Helm

Helm Usage Guidelines for Kubernetes Deployment

Helm is the approved tool for managing Kubernetes deployments across ABC, providing a consistent, scalable, and secure approach to packaging, configuring, and deploying applications. These guidelines establish the mandatory practices that all teams must follow when adopting Helm. They ensure that deployments are streamlined, auditable, and aligned with the organization's security and operational standards across environments.

1. Centralized Helm Chart Repository

ABC operates a centralized Helm chart repository called `abc-helm`. This repository is the single source of truth for all Helm templates and charts used across different environments and projects. Storing Helm charts outside of `abc-helm` is prohibited.

- Centralization: All Helm charts, including base templates, custom configurations, and environment-specific overrides, must be stored in the `abc-helm` repository. This centralization guarantees that all teams have access to the same standardized Helm charts, which have been reviewed for security, consistency, and version control.

- Submission Process: Before a new Helm chart is deployed in any environment, it must be submitted for review. The architecture team will assess all submissions to ensure they comply with organizational standards for security, scalability, and configuration management. Once approved, the chart is added to `abc-helm` for organizational use.

- Update Process: Any updates or modifications to an existing Helm chart must follow the same submission and review process, ensuring that changes are tracked, reviewed, and approved before they are applied in any production environment. This provides full traceability for every deployment.

- Access Control: Strict access control policies must be in place for managing `abc-helm`. Only authorized personnel are allowed to submit, modify, or approve Helm charts. This ensures that charts in the repository meet security and compliance standards.

2. Modular Helm Chart Design

ABC adopts a modular approach to Helm chart design, leveraging a unified template called `unified-helm-template` to cover most standard application deployment scenarios, including creating Kubernetes resources like pods, services, config maps, and persistent volumes.

- Standard Template: All teams are required to use `unified-helm-template` as the base for application deployments. This template is designed to be flexible, with customizable parameters to ensure consistency across all environments while accommodating specific deployment needs.

- Customization: If `unified-helm-template` does not meet the unique needs of a specific application or service, a formal requirement change request must be submitted. This request should clearly outline the limitations of the current template and explain why the changes are necessary. The architecture team will review and approve any valid customization requests, and approved changes will be integrated into the template.

- Reusability: Modular Helm charts allow individual components (e.g., databases, backend services, frontend services) to be managed, updated, and scaled independently. This modular approach also facilitates the reuse of components across projects and environments, improving efficiency and reducing duplication of effort.

3. Parameterization via Values.yaml

Helm charts must be parameterized through the use of `values.yaml` files. This practice ensures that the same Helm chart can be deployed across multiple environments (development, staging, production) with environment-specific configurations.

- Base Configuration: The base `values.yaml` file should contain default settings that apply across all environments, including resource allocations, image names, and other general configurations.

- Environment-Specific Configuration: Separate override files (e.g., `values-dev.yaml`, `values-prod.yaml`) must be created for each environment. These files should include environment-specific values, such as replica counts, resource limits, database credentials, and region-specific configurations. The environment-specific override files will be hosted in the user’s repository to ensure that environments are correctly configured without modifying the base Helm chart.

- No Hardcoding: Environment-specific values, including credentials and sensitive information, must not be hardcoded directly into the Helm charts or `values.yaml` files. Secrets should be managed securely (see Section 5), and Helm charts should remain flexible and scalable across different environments without manual modifications.

- Dynamic Configuration: Helm’s templating system should be utilized to dynamically generate Kubernetes manifests based on values provided in `values.yaml` files. This practice ensures flexibility in deploying different configurations without altering the Helm chart itself.

4. Versioning and Release Management

Helm charts must adhere to semantic versioning to ensure proper version control and traceability of changes. This ensures that every deployment is linked to a specific version of the Helm chart, making it easier to roll back changes when necessary.

- Version Format: Helm charts must follow the semantic versioning format: `MAJOR.MINOR.PATCH`:

- Major version (`X.y.z`): For changes that break backward compatibility or introduce significant architectural updates.

- Minor version (`x.Y.z`): For backward-compatible feature additions.

- Patch version (`x.y.Z`): For bug fixes, minor enhancements, and backward-compatible updates.

