MESHERY

Supporting Design Patterns

An agnostic and holistic approach

Prologue and Challenge

The need for this design specification stems from multiple open needs, one of which is the desire to define a common (and best?) practice of both configuring and operating cloud native infrastructure functionality in a single, universal file. Other specifications exist, but only partially capture these concerns:

- **Kubernetes** manifests describe cluster and container considerations, but leave some workload and application wanting.
    - Issue 1: Applications are left wanting; left under-considered.

- ~~**cloud native infrastructure Interface (SMI)** describes a standard interface for interacting with the functionality of cloud native infrastructurees that run on Kubernetes.~~
    - ~~Issue 1: SMI addresses the lowest common denominator use cases for which service methods are used today. SMI provides a basic feature set for the most common cloud native infrastructure use cases.~~
    - ~~Issue 2: SMI doesn't provide an extensible model for mesh-specific capabilities.~~

- **cloud native infrastructure Performance (SMP)** is a standard format for capturing and characterizing cloud native infrastructure and workload performance in context of the functional value being provided.
    - Issue: SMP focuses on load test profiles and results analysis, which excludes description of traffic split.

Guiding Principles

Adhere to the following design principles:

1. **Establish the world's initial set of design patterns**
   Patterns can be described in an infrastructure agnostic way, while their deployment ultimately is infrastructure-specific.

2. **Reuse where possible. Differentiate where feasible.**
   Build upon the intelligence and functionality of the underlying systems, adding unique value as a solution that offers highly collaborative methods of managing infrastructure and applications.

3. **Be end user-centric; Enable the business function.**
   Simplify. Networking and distributed systems are hard; Connect the dots between dynamic infrastructure and the business.

Design Goals

The designs in this specification should result in enabling:

1. Capture infrastructure *behavior* in a single pattern file.
2. Allow users to access infrastructure-specific differentiation.

Design Objectives

The designs in this specification should result in these specific functions:

1. Define a pattern file format.
2. Identify a pattern file moniker (e.g. MESHERY605 is a two-stage, duration-based Canary) and classification of monikers to allow for variation within a known category (e.g. Meshery600s are Canary patterns) and orchestrate cloud native infrastructure behavior in accordance with the pattern.
3. Create a recognizable suite of pattern icons for common reference and ease of understanding.
4. Unify the specifications listed above into a common pattern file using Meshery Models as an extensible foundation.
5. All components are versioned.

Design Patterns and their Orchestration

The flow depicted in the architecture diagram in Figure 1-1 below outlines the process of registering capabilities (as defined in Meshery Adapters) and the sequence by which their operations are invoked.

Figure 1-1: Interpreting Pattern files in Meshery. Link to slide.

1. **[User]** invokes `mesheryctl pattern apply -f <filename>`; **[mesheryctl]** calls to Meshery REST API OR uses Meshery UI to deploy designs/patterns.
2. **[Meshery Server]** interprets the Pattern and creates an execution plan in the form of a directed acyclic graph (DAG). It will also check the feasibility of the plan execution (i.e. it will check for infinite loops in the graph (pattern)). Once the feasibility of the execution is determined Meshery Server will perform the following checks in the order:
    a. Checks Capabilities Registry for the ability to execute
        i. Answering the question, “Is there an host which is capable of handling a Pattern of this type).
    b. Checks Capabilities Registry if the “Settings” conform to the schema that was registered against this type by the adapter.

   c. Check if the schema of each of the components conforms to the schema registered by the registrant.
3. **[Meshery Server]** invokes Kubernetes / Adapter operations based on the host, acting as an orchestrator in accordance with the planned DAG (akin to Kubernetes CRDs in how they validate CRs and then invoke a controller capable of handling that resource).

Deployment of CRDs from Registrants

Problem: Users face issues when deploying designs which require the presence of CRDs in their cluster. Despite having the ability to tackle this (i.e. provisioning CRDs on their behalf), Meshery server is not acting intelligently. Being an infrastructure management tool Meshery should handle such cases gracefully.

Meshery future will be capable of generating components from different artifact repositories eg: Docker Hub, OCI registries or private repositories. Each component generated, will carry along with it the host information under its metadata.

Based on the host who registered the component, the corresponding operation will be invoked to deploy CRDs/operators/controllers prior to deployment of the pattern.

How will it affect the existing pattern deployment engine?

After patterns are validated and their feasibility is determined, the deployment operations are invoked by corresponding registrants for components. Before starting with component provisioning, each registrant will invoke an operation to deploy CRDs/operators/controllers. This will be performed only once for each unique source.

Caveats/Challenges

1. **Can users configure this behavior? What are the consequences if the user has disabled this behavior?**
   Yes, via a checkbox while confirming their deployment details.
   There is a high chance that deployment will fail, leaving users in a difficult position. Currently, validation checks do not provide guarantees as to whether the design will be deployed or not, and is only evident after invoking deployment operation. So auto deployment of operators should be enabled by default.
2. **Can it affect the user's existing deployment?**
   For helm repositories we interface with helm APIs, and if a release already exists it performs an upgrade.
3. Expose additional behaviors rollback/skip/force if release exist.

Registering of Capabilities: Meshery Models

The intention is to make Meshery act like Kubernetes API however unlike Kubernetes, Meshery Server will invoke RPCs on Meshery Adapters.

In response, Adapters will register Models that contain Component Definitions (along with their schemas), Component Definitions (along with their schemas), and Scope Definitions (along with their schemas). Meshery Server will store/register them in memory. This information is then used by Meshery Server to figure out which meshery adapter is capable of handling which Component / Capability.

Component Definitions that will be registered **must** have the following attributes defined in the spec field:

1. if the resource is a native cloud native infrastructure resource:

```
{
 "definitionRef": {
   "name": "---.meshery.layer5.io"
 },
 "metadata": {
   "@type": "pattern.meshery.io/mesh/workload",
   "meshVersion": "v1.0.0",
   "meshName": "<-- cloud native infrastructure NAME DEFINED IN SMP -->",
   "k8sAPIVersion": "<-- API VERSION OF THE RESOURCE - EXAMPLE:
“istio.xyz/v1alpha1” -->",
   "k8sKind": "<-- KIND OF THE RESOURCE - EXAMPLE: “EnvoyFilter” -->"
 }
}
```

2. If the resource is a native K8s resource

```
{
 "definitionRef": {
   "name": "---.meshery.layer5.io"
 },
 "metadata": {
   "@type": "pattern.meshery.io/k8s",
   "version": "<-- KUBERNETES VERSION -->",
   "k8sAPIVersion": "<-- API VERSION OF THE RESOURCE -->",
   "k8sKind": "<-- KIND OF THE RESOURCE -->"
 }
}
```

