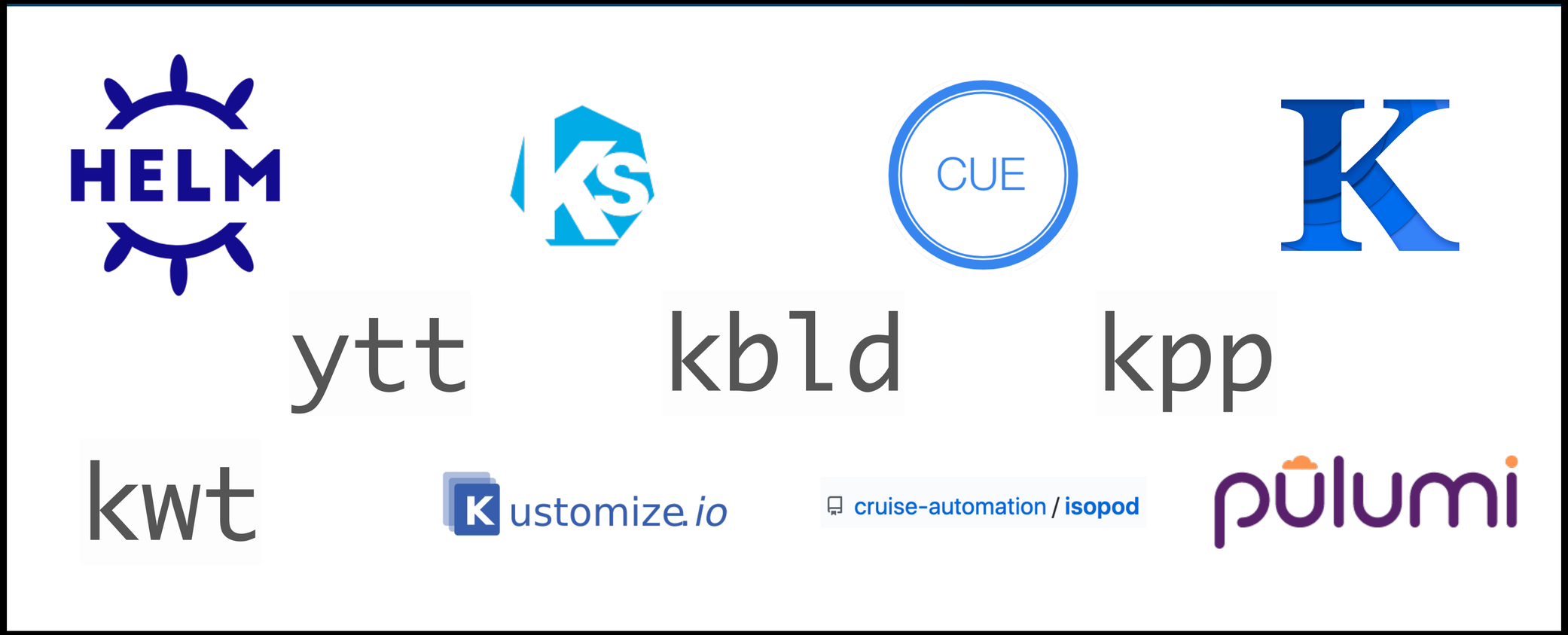
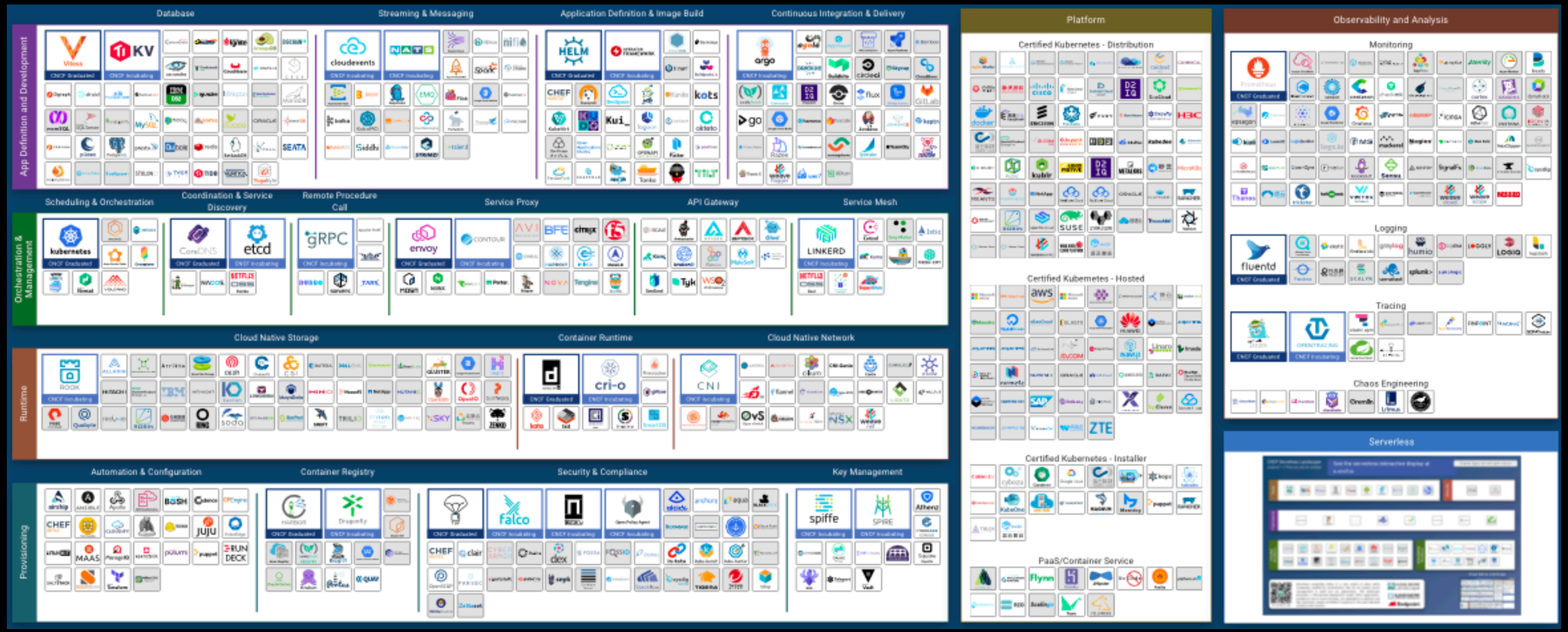
Five Hundred Twenty-five Thousand Six Hundred K8s CLI's