- Versioning Automation: Helm chart versioning is fully integrated into the organization’s CI/CD pipeline. Each time a change is made to a Helm chart, the pipeline will automatically increment the version number and push the updated artifact to JFrog Artifactory. This process ensures that no two deployments use the same version unless explicitly intended.

- Release Tracking: Every Helm chart version must be tagged and logged in the `abc-helm` repository, allowing teams to track the exact version of the chart used in each environment. This ensures full traceability of deployments.

- Rollback Capabilities: In case of a failed deployment, the CI/CD pipeline is configured to automatically roll back to the previous stable version of the Helm chart. This minimizes downtime and ensures that services remain available during error recovery.

5. Security: Secret Management

Helm charts must comply with the organization's security policies, with particular attention to secure secret management.

- Secret Management: Sensitive data such as API keys, passwords, and database credentials must never be hardcoded into Helm charts or `values.yaml` files. All secrets must be securely stored and managed using the organization’s `ABC Credentials Manager`. This tool integrates with Kubernetes secret management features to securely inject secrets into Kubernetes environments at runtime, ensuring that sensitive information is never exposed in configuration files.

- Audit and Compliance: Regular security audits must be performed on Helm charts as part of the review process. Security best practices, including encryption, RBAC enforcement, and proper secret management, must be verified before charts are approved for production use.

6. Deployment via Approved CI/CD Pipeline

All Helm chart deployments must be executed exclusively through ABC’s approved CI/CD pipeline. Manual Helm deployments in staging or production environments are strictly prohibited.

- Pipeline Automation: The CI/CD pipeline is responsible for the entire deployment process, including validating Helm charts, applying environment-specific customizations, performing security checks, and managing version control. Automated deployments ensure that all deployments across the organization follow the same rules and guidelines.

- No Manual Deployments: Manual execution of Helm commands (`helm install`, `helm upgrade`, etc.) is not permitted. This ensures that all deployments are logged, auditable, and consistent with organizational policies.

- Continuous Monitoring: The CI/CD pipeline integrates with the organization’s monitoring tool (e.g., XYZ Monitoring) and alerting system, ensuring that any issues during deployment are immediately detected and flagged. Failed deployments are automatically rolled back to the last stable state.

Thank you for clarifying! Given that `unified-helm-template` is one of the templates available, and teams can choose from other approved templates in the `abc-local` Artifactory, here’s a revised version of the Helm Patterns to reflect this.

---

Helm Usage Patterns

1. Approved Helm Template Pattern

- Description: Teams must select from the approved Helm templates available in `abc-local Artifactory` for their deployments. One of the key templates, `unified-helm-template`, is commonly used for most application scenarios, but other approved templates may be more suitable depending on the application architecture.

- Implementation:

- Teams must review the available Helm templates stored in `abc-local Artifactory` and select the one that aligns with their application's requirements.

- Once a template is selected, it should be customized using `values.yaml` without altering the core Helm template.

- If no approved template fully satisfies the application’s needs, teams must submit a request to the architecture team for a new template or modification to an existing one.

- Advantages:

- Ensures standardization and compliance with organizational guidelines.

- Reduces duplication of effort by allowing teams to use pre-approved, validated templates.

- Promotes consistency in deployment strategies across teams and projects.

---

2. Modular Helm Chart Design Pattern

- Description: Helm charts should be modular, with different components of the application (e.g., databases, APIs, frontend services) handled by individual charts. This ensures that each component can be independently managed, scaled, and updated without impacting others.

- Implementation:

- Create separate Helm charts for different microservices or components of the application (e.g., `api-chart`, `frontend-chart`, `db-chart`).

- Use Helm’s dependency management features when services must interact but avoid over-complicating charts with too many dependencies.

- Modular charts should be versioned and managed independently, allowing teams to deploy and manage components without tightly coupling them together.

- Advantages:

- Supports independent lifecycle management, allowing each service or component to be updated or scaled independently.

- Encourages reusability, as modular charts can be used across multiple projects.

- Improves maintainability by breaking down complex deployments into manageable components.

---

3. Environment-Specific Configuration Pattern

- Description: Externalize environment-specific configurations using `values.yaml` files. This pattern ensures that a single Helm chart can be reused across environments (e.g., development, staging, production) with minimal changes.

- Implementation:

- Use a base `values.yaml` file for common configurations across all environments.

