Harnessing The Power of Crossplane and Dapr

Proof of Concept:

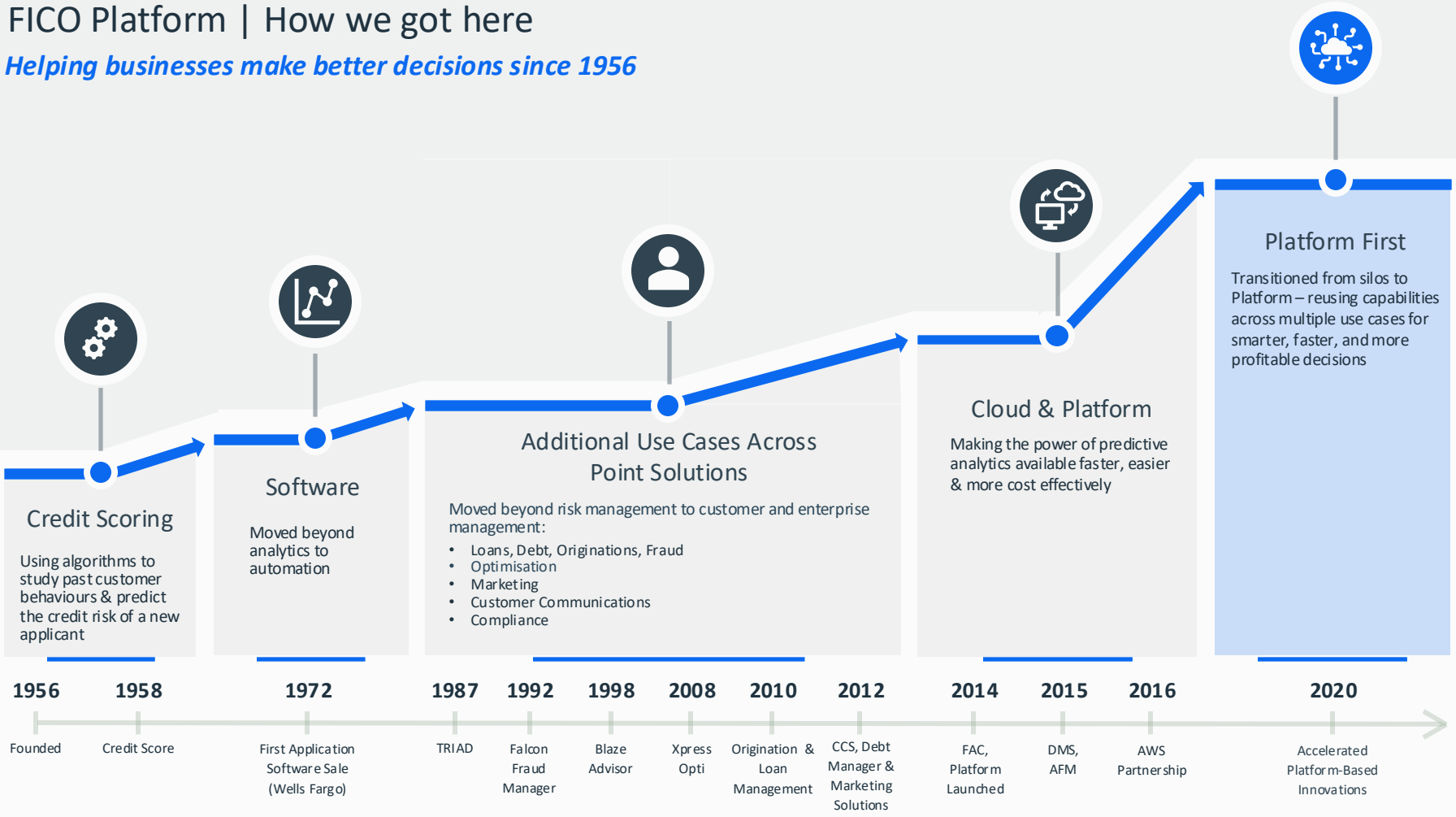
A Kyverno Policy Validator for Crossplane Compositions using Crossplane Functions and Dapr

Hugo Smitter
Platform Architect
hugosmitter@fico.com
(+1) 214 534 5373

Who is FICO and what do we do?