Phil Wittrock @pwittrock
Gabbi Fisher @gabbifish

Act I: 🎶 It's a dizzy merry go round 🎶

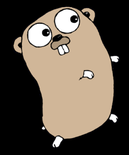


...and Sometimes Feels Like Looking at This



How can we make this landscape less daunting?

- Abstractions



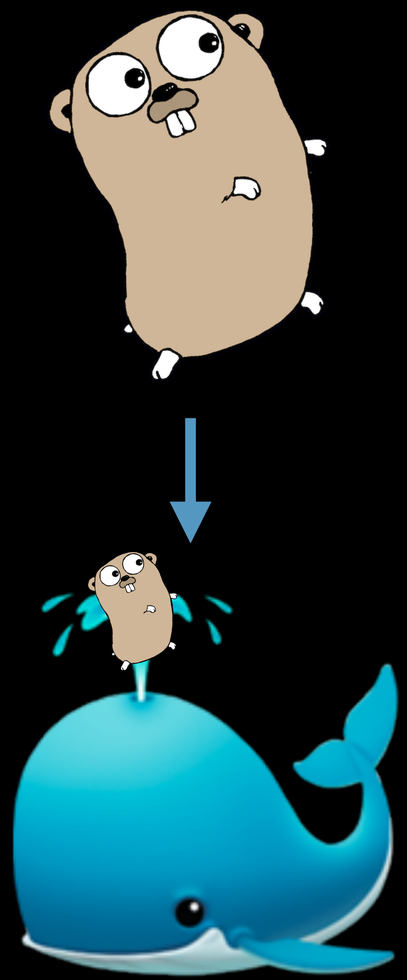
