Kraken

Infrastructure as Code for Kubernetes

Student:	Eoin Fennessy
Student Number:	20100018
Project Coordinator:	Colm Dunphy
Project Supervisor:	John Rellis
Module:	Project and Work Placement
Course:	HDip in Computer Science

Table of Contents

Table of Figures

Declaration of Authenticity

I declare that the work which follows is my own, and that any quotations from any sources (e.g. books, journals, the internet) are clearly identified as such by the use of ‘single quotation marks’, for shorter excerpt and identified italics for longer quotations. All quotations and paraphrases are accompanied by (date, author) in the text and a fuller citation is the bibliography. I have not submitted the work represented in this report in any other course of study leading to an academic award.

Student: Eoin Fennessy

Date: April 4th, 2024

Acknowledgements

Thank you to my Project Supervisor John Rellis and Project Coordinator Colm Dunphy for their help and guidance while working on this project. Thank you to all the lecturers and staff at SETU’s HDip in Computer Science course for all they have done to help me get to the point of taking on this project.

Abstract

Kraken is an Infrastructure as Code (IaC) tool for Kubernetes (K8s) that provides declarative, idempotent APIs for provisioning & managing cloud infrastructure with any cloud provider. It allows infrastructure resources’ generated state to be shared with other, dependent resources and used to dynamically create/update their configurations.

Using Kubernetes’ controller pattern, Kraken reconciles desired state by reacting to both user-provided config updates and dynamic changes to resources’ dependent values. It is also self-healing, and will revert externally-made changes to its managed infrastructure.

Kraken is built around a modular & extensible design, and provides a specification that allows new providers/modules to be developed and integrated. It offers K8s cluster integrations such as configuration import/export, and Kraken configs can be versioned and managed using existing GitOps systems for K8s.

This report will detail the development of Kraken, including information on the initial research and analysis that was carried out, system modelling & design, project planning approach taken, and implementation.

Research and Analysis

This section will describe the first stages of Kraken's development, including the initial concept, analysis carried out on the capabilities & features of existing similar technologies, defining the project's requirements, and choosing suitable technologies to be used.

Context

The initial ideas for the Kraken project came about because of positive and negative experiences I'd had using a variety of different methods for provisioning and managing cloud infrastructure. I wanted to create a tool that would combine the best of each while eliminating any of their major drawbacks, as well as providing new features that could offer improved usability and unique functionality not offered by existing solutions.

Each of the methods I had previously used fitted into one of three main categories: User interfaces such as web consoles and CLIs; imperative, script-based methods such as libraries or SDKs for existing programming languages; and declarative, config-based tools. Each comes with their own advantages and disadvantages.

User interfaces such as web consoles and CLIs are generally easy to use and get started with. They are perfect for doing lots of little things on the fly - performing CRUD operations and observing the state of deployed infrastructure. However, doing anything complex requires lots of manual steps, making it prone to human error and difficult to repeat. Additionally, UIs (especially web-based) can sometimes be unintuitive.

Imperative, script-based tools such as SDKs and libraries for common programming languages enable the automation of infrastructure provisioning. Because these tools are typically available for a variety of common languages, users can generally get started quickly and easily. It also means that state required by and/or produced by the infrastructure can easily be shared by/with larger applications. However, writing robust imperative scripts can be a very tedious and error-prone task requiring lots of checks on existing infrastructure state, setting up "waiters" for conditions to be met, handling different types of client/server errors etc. This can get exponentially more difficult with added scale and complexity.

Declarative tools generally offer the best automation experience, as they provide a higher level of abstraction compared to imperative tools and offer idempotent APIs that are capable of dealing with most or all existing system states. Some solutions also offer Git integration for versioning configs and providing further automation. However, many solutions are cloud vendor-specific or offered proprietary systems using domain-specific languages which required a significant level of learning. Also, some tools in this category had proprietary licensing and/or paid usage plans.

I created the following table to briefly summarise and compare the pros and cons of each method side by side.

Method	Pros	Cons
User Interfaces (Web consoles and CLIs)	• Ease of use • Observability • Convenient queries	• Manual nature • Human error • Sometimes unintuitive
Imperative Scripts (SDKs and Libraries)	• Automation (okay) • Common languages • Integration (state sharing)	• Low level • Tedious • Error-prone
Declarative Tools (config-based)	• Automation (best) • High-Level • Idempotence	• Proprietary systems • Vendor-specific • Learning curve

Figure 1: Summary of Infrastructure Management Methods and their Strengths & Weaknesses

Conception

I wanted to develop a solution that would combine the best of all the aforementioned methods while minimising each's weaknesses. This is how I began thinking about how a set of Kubernetes APIs could achieve this using common patterns and widely known tools.

Firstly, Kubernetes comes with its own user interface in the form of kubectl, a CLI for interacting with the Kubernetes kube-apiserver (web consoles offering similar functionality are also available). Kubectl provides a common interface for all types of Kubernetes resources, and anyone with any experience of the tool would be able to use it to create, update, delete or describe infrastructure resources and carry out other useful tasks like accessing logs, watching resource state, and performing powerful queries on the fly.

Secondly, similar to SDKs for common programming languages, Kubernetes would allow users to use a common "language" in the form of structured YAML or JSON config files to create infrastructure resources. The schema of these configs would adhere to Kubernetes standards. Kubernetes APIs could also offer integrations with the other parts of a Kubernetes cluster, allowing state to be imported/exported to/from other Kubernetes resources or applications running on the cluster.

Lastly, Kubernetes APIs are declarative and idempotent by design, offering a simple, high-level interface for users that makes it easy to automate deployments without having to control the flow of a program by dictating order of events or conditional logic.

Additionally, Kubernetes APIs are designed to be self-healing, allowing them to watch for and react to any changes in internal or external state - something that the three considered methods cannot offer out of the box.

There are also many tools for integrating metrics and observability elements with Kubernetes APIs that could be useful for the likes of creating dashboards for infrastructure administrators or notification services site reliability engineers. Other common tools for Kubernetes, such as GitOps systems could also be used to version and automate the management of infrastructure.

Out of all the cons listed in the methods above, the only one that I thought would apply to a Kubernetes-based tool was the steep learning curve that would be experienced by anyone without prior Kubernetes admin/developer experience. Another potential drawback I thought of was it's requirement of having a Kubernetes cluster to run on, although I figured that many businesses requiring infrastructure automation would be willing to dedicate some compute on existing K8s clusters for this. Additionally, small, low-powered clusters such as KinD could also be used with Kraken, potentially running on single-compute-node clusters.

Similar Technologies

Before defining the initial requirements and targeted features for Kraken, I took some time to research some of the existing IaC tools. This helped me gain a better understanding of the kind of features that are present and missing in existing offerings, and helped inform what could potentially be implemented in Kraken. I limited my investigation to declarative tools, since I knew that is what would deliver the best user experience and would ultimately be expected of a Kubernetes-based tool.

I started my investigation with Terraform, which is a declarative IaC tool developed by HashiCorp. Users write configs in a language called HashiCorp Configuration Language (HCL) (although JSON can also be used). The language breaks configs up into typed blocks composed of attributes. For example, a `resource` block defines a resource of a certain type with a unique name in its header. The main body of the block then contains attributes defining the config for that resource.

Other HCL blocks exist for configuring cloud providers, variables, Terraform itself, etc. Each block gets initialised by Terraform, and when initialised, expected outputs of each resource type can be shared and effectively used as

variables with other resources. HCL also includes a language server enabling intelligent code completion, error-checking, and other static code analysis. Additionally, Terraform offers validation for configs using the `terraform validate` command.

Some of Terraform's most valuable features are in it's `plan` and `apply` commands. When `terraform plan` is called, a dependency graph for the desired infrastructure configuration is built, Terraform then traverses this graph while querying the actual state of the system (external infrastructure services/ APIs), and then forms a new graph which is a diff of the desired state minus the actual state. Terraform then forms an execution plan to reach the desired state by traversing this graph. This plan can then be inspected by the user and applied to their infrastructure. (Hinze, 2016)

AWS CloudFormation is a declarative IaC tool that can be used for provisioning infrastructure on AWS. Users configure infrastructure setups using templates that are written using either JSON or YAML, and apply this configuration using either the AWS console, CLI or web APIs. CloudFormation is free to use, but is limited to managing AWS-provided resources only.

With CloudFormation, infrastructure resources can be grouped together in "stacks", and each resource in a stack can reference values created by other resources in the stack in their config. An update to a stack creates a "change set", which describes the changes that will be made to the resources when the update is applied. AWS also provides "Git sync" which enables synchronising infrastructure with configs maintained in a Git repository.

Requirement Gathering

After reviewing similar IaC technologies on offer, I started to prepare some high-level requirements and baseline features for the project based on what I thought would best serve the end user, and what I thought would lend themselves to a Kubernetes-based app.