- Create separate override files (e.g., `values-dev.yaml`, `values-staging.yaml`, `values-prod.yaml`) for environment-specific configurations like replica counts, resource limits, or secrets.

- Host the `values.yaml` files in the appropriate version control repository alongside the Helm charts to allow easy access and management.

- Advantages:

- Ensures consistency across environments while allowing for environment-specific customizations.

- Prevents duplication of Helm charts for different environments, making them easier to manage and maintain.

- Reduces the risk of errors when deploying to different environments by keeping environment-specific values clearly separated.

---

4. Semantic Versioning and CI/CD Integration Pattern

- Description: All Helm charts must adhere to semantic versioning to ensure that changes are traceable and deployments can be rolled back if necessary. The CI/CD pipeline must automate versioning and ensure that every deployment is properly version-controlled and logged.

- Implementation:

- Versioning should follow the `MAJOR.MINOR.PATCH` convention:

- Major version (`X.y.z`): For changes that introduce backward-incompatible changes.

- Minor version (`x.Y.z`): For new features that are backward-compatible.

- Patch version (`x.y.Z`): For bug fixes or minor changes.

- The CI/CD pipeline must automate version increments and ensure that each deployment is tagged with the correct version and pushed to `abc-local Artifactory`.

- Track all versions in the `abc-helm` repository for auditability and rollback purposes.

- Advantages:

- Provides clear traceability of changes.

- Simplifies rollbacks in case of deployment issues.

- Ensures that all teams can see and track the version history for their deployments.

---

5. Secrets Management Pattern

- Description: Secrets and sensitive information (e.g., API keys, passwords) must never be hardcoded in Helm charts or `values.yaml` files. All sensitive data should be securely managed through `ABC Credentials Manager` and injected into the Kubernetes environment at runtime.

- Implementation:

- Use the organization’s `ABC Credentials Manager` to securely manage secrets, which should then be referenced in the Helm chart using Kubernetes-native secret resources.

- Ensure that secrets are injected into the application at runtime, avoiding the exposure of sensitive information in configuration files.

- Validate that secrets are properly encrypted and managed through regular audits.

- Advantages:

- Protects sensitive data from being exposed in repositories or during deployments.

- Centralizes and secures secret management, reducing the risk of data breaches.

- Simplifies compliance with organizational security policies and regulations.

---

6. Helm Chart Linting and Testing Pattern

- Description: Helm charts must be validated through automated linting and testing to ensure they meet organizational standards and don’t introduce errors into the deployment process.

- Implementation:

- Integrate `helm lint` into the CI/CD pipeline to check Helm chart syntax and structure for common issues.

- Use testing tools such as `Kubeval` or `Helm Test` to validate the chart’s behavior against a Kubernetes cluster before deploying to production.

- Ensure that all Helm charts pass through linting and testing stages before being approved for deployment.

- Advantages:

- Catches configuration issues early, reducing the likelihood of deployment failures.

- Ensures that all Helm charts meet best practices and organizational standards.

- Provides confidence that deployments will behave as expected in production environments.

---

7. Independent Lifecycle Management of Helm Releases Pattern

- Description: Each Helm release should have its own independent lifecycle management. This pattern encourages isolated changes, ensuring that updates or scaling operations on one service do not interfere with others.

- Implementation:

- Each service or component (e.g., `api-chart`, `db-chart`) should have a distinct Helm release name, allowing independent updates, scaling, and rollbacks.

- Use namespaces in Kubernetes to further isolate different services if necessary.

- Manage versioning and configuration for each release independently, allowing for fine-grained control over each service.

- Advantages:

- Reduces risk by isolating services, making it easier to troubleshoot and manage individual components.

- Supports independent scaling of components without affecting other services.

- Allows for flexibility in deployment schedules and update management.

### **Helm Anti-Patterns**

#### **1. Monolithic Helm Charts**

* **Description**: Creating a single Helm chart that deploys multiple services or components together (e.g., bundling databases, backend, and frontend services in one chart).
* **Consequence**:
  + Limits flexibility in scaling or upgrading individual components.
  + Increases the risk of deployment failures due to dependencies between services.
  + Makes maintenance difficult and error-prone.
* **Alternative**: Use modular Helm charts to deploy individual components separately.