- Variants



- Cross-cutting concerns

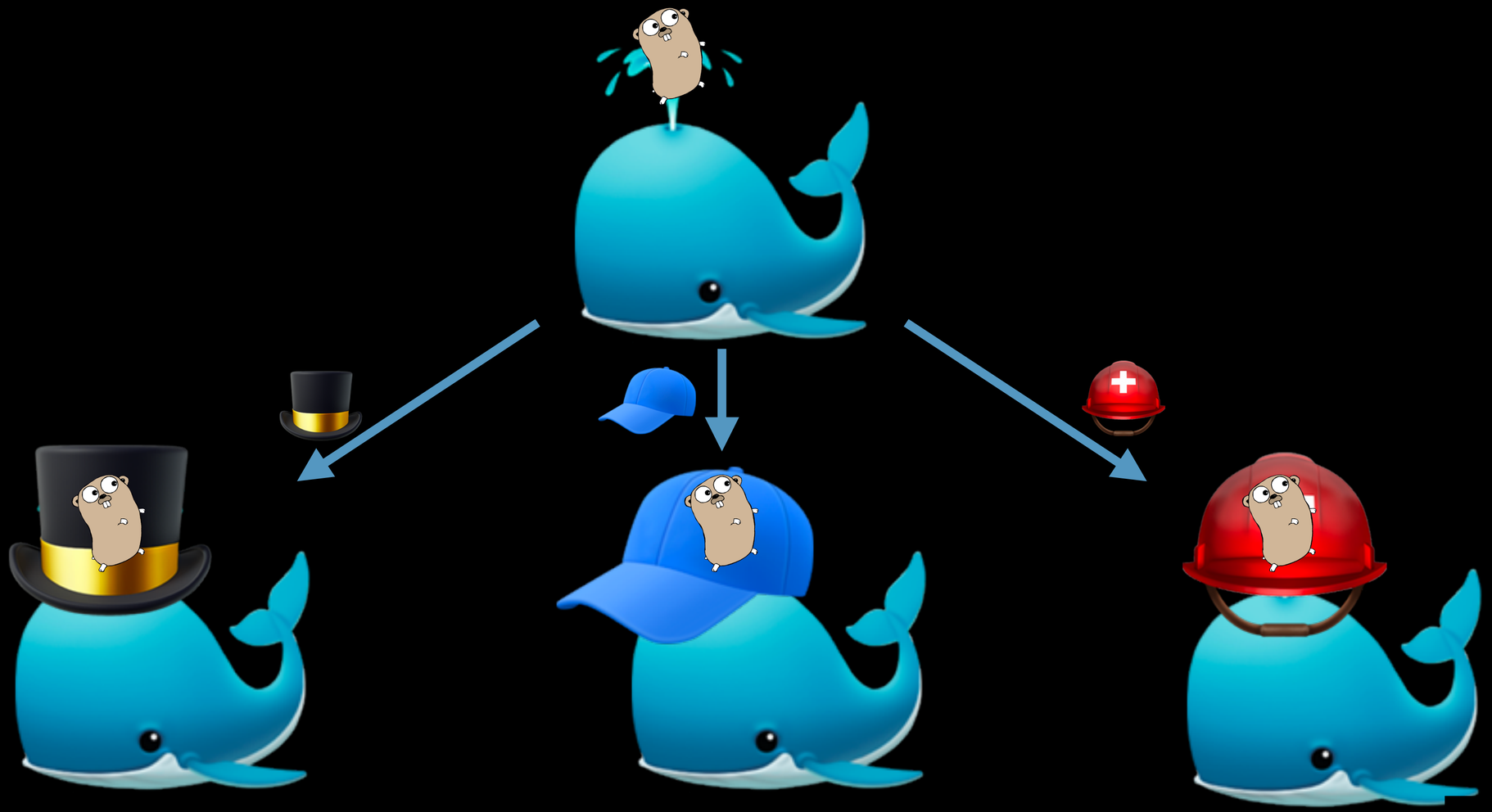
Using CLIs for Configuration

Abstraction



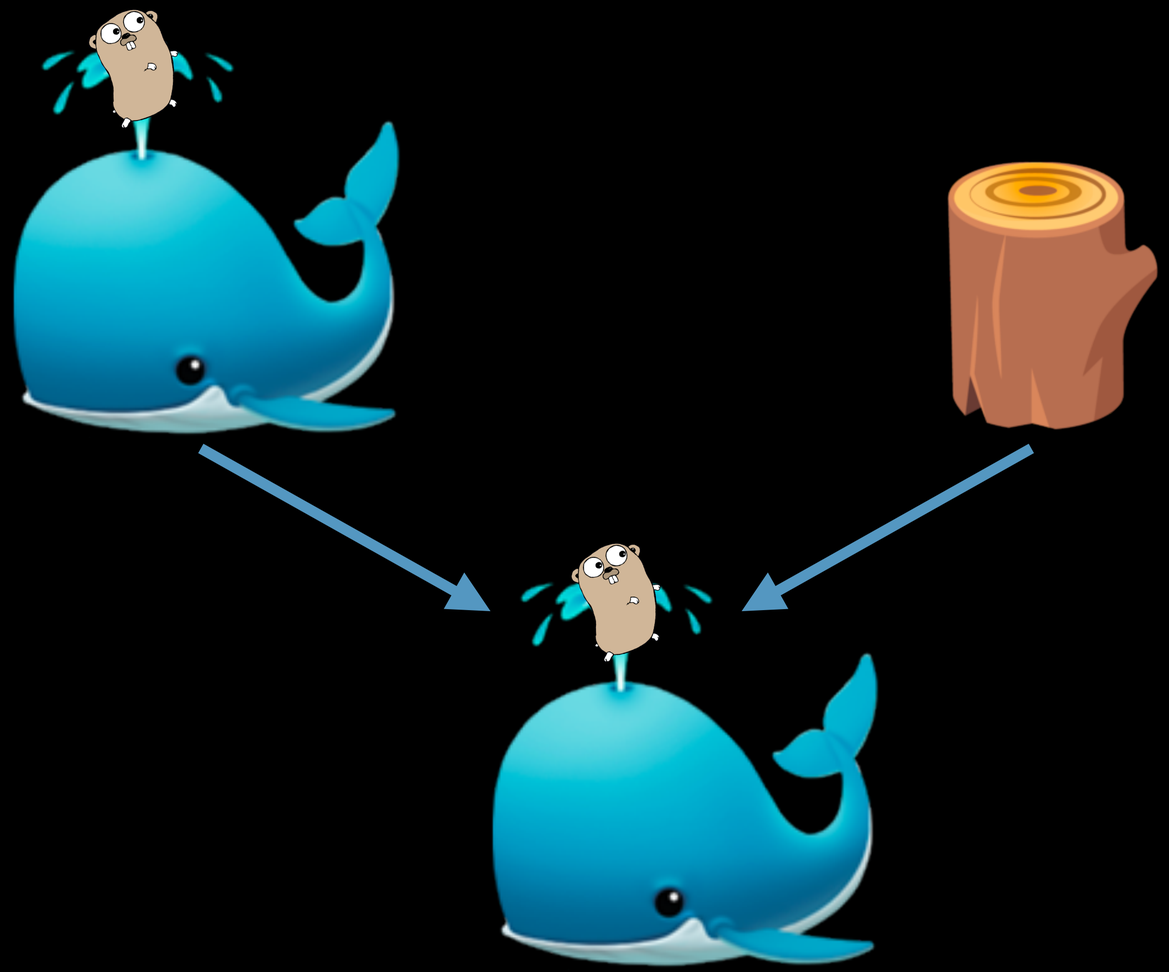
Using CLIs for Configuration

Variance



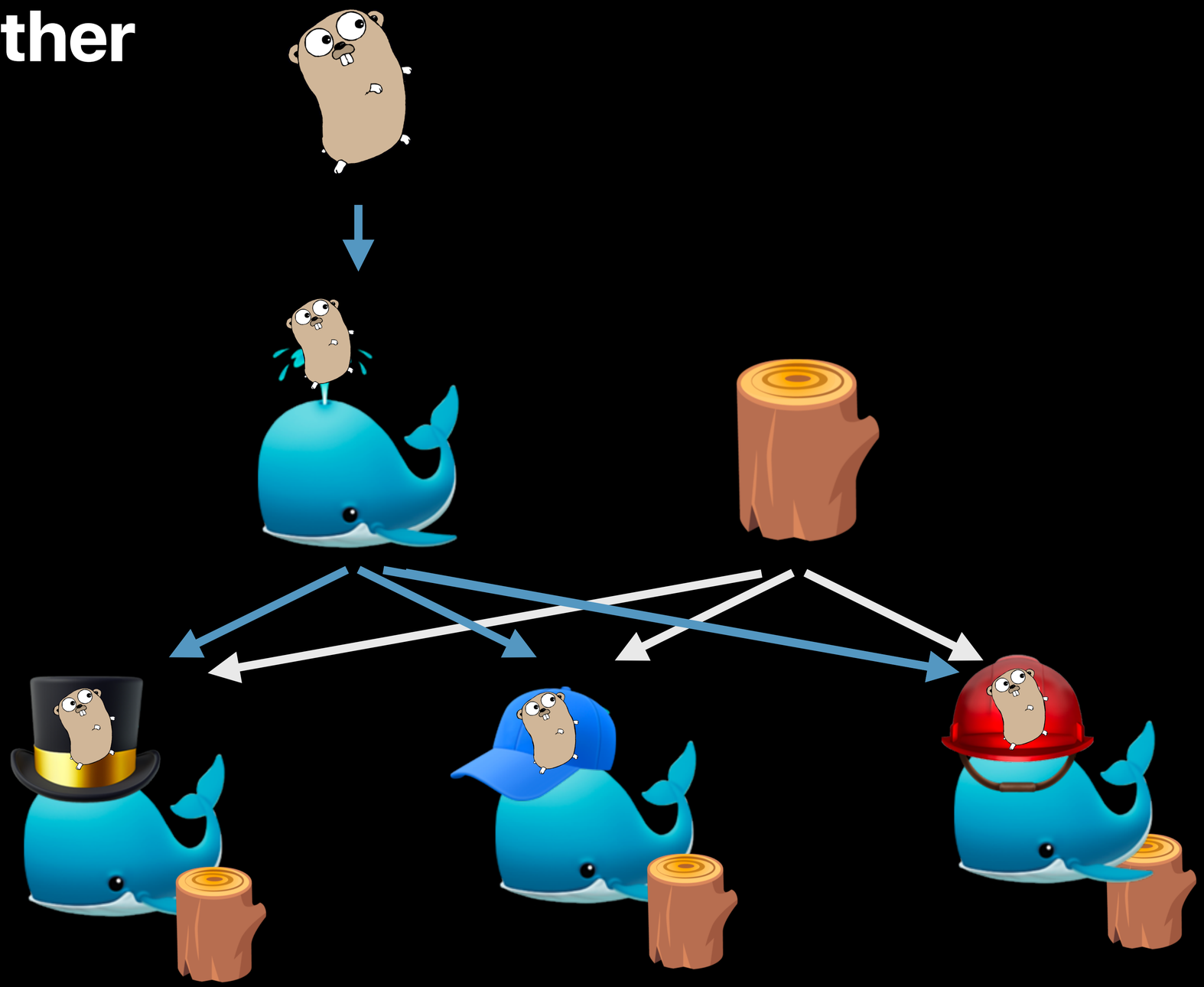