The first requirement I defined was that Kraken's components should be modular. This means that each infrastructure service should have an associated controller and API that can be installed alongside other components, and each of these components should be independent of one another. Controllers should be developed to implement a certain interface that enables them to integrate with the rest of the system. The end user should be able to install, integrate, upgrade, and delete each as required, without affecting the other components.

Another requirement I defined was that Kraken APIs should be idempotent and thread-safe. It should not matter what state the system is in when Kraken configuration is being applied. Kraken should always be able to reconcile the

required state, and it should be able to handle cases where multiple actions are being carried out on an infrastructure resource at the same time.

Finally, I wanted infrastructure resources to be able to share their generated state with other infrastructure resources without needing to be aware of the other resources' existence. Users should be able to add references to values from other Kraken resources in each config and Kraken should be able to handle these dependent values appropriately - watching for value creation/change and queueing a reconciliation when the value is made available or altered.

Differentiating Features

I took some time to think about potential features and characteristics that could differentiate Kraken from existing infrastructure as code tools. The two key differences I focussed on were its integrations with Kubernetes and the k8s ecosystem, and the potential advantages that continuously running IaC processes could offer.

Being able to share the config and generated state of infrastructure resources between Kraken and other Kubernetes apps and resources could offer many benefits to developers. For example, an application load balancer that targets a service running on a number of EC2 instances could be configured using Kraken. Kraken could export the generated IP address of the load balancer to a k8s configMap resource which could be mounted to pods running on the cluster running services that access the external service using the load balancer's IP address.

Having continuously running processes dedicated to managing IaC could potentially offer some advantages when compared to traditional run-once scripts or APIs. One that I was particularly keen to explore with Kraken, was the ability to self-heal.

Self-healing could mean that the system would periodically poll infrastructure services to determine if any changes had been made to the desired state. These changes could have been made in error by people attempting to manually change the state of Kraken-managed infrastructure, or it could be because of cloud provider drift, where IDs or IP addresses change because the cloud provider had to rebuild infrastructure. These changes would then be known to the Kraken and dealt with appropriately.

Another idea that came to mind was the ability to "import" infrastructure config. Perhaps Kraken resources could be configured to discover resources (e.g. by matching specified tags/labels), and manage this pre-existing infrastructure as if it were its own.

Tools & Technologies

Before starting on the modelling and design stage, I took some time to investigate the tools and technologies to could help me implement the features.

After some initial investigation, it seemed that it was possible to write a Kubernetes controller in any language that provides a supported client library for Kubernetes, and these included many common languages such as Python, Java, Ruby, C, and Javascript. However, it seemed that de facto standard for creating k8s controllers is Go (Golang). A benefit of using Go was the officially supported controller-runtime library (absent in other languages), which provides additional tooling for creating controllers. Additionally, Go provides excellent concurrency features for easily and efficiently 'watching' Kubernetes resources.

The controller-runtime library is leveraged by the Kubebuilder and Operator SDK projects, which both provide tooling for Kubernetes controller projects. They both help scaffold an initial project structure with developer tooling including commands for generating API definitions & code, running controllers locally, building a controller's container image, and pushing the container image to a controller registry.

Kraken would require tools for interacting with external services offered by cloud providers. Some cloud providers provided language-agnostic tools for managing infrastructure using HTTP web APIs (e.g. AWS Cloud Formation). Like Cloud Formation, some of these tools also offered some form of declarative infrastructure management. However, these APIs typically returned JSON responses, and I felt that managing and unmarshalling the complex data that was returned would be a very substantial task, and could add to the maintenance burden in the long term as APIs change. I also felt that converting Kubernetes configs to other declarative configs while maintaining records of the state of each could become quite complex. For these reasons, I decided to opt for officially supported SDK libraries, as they typically provided native data types for all required/returned data structures, including errors. Using imperative SDKs would add a little more complexity when programming for idempotency, but I felt the tooling and finer-grained control were probably worth it.

For local development, I would require a tool for building and running containers. Two tools that could be used for this include Docker and Podman, which both offer similar functionality. I would also require a locally-running Kubernetes cluster for testing controllers on. A lightweight option that doesn't require additional virtualisation and runs in a container is KinD (Kubernetes in Docker).

For managing source code, I would need a Git hosting repository that offered tooling for CI/CD. Both GitHub and GitLab provide these features. I would also need a container registry for pushing and pulling containerised builds of code changes. Again, both Docker Hub and Quay.io would be able to provide those services.

Modelling and Design

This section will describe the modelling and design stages that were taken before development of Kraken commenced. The design went through a number of iterations and refinements before I was happy that it had reached the stage where I could start working on code.

Initial Design Model

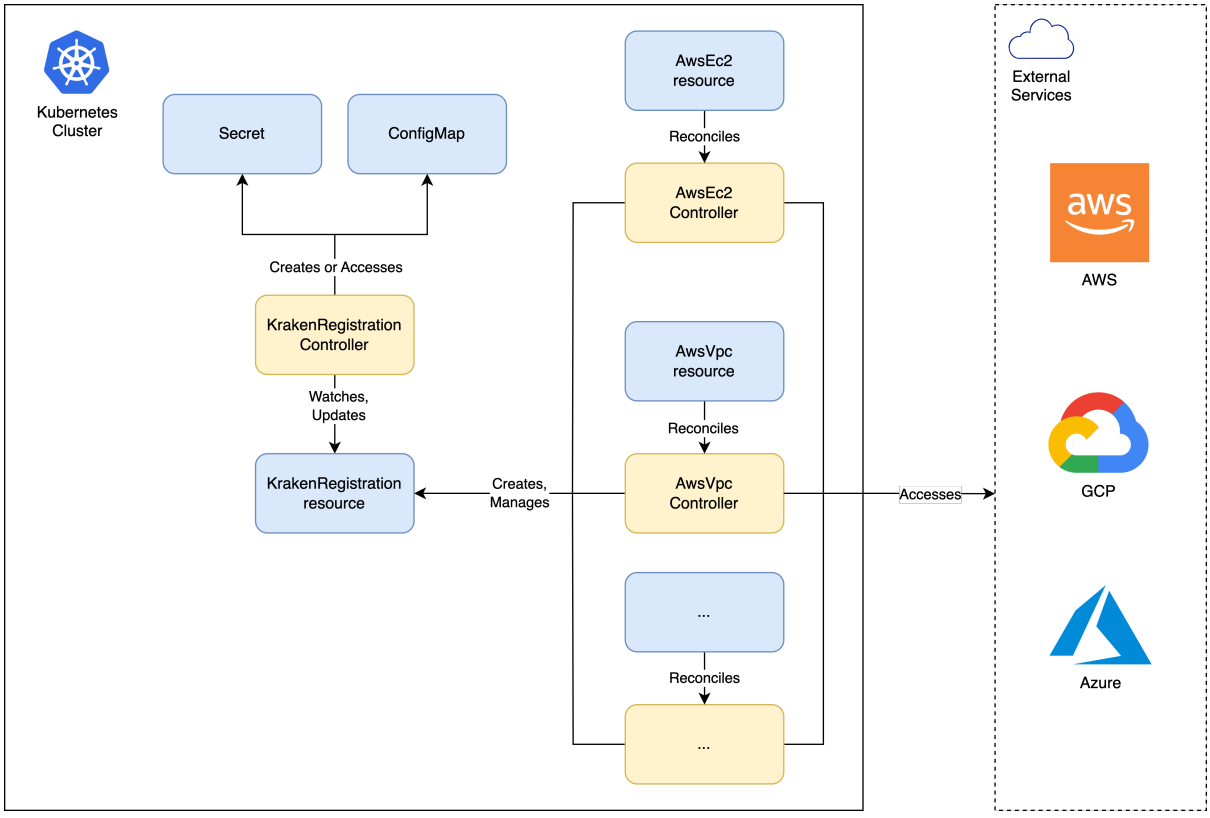

Figure 2: Kraken's Initial Design Model

The initial design model for Kraken was to have one controller per infrastructure type. Each controller would watch for CRUD operations on instances of an associated k8s custom resource. Each of these resource instances would contain the desired state of that infrastructure type as

specified by the end user. The controller would then perform the necessary steps to reconcile the current state of infrastructure with the desired state.

Users could either specify literal values for each field in an infrastructure resource's spec or reference values (dependencies) created by other infrastructure resources. These values could be referenced using a text templating library provided by Go, somewhat similar to the following.

```
apiVerion: kraken.io/v1alpha1
kind: AwsEc2
metadata:
  name: my-ec2-instance
spec:
  vpcId: "{{ .AwsVpc.my-vpc.id }}"
  ...
```

Each infrastructure resource would both share its state and request any state it depends on by creating a `KrakenRegistration` resource. The `KrakenRegistration` controller would update the status of this resource to include depended-upon values when they become available (or change). Once all of these values become available, a reconciliation of the dependent infrastructure resource (the owner of the KrakenRegistration resource) would be triggered and the associated infrastructure could be created/updated.

