Cluster autoscaler

USE CASES

- Karpenter: Better for environments where workload-specific scaling and resource optimization are crucial. Ideal for AWS EKS users seeking advanced scaling features.
    - Checkout blog:
        - https://aws.amazon.com/blogs/containers/how-costar-uses-karpenter-to-optimize-their-amazon-eks-resources/
- Cluster Autoscaler: Suited for scenarios requiring a reliable, straightforward scaling solution that's easy to manage and has broad cloud provider support.
    - Checkout https://aws.github.io/aws-eks-best-practices/cluster-autoscaling/



	Feature/Aspect	Karpenter	Cluster Autoscaler
1	Flexibility & Intelligence	Offers intelligent scaling based on specific workload requirements, optimizing resource division.	More focused on node-level scaling, adjusting cluster size based on overall demand.
2	Downscaling	Provides granular control with customizable rules for scaling down, reducing costs and preventing underutilization.	Less detailed in downscaling specific resources but more established and reliable in production environments.
3	Scheduling Configuration	Allows for scheduling based on various criteria, potentially increasing efficiency but requiring more configuration.	Simpler, more straightforward scaling based on demand, with less need for fine-tuning.
4	Availability & Integration	Built on Kubernetes, uses its infrastructure for high availability.	High availability through leader election and automatic failover, but may need more manual intervention in some cases.
5	Installation	Easy setup, especially for AWS EKS users.	Can be installed on any Kubernetes cluster, with availability for various cloud providers.
6	Maturity	Newer and still in development, may lack the maturity level of Cluster Autoscaler.	More established, battle-tested in many production environments.
7	Cloud Provider Support	Primarily supported on AWS EKS, though designed for other providers.	Broad cloud provider support, available for various providers.
8	Use Cases	Ideal for AWS EKS users needing advanced, workload-specific scaling.	Suited for broader scenarios requiring reliable, easy-to-manage scaling.

Key Differences Highlighted:

- **Upscaling**: Karpenter scales more precisely based on usage and workload needs, while Cluster Autoscaler focuses on adding more nodes as per increased demand.
- **Downscaling**: Karpenter's downscaling is more detailed and customizable, compared to the broader approach of Cluster Autoscaler.
- **Scheduling**: Karpenter offers more advanced scheduling options, requiring specific configurations, whereas Cluster Autoscaler's scheduling is simpler and more general.

This comparison shows that while Karpenter offers more advanced and customizable features, particularly for AWS environments, Cluster Autoscaler provides a simpler, more universally applicable scaling solution.

Example scenario involves an e-commerce application hosted on EKS (Elastic Kubernetes Service) that needs efficient auto-scaling to handle peak load during a high-traffic event. The requirements are:

1. Elasticity to handle peak loads.
2. Efficient bin-packing and right-sizing of EC2 instances to maintain cost efficiency.
3. Scaling based on the number of HTTP requests per second specifically for the Order Service.

Karpenter & HPA (Horizontal Pod Autoscaler) with Prometheus Adapter:

- Karpenter is an open-source Kubernetes cluster autoscaler designed to quickly launch right-sized EC2 instances based on workload needs, which aligns with your efficiency and bin-packing requirements.
- HPA with Prometheus Adapter allows scaling of pods in a deployment based on observed metrics. Prometheus can be configured to **track the number of HTTP** requests per second, meeting specific scaling criteria for the Order Service for example.

Karpenter & VPA (Vertical Pod Autoscaler):

- VPA adjusts the CPU and memory resources of pods, which is useful for right-sizing but doesn't directly address your requirement to scale based on HTTP requests per second.

KEDA (Kubernetes Event Driven Autoscaling):

- KEDA is designed to handle event-driven scaling, including HTTP requests. It can scale both Kubernetes workload (pods) and the underlying infrastructure if properly configured. However, KEDA might require more complex setup for integration with AWS services compared to using AWS-native solutions like Karpenter.

Karpenter with Metrics Server and HPA:

- The Metrics Server collects resource metrics from Kubelets and exposes them in Kubernetes apiserver for use by HPA. While this setup would efficiently scale the pods based on standard metrics like CPU and memory usage, it may not directly scale based on the number of HTTP requests per second unless custom metrics are configured.

Based on the example requirements, you may want to look at Karpenter & HPA with Prometheus Adapter. This combination provides the elasticity and efficiency required for your EKS-hosted application and allows for scaling based on the specific metric of HTTP requests per second, which is crucial for your Order Service during peak times like Christmas.