Using CLIs for Configuration

Cross-Cutting Concerns



Putting it all together

Combining Goals



Outcomes of Combining CLIs

- Architecting your system
- Continuous integration

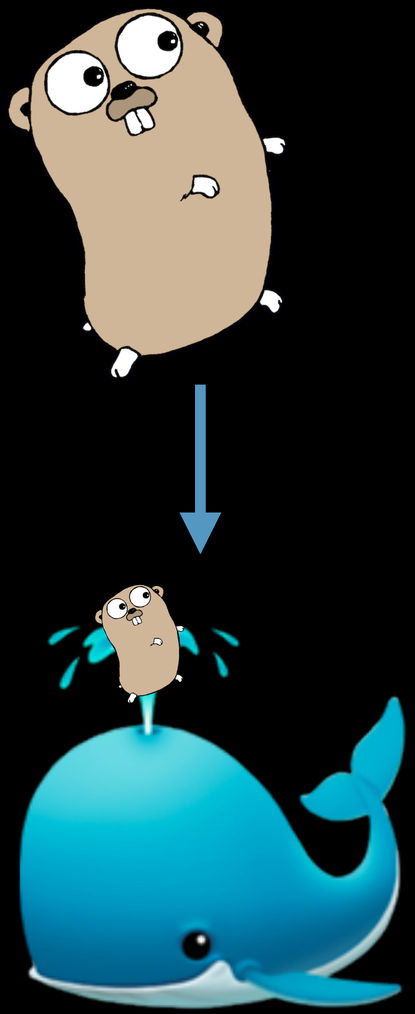
Act II: 🎶 How do you measure, measure a [CLI]? 🎶