Additionally, there would be some kind of functionality to export (part of) an infrastructure resource's state to a Kubernetes ConfigMap or Secret so that it could be used by other Kubernetes resources. This functionality would be provided by the KrakenRegistration controller. Similar functionality would be provided to enable import of data from ConfigMaps or Secrets.

The self-healing behaviour would be made possible by enabling users to provide a configurable time interval on each infrastructure resource specifying how often the resource should be requeued so that its actual state can be measured and re-reconciled with the desired state if required.

Infrastructure resources' scope would be limited so that they could only access state from resources residing in the same Kubernetes namespace. There may also be options provided that allow limiting what state infrastructure resources make available to other Kraken resources.

Refinement

Talk about potential shortcomings of initial design, Single reponsibility principal, Cross field validating web hooks, Ownership, watch

After the initial modelling and design I took some time to consider some of the potential shortcomings of the design and how they could be addressed. I was

largely happy with the majority of the plan, but felt there were two things that could be improved.

The first design issue I saw was in how depended-upon values were referenced in the spec of infrastructure resources. Providing them as string templates meant that every field would have to be unmarshalled into a string data type. This was far from ideal, as many fields would contain (and reference) other data types such as numbers, booleans, lists, maps, etc. and it would require a considerable amount of work managing conversions for each.

I considered two different options for overcoming this. The first was to define data structures wrapping each primitive type that could contain a static, literal value, and a `valueFrom` type where a depended-upon value could be referenced. A validating admission webhook would be used to ensure that only one of these options gets provided. An interface for such a type (which could maybe be called an `IntOption` type) is shown below.

```
...
spec:
  replicaCount:
    value: 42
    # or
    valueFrom:
      krakenResource:
        type: AwsEc2
        name: my-ec2-instance
        field: maxInstanceCount
      # or
      configMapKeyRef:
        name: my-config-map
        key: count
      # or
      secretKeyRef:
        name: my-secret
        key: count
```

And a similar interface for a `StringOption` type, which adds a `templateString` option:

```
...
spec:
  vpcId:
    value: abc123
    # or
```

```
      valueFrom:
        krakenResource:
          type: AwsVpc
          name: my-vpc
          field: id
        # or
        templateString: “{{ .AwsVpc.my-vpc.id }}-567def”
        # or
        configMapKeyRef:
          ...
        # or
        secretKeyRef:
          ...
```

An alternative to this I considered was splitting the specification up into `static` and `dynamic` blocks. Again, validating webhooks would have to be implemented to ensure only one option is provided. Such an interface might look like the following.

```
...
spec:
  static:
    replicaCount: 42
  # or
  dynamic:
    replicaCount:
      fromKrakenResource:
        type: AwsEc2
        name: my-ec2-instance
        field: maxInstanceCount
      # or
      fromConfigMapKeyRef:
        name: my-config-map
        key: count
      # or
      fromSecretKeyRef:
        name: my-secret
        key: count
```

In terms of implementation and work required on the backend, I thought that both options would probably function comparably. The choice seemed to be largely based on UX—a tradeoff between having to add `.value` to each static field, or having to split configs up into `static` and `dynamic` blocks. In the end, I felt that having to add `.value` to each field would be the easier of the two to manage, even though it did seem a little inelegant.

The second major design issue I saw was in the way in which infrastructure resource state was shared and requested with/from other infrastructure resources.

In the initial design model, a `KrakenRegistration` resource gets created by each infrastructure resource. Its spec contains the list of dependencies—values from other resources it depends on. The `KrakenRegistration` controller updates the status of this resource with the depended-upon values when they become available. In addition, this resource also gets updated with the state that gets produced by the owning infrastructure resource—state that may get propagated to other infrastructure resources. This felt like the `KrakenRegistration` resource had too many responsibilities, and it could potentially lead to reconciliation issues where more than one controller is updating the state at the same time.

My solution was to split the `KrakenRegistration` resource up into two parts. First, a `DependencyRequest` resource whose responsibility was to request depended-upon values and receive these values in updates to the resource's status. The remaining responsibility consisted of storing infrastructure resource state so that it could be accessed by the `DependencyRequest` controller and shared with other infrastructure resources. I initially considered using Kubernetes ConfigMaps for this, but these do not allow for any complex nesting of values, and all values must be strings. I decided instead to create a `StateDeclaration` custom resource. This resource would be able to hold any JSON serialisable data, including arrays and nested objects. The `DependencyRequest` controller would watch any `StateDeclaration` resources required by each `DependencyRequest` resource and add any referenced state to the status of the `DependencyRequest`, where it could be consuming and used by the controller that created the `DependencyRequest`.

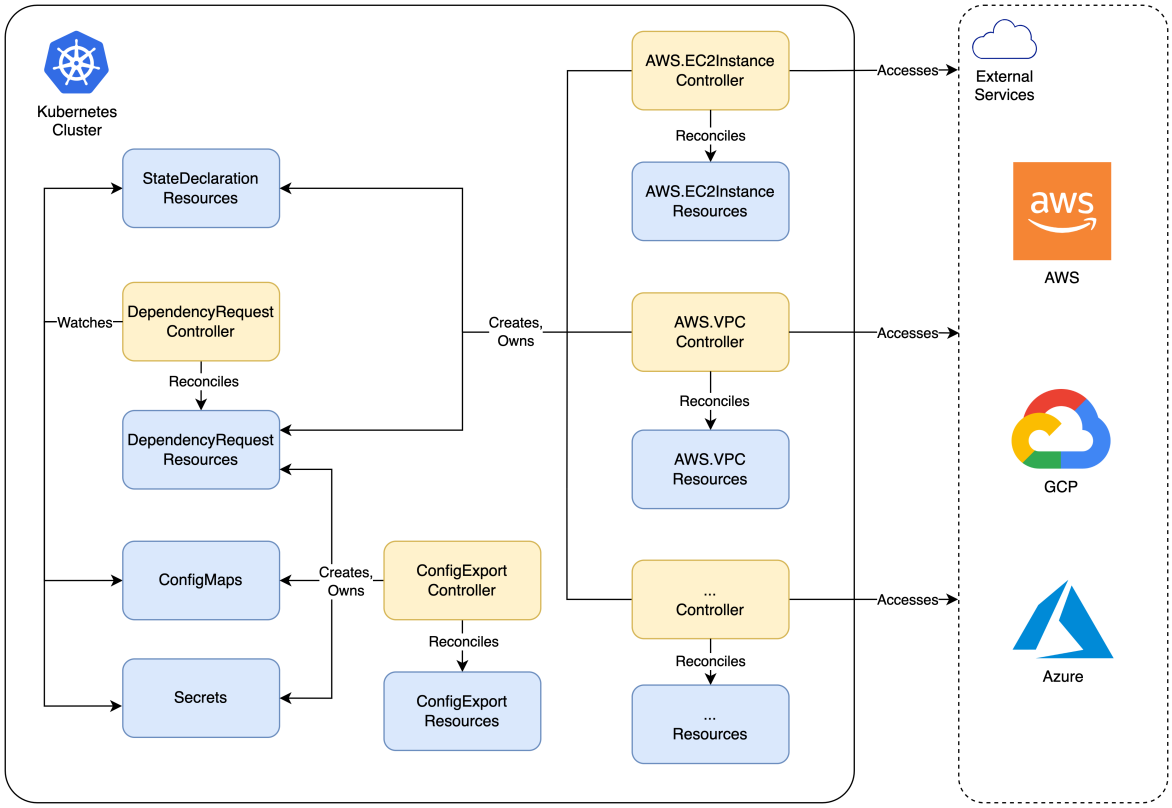

Figure 3: A Refined Design Model for Kraken

The above figure represents a refined design model for Kraken, with relationships between controllers (orange) and resources (blue) included.

Central to the design of Kraken is the DependencyRequest controller, which manages the reconciliation of DependencyRequest resources. These resources can be created by any infrastructure controllers, and reference depended-upon values for an infrastructure resource. The depended-upon values can reference state in both ConfigMaps and Secrets (native Kubernetes resources) and also custom StateDeclaration resources, which represent the state of any infrastructure resource. The DependencyRequest controller watches all resources referenced in a DependencyRequest resources and gathers the required state as it is created or updated. The DependencyRequest controller then updates the DependencyRequest resource's status to include these values, which triggers a reconciliation of the infrastructure resource for whom the DependencyRequest was created.

The controllers for every type of infrastructure resource must implement Kraken's standards for referencing/requesting state (DependencyRequest) and exporting state (StateDeclaration). Any number of new infrastructure

controllers could be developed and integrated as long as they follow this standard.