#### **2. Hardcoding Environment-Specific Values**

* **Description**: Storing environment-specific configurations (e.g., database URLs, resource limits, credentials) directly in Helm charts or hardcoding them in values.yaml.
* **Consequence**:
  + Reduces reusability of the Helm chart across environments.
  + Increases the risk of security vulnerabilities, especially when secrets are hardcoded.
  + Leads to configuration drift across environments.
* **Alternative**: Use values.yaml files to externalize environment-specific values and secure secret management systems for sensitive data.

#### **6. Too Many Conditional Statements in Charts**

* **Description**: Overloading Helm charts with excessive conditional logic (if/else/ternary) to handle too many configuration differences between environments or use cases.
* **Consequence**:
  + Creates complex and hard-to-read charts, making them difficult to debug and maintain.
  + Increases the risk of unintended behavior if the conditions are not thoroughly tested.
  + Leads to reduced readability and maintainability.
* **Alternative**: Simplify charts by externalizing complex configurations in values.yaml files and using separate Helm charts for significantly different use cases or environments.

#### **7. Directly Modifying Generated Manifests**

* **Description**: Editing Kubernetes manifests that are generated by Helm templates rather than modifying the source Helm templates or values.
* **Consequence**:
  + Makes deployments unpredictable and inconsistent because changes bypass Helm's version control and chart management.
  + Removes the ability to trace changes via Helm’s versioning and review process.
  + Leads to configuration drift and inconsistency between environments.
* **Alternative**: Modify the Helm templates or values.yaml files instead of directly editing generated manifests. Always deploy through Helm to maintain consistency.

#### **8. Overuse of Helm Dependencies**

* **Description**: Relying too heavily on Helm’s dependency management to include other charts without fully understanding their impacts.
* **Consequence**:
  + Increases complexity and coupling between services.
  + Makes troubleshooting difficult if a failure occurs within a dependent chart.
  + Introduces the risk of version mismatches and incompatibility between charts.
* **Alternative**: Limit the use of dependencies to well-known and essential services. For custom integrations, prefer managing services independently to maintain clear control over versions and configurations.

#### **9. Insufficient Helm Linting and Testing**

* **Description**: Skipping or insufficiently using helm lint or automated tests to validate Helm charts before deployment.
* **Consequence**:
  + Increases the likelihood of configuration errors being introduced into production.
  + Leads to deployment failures, downtime, or misconfigurations that could have been caught earlier.
* **Alternative**: Integrate helm lint and other testing tools into the CI/CD pipeline to ensure that charts are validated before being deployed.

#### **10. Overly Complex Values Files**

* **Description**: Having values.yaml files that are excessively large and complex, containing hundreds of lines of configuration for many different environments or use cases.
* **Consequence**:
  + Reduces readability and maintainability, making it hard for teams to identify relevant configuration sections.
  + Increases the risk of configuration errors or mismatches, especially when different teams or environments are involved.
* **Alternative**: Split large values.yaml files into more manageable and targeted files for each environment or use case is an alternative approach to avoid big yaml files. Keep the base values.yaml file small and focused on common defaults, using separate files for specific environment overrides.

**Terraform Modularization** allows infrastructure to be defined in reusable, self-contained components (modules) that can be applied across different environments or projects. This ensures consistency, scalability, and maintainability by avoiding code duplication and enabling updates to be made in one place. All modules should be stored in the centralized repository **"terraform-centralized-repo"**, which houses the source code for these modules. After being reviewed and tested, modules are versioned and stored in **JFrog Artifactory**, allowing consumers to directly pull and use these modules with specified release versions. This approach ensures that only approved, version-controlled modules are utilized across teams, promoting standardization and reducing deployment errors.

### **Module Registry Management: Security and Compliance**

**Module Registry Management** is a critical component in safeguarding the security, compliance, and integrity of Terraform-managed infrastructure. Proper management of Terraform modules — including access control, versioning, and auditing — is essential to preventing unauthorized changes, maintaining consistent infrastructure, and adhering to organizational security policies.

#### **1. Centralized Repository for Module Source Code (TCR):**

All Terraform module source code must be centrally stored in the **Terraform Centralized Repository (TCR)**, acting as the single source of truth for infrastructure modules. Access to TCR is strictly controlled through **Role-Based Access Control (RBAC)** to ensure that only authorized personnel can modify or contribute new modules. This guarantees that every module meets the organization’s security, compliance, and coding standards before being released for use.