Templating	YAML Composition	Domain-Specific Languages (DSL)	General Purpose Languages	Controller-esque
 ytt		 cruise-automation / isopod		

Templating

Helm, ytt

- Template accepts pre-defined input values
- Input values expanded, producing output resources
- Most common for defining **Abstractions**



Templating Demo

Helm, ytt

Templating

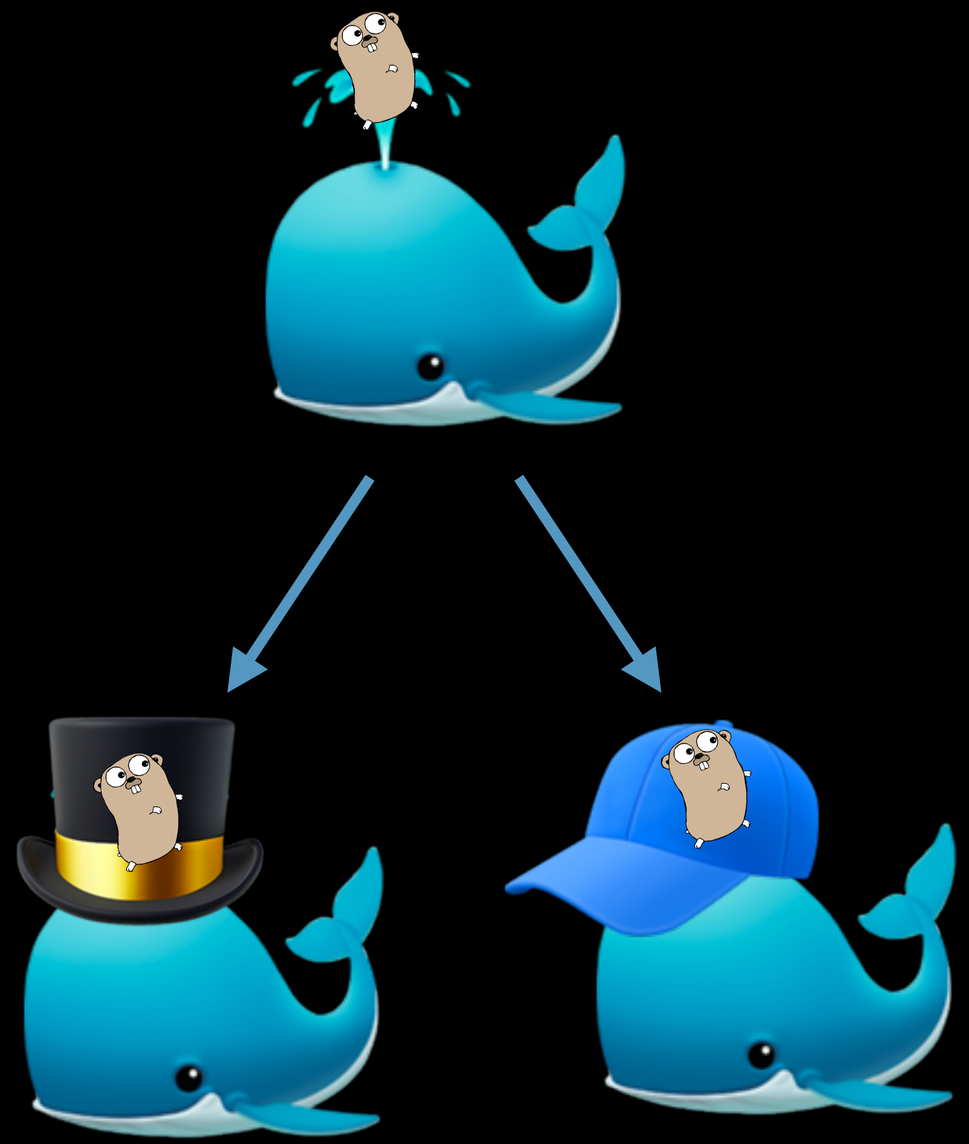
- ✅ **Pros**
  - Simple to use — take a pre-packaged solution and run with it — helm install!
  - Simple to create — relatively mature field with lots of examples and prior art

- ❌ **Cons**
  - Limited ability to support long tail of low-level API options
  - Unable to natively compose opinions of multiple teams into a single template

