

Final Presentation

Justen Stall

Associate Data Science Engineer, UDRI
Master of Computer Science Student, UD

Background

Problem

Solving the complexity problems of large-scale Kubernetes systems.

Approach the complexity problem as a **greenfield** and use it as an opportunity to learn Kubernetes system design.

Plan

1. Describe the problem
2. Evaluate existing solutions
3. Prescribe and implement a solution



A beautiful green field.

Progress

- Attended Kubecon
- Created proof of concept Timoni bundle for ACE/Env
- Created configuration rendering pipeline comparisons
 - Looked at processes for GitLab deployment: Helm Chart, Operator, Big Bang Helm Chart
- Researched declarative API design and its adoption across Kubernetes tools
- Identified standard components for configuration management
- Projects:
 - Configuration-as-Code: [Terraform](#), [Pulumi](#), [cdk8s](#), [KubeVela](#), [kluctl](#)
 - Configuration-as-Data: [kpt](#), [Kustomize](#), [KRM Functions](#)
 - DSLs: [KCL](#), [CUE](#), [CEL](#), [Jsonnet](#), [Tanka](#)
 - Controller SDKs: [Operator SDK](#), [Kubebuilder](#), [metacontroller](#)
 - K8s packaging models: [OAM](#), [OCM](#), [RukPak](#), [ArtifactHub](#), [Skaffold](#)
 - K8s-related tools: [Crossplane](#), [KubeVela](#), [KubeMod](#), [Monokle](#), [helm-docs](#)
 - API development: [OpenAPI](#), [OpenAPI Generator](#), [controller-gen](#), [Kusk](#), [oapi-codegen](#)

Outline

1. Configuration management research
2. Rendering pipeline research
3. Proposal

Configuration Management Research

- Standard components
- Vertical vs horizontal integration
- IaC vs CaC vs CaD
- Static analysis
 - Kubernetes API accuracy
- Feature comparison
- Conclusions
 - Barriers to adoption

Standard components for configuration management

- Authoring format - what an application developer maintains
 - e.g. Helm Chart, Timoni Module
- Extension system - libraries and components to extend functionality
 - e.g. TF/Pulumi/Crossplane providers, Helm library charts, kpt functions
 - Most extension systems include a Helm integration
 - Terraform, Pulumi, Crossplane, kpt, Kustomize, etc.
- Renderer - produces k8s manifests from the input
 - Usually built into a CLI
 - Some renderers provide an interface or SDK for use in other projects
 - e.g. the Helm SDK is used by [helm-controller](#)
- Deployment manager - manages installation and upgrades of k8s manifests
 - Most provide a CLI command for deployment that combines rendering and deployment
 - May generate merges/patches from rendered manifests (or rely on [server-side apply](#))
 - May support pruning resources
 - Can enable GitOps

Vertical vs horizontal integration

Vertical integration

Monolithic stacks

- Baked-in assumptions about usage
- Incorporate many external sources in opinionated way
- Not viable as a distribution format
- e.g. Pulumi:
 - All of the standard components are part of the same ecosystem, purpose built for the Pulumi workflow
 - Render & Deploy: Pulumi CLI
 - GitOps Engine: Pulumi Operator

Horizontal integration

Interoperable components

- Separation of concerns
- UNIX philosophy
- Unopinionated about usage
- Allows incremental adoption
 - Easier onboarding
- Decreased ecosystem lock in
- e.g. Flux:
 - Flux's GitOps engine is renderer-independent

Configuration-as-Code vs Infrastructure-as-Code

Configuration-as-Code (CaC)

- K8s resource configuration is abstracted
- Introduces authoring features:
 - Syntax: more efficient syntaxes than YAML
 - Composability: DRY configuration
- Can introduce consumer features:
 - Templating: reduced consumer interface
- e.g. Helm, cdk8s, Timoni



kubernetes

Infrastructure-as-Code (IaC)

- Superset of CaC: adds infrastructure management
 - Infrastructure = out-of-cluster resources
- Out-of-cluster resource configuration is abstracted:
 - Combined declarative interface for imperative resource management tasks
 - Efficient alternative to GUIs, shell scripts, or directly making API calls
- e.g. Pulumi, Terraform, Crossplane



Infrastructure management is **out-of-scope** for this project.

Configuration management is **in-scope** for this project.

IaC tools will be evaluated exclusively for their CaC features.