Here's a table outlining the differences and use cases for each of the auto-scaling mechanisms mentioned:

	Scaling Mechanism	Key Features	Use Cases
1	Karpenter & HPA with Prometheus Adapter	- Fast and efficient instance provisioning. - Custom metric scaling with Prometheus. - HPA scales pods based on metrics.	- Ideal for applications needing scaling based on custom metrics like HTTP requests. - Useful when precise, metric-driven scaling decisions are required. - Efficient instance management during variable workloads.
2	Karpenter & VPA	- Efficient instance provisioning. - VPA adjusts pod resources.	- Suitable for applications with varying resource requirements per pod. - Good when pods need resizing rather than increasing the number of pods. - Not ideal for scaling based on specific metrics like HTTP requests.
3	KEDA	- Event-driven scaling. - Scales both Kubernetes workload and infrastructure.	- Best for event-driven applications (e.g., queues, HTTP requests). - Useful when scaling needs to be triggered by external events or metrics. - Requires more setup for integration with AWS services.
4	Karpenter with Metrics Server and HPA	- Efficient instance provisioning. - Standard metric scaling with HPA.	- Good for applications where standard metrics like CPU and memory usage are the primary scaling indicators. - Suitable for straightforward scaling needs without the complexity of custom metrics.

Each of these mechanisms has its strengths and specific scenarios where it shines. The choice largely depends on the particular scaling needs of the application, including the metrics that are most relevant for scaling decisions and the complexity of the scaling requirements.

7. Istio and AWS App Mesh

Both Istio and AWS App Mesh provide service mesh functionality, but they are designed and integrated differently within the AWS ecosystem. Here's a comparison of their key features.
AWS App Mesh is a managed service providing application-level networking and service communication support for Microservices architecture. It also uses Envoy proxy making it compatible with multiple solution approaches. It supports h AWS Fargate, Amazon EC2, Amazon ECS, Amazon EKS, and Kubernetes running on AWS.

	Feature/Aspect	Istio	AWS App Mesh
1	Primary Function	Open-source service mesh providing traffic management, security, and observability for microservices.	AWS-managed service mesh for traffic management, security, and observability within microservices.
2	Scope	Works with various platforms, including Kubernetes, VMs, and other cloud environments.	Primarily focused on AWS, seamlessly integrates with AWS services like ECS, EKS, EC2, and Fargate.
3	Traffic Management	Provides advanced traffic management features like canary deployments, circuit breakers, A/B testing, and fault injection.	Offers similar traffic routing capabilities including canary deployments, retries, and timeouts.
4	Security	Supports strong identity and security policies with mutual TLS (mTLS) for service-to-service encryption.	Provides mTLS for secure communication between services, integrated with AWS security services like IAM.
5	Observability	Offers detailed telemetry, logging, and tracing for monitoring microservices.	Integrates with AWS CloudWatch for logging and monitoring, providing insights into service performance.
6	Ease of Use	Can be complex to configure and manage, especially in multi-cloud or hybrid environments.	Designed to be simpler to configure within the AWS ecosystem, leveraging AWS management tools.
7	Flexibility	Highly customizable and flexible, suitable for complex, cross-platform environments.	Optimized for AWS environments, might be less flexible for cross-platform scenarios.

8	Community Support	Broad community support and wide adoption in various environments.	Strong support within the AWS ecosystem, continuous improvements and updates from AWS.

"Must-Have" and "Nice-to-Have"

Breaking down the recommendations for the EKS architecture into "Must-Have" and "Nice-to-Have" categories will help prioritize the changes. Here's a table outlining these categories along with detailed implementation steps for each recommendation:

	Category	Recommendation	Implementation Steps
1	Must-Have	Auto-Scaling	1. Configure Horizontal Pod Autoscaler in Kubernetes. 2. Set up Karpenter for node-level auto-scaling. 3. Define metrics for scaling (CPU, memory usage). 4. Test auto-scaling with simulated load.
2	Must-Have	Security Enhancements	1. Conduct an IAM roles audit. 2. Implement strict Network Policies in Kubernetes. 3. Use AWS Shield for DDoS protection. 4. Regularly update AMIs and Kubernetes for security patches.
3	Must-Have	Disaster Recovery	1. Set up cross-region EKS clusters for failover. 2. Implement regular backups of EKS data and configurations. 3. Test failover procedures.
4	Must-Have	Monitoring and Alerting	1. Integrate Prometheus with EKS for metrics collection. <br>2. Set up Grafana for visualizations. <br>3. Define alert rules in Prometheus. <br>4. Connect alerts to notification channels (e.g., Slack, email).
5	Must-Have	Cost Optimization	1. Analyze current resource usage with AWS Cost Explorer. <br>2. Implement spot instances for eligible workloads. <br>3. Right-size resources based on utilization metrics. <br>4. Turn off or scale down non-essential resources during off-peak hours.
6	Nice-to-Have	CI/CD Improvements	1. Automate deployment pipelines using Jenkins or AWS CodePipeline. <br>2. Implement blue-green or canary deployments for safer rollouts. <br>3. Use Infrastructure as Code (IaC) for consistent environment setups.
7	Nice-to-Have	Performance Tuning	1. Profile application performance using AWS X-Ray or similar tools. <br>2. Optimize application code and resources based on findings. <br>3. Implement caching where applicable (e.g., Redis). <br>4. Regularly update to the latest stable Kubernetes and Docker versions for performance improvements.
8	Nice-to-Have	Use of Managed Services	1. Migrate databases to Amazon RDS or Aurora. <br>2. Use Amazon ECR for Docker container registry. <br>3. Consider Amazon MSK for Kafka workloads.
9	Nice-to-Have	Regular Audits	1. Schedule quarterly security and compliance audits. <br>2. Conduct performance and cost audits bi-annually. <br>3. Use AWS Trusted Advisor for ongoing recommendations.
10	Nice-to-Have	Documentation and Training	1. Create detailed documentation of the EKS setup. <br>2. Conduct regular training sessions for the team on Kubernetes best practices and AWS services. <br>3. Keep documentation and training materials updated with latest changes and technologies.

ADDITIONAL NOTES:

- Prioritization: The "Must-Have" recommendations are crucial for the stability, security, and efficiency of your EKS architecture. These should be implemented as soon as possible. The "Nice-to-Have" items, while beneficial, can be implemented as per the available resources and timelines.
- Customization: Tailor the implementation steps based on your specific environment, workload types, and organizational policies.
- Testing and Validation: Ensure thorough testing and validation at each step of implementation to avoid disruptions.
- Continuous Improvement: Regularly revisit your architecture for potential improvements and updates.

By following these steps, you can systematically enhance and optimize your EKS architecture.

8. Review of Excel document results

MUST-HAVE RECOMMENDATIONS

Use layered Constraints for Karpenter: Implement this to ensure pods are scheduled across multiple availability zones for higher availability.

- Implementation: Define a Provisioner with requirements for each AZ and label your pods accordingly to guide Karpenter in scheduling.

Ensure Pods are Deregistered from Load Balancers *before* Termination: Crucial for zero downtime deployments.

- Implementation: Implement a `preStop` hook in your pod specifications that deregisters the pod from the load balancer *before* the pod is terminated.

Implement Readiness and Liveness Probes: Vital for ensuring traffic is only sent to ready pods and unhealthy pods are restarted.

- Implementation: Define `readinessProbe` and `livenessProbe` in your pod specifications.

Horizontal Pod Autoscaler (HPA): Important for automatic scaling of applications based on demand.

- Implementation: Deploy HPA for your deployments, setting appropriate metrics for scaling.

Rollback Mechanism in CI/CD: Essential for reverting to a previous stable state in case of deployment failures.

- Implementation: Incorporate rollback strategies in your deployment pipelines, such as using Helm's rollback features. ArgoCD is used today.

Pod Disruption Budgets (PDB): Necessary to prevent accidental deletion of critical pods.

- Implementation: Create PDBs for your deployments to ensure a minimum number of replicas are always running.

Implement Node Problem Detector: To enhance the reliability of the nodes.

- Implementation: Deploy the Node Problem Detector in your cluster to automatically detect and handle node issues.

Monitor CoreDNS Metrics: Critical for DNS reliability.

- Implementation: Set up monitoring for CoreDNS latency and failure metrics using Prometheus.

NICE-TO-HAVE RECOMMENDATIONS

Chaos Engineering: Helps in identifying failure points.

- Implementation: Introduce tools like Gremlin in a controlled environment to test cluster resilience.