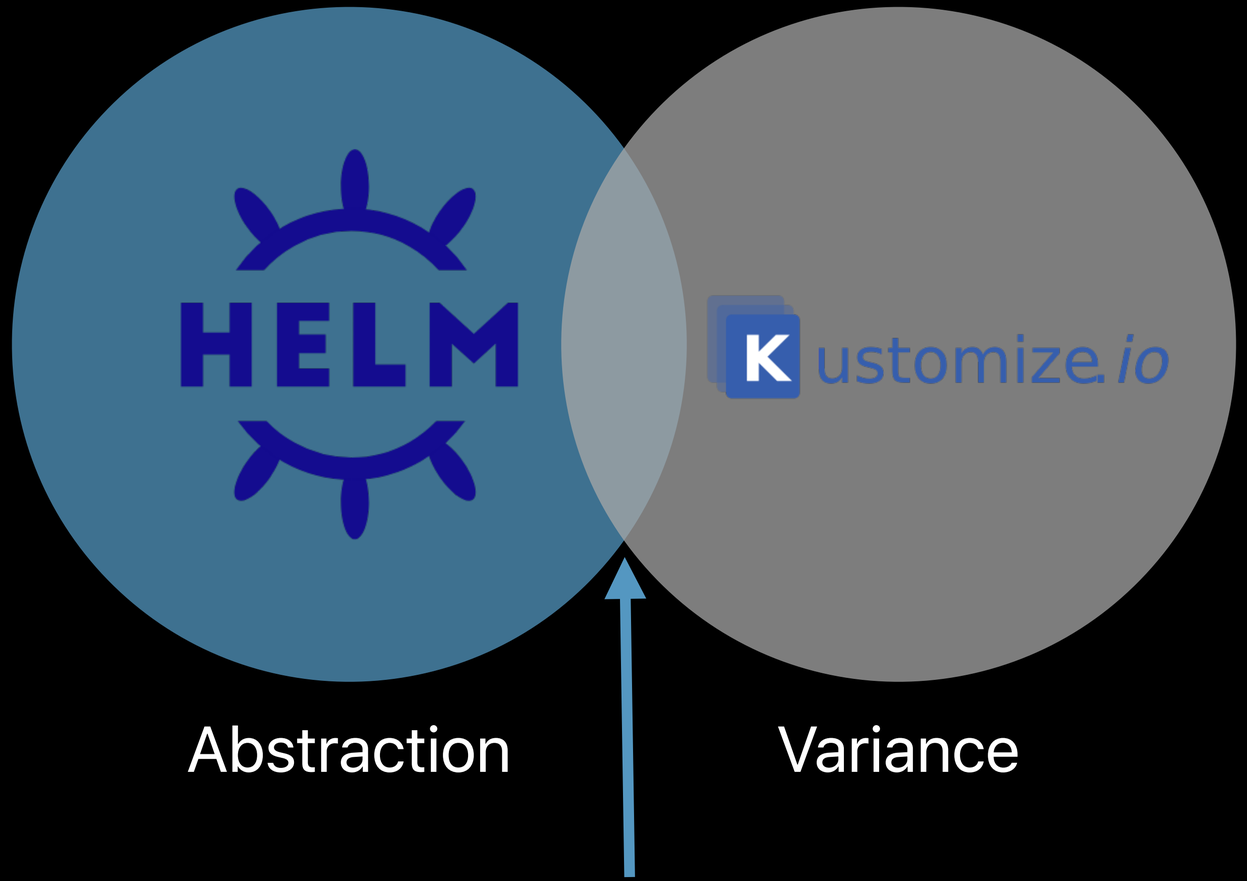
YAML Composition

Kustomize

- Express the system in the native APIs supported by the server — Deployment, Service, ConfigMap
- Break out common YAML pieces into separate files
- Break out varying pieces into separate files that can be applied for different environments



Chaining CLIs



Cross-CLI behaviors are possible and in fact, supported!