FICO Platform | How we got here

Helping businesses make better decisions since 1956



Agenda

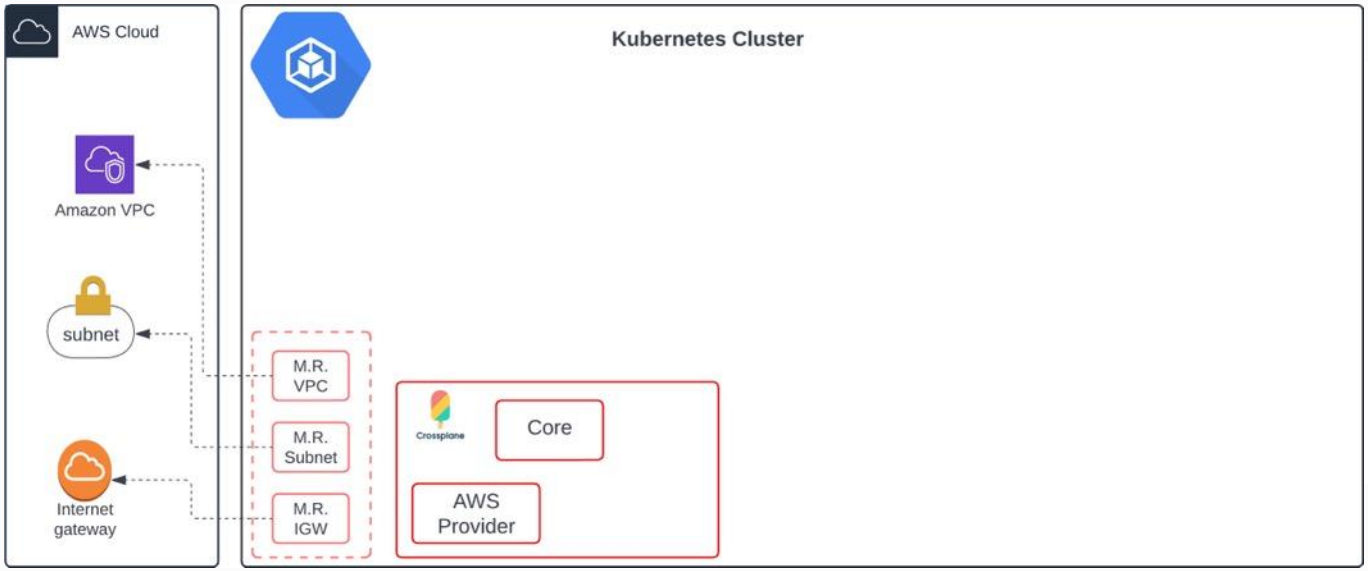
- Crossplane overview
- Opportunities and challenges when using Crossplane (Composition) Functions
- Dapr overview: Build more complex but maintainable Crossplane Functions
- How to integrate Dapr and Crossplane
- Proof-of-Concept - A Kyverno Policy Validator for Crossplane Compositions:
  - Problem we're trying to solve
  - Challenges and solutions
  - Architecture diagrams
- Demo
- Next steps
- Q&A
- Appendix

Quick Crossplane Overview

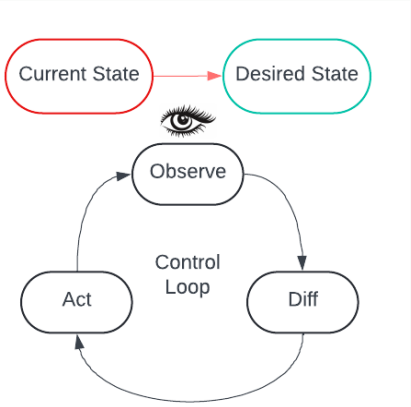
Crossplane Overview

Managed Resources (M.R.) are representations of external resources in Crossplane

Providers enable Crossplane to provision things (e.g.: infrastructure on an external service). They create new K8s APIs and map them to external APIs (*).



