Kubernetes Configuration Management: Simplicity vs. Complexity

Helm

Helm was designed to simplify Kubernetes application deployment, but it has become another abstraction layer that introduces unnecessary complexity. Helm charts often hide the underlying process with layers of Go templating and nested `values.yaml` files, making it difficult to understand what is actually being deployed. Debugging often requires navigating through these files, which can obscure the true configuration. This approach shifts from infrastructure-as-code to something less transparent, making it harder to manage and troubleshoot.

```
imagePullPolicy: {{ .Values.defaultBackend.image.pullPolicy
↪  }}
{{- if .Values.defaultBackend.extraArgs }}
  args:
  {{- range $key, $value := .Values.defaultBackend.extraArgs
  ↪  }}
    {{- /* Accept keys without values or with false as value
    ↪  */}}
    {{- if eq ($value | quote | len) 2 }}
    - --{{ $key }}
    {{- else }}
    - --{{ $key }}={{ $value }}
    {{- end }}
  {{- end }}
{{- end }}
```

YAML itself isn't inherently problematic, and with modern IDE support, schema validation, and linting tools, it can be a clear and effective configuration format. The issues arise when YAML is combined with Go templating, as seen in Helm. While each component is reasonable on its own, their combination creates complexity. Go templates in YAML introduce fragile constructs, where whitespace sensitivity and imperative logic make configurations difficult to read, maintain, and test. This blending of logic and data undermines transparency and predictability, which are crucial in infrastructure management.

Helm's dependency management also adds unnecessary complexity. Dependencies are fetched into a `charts/` directory, but version pinning and overrides often become brittle. Instead of clean component reuse, Helm encourages nested charts with their own `values.yaml`, which complicates customization and requires understanding multiple charts to override a single value. In practice, Helm's dependency management can feel like nesting shell scripts inside other shell scripts.

Kustomize

Kustomize offers a declarative approach to managing Kubernetes configurations, but its structure often blurs the line between declarative and imperative. Kustomize applies transformations to a base set of Kubernetes manifests, where users define overlays and patches that *appear* declarative, but are actually order-dependent and procedural.

It supports various patching mechanisms, which require a deep understanding of Kubernetes objects and can lead to verbose, hard-to-maintain configurations. Features like generators pulling values from files or environment variables introduce dynamic behavior, further complicating the system. When built-in functionality falls short, users can use KRM (Kubernetes Resource Model) functions for transformations, but these are still defined in structured data, leading to a complex layering of data-as-code that lacks clarity.

While Kustomize avoids explicit templating, it introduces a level of orchestration that can be just as opaque and requires extensive knowledge to ensure predictable results.

The Configuration Pipeline Problem

In many Kubernetes environments, the configuration pipeline has become a complex chain of tools and abstractions. What the Kubernetes API receives — plain YAML or JSON — is often the result of multiple intermediate stages, such as Helm charts, Helmsman, or GitOps systems like Flux or Argo CD. As these layers accumulate, they can obscure the final output, preventing engineers from easily accessing the fully rendered manifests.

This lack of visibility makes it hard to verify what will actually be deployed, leading to operational challenges and a loss of confidence in the system. When teams cannot inspect or reproduce the deployment artifacts, it becomes difficult to review changes or troubleshoot issues, ultimately turning a once-transparent process into a black box that complicates debugging and undermines reliability.

Other Approaches

Apple's pkl (short for "Pickle") is a configuration language designed to replace YAML, offering greater flexibility and dynamic capabilities. It includes features like classes, built-in packages, methods, and bindings for multiple languages, as well as IDE integrations, making it resemble a full programming language rather than a simple configuration format.

However, the complexity of pkl may be unnecessary. Its extensive documentation and wide range of features may be overkill for most use cases, especially when YAML itself can handle configuration management needs. If the issue is YAML's repetitiveness, a simpler approach, such as sandboxed JavaScript, could generate clean YAML without the overhead of a new language.

KISS Principle

Kubernetes configuration management is ultimately a string manipulation problem. Makefiles, combined with standard Unix tools, are ideal for solving this. Make provides a declarative way to define steps to generate Kubernetes manifests, with each step clearly outlined and only re-run when necessary. Tools like `sed`, `awk`, `cat`, and `jq` excel at text transformation and complement Make's simplicity, allowing for quick manipulation of YAML or JSON files.

This approach is transparent — you can see exactly what each command does and debug easily when needed. Unlike more complex tools, which hide the underlying processes, Makefiles and Unix tools provide full control, making the configuration management process straightforward and maintainable.