Configuration-as-Code vs Configuration-as-Data

Configuration-as-Code (CaC)

- K8s resource configuration is abstracted
- Introduces authoring features:
 - Syntax: more efficient syntaxes than YAML
 - Composability: DRY configuration
- Can introduce consumer features:
 - Templating: reduced consumer interface
- e.g. Helm, cdk8s, Timoni



Configuration-as-Data (CaD)

- K8s resource configuration is maintained as YAML
 - Full complexity is surfaced
 - Higher cognitive load for consumers
- Guaranteed transparency for ground truth manifests
- Provides tools for batch configuration maintenance and modification
- e.g. kpt, Kustomize, an insane person managing a directory of YAML with yq commands



Configuration explosion

Taking full advantage of Kubernetes' declarative nature requires the maintenance of a massive amount of configuration.

Real-world example: ACE/Env line counts

Fully-rendered manifests	35,176 lines (49,433 including Grafana dashboards)
Pre-rendered Charts/manifests	71,616 lines (additional 165,837 lines of CRDs)



`kubectl cluster-info dump` for meerkat:

2.3 million lines of YAML

Static Analysis

What is static analysis?



Analyzing code before executing.

Static analysis is an example of the “shift-left” testing approach, because it allows validation earlier in the development process. Static analysis is also useful for comprehending existing code, which improves developer efficiency and learning speed.

Examples:

- IDEs perform static analysis of code using [language servers](#)
- IDEs perform static analysis of config files using [declarative schemas](#)
- The [VS Code Kubernetes extension](#) enables static analysis for YAML manifests with access to a cluster API server

Static analysis for Kubernetes



The most documented and thus understood interface for the Kubernetes API is the native manifests in YAML. – Tony Gilkerson

- Kubernetes API is defined in a declarative format
 - Kubernetes-specific extended form of OpenAPI called “Structural Schema”
- Native types are defined as a best-effort replication of their handwritten validation rules
 - Ongoing improvements: [KEP-2896](#), [KEP-4153](#), [wg-api-expression](#)
- CRDs are comprehensively defined







However, Kubernetes does not provide a tool for client-side static analysis of YAML manifests. (there is ongoing work on a [kubect1 validate](#) command)

Many Kubernetes tools fill this gap with their own client-side validator or by using a non-YAML interface for authoring manifests (such as a strongly-typed programming language).

Do static analysis tools for Kubernetes provide
1-to-1 accuracy with the Kubernetes API?
I researched each codebase to find out.

Kubernetes API 1-to-1 accuracy?



 cdk8s	NO	Types are generated by cdk8s import k8s crd , both lossy converters.
 Pulumi	NO	Native types are generated by pulumigen , a lossy converter. CRD types are generated by crd2pulumi , a lossy converter.
 {timoni}	NO	CUE's OpenAPI importer is lossy. Timoni extends its functionality in timoni mod vendor k8s crd , which is still a lossy converter.
 Kubeconform	NO	Validator schema is generated with openapi2jsonschema , which is a lossy converter.
 MONOKLE by Kubeshop	NO	Validation command monokle validate is an incomplete validator. (working on incorporating CEL: branch)
 kcl	NO	CRD models are generated using kcl-openapi , a lossy converter. (open issue for incorporating CEL: #22)

Links are to the offending file or repository.