Kubernetes Control Loop



(*) https://marketplace.upbound.io/providers

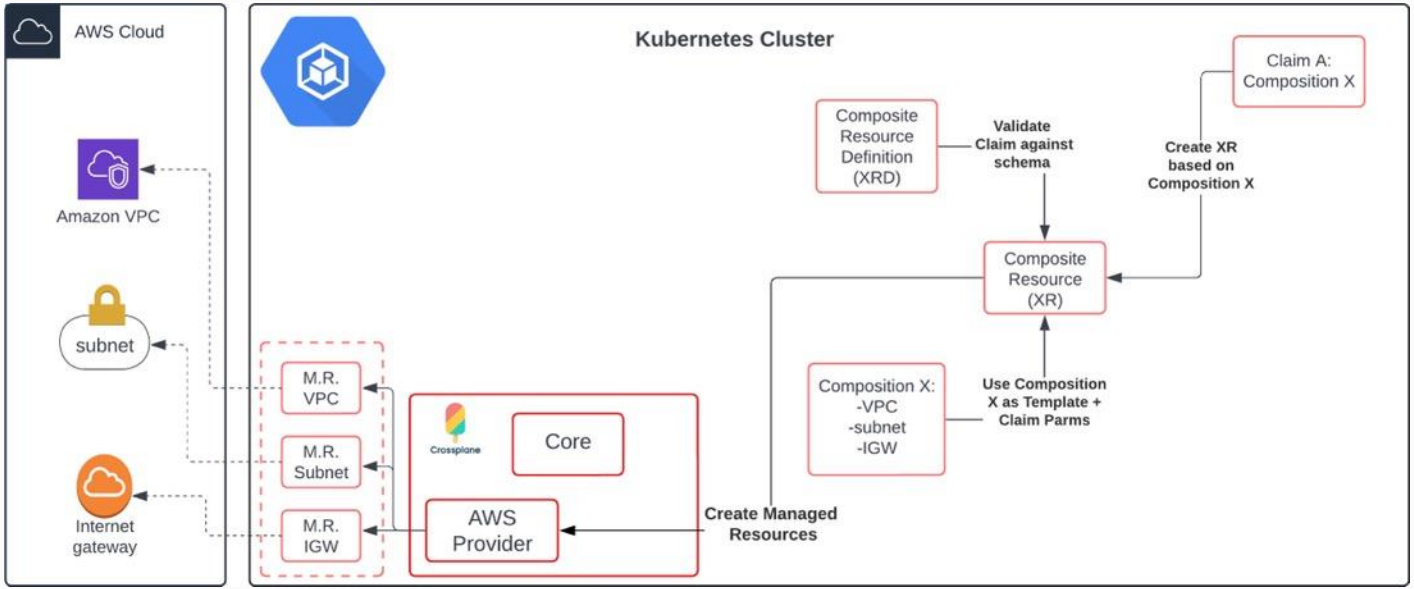
Crossplane Overview (cont.)