#### **2. Module Publishing to JFrog Artifactory:**

After modules are reviewed and approved in TCR, they are versioned and published to **JFrog Artifactory**, which serves as the official, secure distribution platform for all infrastructure modules. Artifactory ensures that only approved and compliant versions of modules are available for use by various teams. Versioning prevents the inadvertent use of insecure or outdated modules. Artifactory also enables controlled, traceable module releases, promoting consistency and compliance across the organization.

#### **3. Access Control and Logging:**

Strict access control must be applied to both **TCR** and **JFrog Artifactory**. Only authorized pipelines, such as CI/CD systems, should have permission to publish modules to JFrog, ensuring no manual interventions compromise security. All interactions — including module publishing, updates, and downloads — must be logged and monitored via **Sumo Logic**. This provides real-time visibility into access patterns and module usage, helping security teams identify and respond to unauthorized access or anomalies quickly.

#### **4. Approval Workflow and Version Control:**

Each module must pass through a rigorous review and approval process before publication in JFrog. This process includes **security audits**, **compliance checks**, and **functional testing** to ensure that the module adheres to the organization's security policies and regulatory standards. Once approved, the module is versioned, and each version is **immutable**. Teams should always use the approved version stored in JFrog to avoid the risks associated with unapproved or insecure modules.

#### **5. Regular Security Audits and Compliance Monitoring:**

Both TCR and JFrog Artifactory should be subject to regular security audits to ensure that all stored modules meet the latest security and compliance standards. Modules that are outdated, vulnerable, or no longer compliant must be flagged and deprecated to prevent their use. Continuous compliance monitoring helps ensure that infrastructure deployments meet both organizational and industry security requirements.

#### **6. Immutable Module Versioning:**

Once a module version is published to **JFrog Artifactory**, it must remain **immutable**, meaning no modifications can be made to that version. If updates are needed, they should be published as a **new version**, ensuring the integrity of previous versions. This ensures stability and consistency across environments, with the guarantee that previous versions remain unchanged and traceable for historical deployments.

#### **7. Logging and Auditing with Sumo Logic:**

All activity related to module management and usage should be fully logged and auditable through **Sumo Logic**. This includes tracking who publishes, updates, or downloads modules, providing a detailed and auditable trail for security and compliance purposes. Sumo Logic enables real-time monitoring and reporting, allowing security teams to detect unauthorized changes or access. Regular audit log reviews ensure continued adherence to security policies and regulatory requirements.

**hardening spec**

For \*\*Infrastructure as Code (IaC) Hardening\*\*, a combination of \*\*proactive\*\* and \*\*reactive\*\* controls is essential to maintaining robust security and compliance across infrastructure deployments. \*\*Proactively\*\*, the organization has implemented \*\*Checkov\*\* as a \*\*Policy as Code\*\* tool within the \*\*Jenkins pipeline\*\*. Checkov automatically scans IaC configurations, such as \*\*Terraform\*\* and \*\*CloudFormation\*\*, for misconfigurations and potential security issues before they are deployed. This integration ensures that any IaC changes are validated against organization-specific security policies. The rules enforced by Checkov are based on the organization's \*\*Reference Architecture (RA)\*\* and \*\*Solution Architecture (SA)\*\*, which define the security and compliance requirements for each resource and solution. This ensures that all IaC deployments adhere to best practices and remain compliant with regulatory standards.

On the \*\*reactive\*\* side, the organization has deployed \*\*QWE\*\*, a continuous monitoring tool, to ensure that any misconfigurations or noncompliance issues that occur post-deployment are swiftly detected. QWE constantly evaluates the deployed infrastructure for deviations from the predefined security baselines. If any noncompliance or misconfiguration is detected—such as unauthorized changes or drift from the IaC-defined state—QWE generates alerts and initiates remediation workflows to resolve the issues automatically or notify the appropriate teams.

Together, these \*\*proactive\*\* (Checkov) and \*\*reactive\*\* (QWE) mechanisms ensure that infrastructure security is maintained at all stages of the deployment lifecycle. This dual-layered approach not only prevents misconfigurations from being deployed but also actively monitors live infrastructure for any potential risks. All policies and rules governing these controls are resource-specific and tied to the RA and SA documentation, ensuring that security is tailored to each solution’s unique requirements.