3. If the resource is a native pattern resource

```
{
 "definitionRef": {
```

```
    "name": "---.meshery.layer5.io"
  },
  "metadata": {
    "@type": "pattern.meshery.io/core",
    "version": "<-- ???? VERSION -->",
  }
}
```

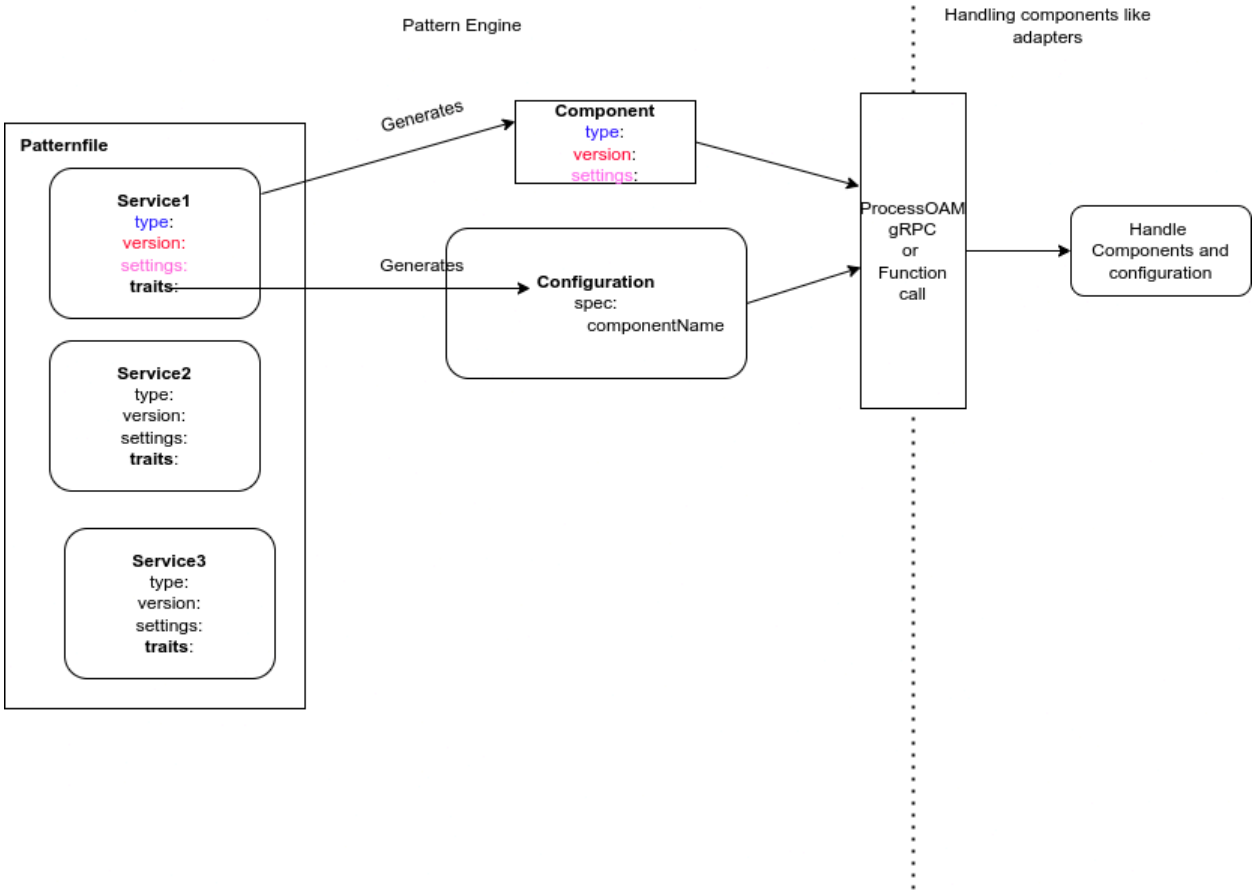
“Core” Components - - Non-Custom Resources

Some core necessary for lifecycle management, but are not represented by custom resource definitions, these operations are statically defined within each cloud native infrastructure and are cloud native infrastructure specific. Common examples of the type of operations that fall into this category are those of cloud native infrastructure provisioning, deprovisioning, for the control plane, data plane and add-ons.

Properties

All components are versioned. “**Core**” components are those that are Meshery Server/Meshery Adapter/Meshery Component-defined capabilities.

The type below is the same as from metadata.name from definition json. The version comes from spec.metadata.version as seen above.



Capability Registry (Internal)

Prelude
Purpose
Scope
Guiding Principles

1. Registrants are the foremost authority of their components.

All the above definitions and schemas are stored inside the capability registry at the time of component registration. This registry is an in-memory key- value store which acts as a single source of truth at various places like the Provisioning stage of pattern engine or while K8s->Pattern conversion. The methods on this struct allow getting registered workloads against a k8s yaml or a pattern's service. Once we have the appropriate workload, this struct also carries information of the

host i.e who registered this capability. Therefore the component-configuration pair generated is sent to that host
Key: /meshery/registry/definition/<apiVersion>/<kind>/<name>

Subkey: md5Hash of the entire schema, such that any changes in the schema produces another subkey for the same key.
Value: The json form of an instance of this struct

Exposing Capability Registry

Capability registry is an internal in-memory store which requires some considerations/caveats before directly exposing it via a CRUD API.

1. Create- Registering components is CREATE operation. Creating a component with name R in the registry sends an api request to `/api/oam/workload/R`. This endpoint already exists and is used by adapters at startup. Along with the definition and schema, a host field is passed while creation which specifies the address of the handling component. For adapters, this is a remote URI to which RPCs will be initiated. For Meshery/K8s components, this is set to <none/local>, and a normal function call is made internal to Meshery when these components are deployed instead of an RPC.
    a. Exposing registration outside of adapters: Currently Meshery either self registers(or on behalf of k8s), or adapters register the components. Should this registration allow users to pass on the host? What should be the host in that case? Host override?
    b. There are two purposes of creating Resource in the capability registry-
        i. To allow people to deploy designs by validating and delegating to adapters.
        ii. To allow people to design even if they do not have the adapters running.
    c. There should
2. Read- Lookup is also supported currently for any resource R at `/api/oam/workload/R`.
    a.