Configuration manager feature comparison

Component	Kubectl	Kustomize	Helm	KPT	CUE	Timoni	KCL	cdk8s	Terraform	Pulumi	Crossplane	Jsonnet	Tanka	kluctl
Authoring		Kustomizations	Chart	KPT packages	Module	Module	Module	app	Module	Program	Custom resource	Project	Project	project
Language	YAML/JSON	YAML	Go templates	YAML	CUE	CUE	KCL	Many	HCL	Many	Many	Jsonnet	Jsonnet	Jinja
Language server?	Yes	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
Can include cluster state?	No	No	Yes	No	No	Yes	No	No	Yes	Yes	Yes	No	No	No
Valid YAML encoding?	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Validate of K8s schemas?	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	No	Partial	Partial	No
Validate CRD schemas?	No	No	No	No	Partial	Partial	Partial	Yes	No	Yes	No	Partial	Partial	No
Declarative validation support?	No	No	No	No	No	No	No	No	No	No	No	No	No	No
Integrate other package formats?	Yes	Yes	No	Yes	No	No	Partial	Yes	Yes	Yes	Yes	No	Yes	Yes
Extensions	Plugins	KRM Functions	Sub-charts	KRM functions	Modules	Modules	Modules	Constructs	Providers	Providers	Packages			
Extension system?	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Central extension catalog?	Yes	No	Partial	Yes	No	No	Yes	Yes	Yes	Yes	Yes	No	No	No
Distribution														
Defined packaging format?	N/A	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
OCI compatible?	N/A	No	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes	No	No	Yes
Renderer														
Client-side	N/A	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
Renderer SDK	N/A	Yes	Yes	Yes	Partial	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Render without cluster?	N/A	Partial	Partial	Partial	Yes	Yes	Yes	Yes	Partial	Partial	No	Yes	Yes	Yes
Available as a KRM Function?	N/A	No	Partial	No	No	No	Yes	No	No	No	No	No	No	No
Deployment														
Built-in deployment command?	Yes	Partial	Yes	Yes	No	Yes	No	No	Yes	Yes	Yes	No	No	Yes
First-party GitOps engine?	No	No	Yes	Yes	No	No	No	No	Yes	Yes	No	No	No	Yes
Flux-compatible controller?	Yes	Yes	Yes	No	Yes	No	No	No	Yes	Yes	Yes	Yes	Yes	No
Merge/patch logic?	Yes	N/A	Yes	Yes	N/A	Yes	N/A	N/A	Yes	Yes	Yes	N/A	Yes	Yes
Native k8s merge/patch logic?	Yes	N/A	No	Yes	N/A	Yes	N/A	N/A	Yes	Yes	Yes	N/A	Yes	Yes
Resource pruning?	Yes	N/A	Yes	Yes	N/A	Yes	N/A	N/A	Yes	Yes	Yes	N/A	Yes	Yes
Supports server-side apply?	Yes	N/A	No	Yes	N/A	Yes	N/A	N/A	Yes	Yes	Yes	N/A	Yes	Yes
Defaults to server-side apply?	No	N/A	No	No	N/A	Yes	N/A	N/A	Yes	Yes	Yes	N/A	No	Yes

[Spreadsheet link](#)

Conclusions

It's chaos out there.

There is no clear consensus from the endless stream of configuration management ideas.

Kubectl, Kustomize, and Helm remain the only widely-adopted tools.

I have shifted focus to understanding the barriers to the adoption of alternative tools.



Barriers to adoption

- No idea can have an impact without wide adoption
 - Apps maintain a single deployment strategy for developer efficiency and deployment testing purposes
 - Developers do not want to add prerequisites and background knowledge for consumers deploying their apps
- Claim: lack of interoperability between configuration managers is the main obstacle for their adoption
 - Almost all alternatives include an integration with Helm, since it is the de-facto tool
 - Most alternatives' catalogs are filled with wrappers for Helm Charts

Compare rendering pipelines to look for a path to interoperability.