Use Blue/Green or Canary Deployments: For safer deployment strategies.

- Implementation: Set up your deployment process to support blue/green or canary strategies.

Multi-Stage Docker Builds: To create minimal and secure container images.

- Implementation: Refactor your Dockerfiles to use multi-stage builds, separating the build environment from the runtime environment.

Regularly Scan Images for Vulnerabilities: To maintain security.

- Implementation: Set up automated scanning of container images in your CI/CD pipeline or upon push to the registry.

Limit Dynamic Admission Webhooks: To avoid API request delays.

- Implementation: Review and optimize the number of webhooks and their failure policies.

For each of these recommendations, it’s essential to plan, review the current state, implement changes, test in a development environment, and monitor the impact after deployment to production. Automation where possible is key to maintaining consistency and reliability. Ensure all changes are documented and that team members are trained on new procedures.

9. Implementation Steps (Must have)

1. USE LAYERED CONSTRAINTS FOR KARPENTER

OBJECTIVE: SCHEDULE PODS ACROSS MULTIPLE AVAILABILITY ZONES (AZS) TO ENHANCE HIGH AVAILABILITY.

Steps:

- Open the AWS Management Console.
- Navigate to the Amazon EKS section and select your cluster.
- Install Karpenter on your cluster if not already installed.
- Create a Provisioner specification that includes constraints for each AZ.
- Label your pods with corresponding AZ labels.
- Apply the Provisioner configuration using `kubectl apply`.

2. ENSURE PODS ARE DEREGISTERED FROM LOAD BALANCERS BEFORE TERMINATION

OBJECTIVE: GRACEFULLY DEREGISTER PODS FROM THE LOAD BALANCER TO MAINTAIN SERVICE AVAILABILITY DURING UPDATES OR TERMINATIONS.

Steps:

- Update your pod specifications to include a `preStop` lifecycle hook.
- The `preStop` hook should include commands to deregister the pod from the load balancer.
- Apply the updated pod specifications to your deployments.

3. IMPLEMENT READINESS AND LIVENESS PROBES

OBJECTIVE: ENSURE THAT TRAFFIC IS DIRECTED ONLY TO PODS THAT ARE READY AND THAT UNHEALTHY PODS ARE RESTARTED.

Steps:

- Define `readinessProbe` in your pod specification with appropriate settings (HTTP GET, TCP Socket, or Exec).
- Define `livenessProbe` in your pod specification to check the health of your application.
- Configure the probes with initial delays and period checks according to your application startup time and health check requirements.
- Apply the updated pod specifications.

4. HORIZONTAL POD AUTOSCALER (HPA)

OBJECTIVE: AUTOMATICALLY SCALE APPLICATIONS BASED ON DEMAND.

Steps:

- Ensure metrics-server is installed in your cluster for HPA to retrieve metrics.
- Create an HPA resource defining the minimum and maximum number of pod replicas, and the metrics to scale (CPU

utilization, custom metrics).
- Apply the HPA configuration using `kubectl apply`.

5. ROLLBACK MECHANISM IN CI/CD

OBJECTIVE: QUICKLY REVERT TO A STABLE RELEASE IF A NEW DEPLOYMENT FAILS.

Steps:

- Implement a CI/CD pipeline that supports rollback, using tools like Jenkins, GitLab CI, or AWS CodePipeline.
- Include steps in the pipeline to deploy new versions.
- Include a rollback step that can be triggered manually or automatically based on failure conditions.
- Test the rollback mechanism in a non-production environment.

6. POD DISRUPTION BUDGETS (PDB)

OBJECTIVE: ENSURE THAT A MINIMUM NUMBER OF POD REPLICAS ARE ALWAYS RUNNING.

Steps:

- Create a PDB manifest file specifying the minimum available replicas or maximum unavailable replicas.
- Apply the PDB to your deployments using `kubectl apply`.
- Monitor the PDB status with `kubectl get pdb`.

7. IMPLEMENT NODE PROBLEM DETECTOR

OBJECTIVE: DETECT AND HANDLE NODE ISSUES TO IMPROVE NODE RELIABILITY.

Steps:

- Consider installing the Node Problem Detector on your cluster
  - https://kubernetes.io/docs/tasks/debug/debug-cluster/monitor-node-health/
- Configure the Node Problem Detector to monitor specific logs or system conditions.
  - https://github.com/kubernetes/node-problem-detector/blob/master/README.md
- Set up automated actions such as node repair or replacement based on detected problems.