Claims represent a set of managed resources as a single K8s object *inside a namespace*

Composite Resource Definitions (XRDs) define the schema for a custom API (claim)

Compositions are templates for creating multiple Managed Resources (MR)

Composite Resources (XRs) represent a set of MRs as a single K8s object



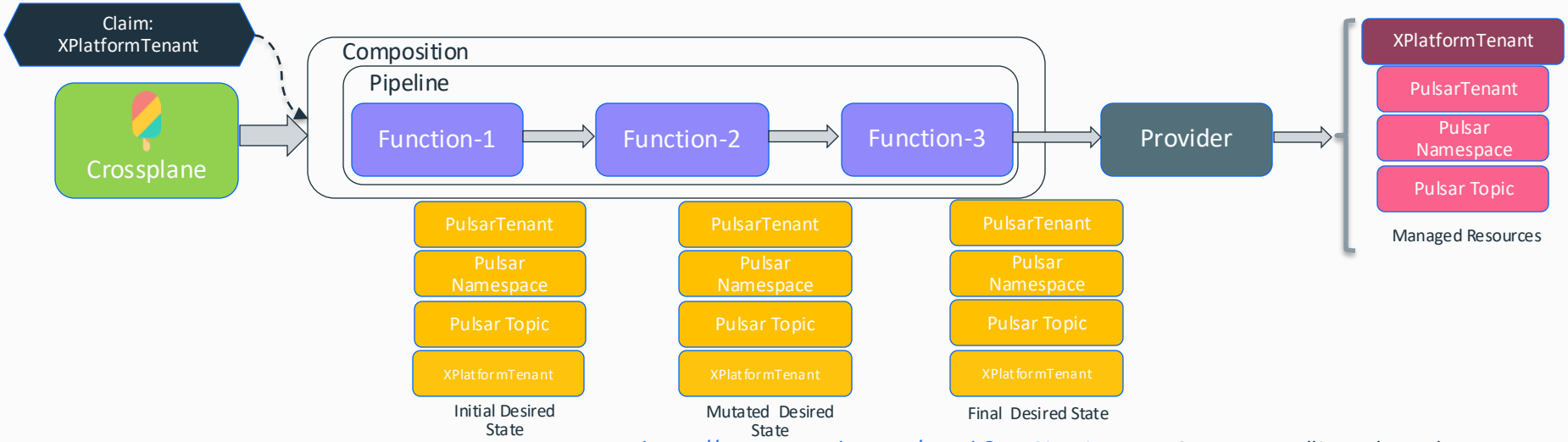
See - https://kubecharm.com/blog/what-is-crossplane

Crossplane Overview (cont.)

https://marketplace.upbound.io/functions?query=functions

Composition Functions are custom programs that template Crossplane resources.

- Functions dynamically determine what resources should be created when you create composite resource (XR).
- You can write Functions in Go or Python (more languages to come).
- You can write advanced logic to template resources, like loops and conditionals. You can also access data or call other programs.
- Compositions can have pipelines of functions assembled in steps, where the output of a function is the input to the next function.