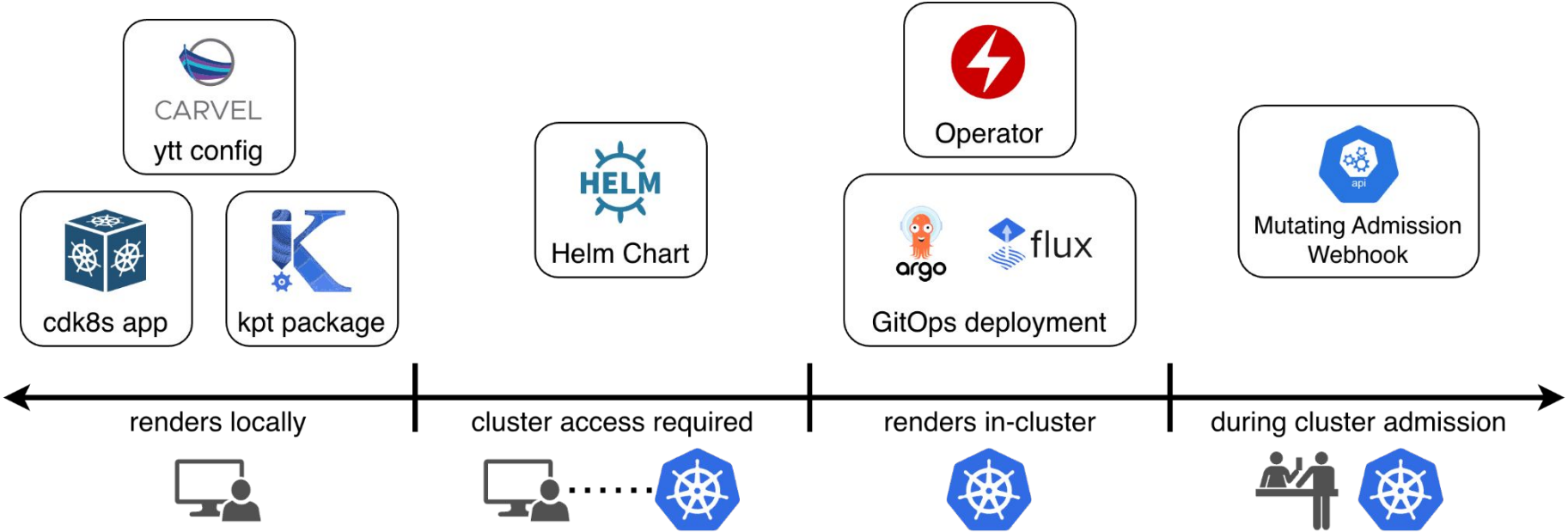
Rendering Pipelines

- Definition
- Render time comparison
- Rendering pipeline comparisons
- Desirable properties
- Conclusions

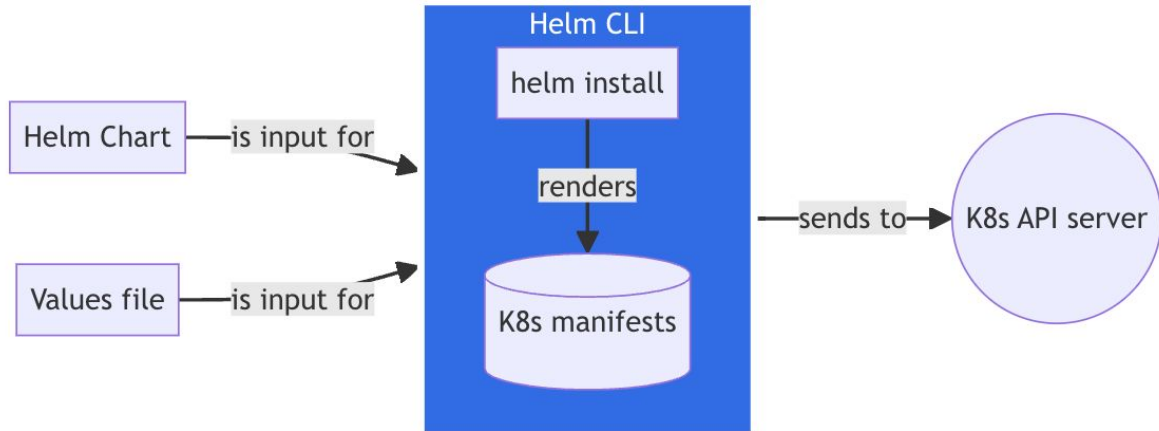
What is a rendering pipeline?

The process of generating ground truth Kubernetes manifests from an arbitrary input format.