Action Items:

1. Replacing “OAM” with “MeshModel”. Rename Workloads and Traits.
    a. Incorporate the terms that are defined in MeshModel.
2. Propose a definition terms: registry, registrar, registrant, registered component?
3. API Endpoint breakdown (use spreadsheet).
    a. Move away from the massively large queries.
    b. Bring standard operations to each object / endpoint (`/<name of registrant>/<id>/components/<component-name>`)
        i. component version, pagination, string **filter**ing, filter on version, **count** in addition to CRUD.
4. Lifecycle of Capabilities Registry
    a.
5. Lifecycle of components.

    a. Do components expire? When do components eject / archive? Mark as invalid? Mark as not able to satisfy.
    b. Shipping Meshery with pre-generated components and pre-populated registry.
        i. Not just for Kubernetes but all the CRDs (in the world). Crd.dev
        ii. 1st Kubernetes (and maybe many versions)
        iii. 2nd Adapter components
    c. Guaranteed deployment? No, but we don't guarantee it today.
    d. Validation needs to change anyway.
    e. Caching considerations?
    f. *Future:* Move to a model where caching in-memory database in Meshery Server backed by postgresql database.
6. Registry to only support components?
    a. Example: `meshery-perf`
7. Metadata of every registered item.
    a. Capture the Kubernetes Distribution and Version
    b. UX: Environment → contains 3 kubernetes clusters → components generated
        i. (hide / show components in Designer with respect to the currently selected K8s context(s) (Environment)
8. Administrative controls over capabilities registry.
    a. # of registered 1,000 components for design

Capabilities

Types

Approval policy
OPA policy
Infrastructure components

—

Standard Set of Actions on a Workflow Operation: *Retry, Rollback, Ignore, Cancel*

Separate considerations

1. OAM Scope to Permissions
2. Define Content Registry as synonymous with Meshery Catalog.
3. Dependencies
    a. There will be UX controls over greenfield and brownfield dependencies. (e.g. "Istio Mesh" component and automatic vs. not-automatic depends_on).

Pattern Example

Duration-based Canaries

* required by Pattern

** required contingent upon specific Meshery Adapter

```
### OAM ApplicationConfiguration
service-mesh:                          # (required*) one or more cloud native infrastructurees
  istio-acme:                          # (required*) name of the instance mesh
    type: IstioMesh                    # (required*) type of cloud native infrastructure
    namespace: istio-system            # (required*) control plane namespace required
    settings:                          # (optional) control and data plane settings
      version: 1.8.2                   # (required**) cloud native infrastructure build version
      profile: demo                    # (optional) reference to cloud native infrastructure
configuration
    traits:                            # (required*) one or more configuration
      mTLS:                            # (optional)
        policy: mutual                 # (optional)
        namespaces:                    # (optional)
          - istio-test                 # (optional)
      automaticSidecarInjection:       # (optional)
        namespaces:                    # (optional)
          - default                    # (optional)
          - istio-test                 # (optional)

### cloud native infrastructure Behavior (function)
### OAM WorkloadDefinition
  behavior:                            # (required)
    type: Rollout### Pattern Metadata
name: Istio                            # friendly name of pattern (user-prescribed)
identifier: SMP105                     # unique identifier for pattern
version: 1                             #

### cloud native infrastructure Configuration
### Pattern Component
### OAM ApplicationComponent

    namespace: istio-test
    settings:
      replicas: 5
      containers:
        - name: svc-demo
          image: utkarsh23/meshy:v5
          ports:
          - name: http
            containerPort: 8080
            protocol: TCP
          resources:
            requests:
```

```
          memory: 32Mi
          cpu: 5m
    svcPorts:
    - 8080:8080
  traits: -- need another color here? Utkarsh?
    strategy:
      canary:
        - setWeight: 20
        - pause: {duration: 60}
        - setWeight: 40
        - pause: {duration: 10}
        - setWeight: 60
        - pause: {duration: 10}
        - setWeight: 80
        - pause: {duration: 10}
```

Brainstorm:
SMP Trait: Adaptive Rate Limiting
slime/slime-io

Metrics
Traces
Logs

```
  Prometheus:
    type: PrometheusIstioAddon
    namespace: istio-system
    dependsOn:
      - istio
      - svc
```

```
# PerformanceBudget defines the normal operating conditions for the application
# it could also define a load test or performance analysis test.
#
# In the instance of normal operating conditions the performance budget would
# configure something like AlertMonitor or PagerDuty integrations, it could
# also define the success or fail for a canary deployment
#
# In the instance of a performance test, the performanceBudget would define the success
# or fail for a test
performanceBudget:
  metrics:
    # Metrics are defined a follows
```

```
    # `request_duration_p50` is not the absolute name of the metric in the storage db
    # but an abstraction which relates to the duration for an individual request
    # For example: Istio might use the metric istio_request_duration_milliseconds
    # Consul might use envoy_cluster_upstream_rq_time_bucket
    # Inside the brackets allows filtering of a bucket, again the "name" is an abstraction
    # this could be pod_name or service_name or something different dependent on implementation
    #
    # Different metric types will have different properties, for example request metrics would
    # have properties related to the request, e.g. status_code. The translator of the OAM spec
    # would be responsible for translating this SMP metric into the correct query depending
    # on the metric provider
  - request_duration_p50["name=payments_v1"]:
      value: < 100ms
  - request_duration_p50["name=payments_v2"]:
      value: < 100ms
  - request_count["name=payments_v1"]:
      value: 50%
      tolerance: 0.1%
  - request_count["name=payments_v2"]:
      value: 50%
```

Design Pattern Lifecycle

Design Goals

1. Facilitate pattern lifecycle by recognizing that patterns are a living document, published from a central source that may be changed by users.
    a. Maintain integrity from a central source.
    b. Version from central source.
    c. Allowing mutation by consumers via copy of patterns.
2. Provide convenient mechanisms for importing, modifying, exporting, versioning.

Design Objectives

The designs in this specification should result in enabling:

1. Referential Integrity
    a. Pattern file specification should include a field for unique, identifying moniker.
2. Import
    a. Users should be able to easily import their patterns from:
        i. any public http/s endpoint as a **single file reference**.

ii. any public GitHub.com repository as either a single file reference or as a **recursive directory search** for valid pattern files.
iii. ~~? Github.com repository for all pattern files.~~
iv. ~~? generic, remote git repository?~~

b. Facilitate for Remote Providers extending the pattern import API in order to support additional use cases (e.g. import from private repository).