A builtin ConfigExport controller is also represented in the above figure. This controller reconciles ConfigExport resources. These resources contain references to values provided by any other infrastructure resource. The ConfigMap controller creates a DependencyRequest for these values, and creates & maintains either a Configmap or Secret from the state that is added/ updated in the status of the DependencyRequest resource.

Planning

This section will outline the expected scope of the project, testing strategies, release processes, and how progress the project will be managed, including agile methodologies, issue tracking and planning.

Scope

The goal of this project is to produce a set of proof of concept applications that demonstrate the core functionalities of Kraken. These core functionalities include allowing generated state to be shared with infrastructure resources that reference it in their configurations, and allow state to be imported from/ exported to standard Kubernetes config resources such as ConfigMaps and Secrets.

This will require the development of at least three APIs and associated controllers:

1. A ‘hub’ controller that provides a standard interface for all infrastructure resources to communicate: It watches for requests for referenced state from all infrastructure resources, watches the source of the referenced state, and shares the up-to-date state with the requesting infrastructure resource.

2. Any number of ‘spoke’ infrastructure controllers: These will interact with cluster-external cloud services to provision and manage infrastructure resources. For each configured infrastructure resource, the controller may create a request for dependent state referencing some other state of the system. The controller will watch for updates to this request for state and provision the associated infrastructure when all required state has been gathered. It will continuously watch for changes (updates/deletions) to dependent state and react accordingly to update the provisioned infrastructure.

3. A controller that will import and export referenced state to standard Kubernetes resources such as ConfigMaps and Secrets.

The system should be fully event-driven and react appropriately to all state changes to ensure that the actual state of the system is always reconciled with the desired state. If the desired state is not possible, possibly due to a misconfiguration or service-provider error, it should be easy for users to track down the source of the error by inspecting the status of their configured resources.

Because this is a proof of concept, implementing the core functionality is more important than having a fully production-ready system. While it would be nice to have a variety of infrastructure resources on different cloud platforms to demonstrate, it would be sufficient to have one or two well developed infrastructure resources that provide a blueprint for the general implementation pattern and utilise the standard Kraken interfaces. Additionally, infrastructure resources can offer opinionated defaults and/or expose a reduced set of configuration options to allow development efforts to focus on more key features.

Project Management and Agile Methodologies

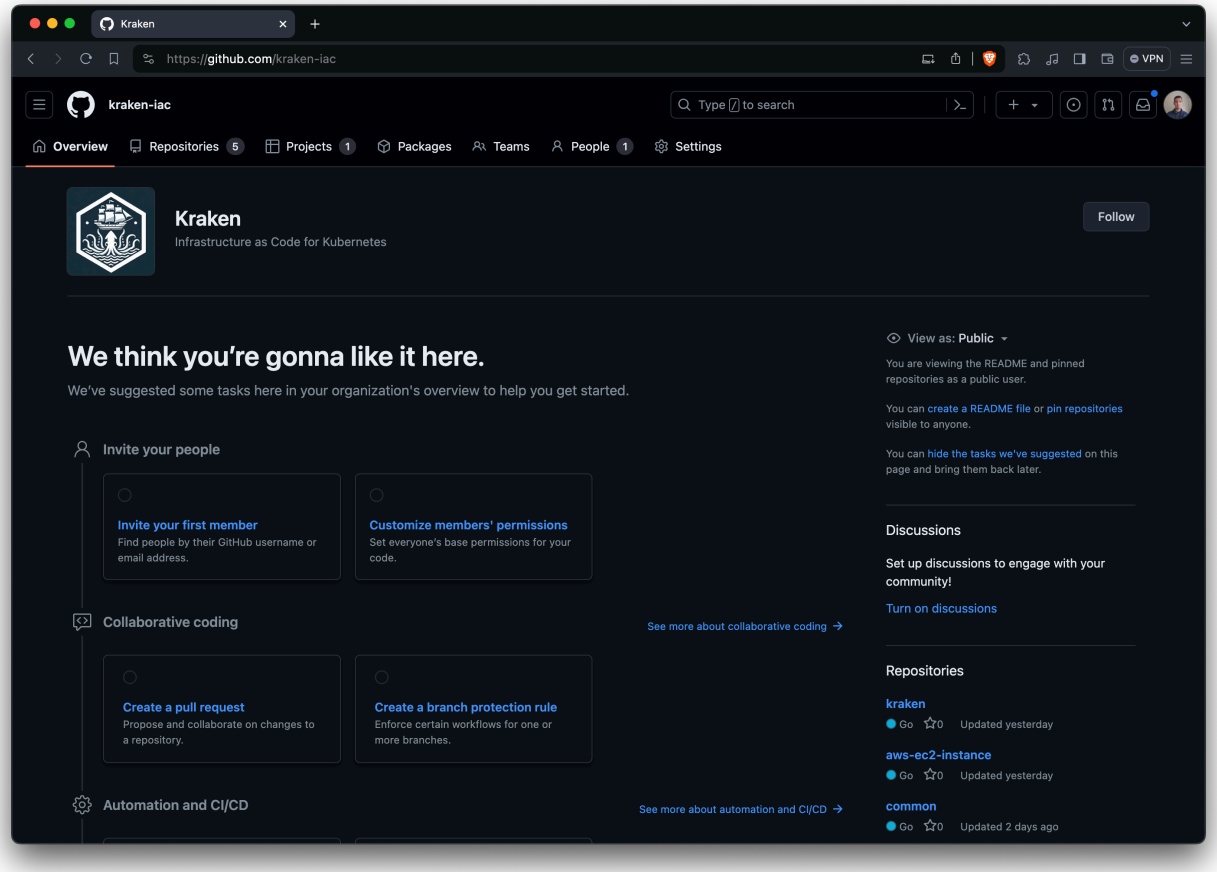

Figure 4: Kraken's GitHub Organisation

A GitHub organisation will used to group all the code repositories for the project. This organisation will include shared configuration such as variables and secrets to be used by all repositories in CI/CD pipelines.