Render time comparison

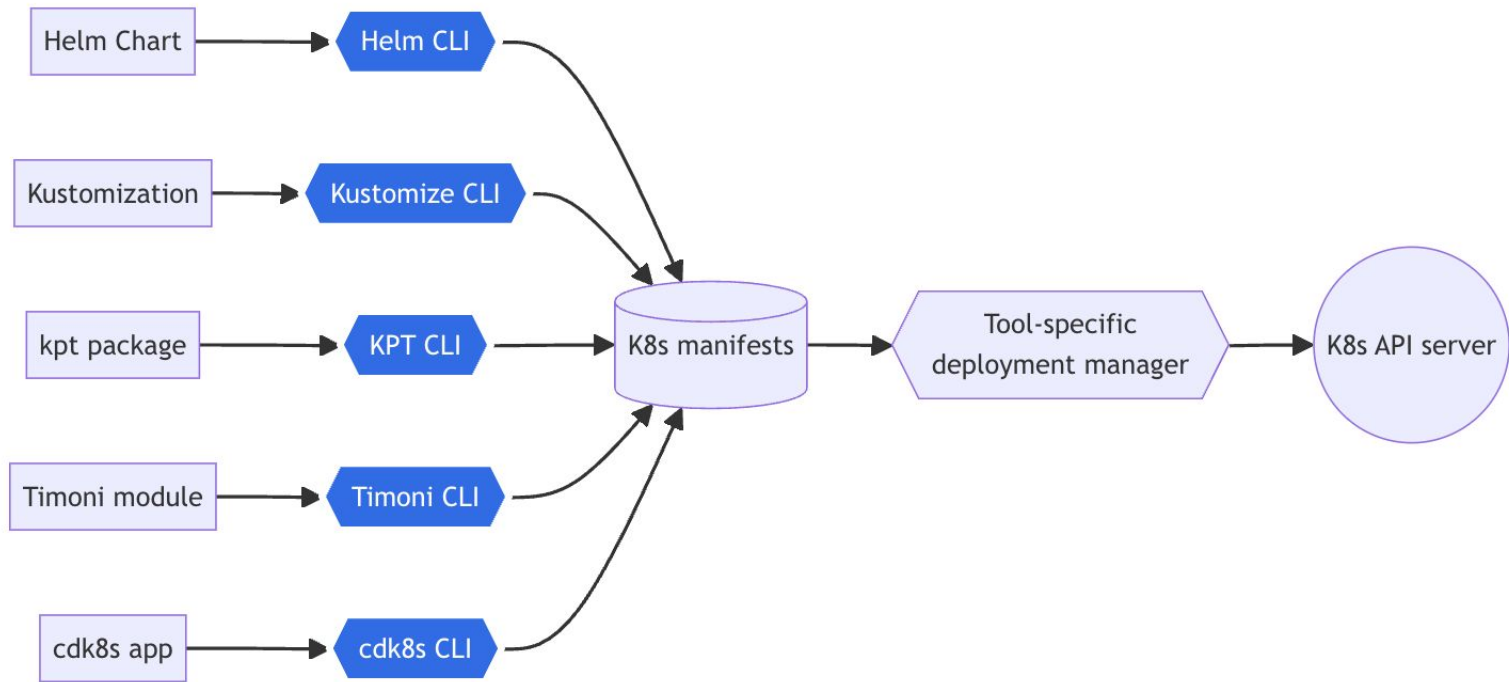


Rendering Pipeline Comparisons

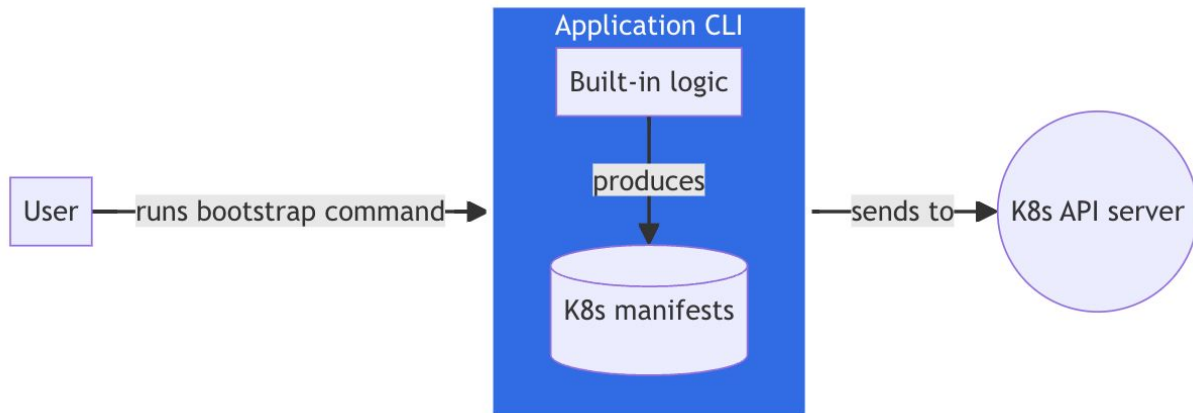


Generalized rendering pipeline of a Helm Chart

Omitted for clarity: Helm reads cluster state while rendering

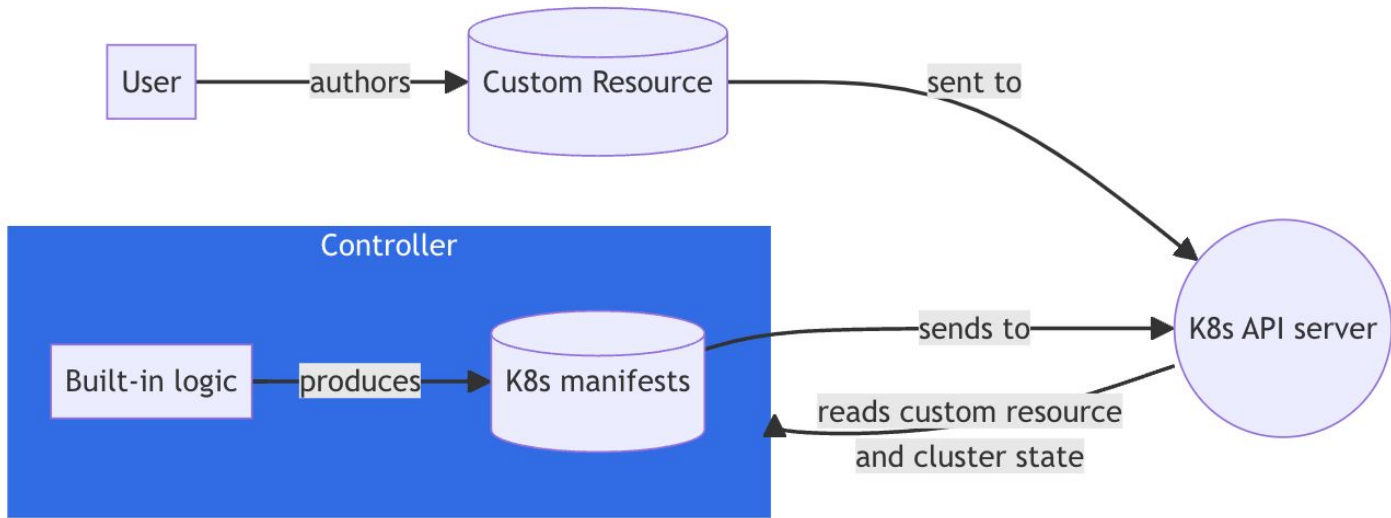


Generalized rendering pipeline for configuration manager CLIs



Rendering pipeline of an application's custom CLI

e.g. deploying Istio with ["istioctl"](#) (which is implemented as a wrapper for the Istio Helm Chart)



Rendering pipeline of a controller (or operator)

Desirable properties

The following software properties are valuable for designing renderers:

Deterministic

- The same input should always produce the same output
- e.g. a renderer that adds a timestamp to resources is not deterministic

Idempotent

- Repeat executions should not produce any change
- e.g. a renderer that increments the value of the replicas field by 1 is not idempotent
 - Instead, the replicas value should be provided as an input

Hermetic

- Renderers that require access to the host system cannot be executed declaratively and pose challenges for security, correctness, portability, and speed
- e.g. a renderer that reads cluster state requires read permissions to the cluster and cannot be reproduced without network access

Conclusions

- Rendering manifests is a part of every deployment process whether it is made visible to the user or not