Out of Scope

1. Version Control - out of scope for Local Provider.

Meshery Users See:
MeshModel (Components)
- OpenAPI spec requirements
- ***ref*** *(dependsOn)*
- *environment (k8s contexts, credentials)*
- ***scope*** *(versions)*

- Applications (helm, manifests) →
- ~~Traits:~~ Patterns (behaviors)
Profile
- Type: performance
MeshMaps:
(x,y, layout,
Can-be-NodegroupOfNodeGroup?
isNodeGroup?
isNode?
colors, icon,)
Workloads:
Policy:
type: security-opa
type: security-kyverno
type: security-falco
category:

Policies:
Name: asdf
Category: security
Type: opa

Category: slack
Type: notification

Filters:
- Type: wasm
- Type: ebpf

Integrations (e.g. Slack - credentials)

Meshery has three tiers of support for cloud native infrastructure that it manages:

Least - **Generic** (any discovered CRD, not currently registered), which supports Day 2 configuration operations.
To - **Meshery Core**, which supports MeshSync fingerprint, infrastructure-specific color/icon, and Day 2 configuration operations.
Most - **Adapter-based tier**, which supports MeshSync fingerprint, lifecycle and configuration, operations (greenfield provisioning of infrastructure), and complex operations.

MeshModel Component Generator

1. K8s core resource vs all other custom resources
2. Component's Managed Version #
3. Client

Design Pattern Lifecycle User Stories

Story 1: Import Design from a remote location

As a Meshery user, I would like to import pattern from a remote location, so that I can *apply* it and keep its track (only if it was imported from a remote git repository)

Implementation:

1. API client will send a **POST** request to meshery server on the endpoint */api/experimental/pattern*
2. The request body may look like following:
    a.
```
{
  "url": "https://github.com/owner/repository/branch",
  “path”: “dir1/dir2/**”, // or it could be “dir1/file.yml”,
  “save”: false
}
```
    b.
```
{
  "url": "https://example.com/file.yml",
  “save”: true
}
```
    c.
```
{
  "url": "https://gitlab.com/owner/repository",
  “path”: “dir1/dir2”, // or it could be “dir1/file.yml”,
  “save”: false
}
```
3. If the request body is of type:
    a. 2.a and the user is logged in with:
        i. None provider: meshery server will use github API to try to traverse the repository and return all of the pattern files. For identifying pattern files, the meshery server will look at all the files with extension ".yaml" or ".yml" and will try to look for certain parameters in that file. These parameters are described later. Meshery server will **not** be able to access private repositories.
        ii. Remote provider: meshery server will look for capability called "import-remote-meshery-pattern" in the capabilities.json. If it doesn't then the API will return an error. If it does, then the meshery server will forward the ***same*** request to the remote provider.

b. 2.b and the user is logged in with:
    i. None provider: meshery server will send a **GET** request to the given endpoint and expect a response of type "Content-Type: plain/text".
    ii. Remote Provider: meshery server will look for capability called "import-remote-meshery-pattern" in the capabilities.json. If it doesn't then the API will return an error. If it does, then the meshery server will forward the ***same*** request to the remote provider.
c. 2.c and the user is logged in with:
    i. None Provider: Does not support any other git remote repository.
    ii. Remote Provider: meshery server will look for capability called "import-remote-meshery-pattern" in the capabilities.json. If it doesn't then the API will return an error. If it does, then the meshery server will forward the ***same*** request to the remote provider. It will then be up to the remote provider to verify if it supports the given git remote repository or not and respond back appropriately.

Story 2: Import pattern from a Cytoscape JSON

As a meshery developer, I would like to convert my cytoscape JSON into a pattern file , so that it can be saved or can be applied from the UI.

Implementation:

1. API client will send a **POST** request to meshery server on the endpoint */api/experimental/pattern*
2. Request body must look like:
   `{ "cytoscape_json": "// cytoscapejson blob" }`
   `The cytoscapeJSON must have position defined as well as have a field called "scratch" defined in it must have a field called "data" which should be of type Service.`
3. Meshery server will convert the cytoscape into a pattern file and will **only return the file (i.e it will not save the pattern).**

https://react-jsonschema-form.readthedocs.io/en/latest/

Story 3: Export pattern to Cytoscape JSON

As a meshery developer, I would like to convert my pattern into Cytoscape JSON , so that it can be shown in the meshmap visualizer.

Implementation:

4. API client will send a **POST** request to meshery server on the endpoint */api/experimental/pattern?output=cytoscape*
5. Request body must look like:
   `{ "pattern_data": "// pattern_data along with metadata" }`
6. Meshery server will convert the pattern into cytoscape json and will **only return the file (i.e it will not save the pattern).**

Questions

What's the behavior of concurrency across multiple different types of cloud native infrastructurees?

Can you run the same pattern on the same application at the same time?

- How do you manage cross Pattern dependencies?

Can you rollback a failed pattern?
=> Rollbacks aren't yet implemented partially because we don't store the previous state of the individual components.

- Can we add support for rollbacks?
  => Yes, we can. Each workload is allowed to mention the number of its previous states that can be or should be stored by the runtime (Meshery). This can be leveraged for rolling back the changes (DAG should be able to support this too).

List of each code file and why the file exists:

- patterns or meshflow branch

How will OAM Scopes be used in Patterns?
=> OAM ScopeDefinitions are yet to be implemented. But, we can have another field in the pattern something like "scopes" which could be an array of registered scopes.

Do all patterns fit into this?
=> As this Patterfile is capable of describing all the types of applications (except multi-cluster), it should be able to describe all of them. However, a few of the patterns may amalgamate into one Pattern component, for example a rollout may have pre-defined retries, circuit breaking, etc.

Examples of Patterns using OAM Components?

Do each of the patterns require their own MeshFlow?
No, they don't. Each pattern's requirements is fulfilled by a set of Meshery Adapters. So a pattern may trigger an action on all of the Adapters, some may not invoke only a few. Meshflow will be invoked only if the "type" is *Rollout.*

How to control parallel execution of MeshFlow?
Meshery server creates a DAG and controls the parallel execution of the commands on the Adapters. It will ensure that all of the subsequent workflows also fail if one the dependency fails. However, this behavior of "chain" failure is controlled by the execution function (the function that is invoked by the Meshery Server on each DAG node/vertex).

Meshery server creates the diag.

Current discrepancies b/w Designs and Patterns

Guiding Principle: Hydration / Dehydration - - Translation - - -

Below are listed the feature differences/incompatibilities along with how to resolve them. If there is a good enough reason to drop any of this feature then please list them out as well.