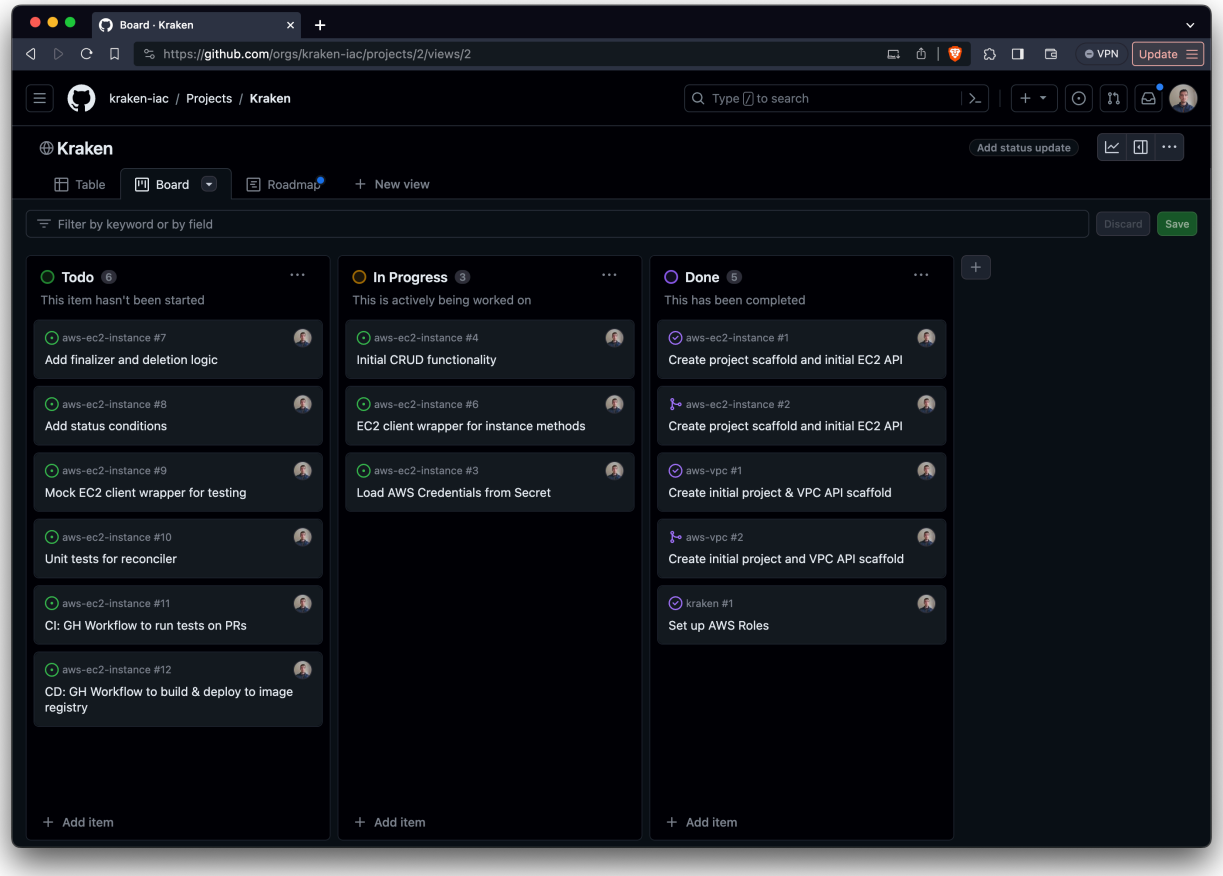

Figure 5: Project Kanban board view

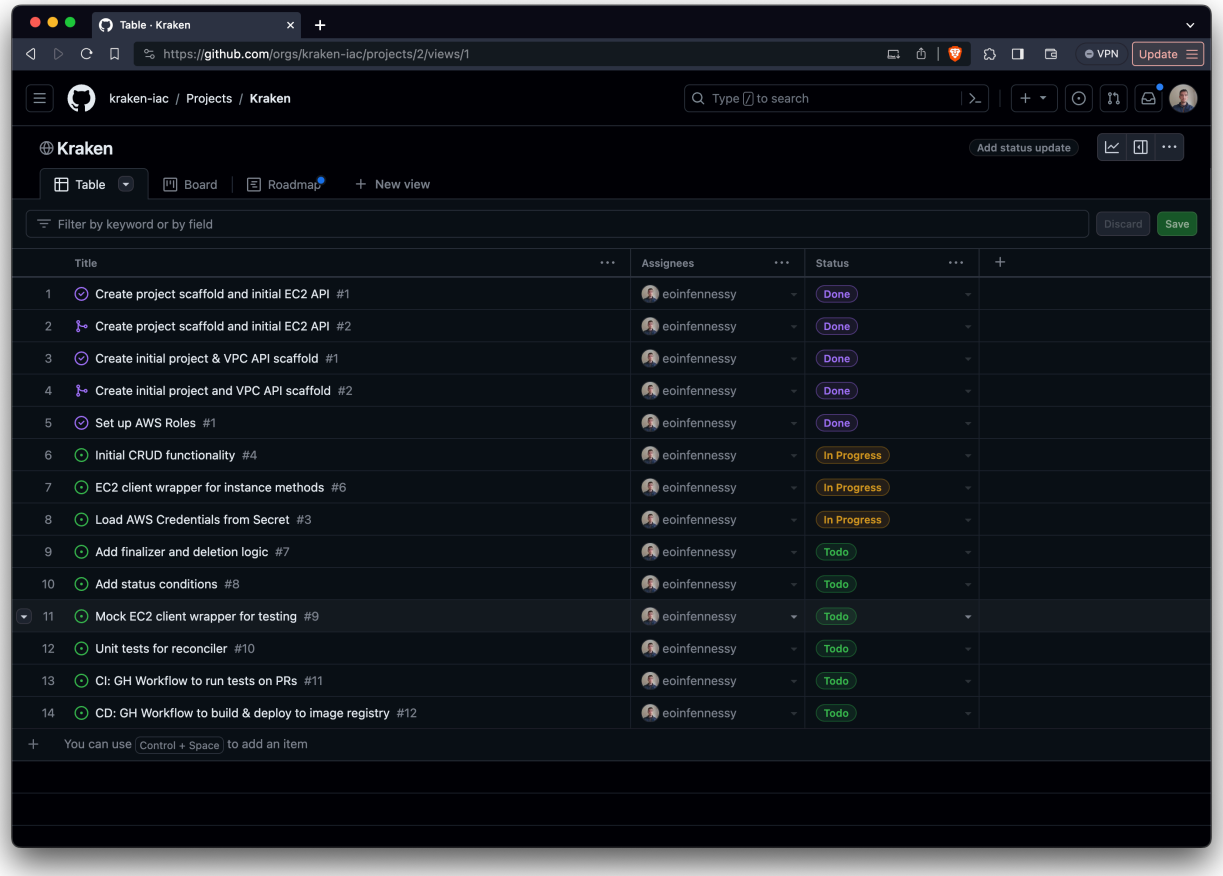

Figure 6: Project backlog view

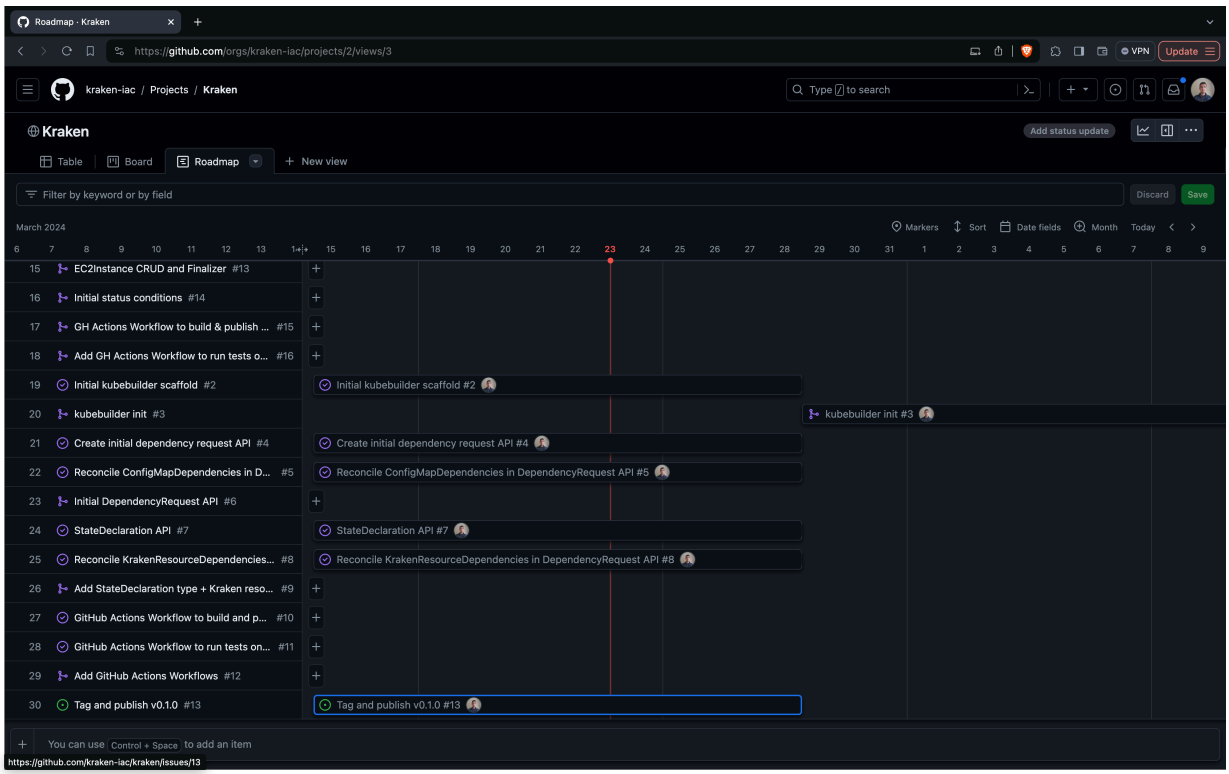

Figure 7: Project timeline view

A GitHub Project will be used to keep track of all project management and progress in all repos. Issues created in each repo will be linked to this project so that all progress can be seen and planning carried out in a unified dashboard. The views above show some of the project views that will be used for planning and progress tracking, including a Kanban board view to show at a glance the state of all work, a list view to prioritise items at backlog refinements, and a timeline view used for sprint planning.

I will be adopting a number of agile processes to manage work on the project. Sprints will be used for planning short bursts of prioritised work, while backlog refinements will regularly take place to line up work to be pulled into each sprint. The Kanban board will largely just be used to get an at-a-glance view of the current state of work.

Testing and Continuous Integration

Testing will be carried out in two main forms. Firstly, table-driven unit tests will be written to test small units of functionality against a varied set of inputs. Secondly, behaviour-driven integration tests will be used to test the reconciliation logic of controllers by providing well-defined narratives on what the expected behaviour should be in different situations.

For testing components that access external services such as those from cloud providers, mock clients will be developed that implement the same interface as the real clients. These mock clients will be passed to consumers in place of the actual client so that the likes of infrastructure resource reconciliation can be tested without depending on cloud providers' services.

End-to-end tests would also be desirable in this project but a decent implementation of such a suite is probably out of the scope of this project.