HashiCorp Vault is a tool for managing secrets and sensitive data, offering features like encryption, access control, and secure storage. It was used as an example of critical infrastructure deployed on Kubernetes without Helm, emphasizing manual, customizable management of resources.

```
define rbac
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: vault-service-account
  namespace: ${VAULT_NAMESPACE}
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: vault-server-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
```

```
  name: system:auth-delegator
subjects:
- kind: ServiceAccount
  name: vault-service-account
  namespace: ${VAULT_NAMESPACE}
endef
export rbac
```

```
manifests += $${rbac}
manifests += $${configmap}
manifests += $${services}
manifests += $${statefulset}

.PHONY: template apply delete

template:
    @$(foreach manifest,$(manifests),echo "$(manifest)";)

apply: create-release
    @$(foreach manifest,$(manifests),echo "$(manifest)" |
     ↪  kubectl apply -f - ;)

delete: remove-release
    @$(foreach manifest,$(manifests),echo "$(manifest)" |
     ↪  kubectl delete -f - ;)

validate-%:
    @echo "$$$*" | yq eval -P '.' -

print-%:
    @echo "$$$*"
```

The `manifests` array holds the multi-line YAML templates for Kubernetes resources, including RBAC, ConfigMap, Services, and StatefulSet. In the `apply` target, each manifest is processed and passed to `kubectl apply` to deploy them to the Kubernetes cluster. This approach uses `foreach` to iterate over the `manifests` array, applying each resource one by one. Similarly, the `delete` target uses `kubectl delete` to remove the resources defined in the manifests.

Using Make with tools like `curl` is a super flexible way to handle Kubernetes deployments, and it can easily replace some of the things Helm does. For example, instead of using Helm charts to manage releases, we're just using `kubectl` in a Makefile to create and delete Kubernetes secrets.

```
.PHONY: create-release
create-release:
    @echo "Creating Kubernetes secret with VERSION set to Git
    ↪  commit SHA ... "
    @SECRET_NAME="app-version-secret"; \
    JSON_DATA="{\"VERSION\":\"$(GIT_COMMIT)\"}"; \
    kubectl create secret generic $$SECRET_NAME \
        --from-literal=version.json="$$JSON_DATA" \
        --dry-run=client -o yaml | kubectl apply -f -
    @echo "Secret created successfully: app-version-secret"

.PHONY: remove-release
remove-release:
    @echo "Deleting Kubernetes secret: app-version-secret ... "
    @SECRET_NAME="app-version-secret"; \
    kubectl delete secret $$SECRET_NAME 2>/dev/null || true
    @echo "Secret deleted successfully: app-version-secret"
```

Since the `manifests` array contains all the Kubernetes resource definitions, we can easily dump them into both YAML and JSON formats. The `dump-manifests` target runs `make template` to generate the YAML output and `make convert-to-json` to convert the same output into JSON.

```
.PHONY: dump-manifests
dump-manifests: template convert-to-json
    @echo "Dumping manifests to manifest.yaml and
    ↪  manifest.json ... "
    @make template > manifest.yaml
    @make convert-to-json > manifest.json
    @echo "Manifests successfully dumped to manifest.yaml and
    ↪  manifest.json."
```

With the `validate-%` target, you can easily validate any specific manifest by piping it through `yq` to check the structure or content in a readable format. This leverages external tools like `yq` to validate and process YAML directly within the Makefile.

```
# Validates a specific manifest using `yq`.
validate-%:
    @echo "$$$*" | yq eval -P '.' -

# Prints the value of a specific variable.
print-%:
    @echo "$$$*"
```

With Makefile and simple Bash scripting, you can easily implement auxiliary functions like getting Vault keys.

```
.PHONY: get-vault-keys
get-vault-keys:
    @echo "Available Vault pods:"
    @PODS=$$(kubectl get pods -l
     ↪  app.kubernetes.io/name=vault -o
     ↪  jsonpath='{.items[*].metadata.name}'); \
    echo "$$PODS"; \
    read -p "Enter the Vault pod name (e.g., vault-0): "
     ↪  POD_NAME; \
    if echo "$$PODS" | grep -qw "$$POD_NAME"; then \
        kubectl exec $$POD_NAME -- vault operator init
         ↪  -key-shares=1 -key-threshold=1 -format=json >
         ↪  keys.json; \
        VAULT_UNSEAL_KEY=$$(cat keys_$$POD_NAME.json | jq -r
         ↪  ".unseal_keys_b64[]"); \
        echo "Unseal Key: $$VAULT_UNSEAL_KEY"; \
        VAULT_ROOT_KEY=$$(cat keys.json | jq -r
         ↪  ".root_token"); \
        echo "Root Token: $$VAULT_ROOT_KEY"; \
    else \
        echo "Error: Pod '$$POD_NAME' not found."; \
    fi
```

Conclusion

Sometimes, the simplest way of using just Unix tools is the best way. By relying on basic utilities like `kubectl`, `jq`, `yq`, and Make, you can create powerful, customizable workflows without the need for heavyweight tools like Helm. These simple, straightforward scripts offer greater control and flexibility. Plus, with LLMs (large language models) like this one, generating and refining code has become inexpensive and easy, making automation accessible. However, when things go wrong, debugging complex tools like Helm can become exponentially more expensive in terms of time and effort. Using minimal tools lets you stay in control, reduce complexity, and make it easier to fix issues when they arise. Sometimes, less really is more.