- **Depends-on:** Each service inside of patternfile can have a depends-on field to imply it's dependency on some other service. Designs currently do not support this feature
    - **Merits:** The sophisticated DAG at the core of pattern engine is designed to support this feature so that people can have all their applications and infra related things in one single pattern/design. It adds huge value in terms of saving time. In a huge number of cases, there is a set sequence of installation and **depends-on** helps with that.
    - How to implement it in design? : (Suggestions) (UX)
        -
        -

- **Referencing:** Patternfiles allow referencing other fields from any given field across services within and between patternfiles.
    - **Merits**: It increases the re-usability of a pattern/design as variable-izing the values at too few places reduces the area of change in case a single field has to be replicated a number of times. People cannot do this with simple k8s manifest.
    - How to implement it in design? : (Suggestions) (UX)
        - Each service needs to be context aware of all the fields that are already filled by the user. With some hotkey(or some other action), they can be given the option to either manually fill a field in RJSF form as they do now OR reference an already existing field that they have filled in this service or any other service.
        - Vars field should be saved along with elements in cytoscape.

- **Pattern/Design imports**: If you have an existing pattern that you want to reuse, then patternfile offers a feature to import the pattern by URL in a single service on a new pattern. This way, you can have patterns inside of patterns, recursively.
    - **Merits:** As the design gets bigger and bigger, it can be broken down into re-usable smaller designs which fulfill a unique purpose. For instance, one design whose sole job is to install SPIRE for the upcoming Istio. Another design which installs Istio and another which makes changes to the workload API. All these can be then imported into one design as three services and along with the **depends-on** feature, we can use this one simple design to do everything. So composing designs is a great feature to

have. People can create their own custom designs and externalize a few variables for other people to configure and use.

Implementing design composition via pattern import

The above mentioned import feature is used by designs to generate higher order components on the fly by following mechanism:

1. Each patternfile that is "imported" into a service has a `vars` field which specifies the configurable variables in that pattern.
2. We add an optional field on services: `schema`
3. The `schema` field can be used by any specific service to bypass the static check for type and can directly provide its own schema. This can be a use case for handlers of a service type which compute the schema based on some factors subject to change over time.
4. Using the dynamic schema approach: When a pattern is imported via URL, Meshery server fetches the patternfile and generates a schema based on variables declared by the imported patternfile inside the vars field. As the imported patternfile changes in future and adds new fields into `vars`, we would have dynamically generated schemas compatible with the new patternfile automatically.
5. That schema is then pushed to service.schema and sent back to the client.
6. The change in logic on client would be: Before defaulting to the behavior of fetching schema based on type, client first checks if the service provides a schema explicitly and if yes then it uses that schema to generate the RJSF form.

Component Generation Factory

Abstract

We want the ability to add support for various other systems into Meshery. Meshery Components are the way to do that. So, we achieve this by *Generating* Meshery Components from various sources.

Architecture

Although right now the only source we will be using is CRDs, this will change in the future, so the architecture should take that into account.

Our sources can be anything. It can be ArtifactHub packages, DockerHub etc. Depending on the source we use, the data type varies along with the logic.

Package
Any entity that is used to expose a particular systems capabilities in Meshery is a `Package`.
A Package should have all the information that we need to *generate the components*.

Package Manager (Meshery Server)
Supports pulling packages from Artifact Hub and other sources like Docker Hub.
Should envelope Meshery Application importer.

```
mesheryctl repo add meshery/meshery
```

Tracks a list of packages (e.g. helm chart names) to import or not import from the given Package Importer.

Package Manager will be responsible for tracking and managing the list of *Packages.*

So, A `PackageManager` knows how to:

1. Get the packages

The PackageManager interface will look something like this:

```
type PackageManager interface {
        GetPackages() (Package, error)
}
```

Package Importers (Meshery Server)
These importers are mechanical in nature; machines to configured to interact with the repo-specific APIs

E.g. Artifact Hub, Docker Hub

Component Generator (**Meshery Server) using MeshKit's functions**
Meshery Server is scheduled to run on schedule, generate components, persist components in **Meshery Cloud**.

As a user of MeshMap, I would like to mark some categories and components as displayed or not.
Prioritize component generation for "Official" repos first.

Components are generated:

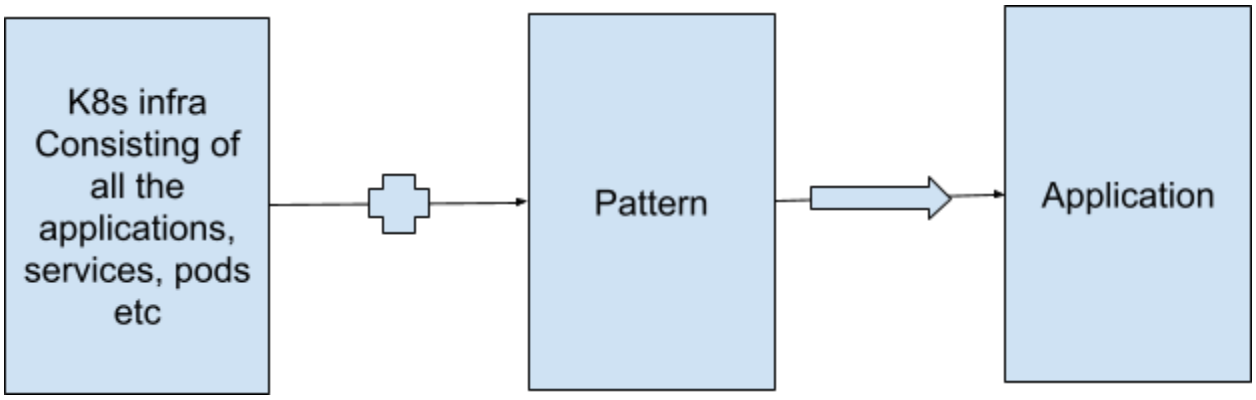
1) Pre-generated components
2) On-connection to Kubernetes context
3) On-demand by user, who imports a Meshery Application as a single file reference, folder reference (to be recursively searched), or a git repository to be walked.

Draft: Patterns, Designs, Applications,

MeshMap users need the ability to drag and drop their imported Kubernetes manifests onto the MeshMap canvas.

What is a Meshery Application?

A single file (future: which could be split into multiple files upon user request) that comprises Kubernetes objects representative of a complete set of Kubernetes workload resources.



1. ***How an application should like on yaml??***
2. ***What Should be an application deploy behaviour?***

Comments:
From the last discussion on team call, I inferred the above diagram and thus writing down here.
Q. What should be a raw infrastructure be called? (the first block in the above diag)
Q. What should that contain exactly? Is any kubernetes yaml valid under this? (1st block)
Q. How an application yaml should look like ?