8. MONITOR COREDNS METRICS

OBJECTIVE: ENSURE THE DNS SERVICE WITHIN THE CLUSTER IS FUNCTIONING CORRECTLY.

Steps:

- Enable Prometheus metrics for CoreDNS if not already enabled.
- Configure Prometheus to scrape CoreDNS metrics.
- Set up alerts in Prometheus for high latency or failure metrics.
- Create a Grafana dashboard for visualizing CoreDNS metrics.

These steps are high-level and will need to be adjusted to the specifics of your environment and applications. It is also essential to perform these changes in a controlled manner, preferably starting in a staging environment before promoting to production. Make sure to test each change thoroughly and have monitoring in place to detect any unforeseen issues.

10. Implementation Steps (Nice to have)

1. CHAOS ENGINEERING

OBJECTIVE: TEST THE RESILIENCE OF YOUR KUBERNETES CLUSTER AND UNCOVER SINGLE POINTS OF FAILURE.

Steps:

- Select a chaos engineering tool suitable for Kubernetes, like Gremlin or Chaos Mesh.
- Install the chaos engineering tool following the provider's instructions.
- Define chaos experiments, which could include pod failures, network latency, or resource exhaustion.
- Start with small, controlled experiments in a non-production environment.
- Gradually increase the scope and impact of your experiments as you become more confident.
- Monitor the system's response to experiments and refine your recovery processes accordingly.

2. BLUE/GREEN OR CANARY DEPLOYMENTS

OBJECTIVE: DEPLOY NEW APPLICATION VERSIONS WITH MINIMAL RISK AND EASY ROLLBACK CAPABILITIES.

Steps:

- Define two separate environments: one for the current (blue) version and one for the new (green) version.
- Deploy the new version to the green environment without affecting the blue environment.
- Gradually shift traffic to the green environment according to your strategy (instant for blue/green or incremental for canary).
- Monitor the new version's performance and error rates closely.
- If an issue is detected, shift traffic back to the blue environment.
- Once confident in the new version, decommission the blue environment.

3. MULTI-STAGE DOCKER BUILDS

OBJECTIVE: CREATE MINIMAL AND SECURE DOCKER IMAGES.

Steps:

- Update your Dockerfile to a multi-stage format, defining a build stage with all the necessary build tools and dependencies.
- Copy only the artifacts needed for running the application into a final stage based on a minimal base image.
- Configure your build pipeline to build the Docker image using the updated Dockerfile.
- Push the built image to your container registry.
- Deploy the image to a test environment to verify its functionality.

4. REGULARLY SCAN IMAGES FOR VULNERABILITIES

OBJECTIVE: IDENTIFY AND FIX SECURITY VULNERABILITIES WITHIN CONTAINER IMAGES.

Steps:

- Choose a container image scanning tool like Trivy, Clair, or the scanning features provided by ECR.
- Integrate the scanning tool into your CI/CD pipeline so that images are scanned after each build or before deployment.

- Configure the scanner to report vulnerabilities and fail the build if critical issues are found.
- Review and address the reported vulnerabilities by updating dependencies or altering the Dockerfile.

5. LIMIT DYNAMIC ADMISSION WEBHOOKS

OBJECTIVE: REDUCE API REQUEST DELAYS CAUSED BY WEBHOOKS.

Steps:

- Audit your existing webhooks to understand their purpose and necessity.
- Remove any unnecessary webhooks.
- Ensure that webhooks have appropriate timeout settings to prevent long delays.
- Configure webhooks with a `failurePolicy` that aligns with your reliability requirements.
- Test the behavior of your webhooks in a staging environment to ensure they do not introduce unacceptable latency or failures.

Links

EKS OSS Metrics Best Practices:
https://aws-observability.github.io/observability-best-practices/guides/containers/oss/eks/best-practices-metrics-collection/

EKS Native observability best practices :
https://aws-observability.github.io/observability-best-practices/guides/containers/aws-native/eks/amazon-cloudwatch-container-insights/

ECS Native best practices :
https://aws-observability.github.io/observability-best-practices/guides/containers/aws-native/ecs/best-practices-metrics-collection-1/

ECS OSS Best Practices:
https://aws-observability.github.io/observability-best-practices/guides/containers/oss/ecs/best-practices-metrics-collection-1/