**Security**

#### **Encryption**

**1. Encryption in Transit**:  
Encryption in transit ensures that data moving between services, users, or infrastructure resources is protected from interception. In AWS, encryption in transit should be enforced through secure protocols such as **TLS (Transport Layer Security)**. For IaC, tools like **Terraform**, **CloudFormation**, or **AWS SDK** should be configured to ensure that services like **S3**, **RDS**, **Elastic Load Balancers (ELBs)**, and **API Gateway** are using TLS encryption. **AWS Certificate Manager (ACM)** can be used to manage SSL/TLS certificates across services. Additionally, all internal communication between AWS services or between Kubernetes clusters (via Helm) should implement **mutual TLS (mTLS)** to safeguard communication at every layer of the infrastructure.

**2. Encryption at Rest**:  
Encryption at rest ensures that stored data is unreadable to unauthorized users or attackers. For IaC, this should be enabled by default for all critical resources. In AWS, services such as **S3**, **EBS volumes**, and **RDS databases** can be encrypted at rest using **AWS KMS (Key Management Service)**. IaC tools like **Terraform**, **CloudFormation**, and **AWS SDK** can define encryption parameters directly, ensuring that data is automatically encrypted at rest. Whether using AWS-managed keys or customer-managed keys (CMKs), encryption at rest can be applied to all sensitive storage systems. Specific encryption rules should be incorporated into the IaC templates as part of the compliance strategy, ensuring that all resources meet the required security standards.

#### **Identity and Access Management (IAM) for Infrastructure as Code**

**1. Fine-Grained Access Control**:  
Identity and Access Management (IAM) is critical for controlling access to AWS resources in an Infrastructure as Code (IaC) environment. When deploying infrastructure via **Terraform**, **CloudFormation**, or the **AWS SDK**, ensure that IAM roles, policies, and permissions follow the **principle of least privilege**. This means that all resources, including AWS services such as EC2, S3, RDS, and Lambda, should be granted only the minimum permissions necessary to perform their intended functions. IAM policies should be written and maintained within IaC templates to ensure consistent and secure deployment across environments (e.g., dev, staging, production).

**2. IAM Role Delegation and Automation**:  
For automated IaC deployments via pipelines like Jenkins, it's essential to grant **delegated roles** to the CI/CD system. These roles should be provisioned with precise access controls to limit interactions to only the necessary AWS resources. In a typical workflow, the Jenkins pipeline, executing Terraform or CloudFormation, would assume a specific role with access to deploy infrastructure components (e.g., creating VPCs, managing S3, provisioning EKS clusters). Each action is governed by policies that are embedded within the IaC code to maintain control and visibility over infrastructure changes.

**3. IAM Policies and Templates in IaC**:  
IAM roles, policies, and groups should be defined as part of the IaC codebase itself, ensuring access controls are versioned and tracked. In **Terraform**, IAM roles and policies can be defined using the aws\_iam\_role and aws\_iam\_policy resources, while in **CloudFormation**, the AWS::IAM::Role and AWS::IAM::Policy resources ensure consistency across environments. This allows access controls to be managed centrally, reducing the risk of drift between different environments and ensuring that permissions are consistently applied across the infrastructure lifecycle.

**4. Multi-Factor Authentication (MFA) Enforcement**:  
To further secure privileged accounts and roles, **Multi-Factor Authentication (MFA)** must be enforced for all IAM users with elevated access or administrative privileges, including those involved in deploying infrastructure via IaC. MFA ensures that even if IAM credentials are compromised, attackers cannot gain access without a secondary authentication factor. This can be enforced through **AWS IAM** for both individual accounts and roles that execute automated deployments via CI/CD systems.

**5. Auditing IAM Activities with CloudTrail**:  
Tracking and auditing IAM-related activities is critical in maintaining a secure infrastructure. **AWS CloudTrail** should be enabled to log all actions taken by IAM users, roles, and services. Every interaction with AWS resources—including who accessed which resources, when, and what changes were made—should be captured and analyzed. Logs from CloudTrail should be integrated with tools like **Sumo Logic** to provide real-time monitoring and analysis of IAM activities, enabling rapid identification and response to suspicious behavior. Additionally, security teams can set up alerts for any deviations from established policies or attempts to access unauthorized resources.