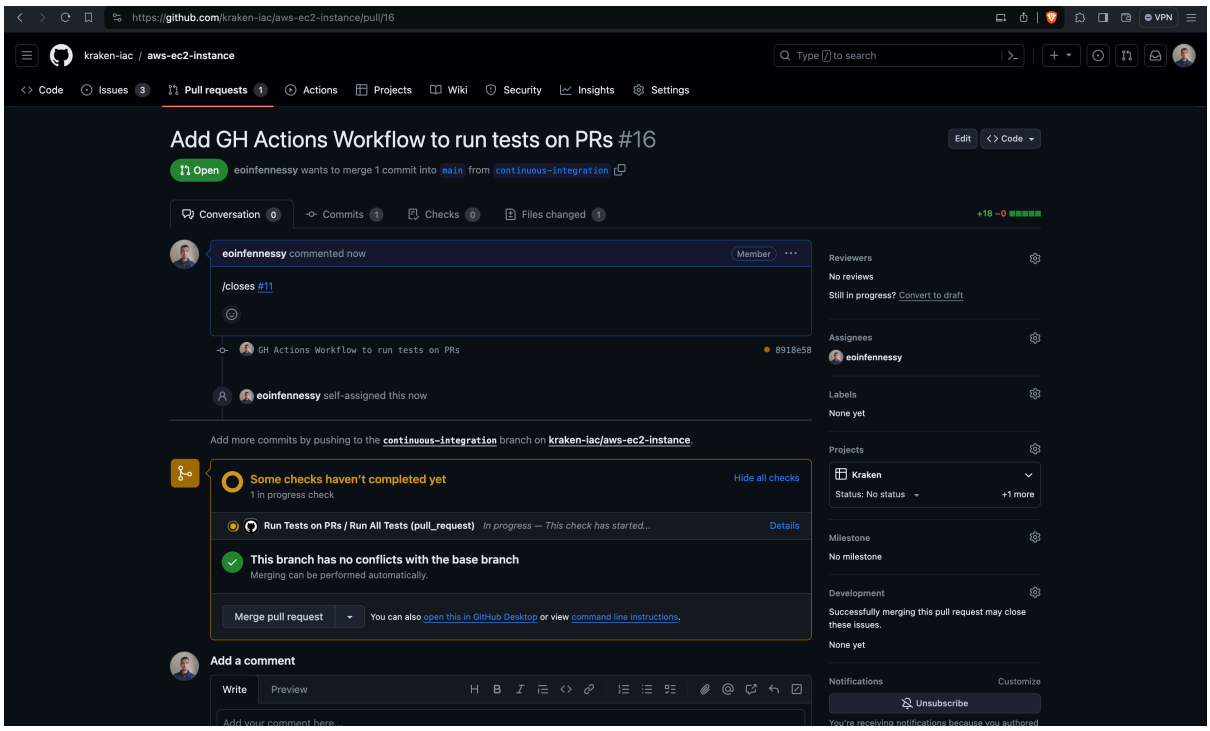

Figure 8: Automation running tests against pull requests

A continuous integration workflow will be used to build the project and run all tests against pull requests. GitHub Actions Workflows will be used to configure continuous integration workflows.

Release Process - Container Images and Packages

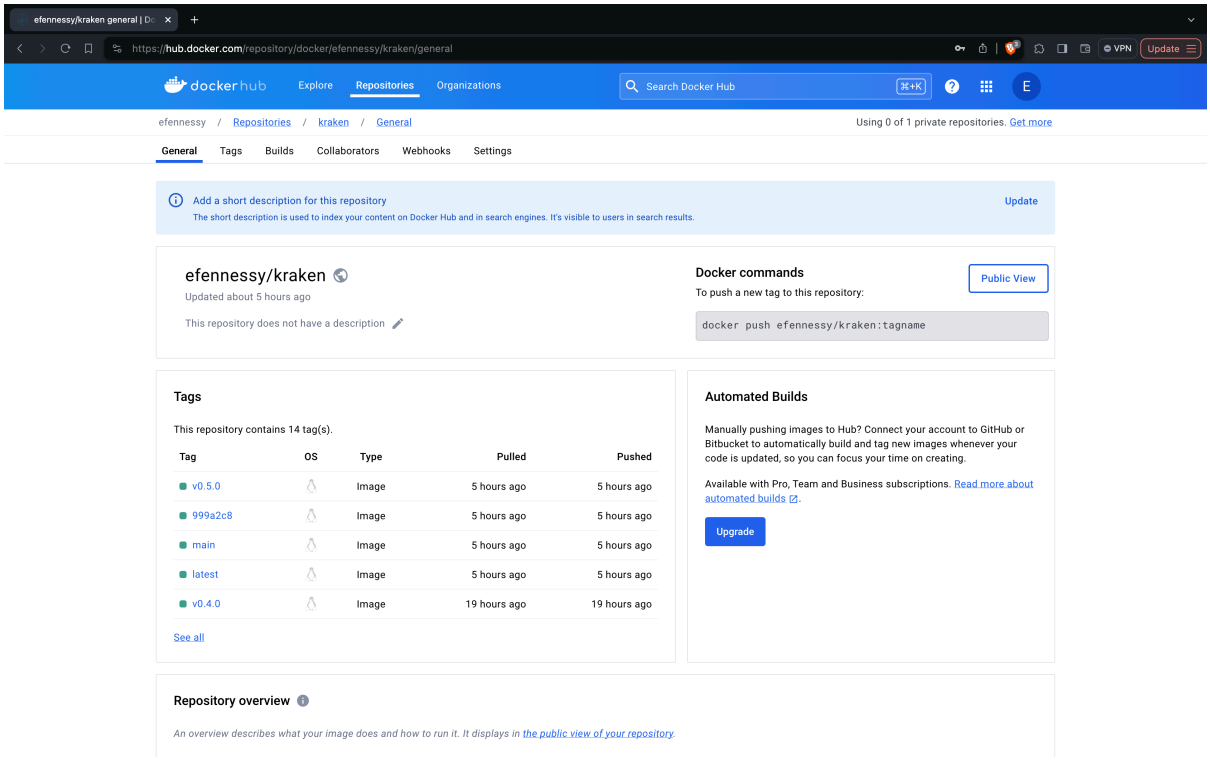

Figure 9: Kraken container repository on the Docker Hub image registry

Container images for releases will created using GitHub Actions Workflows and hosted in public repositories on the Docker Hub image registry. For each merge to the main branch, an automated CD pipeline will be triggered that will build a new container image and push it to the image registry with the `latest`, and `main` tags and a tag of the Git commit's SHA. The workflow will also be triggered when a new Git tag (e.g. `v0.1.0`) is created, and the container image will be tagged accordingly.

Released packages will be available on GitHub, and package documentation will be available on go.dev.

User installations of Kraken components will be managed with Kustomise. Users will be able to specify the versions of components they want to install by pointing Kustomize to install a versioned configuration on a tagged GitHub repo reference. This config file will install to the user's cluster the specified container versions, custom resource definitions, and other configurations.

Implementation

GitHub Org and Project Views

The first step I took was to create the `kraken-iac` org on GitHub. This org contains all of the code repositories related to the Kraken project.

I also created an org-level GitHub Project, which contains unified views used for tracking work across all repos in the org. I created a backlog/list view, a Kanban board view, and a sprint planning/timeline view. When creating issues and PRs in each repository, I assign them to this project so that each can be prioritised, planned, and tracked at appropriate stages of sprint planning and backlog refinement.

Scaffolding Components

Initially, I created repos for the core kraken component and the aws-ec2-instance infrastructure component. For each repo I scaffolded out an initial project structure using the Kubebuilder CLI, adding appropriate metadata to uniquely identify each component under a specified domain and group. I also scaffolded placeholder APIs for the EC2Instance and DependencyRequest custom resources and versioned them as `v1alpha1`.

AWS Configuration

Using my personal AWS account, I created an organisation to manage Kraken configurations. I created an IAM permission set with limited permissions for creating, listing, updating and deleting EC2 Instances. I assigned this permission to a Kraken user group and created a user that would assume these permissions. I then generated credentials to be used by the AWS EC2 client to manage EC2 instances.

Initial EC2Instance Functionality

The initial objective was to incrementally build up a relatively full-featured EC2Instance API that allows EC2 instances to be created declaratively. However, this initial iteration would not offer any functionality to reference external state via the core Kraken APIs or share state with other infrastructure resources. The sub-sections below will outline the work carried out to reach the objectives of this initial iteration.

EC2 Client Wrapper and Interface

I used the official EC2 SDK from AWS to create a wrapper client that offers a reduced interface designed to specifically serve the functionality required in the EC2Instance controller.

The wrapper client implements an interface that the consumer (the controller's reconciler struct) expects. I hid some complexity behind this interface by doing work like type conversions, providing opinionated defaults, etc. in the client wrapper. Because the consumer of the client wrapper expects an interface type rather than a concrete type, a mock client can later be provided for testing purposes.