- Competing ideas about how and when to render create confusion



Proposal

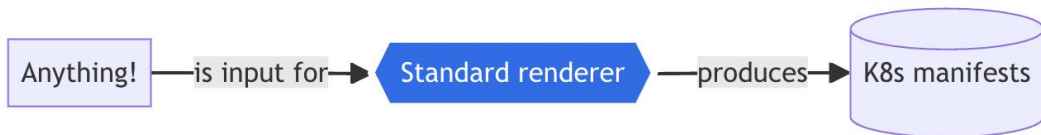
A standard renderer

- Need for a standard renderer
- Requirements
- Competing implementations
- Proposed solution
- Next steps

Need for a standard renderer

Simplify rendering to its fundamental process.

- Encapsulate various configuration management tools as extensions of a standard renderer framework
 - Extensions should be consistent, composable, and reusable
 - Aim to be deterministic, idempotent, and hermetic
 - Containerize extensions to minimize dependencies
- On-demand access to ground truth manifests
- Portable into any deployment process
 - Run locally, in CI, incorporate into controllers, etc.
 - Application consumer decides when and where an application is rendered



Requirements

Developer requirements

- Reduce complexity when possible with reusability and composability
- Enable adoption of alternative tools
- Shift-left: testing and validation earlier in the development process
- Manageable dependencies
 - Able to tweak output from an upstream dependency
 - Easily identify breaking changes from updated interface and resulting manifests
 - Easily update to incorporate improvements and patched vulnerabilities
- Full control of ground truth manifests
 - Small changes to the Kubernetes manifests should be easy in all cases