Currently the yaml file (the kubernetes yaml file) that consists of the list of services/pods/deploymentsets/etc,etc are not able to be deployed. If the above things are true, we have to create an engine that takes a regular yaml and be able to parse it.

What is a Meshery Rollout?

"v1" Application = Services + Deployments + ReplicaSets
"v2" Application = Services + Deployments + ReplicaSets (Rollout Strategy)

Implementation:
Acceptance Tests:

1. *Transposed manifests will show as Meshery Application, persisted in MeshMap/pattern format.*

Caveat:

1. *Rollouts do not support custom ServiceAccount names. User's ServiceAccount name must be the same as the Rollouts name.*
2. *Rollouts: DNSPolicy*

Appendix A: OAM Examples

Example of a workload definition:

```
{
    "apiVersion": "core.oam.dev/v1alpha1",
    "kind": "WorkloadDefinition",
    "metadata": {
        "name": "Rollout"
    },
    "spec": {
        "definitionRef": {
            "name": "rollout.meshery.layer5.io"
        }
    }
}
```

Example of a trait definition:

```
{
    "apiVersion": "core.oam.dev/v1alpha1",
    "kind": "TraitDefinition",
    "metadata": {
        "name": "strategy"
    },
    "spec": {
        "appliesToWorkloads": ["Rollout"],
        "definitionRef": {
            "name": "strategy.meshery.layer5.io"
        },
        "revisionEnabled": true
    }
}
```

Structure of API

Need-
Ability to have more fine grained queries to backend for oam workloads-

- Name of the resource like VirtualService.Istio is the last string in the key with which the schema for this workload is stored.VirtualService.Istio is given as a type in the pattern file. We need a pattern file that does not make any assumptions about what component(adapter) handles any given workload. Any component that has registered capability for that workload is a potential candidate. So instead of knowing who registered anything, much more useful information is- WHO IS GOING TO HANDLE THIS? In the current scenario, both registrant and handler are exactly the same, so we do not require any additional field there.
- Question- If there are 10 running adapters, should the user be able to provide which adapter needs to handle any given pattern file?
    - Answer- NEVER! Pattern files are reusable declarations. They should not carry with them any data that is local to the current state of the cluster. If in any pattern file, we give the ability to specify any specific adapter local to that cluster -in the pattern file then that pattern file will most probably break when used somewhere else. The information about who is handling any given service type is and should be transparent to the user if we want to maintain reusability of pattern files. Users should only know about the meshery server. When any adapter fails to register it's capabilities and we are not able to handle the pattern then we do not get opinionated and tell them to install any specific adapter. Instead the error shows that this particular type is invalid. Users can have their own custom adapters or something registering and handling that given workload type.
    - To summarize, what we care about are handlers of the workload and not registrants of the workload. Still in all current use cases, they are the same so relying on handlers is much more sensible. Secondly, the handlers are and should be transparent to the user. We do not bind any specific workload_type to any specific handler. And this is what makes patterns powerful. If a user has 10 adapters(which will be rare) and all of them have registered the capability for the same workload type then from the available handlers we will delegate the request to the first one.
- Given all the above information, we still need a fine grained control over the API. The extensibility and usability of patterns have much to do with how they are stored in the database(like key-value pairs) but this capability does not make up for a friendly API that we expect. Although this way of storing is the best way for handling the patterns. Hence I think we should have the best of both worlds. We let the way we store components in the database the same, for the best of pattern engine(As pattern engine is a client of the database and changing the way we store components will require rethinking the pattern engine-THERE IS NO NEED FOR THAT). Instead we write a thin abstraction on time for it, for the best of UI and other api consumers.

- What we really want to know in a much fine grained manner is:
    - What are all the types available- like Istio, Traefik, Core, Kubernetes
    - For any given type, what are all the versions available.(Except for core-core components do not have version)

    - For any given version, what are all the workloads available

- So any given request might look like- /api/oam/workload/Istio/latest/VirtualService
- /api/oam/workload will return all workloads
- /api/oam/workload/data/type will return an array [Core,Istio,Kubernetes,Linkerd...]
- /api/oam/workload/data/count will return the count of the above array only
- /api/oam/workload/data/Istio will return all components handled by handler Istio -all versions
- /api/oam/workload/data/Istio/<version> will return all components specific to that version
- For each /data , we can have /count. Instead of entire data, /count will only return the count- this will make the API calls faster.

The above changes would leave the database for the best of pattern engine. The way we will do this is by using a middleware pattern and after a component is successfully stored, we will extract the relevant information that is needed to serve the above API calls back. When the above API calls are made, the request will be handled accordingly by the information that we stored earlier.

The extensibility of the pattern engine relies on the clean and simple way the data is stored in the database. Any compromise with that will be a step in the deprecation of the pattern engine's power.

Just like a pattern file handler is a client to the database(it looks into the database to find the handler of any given service type), the UI is also a client. Both have different needs. Giving priority to one over the other is being myopic. The concerns of UI and pattern engine are orthogonal. Pattern engine does not care about the "available versions","types",etc. The UI does because it is our UI, and we know that these constructs will exist. This concern is completely orthogonal and hence should be dealt with that manner.

The middleware-like(not a very good name) approach will not be inefficient as we are not going to serialize and deserialize jsons. We have the OAMDefinition struct with all its data available. That is all we need. This way one can design the API without worrying about how that will impact the pattern engine. The database is closely tied to the pattern engine.