Reading AWS Credentials from Secrets

When running locally, the EC2 client expects the secret AWS credentials to either be exported as environment variables or contained in a `~/.aws/credentials` file. When the EC2 client is running in the controller's container on the Kubernetes cluster, the credentials need to be provided in the environment of the container. To achieve this, I configured the controller deployment to take it's environment from a reference to a Kubernetes Secret:

```
envFrom:
  - secretRef:
      name: aws-credentials
```

A user creates this Secret on the cluster before deploying the controller, which gets loaded with environment variables taken from the Secret. If the Secret is not provided, the deployment logs a suitable error message and status condition to let the user know that the Secret must be provided. I also created a "local" deployment config that generates the secret from a file reference and applies it to the cluster:

```
secretGenerator:
- envs:
  - aws-credentials.env
  name: aws-credentials
```

Initial API Definition

I created a Go struct with JSON tags and that defined an initial spec for the EC2Instance resources that included the image ID, instance type, max & min counts of instances, and AWS tags, as well as standard Kubernetes object metadata such as name, namespace and labels.

The Kubebuilder CLI generates from this struct a Custom Resource Definition that contains an OpenAPI spec with definitions of expected datatypes, required/optional fields and additional validation that I specified using Kubebuilder comments, such as only allowing numbers greater than zero for the maxCount field.

Create/Update Functionality

Using the API definition I started work on the reconciliation logic for the EC2Instance kind.

In `cmd/main.go`, an `EC2InstanceReconciler` object is created by passing in client objects for EC2 and Kubernetes. The reconciler has a `SetupWithManager` method that, among other things, is used to specify to the controller manager the type of resource the controller reconciles. Whenever a resource of the associated type is created, updated, or deleted, the `Reconcile` method of the `EC2InstanceReconciler` is called passing in the key (namespaces name) to retrieve the object using the K8s API & client. It is within this `Reconcile` method found in `internal/controller/ec2instance_controller.go` that most of the key functionality and logic of the API is defined.

In the `Reconcile` method, the initial functionality consisted of setting up a logger from the provided context, fetching the EC2Instance resource, fetching any running EC2 instances associated with the resource, update/recreate/scale up/scale down the resources to reach the provided state. Any errors encountered at any stage in the function cause the reconciliation of the resource to be requeued. When writing the reconciliation logic, I am conscious to make each step idempotent and thread-safe to ensure the desired state can be reached regardless of the actual state of the world and to try to avoid reconciliation loops where possible.

Finalizers & Deletion Logic

Finalizers are attributes that can be added to a resource that must be removed before the object can be deleted. In the `Reconcile` method, I added a step after fetching the EC2Instance resource to add a finalizer if one is not already present.

When an attempt is made to delete an EC2Instance resource, the resource is given a deletion timestamp and the `Reconcile` method is called on it. I added a check to see if the object is marked for deletion near the start of this method and if so perform the deletion logic (terminate any running instances) and then remove the finaliser so that the resource can be fully deleted. If an error occurs while terminating instances, the finalizer is not removed and the reconciliation is requeued.

Status Conditions

Status conditions are a Kubernetes-standard way of representing the state of a resource. Each condition includes a condition type, such as “Ready” or “Error”, the status of that type (either true, false or unknown), the reason for the status, such as “Reconciled” or “CrashLoopBackOff”, and a human-

readable message. Each resource can have a list of status conditions of different types.

For the EC2Instance resource, I created one status condition with condition type “Ready”. When the resource is first reconciled, this status condition is added with condition status unknown and the reconciliation is requeued. If there is an error in the reconciliation, the status is set to false and updated with an appropriate reason and message. When the status is fully reconciled the status of the “Ready” condition is set to true.

Integration Tests and Mock EC2 Client

To test the reconciliation logic, I created a mock EC2 client that implements the interface expected by the consumer, the `EC2InstanceReconciler` object. Using the behaviour-driven Ginkgo testing framework, alongside a mock Kubernetes client, I wrote some integration tests for the controller.

Before each test I recreated the test environment and any sample data. I then performed tests to ensure the reconciliation functionality would behave as expected when given certain inputs and that no errors would be returned after the `Reconcile` method was called with good input data.

CI/CD Workflows

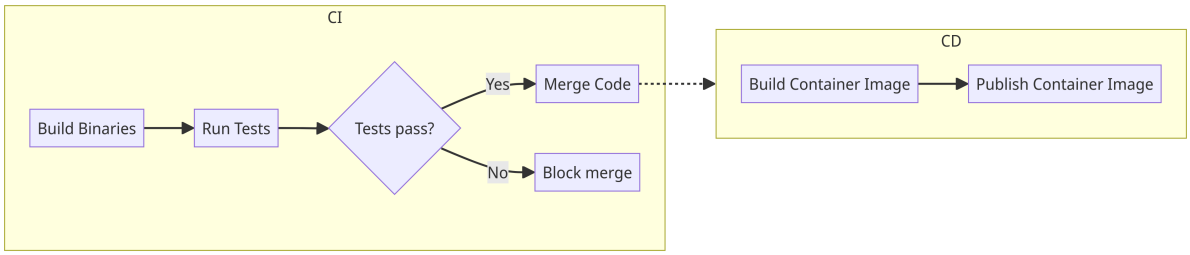

Figure 10: CI/CD pipeline for pull requests

Using GitHub actions workflows, I configured Continuous Integration and Continuous Delivery pipelines for the aws-ec2-instance repository.

Pull Request Tests

The CI workflow is configured to run on pull requests to the main branch. Using a base Ubuntu container, the code for the PR is checked out, the specified version of the Go tool is installed, and the code is built and all tests ran using the `make test` command. This job can be seen running on the pull request view on GitHub, and if the job fails, merging the PR is blocked.

Automate Container Image Build & Publish

The CD workflow runs on merges to the main branch or release branches, and also when tags are created. It uses a base Ubuntu image to checkout the code, generate tags for the container image, log in to the image registry, and build and push the code using Docker and the repo's Dockerfile.

In order to authenticate to the container image registry, GitHub-organisation-wide secrets and variables are configured on GitHub that contain the image registry host, org, username, and credentials. A git-repo-specific variable is also set that specifies the image registry repo to push to.

DependencyRequest API

After laying the groundwork with the first iteration of the EC2Instance API, I started work on the core Kraken APIs that provide a standard by which any and all infrastructure resource types can import and export state, both with other infrastructure types and Kubernetes-native resources.

The DependencyRequest API is an internal API that an end user should never need to create, examine, or manage in any way. It's API enables any implementation of an infrastructure controller to request & receive values that it depends on when reconciling resources. Values can be requested from references to state exported by any Kraken infrastructure resource, without the requirement for the requester to be aware of anything about the resource it is requesting values from. Values can also be requested using references to Kubernetes ConfigMaps or Secrets.

Reconciling ConfigMap Dependencies

The DependencyRequest API allows dependent values to be retrieved from ConfigMaps by specifying the ConfigMap's name and the key whose associated value should be retrieved.

The first iteration of the DependencyRequest `Reconcile` method fetches the DependencyRequest resource, adds an initial status condition if not present, defines a `defer func()` that updates the resource's status on each return. If there are ConfigMap dependencies present in the spec for the resource it will attempt to retrieve each value and add it to the resource's status. If it encounters an unreconcilable error such as a missing ConfigMap or key, it will set the status condition "Ready" to false with an appropriate reason and message and return early without requeueing.

ConfigMaps that each DependencyRequest resource references need to trigger a reconcile for that resource. To do this, an index is set up for DependencyRequest resources on the `ConfigMapDependencies` field. The controller is set up to watch all ConfigMap resources and if the ConfigMap's

name is contained in the ConfigMapDependencies index of any DependencyRequest resource, that resource is queued for reconciliation.

StateDeclarations and Reconciling KrakenResource Dependencies

References to values from other Kraken infrastructure resources can be requested in a similar way to ConfigMap values. This was implemented in a similar way to the ConfigMap dependency, but instead of providing a name and key to request a value, the resource's kind, name, and JSON path to the value requested are all provided. The JSON path field allows users to specify a path to any nested value in a complex structure.

The KrakenResource dependency also requires an expected data type to be provided in the form of a `reflect.Kind` value (underlying type int). The reflect package includes methods for runtime type checking, and these methods are used in the DependencyRequest controller to compare the expected data type to the actual data type of the referenced value and reject the value if it does not match the expected type. This enables end-to-end type safety for the standard Kraken interface—a requested value must always match its expected datatype; otherwise it is not provided to the controller that made the request and an appropriate error message is returned in the form of a status condition.

A StateDeclaration resource is the source of the referenced values. I created the StateDeclaration resource to allow arbitrary JSON-serialisable data to be exported by infrastructure resources and referenced by other resources. It is a controller-less custom resource that is simply used for storing data of any schema.