YAML Composition

Kustomize

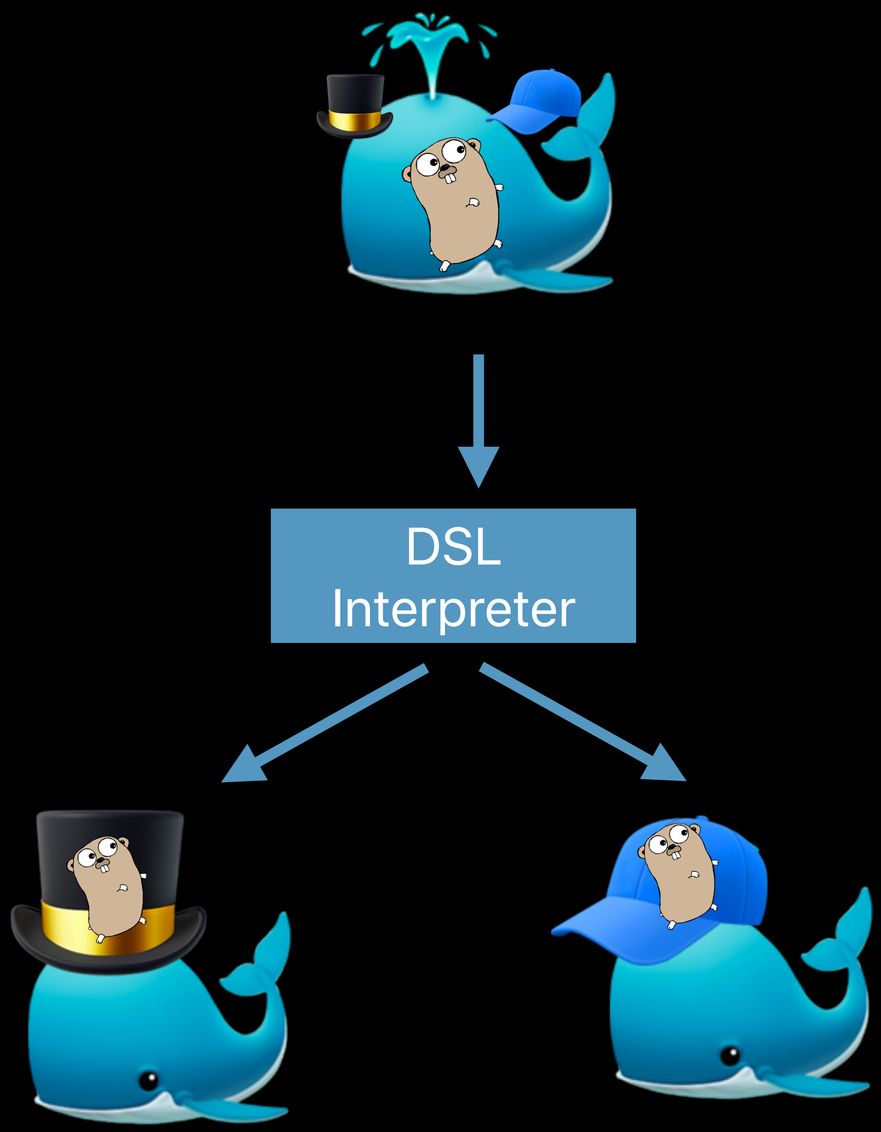
YAML Composition

- ✅ **Pros**
  - Highly customizable
  - Does not hide the underlying APIs that are used
- ❌ **Cons**
  - Static definitions — later layers do not self-reconfigure in response to changes in earlier layers.
  - Do not provide high-level definitions — only operate on native API types

Configuration Domain-Specific Languages (DSLs)

Cuelang, Isopod, JSonnet

- Languages with a specific focus on solving configuration problems
- Provide built-in solutions for compose and transforming configuration data
- *Often use either inheritance or unification as the primary mechanism for abstraction*
- Similar to YAML composition with dynamic logic and abstractions



Domain-Specific Languages (DSLs) Demo

Cuelang, Isopod, JSonnet

Domain-Specific Languages (DSLs)

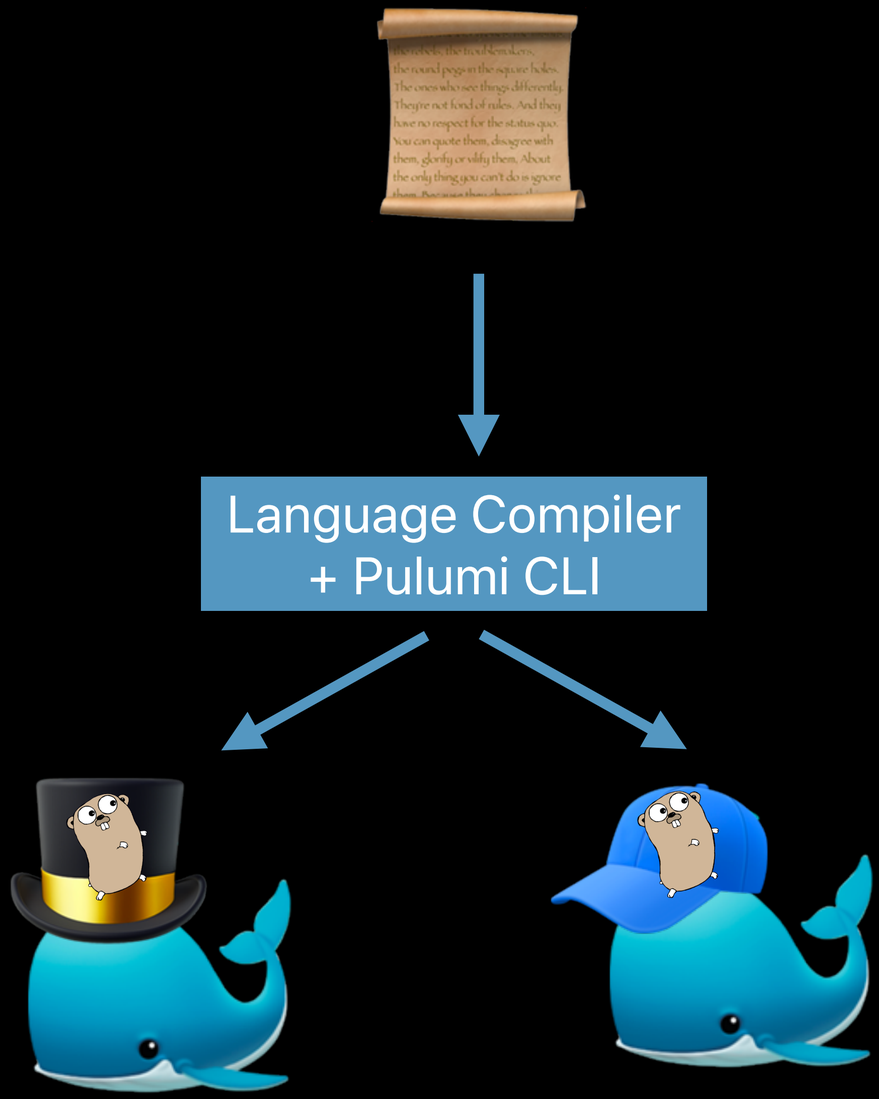
Cuelang, Isopod, JSonnet

- ✅ **Pros**
  - One-stop-stop for Abstraction, Variance, Cross-Cutting Concerns
  - Powerful and expressive
- ❌ **Cons**
  - Different mental model than imperative general purpose languages
  - Need to be a standard — even just 2 different DSLs in an organization is too many and create problems
- Tips
  - Avoid doing “clever” things that make the control flow hard to follow