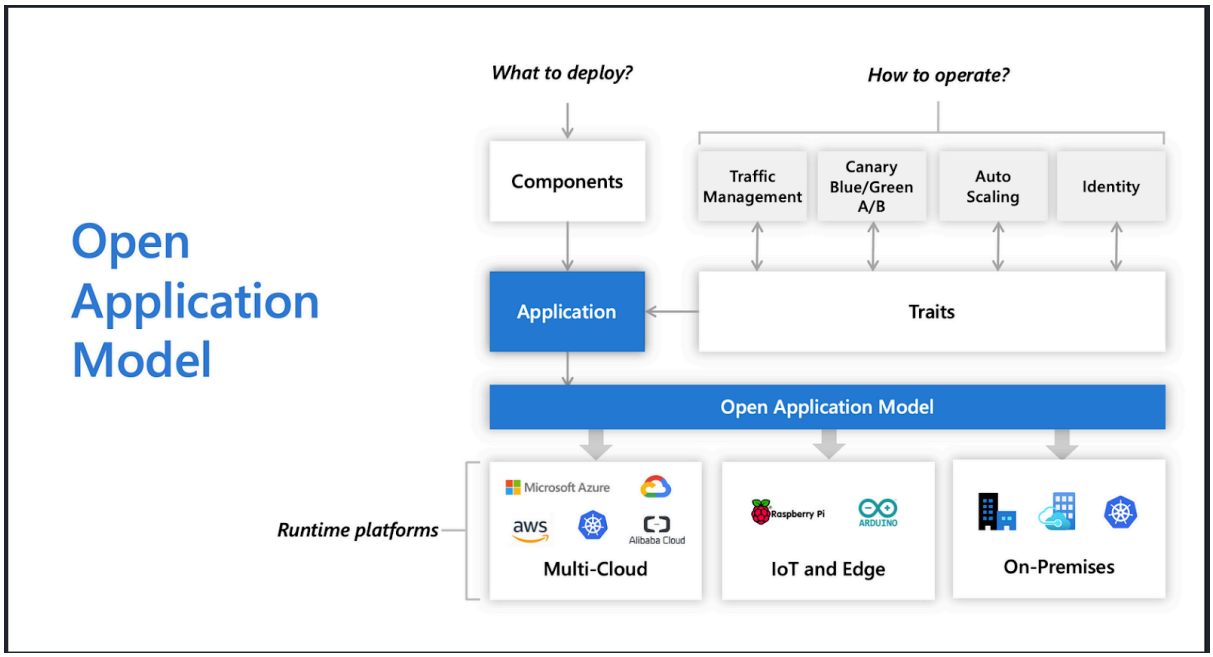
This is the cleanest way to do it. Requires no change in the pattern engine and database. Maintains the extensibility of the pattern engine. And fulfills UI specific needs. This will guarantee to not break anything. And in future can be easily modified to cater to UI needs without changing any part of the pattern engine. Most intuitive.

/api/components/types => Returns array ["istio","core","linkerd"...]
/api/components/{type}/versions => Returns array of available versions for that type
/api/components/{type}/{version} => Returns all components of given version for that type
/api/components/{type}/{version}/{componentName} => Returns all components of given name of given version

Meshery Pattern Model Inspiration

Open Application Model

Open Application Model is a Team-centric/App-centric model which focuses on separation of concerns between *platform engineers*, *application developers* and *infrastructure operators*.



In this figure:

- **Traits** and **Components** are created/registered/managed by the *operators.*
- **Applications** are used by the *application developers.*
- **Workloads** are registered/created/managed by the *platform engineers.*

For a more detailed understanding, look at the [oam-spec]

Crossplane

Crossplane is a project that has the same core idea as OAM. It essentially wants to achieve separation of concern between *platform engineers*, *application developers* and *infrastructure*

operators. Crossplane is a Kubernetes implementation of OAM even though their implementation details are not in strict adherence with the OAM spec.

Crossplane has some core constructs.

Managed Resources

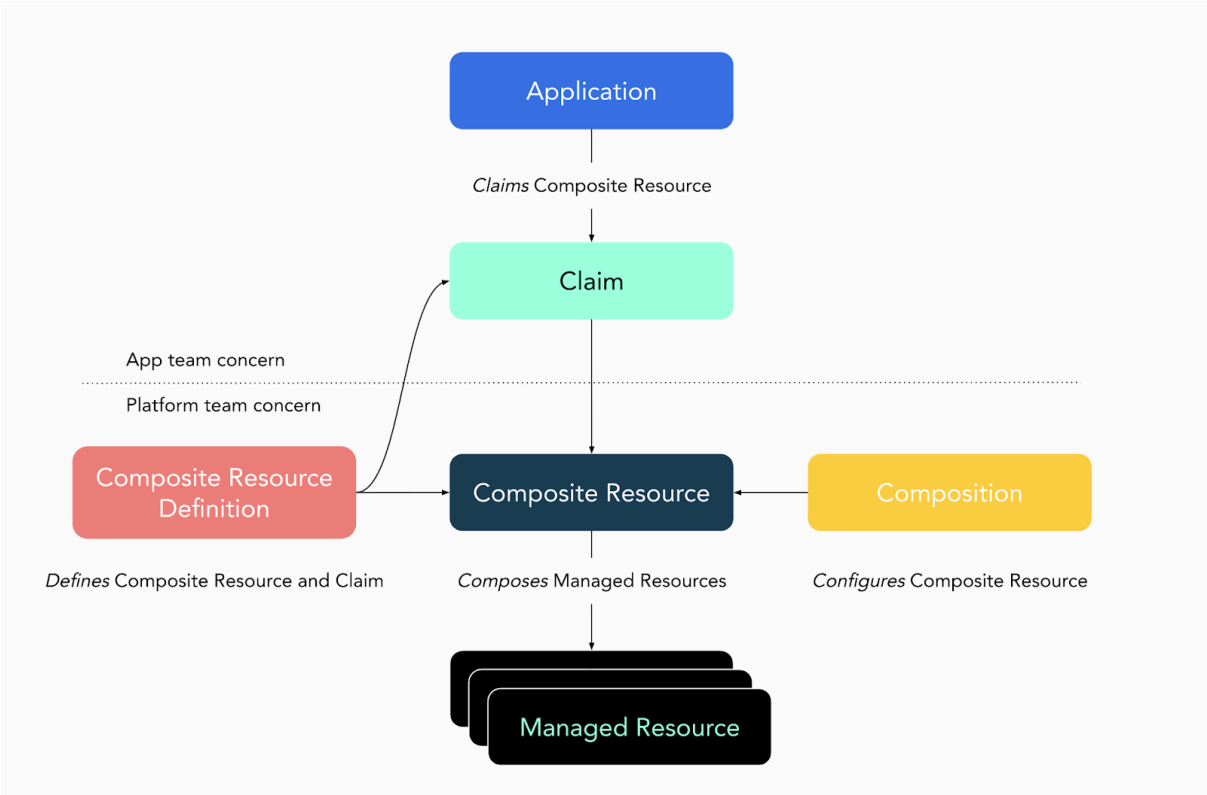
A Managed Resource (**MR**) is Crossplane's representation of a resource in an external system - most commonly a cloud provider. Each provider such as AWS, GCP, Azure etc. will provide their own Managed Resources.

Composite Resources

These are Kubernetes custom resources that are composed of Managed Resources. Composite Resources are a way of defining the desired state of the application(an application can be a workload, a service etc.) declaratively without worrying about the underlying platform specifics. Hence, Composite Resources separate the concerns of platform operators and application developers.

Composition

Composition is a Kubernetes Custom Resource that serves as a link between Composite Resources and Managed Resources. Compositions configure what Managed Resources should be created/modified/deleted when a Composite Resource is created/modified/deleted. This takes away the need to write your own Kubernetes custom controller. Rather, you just declaratively specify the relationship and Crossplane will take care of maintaining the desired state.