Consumer requirements

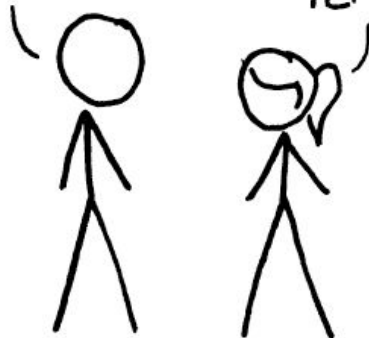
- Do not introduce dependencies to deployment process
- Push complexity down – away from the consumer
- Control over when manifests are rendered
 - e.g. locally by the consumer or in-cluster by a controller
 - Enable transitions between the options
- Control over how manifests are deployed
 - Single deployment and continuous deployment should be equally full-featured

HOW STANDARDS PROLIFERATE:

(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)

SITUATION:
THERE ARE
14 COMPETING
STANDARDS.

14?! RIDICULOUS!
WE NEED TO DEVELOP
ONE UNIVERSAL STANDARD
THAT COVERS EVERYONE'S
USE CASES.



SOON:

SITUATION:
THERE ARE
15 COMPETING
STANDARDS.

Proposed Solution

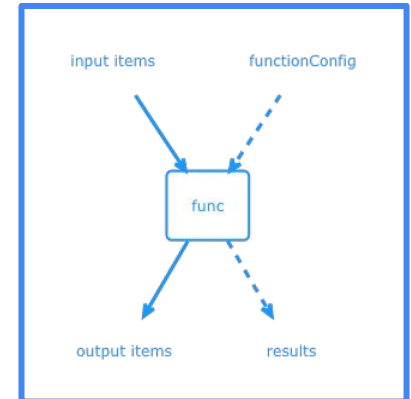
What if I told you there was an existing standard already built into kubectl?



Proposed solution: KRM Functions

There is an existing standard built into to kubect1: [KRM Functions](#)

- KRM Functions
 - Specification was created by the kpt project and donated to the CNCF
 - Defines inter-process communication between an orchestrator (e.g. kpt CLI) and functions
 - Can be developed in different toolchains and languages (with containerization)
 - Interoperable and backwards compatible
- Kustomize plugin system
 - Kustomize is adopting KRM Functions for its plugin system
 - The KRM Function-based plugin system is still in alpha
- Currently available in kubect1, Kustomize, and kpt
 - Kustomize has been built into kubect1 since 2019



What are KRM Functions?

KRM Functions are container images with a well-defined input and output format.

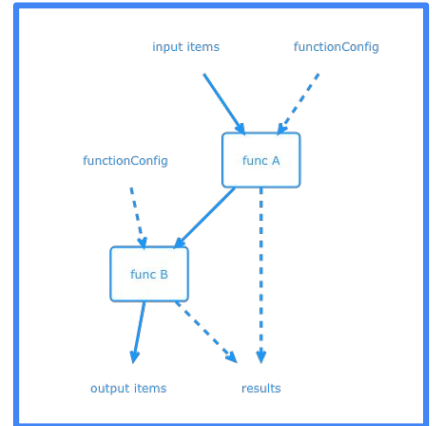
- kpt's KRM Function catalog has examples: catalog.kpt.dev
- kpt includes an experimental WASM runtime: [kpt PR #3450](#)
 - Still in alpha and not part of the core specification yet (Kustomize issue: [#4956](#))