General-Purpose Languages

Pulumi, kpt-sdk

- Use a general purpose programming language such as Python, TypeScript, Golang, etc to express your configuration
- Familiar model for reusing libraries and developing modules
- Express configuration in code rather than data
- May be part of an all-in-one orchestration framework
- Often able to ingest other sources of input — Helm, Kustomize, etc



General-Purpose Languages Demo

Pulumi, kpt-sdk

General-Purpose Languages

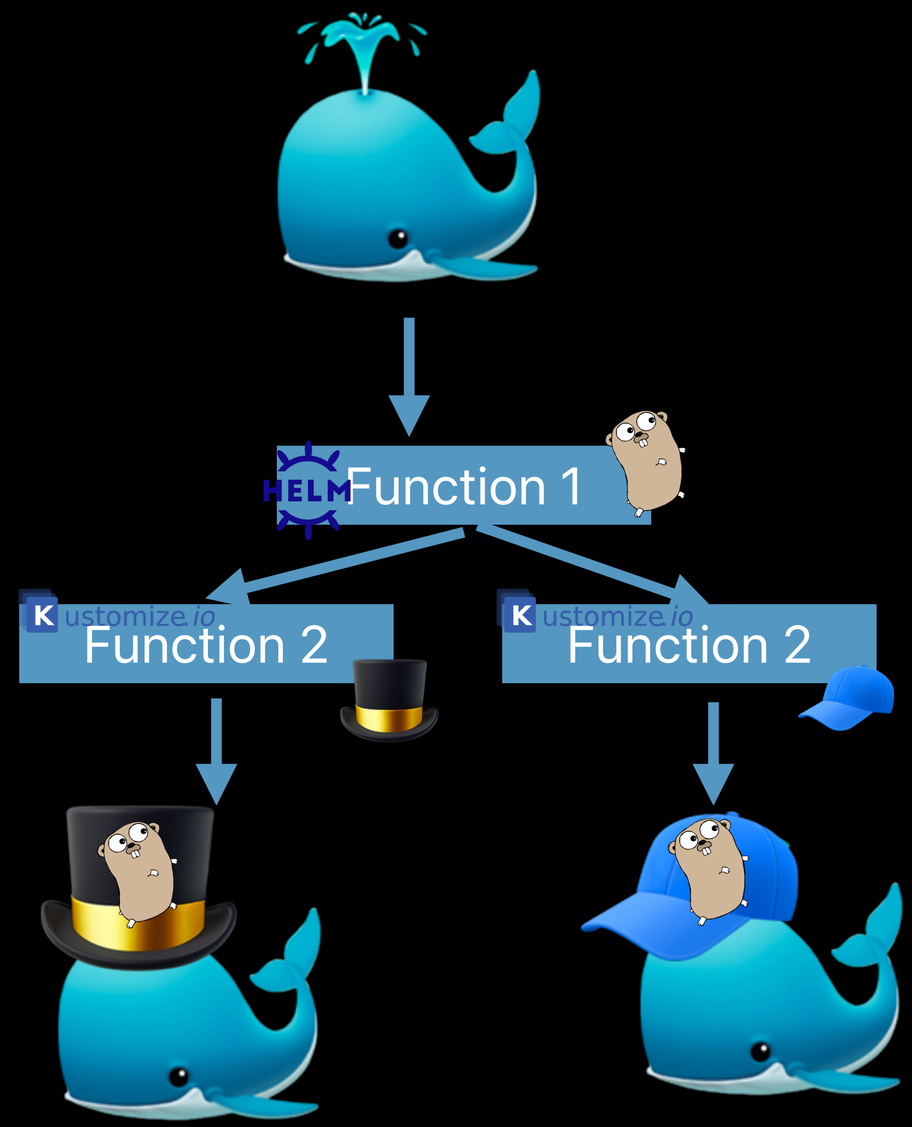
Pulumi, kpt-sdk

- ✅ **Pros**
  - Leverages existing libraries and tools for writing software that folks are already proficient in — e.g. IDEs, linters, testing libraries
  - May perform sophisticated and complex tasks outside of generating configuration — CICD, provisioning cloud infrastructure, etc
  - May ingest configuration from other sources and apply orchestration
  - ❌ **Cons**
  - Evaluating complex programs for correctness can be difficult
  - Syntax of languages rarely optimized for defining configuration data vs DSLs
  - May have friction if using another orchestrator

Controller-esque

Kpt

- Model Kubernetes APIs — read input resources and modify them to match some desired outcomes
- May read both input values and secondary values — e.g. both the “Golang Program” (high-level) and a “Deployment” low-level to turn into a Golang program



Controller-esque Demo

Kpt

Controller-esque

Kpt

- ✅ **Pros**
  - Similar capabilities to Kubernetes APIs themselves, but limited to configuration
  - Highly composable declarative format
- ❌ **Cons**
  - Can be more difficult to write modules using this approach than others — module must consider the full state of the system and be capable of changing it rather than generating new state
  - New and experimental — limited resources and best practices

Act III: 🎶 You could use a little flow 🎶

	Abstraction	Cross-cutting Abstraction	Variance	Familiar
Templating	✅			✅
YAML Composition			✅	✅
Domain-Specific Languages (DSL)	✅	✅	✅	
General Purpose Languages	✅	✅	✅	✅
Controller-esque	✅	✅	✅	

🎶 I try to open up to what I don't know 🎶

Thank you!

We’re Hiring

Stop by the Apple Booth