In this figure, you can see how concerns of the Platform teams and the Application Teams are separated. The Application team can make use of these Composite Resources via Claims which are offered by the platform team.

Adherence to OAM

We want to take inspiration from the idea that both OAM and Crossplane have, which in essence, is to decouple different teams in an organization (operators, developers, platform engineers, security engineers, product managers) and enable them to become more productive. It is also important to note that the boundaries are not necessarily rigid. Sometimes, an application developer will assume the role of an operator and vice versa and hence our model should be flexible. But, we need not strictly adhere to the implementation details given in the spec.

Models

Abstract

The goal is to have a universal model for Meshery that sets a foundation for the project to build and expand on. The model should be *fluid.* We should be able to easily shape the model according to our use cases. It should facilitate a consistent way of communicating between multiple components of Meshery.

Component

Resource: [Static list of components]

Components are a way of exposing the capabilities of the underlying platform. In case of application development, Components are entities that are created/registered by the platform engineers and used by the operators.

ComponentDefinition, as the name implies, is just the definition of a Component which will be stored in the Capabilities Registry.

In other words, Components are instances of ComponentDefinitions.

Components can be Filters, eBPF programs etc.

A Component on evaluation will result in a set of manifests(can be anything) which will be sent to the host specified.

Components should be *PORTABLE, ACCESSIBLE* and *EASILY MANAGEABLE.*
Any user should be able to easily create his own ComponentDefinition and publish it so that others can make use of it.

```
{
      "kind": "ComponentDefinition",
      "apiVersion": "core.meshery.io/v1alpha1",
      "metadata": {
         "name": "ScopeDefinition",
         "version": "1.6.0-alpha.2",
         “icon”: “https://iconurl.com”,
          “primary_color”: “red”,
           “secondary_color”: “red”,
           “source”:”kubevela.artifacthub.meshery”
```

```
    },
    "spec": "...",
    "id": "5b64f71a1823f38f7d2715f6de3fe165",
    "host": "<none-local>"
}
```


Design

A Design is a declarative generic representation of the desired state of the services that provide value to an organization/team.

- Components
  - Coordinates
- Patterns
- Metadata

Pattern

Rollout - - abstracting ArgoCD - instantiating its own components
Rollout - - reveals ArgoCD - configuring other components from other adapters

A Pattern is an entity that augments the operational behavior of a deployed instance of a Design. A Pattern can be applied to a Component or a Design. Patterns define a common (and best) practice of both configuring and operating cloud native application functionality. Patterns are read-only.

Components and Capabilities

Component will be consumed by more than one consumer (MeshMap and the design processing engine for instance). A ComponentDefinition will have information that may be of use of one engine and not to the other. For example, the Display related information will be of use to MeshMap but not to the design processing engine in Meshery Server.

Capabilities and **Components** are two different things. Whatever that the adapters are registering are all **Capabilities.** A capability can be a trait, workload, edge, policy, workflow etc. A **Component** is a type of capability that the adapters usually register.

Adapters flow:

Whenever an adapter starts, using some component generation methodology, it will register all the ComponentDefinitions. They will be registered in capabilities registry namespaced by hosts. For example, meshery-istio adapter will register

```
apiVersion: core.meshery.io/v1
kind: ComponentDefinition
metadata:
  name: VirtualService
  Version: 1.19
Spec:
  schematic:
          cue:
          template: |
          hosts: [for k in outputs {String("/(k)"): kubernetes}]
          outputs : vs :  {
            apiVersion: networking.istio.io/v1alpha3
            kind:           VirtualService
            spec: {
                    ........
                }
          }
          parameters: {
                    ......
          }
```

as VirtualService component. This is saying that, on instantiating this Component with the necessary parameters, you will get some manifest(a Kubernetes `VirtualService` Object in this case) which will be sent to `Kubernetes` host. The `Kubernetes` host should know how to deal with this manifest. There can be some components which will output some manifests that will be sent to meshery-istio adapter itself and it will know how to handle it.

Design Evaluation Engine

- Design Validation
- Pattern Evaluation
- Components Evaluation

Capabilities Registry

Component Generation factory is now capable of generating MeshModel Components. These Components have to be registered in the Capabilities Registry. Right now, the capabilities registry that we have are

Deployment Strategy

The fundamental constructs that are present now are Design, Components and Patterns. There is no concept of *deploying a Component* or *deploying a Pattern* as such. Only **Designs are Deployable.**

Designs are not just a way to express what needs to be deployed, because that is not providing any value to Meshery. We have a lot of different types of manifests, and engines that interpret them, for deploying a set of infrastructure components. Designs help us create abstractions such as Policies, Workflow etc. We will have a whole range of areas in the Cloud Native ecosystem to play with.

As a user, I have described what needs to be deployed, and some other things such as policies, through a Design. I expect Meshery to make the desired state of my infrastructure a reality.

Component Definitions will be registered in the capabilities registry. There will be a Design Evaluation Engine. It will know **how to interpret the Designs**. A Design may be comprised of many fundamental entities such as Components, Patterns, Policies etc. The Engine should have access to the capabilities registry since all these definitions will be stored there.

The implementation of the Engine will be using Temporal.

Now, if we take an example design,

```
`apiVersion: core.meshery.io/v1alpha1
kind: Design
metadata:
        description: A sample design for testing
        name: sampledesign
spec:
  components:
  - name: httpbin
        type: server
        settings:
          port: 80
```

```
        replicas: 3
        image: docker.io/kennethreitz/httpbin
patterns:
      - name: simplecb
        type: circuitbreaker
        settings:
          applyTo: httpbin
          interval: 3s
          consecutive5xxErrors: 2`
```

Meshery will just be delegating the deployment needs to an underlying orchestration system such as Kubernetes.

We will have something called a `Components Controller`. Similarly `Patterns Controller` and `Policies Controller` will also be there.

The order of evaluation of the constructs in the runtime will be:

1. Components
2. Patterns

Components Controller

In this given example, when the component `httpbin` is given as the input to the Components Controller, it will

1. Get the Component Definition for `server`
2. Instantiate the component - Create a manifest that can be sent to the host which knows how to deploy that
3. Delegate the manifest to the host

Step 2(Instantiation of the component) will include Cue logic to extract information to generate the required output manifests. If JSONSCHEMA is used, then we wil just generate a manifest using the schema and the given parameters.
Every component definition will have a `host` field which will have the information as to where that component has to be delegated to.