Imperative execution: manually run a function once

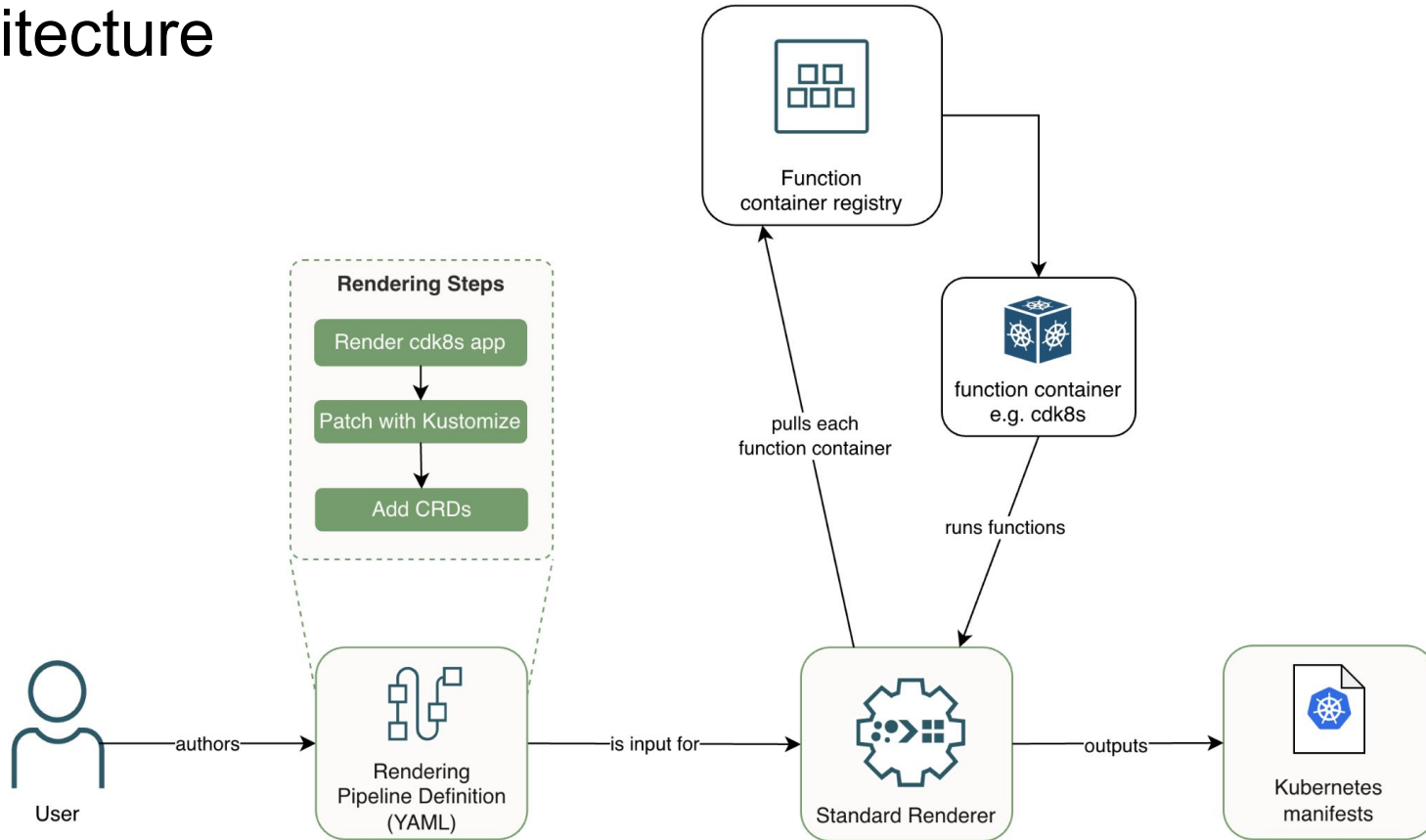
- [kustomize fn run](#)
- [kpt fn eval](#)

Declarative execution: run functions as part of a defined pipeline

- [kubectl kustomize --enable-alpha-plugins](#)
- [kustomize build --enable-alpha-plugins](#)
- [kpt fn render](#)



Architecture



State of the KRM Function specification

Functions have been stable in kpt for a while, but Kustomize's plugin system is still in development, with many unimplemented proposals and open issues:

- [KEP-2299](#) Kustomize Plugin Composition API
- [kpt #3118](#) Accept non-YAML files
- [KEP-2906](#) Kustomize Function Catalog
- [KEP-2299](#) Public KRM Functions Registry
- [KEP-2953](#) Kustomize plugin graduation



Next steps

Work will continue next semester as CPS 596.

- Currently researching KRM Functions, function SDKs, and similar projects
 - KRM Function SDKs: [Kustomize fn/framework](#), [kpt fn](#)
- Create standard renderer POC that accomplishes the following:
 - Supports various configuration renderers through containerized functions
 - Defines rendering pipelines as compositions of functions
 - Standard renderer itself packaged as a containerized function
 - Runs locally with a CLI and runs in-cluster with a controller

Goal: POC renders with `kubectl` (using built-in Kustomize)

Deliverables

1. Proposal presentation 
2. Progress presentation 
3. Final design presentation 