To reconcile a KrakenResource dependency, the following steps are carried out (assuming no errors):

1. Fetch the referenced StateDeclaration
2. Access the value at the JSON path provided
3. Check that the actual datatype of the value matches the `expectedKind` datatype using the `reflect` package
4. Add the value to the DependencyRequest's status

As described in the ConfigMap dependencies section above, a similar field-indexing and "watch" pattern is implemented to reconcile DependencyRequests when a change is made to their referenced StateDeclarations.

Status Conditions

If any of the reconciliation steps results in an unreconcilable error (missing resource, key, invalid datatype, etc.), the error is represented in the DependencyRequest's status condition and the resource is not requeued.

CI/CD Workflow

A similar CI/CD workflow to the EC2Instance resource was implemented: All tests are run against PRs and merges to main and tag creations trigger a container image build that gets pushed to the kraken repository on the Docker Hub image registry.

Package Releases

Because the API and type definitions for DependencyRequests and StateDeclarations are required by infrastructure controllers such as the EC2Instance, I maintained tagged releases for them on GitHub so that they could be imported in other projects. I also listed the package docs on go.dev.

Export EC2Instance State using StateDeclaration

After creating some of the core Kraken APIs, I turned my attention back to the EC2Instance controller so that it could make use of them.

Option Types

In the first iteration of the EC2Instance I used standard primitive types for the API such as ints and strings. To integrate the core Kraken APIs, I needed to replace these primitive types for new "option" types that I would define. They are called option types because they are (sort of) analogous to the algebraic data types of the same name often found in functional-paradigm languages such as OCaml and Scala.

For example, the `option.Int` type I defined allows users to provide either a concrete int value, a reference to a value found in a ConfigMap, or a reference to a value provided by another Kraken infrastructure resource. Similar types were defined for other types such as `option.String`, and the API would be extended in the future as required.

I started off by including this type in the EC2Instance package but later moved it to its own repo and released a package so that it could be utilised by any infrastructure resource.

Conversion Methods

The option types include methods for converting them to types that can be added to DependencyRequest specifications. They also provide methods for

returning a concrete value or an error when provided a DependencyRequest status that should contain the value the option type references.

Using DependencyRequest API in EC2Instance Controller

To make use of the DependencyRequest API, some extra steps were added near the start of the EC2Instance's `Reconcile` function:

1. Construct DependencyRequest spec from all value references in EC2Instance spec
2. Fetch existing DependencyRequest if one exists
3. Delete old DependencyRequest if it exists and the new DependencyRequest spec contains no dependencies; return
4. Create DependencyRequest owned by EC2Instance resource if one does not already exist and the new DependencyRequest spec is not empty; return
5. Update DependencyRequest if one exists and the new DependencyRequest spec is different; return
6. If DependencyRequest's status is not ready, return without requeue (eventual update to DependencyRequest should requeue)
7. Get applicable values using EC2Instance's spec and DependencyRequest's status and continue with reconciliation

Validating Webhooks for Option Types

Because only one "option" (either a concrete value or a single reference) should be provided for each option type, a more complex validation than the simple OpenAPI spec provides was required. To achieve this I created validating webhooks that watch for create and update events and carry out programmatic cross-field validation to ensure only one "option" is provided. Each option type provides a `Validate` method to make this process simpler.

ConfigExport API

The final feature I wanted to implement was the ConfigExport API. It allows users to reference values exported by Kraken infrastructure resources and exports them to Kubernetes-native configuration resources called ConfigMaps and Secrets that other Kubernetes resources can consume.

The pattern it uses to achieve this is similar to that of the EC2Instance controller: It creates a DependencyRequest for the required values and creates/updates/deletes either a ConfigMap or a Secret as the dependent

values become available/change/become unavailable. The `spec.configType` value provided by the user is a string enum type, that only allows the string literals “ConfigMap” or “Secret” to be provided. An example spec for this API is shown below.

```
spec:
  configType: Secret
  configMetadata:
    name: my-exported-secret
    namespace: default
  entries:
    - key: someKey
      valueFrom:
        krakenResource:
          kind: ec2-instance
          name: my-ec2-instance
          path: myList.0.myMap.foo
```

Reflection

Working on this project, I learnt a tremendous amount about designing Kubernetes controller APIs, operator patterns, creating modular & decoupled systems, defining standard interfaces, hub-spoke architectures, idempotent programming patterns, event-driven programming patterns, behaviour-driven development & testing, CI/CD pipelines, agile methodologies, etc.

I’ve achieved what I believe is a decent proof of concept for a novel approach to infrastructure as code that integrates with Kubernetes clusters and can be used with multiple cloud providers. If more time was available, I would have loved to have spent more time fleshing out the capabilities of the core Kraken interface—supporting references to complex datatypes and things like string templates. I’d also like to have created more infrastructure modules on different cloud platforms so that I could demonstrate them working together and referencing each other.

I encountered many problems along the way—problems working with schema-less JSON data and JSON paths while ensuring end-to-end type safety is one to mind, as well as addressing some tricky reconciliation loops—but all were dealt with in a methodical way by isolating the problem, splitting it into smaller parts, and using tools like debuggers and writing helpful logging and error messages for myself. I think that spending time at the start of the process planning the key design and scrutinising it while iterating on it helped to avoid any insurmountable problems.

Glossary of Terms and Abbreviations

CI/CD (Continuous Integration/Continuous Delivery): CI is the process of automating the integration of code into a code repository, and may include automated builds and tests. CD is the automated deliver and/or deployment of code changes.

CLI: see Command-Line Interface

Command-Line Interface: a text-based interface for interacting with a computer program using a computer's terminal

Container (Docker): In computing, a container is an isolated environment for running an application containing all of the binaries and libraries it requires to run. Containers run on a shared container runtime, and unlike virtual machines, each container does not require a full operating system and hypervisor to run, which makes them much more efficient. (Docker Inc., 2023)

Controller Pattern in Kubernetes: a pattern in which a control loop watches for create, update and delete operations on K8s resources of the controller's associated type (aka kind). When such operations are detected, the controller attempts to bring the current state of the system closer to the desired state that is declared in the resource's specification.

Declarative Programming: a programming paradigm in which the definition of what needs to be accomplished by a program is provided without requiring implementation details that specify how it should be accomplished (Rouse, 2020)

IaC: see Infrastructure as Code

Idempotence: a property of certain operations in mathematics and computer science, that can be applied multiple times without changing the result beyond the initial application (Wikipedia contributors, 2023)

Imperative Programming: a programming paradigm in which code defines a step-by-step process dictating how a program should carry tasks out; the programmer uses statements and expressions to change the state of the program and control its flow

Infrastructure as Code: a method of provisioning and managing cloud computing infrastructure using plain text-based code; an alternative to graphical/UI-based methods of managing cloud infrastructure

Kubectl: a command-line interface used for interacting with Kubernetes' kube-apiserver. It provides a common interface for performing operations on all types of Kubernetes resources including both idempotent and imperative CRUD operations, querying, accessing logs, reading documentation, waiting for status conditions, etc.

K8s: see Kubernetes

kube-apiserver: an application that serves as the frontend to Kubernetes by exposing a RESTful interface through which all external clients and cluster components interact with the cluster resources

Kubebuilder: a CLI that provides tools for scaffolding & managing projects that follow the Kubernetes controller pattern

Kubernetes: a platform that provides declarative, idempotent APIs for managing containerised workloads and services; offers an application-centric approach to service deployment by abstracting underlying computing infrastructure (The Kubernetes Authors, 2023)

Kustomize: a CLI tool that can be used to customise application configuration that gets applied to a Kubernetes cluster. Kustomize is built in to the Kubectl command line tool and allows users to reference remote configs by specifying URLs to configuration files.

Kind (disambiguation: K8s API type): a Kubernetes API type that consists of a schema that defines a declarative interface for creating corresponding resources

KinD (disambiguation: K8s cluster tool): ‘Kubernetes in Docker’ is a tool for running Kubernetes clusters inside Docker containers that function as a cluster’s compute nodes. This tool is typically used for creating local development clusters and in CI workflows such as integration tests.

SDK: see Software Developer Kit

Software Developer Kit: a set of software development tools for a specific platform in a single installable package

References

Docker Inc. (2023). *What is a container?* [online] Docker Documentation. Available at: https://docs.docker.com/guides/walkthroughs/what-is-a-container/.

Hinze, P. (2016). *Applying Graph Theory to Infrastructure as Code*. [online] www.youtube.com. Available at: https://www.youtube.com/watch?v=Ce3RNfRbdZ0 [Accessed 31 Jan. 2024].

The Kubernetes Authors (2023). *Overview*. [online] Kubernetes. Available at: https://kubernetes.io/docs/concepts/overview/.

Rouse, M. (2020). *Declarative Programming*. [online] Techopedia.com. Available at: https://www.techopedia.com/definition/18763/declarative-programming.

Wikipedia contributors (2023). *Idempotence*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Idempotence.