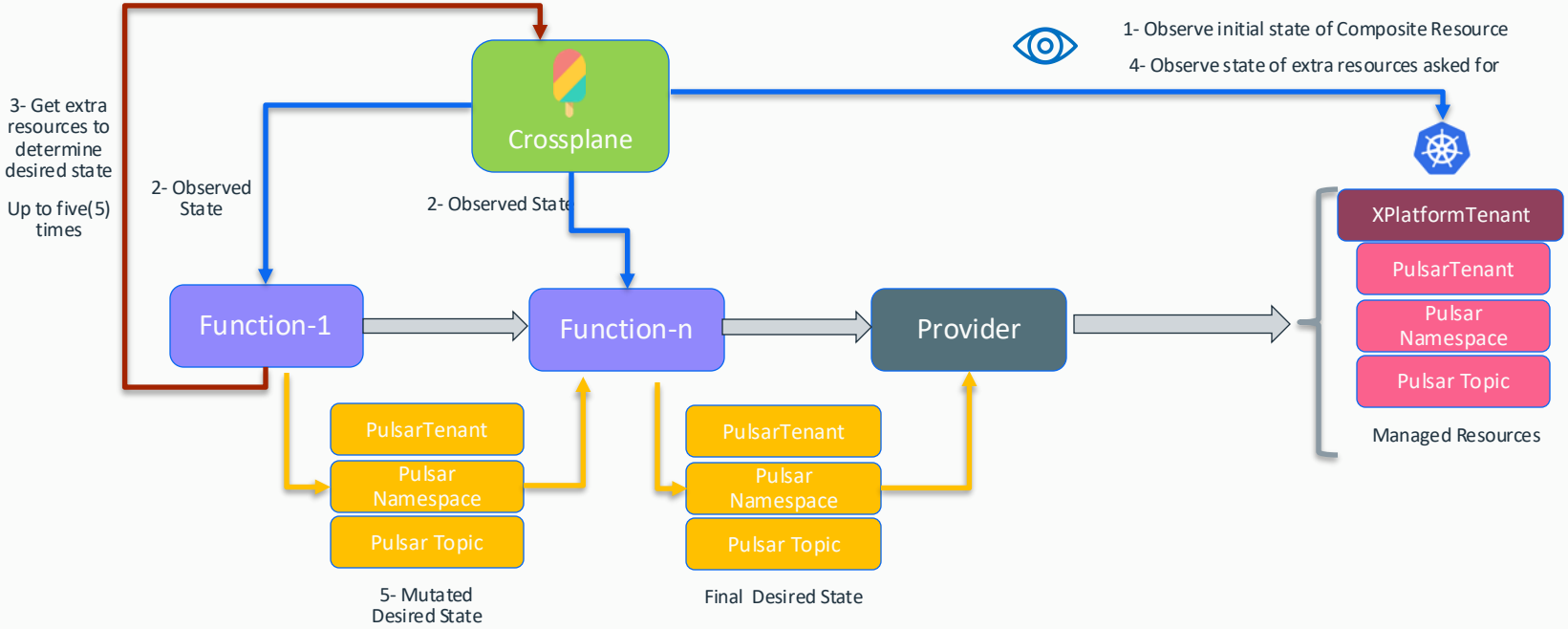


https://www.youtube.com/watch?v=n0j5WiavXEI = Steven Borrelli – Upbound Inc.

Crossplane Overview (cont.)

Empirical observations on how compositions and functions work:

- They work in loops! Their behavior is more complex than a sequence of steps in a pipeline (*)
  - Composed/Managed Resource **provisioning loop**: Five (5) iterations until stability is reached or error
  - **Reconciliation loop** once resources are created (Default: every minute)

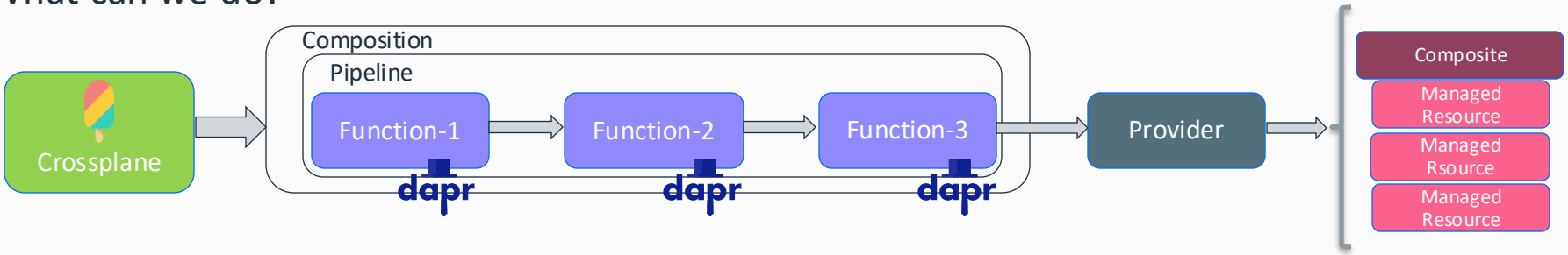


(*) https://docs.crossplane.io/latest/concepts/compositions/#how-composition-functions-work

Some Challenges Provisioning Anything-As-Code Even With Functions

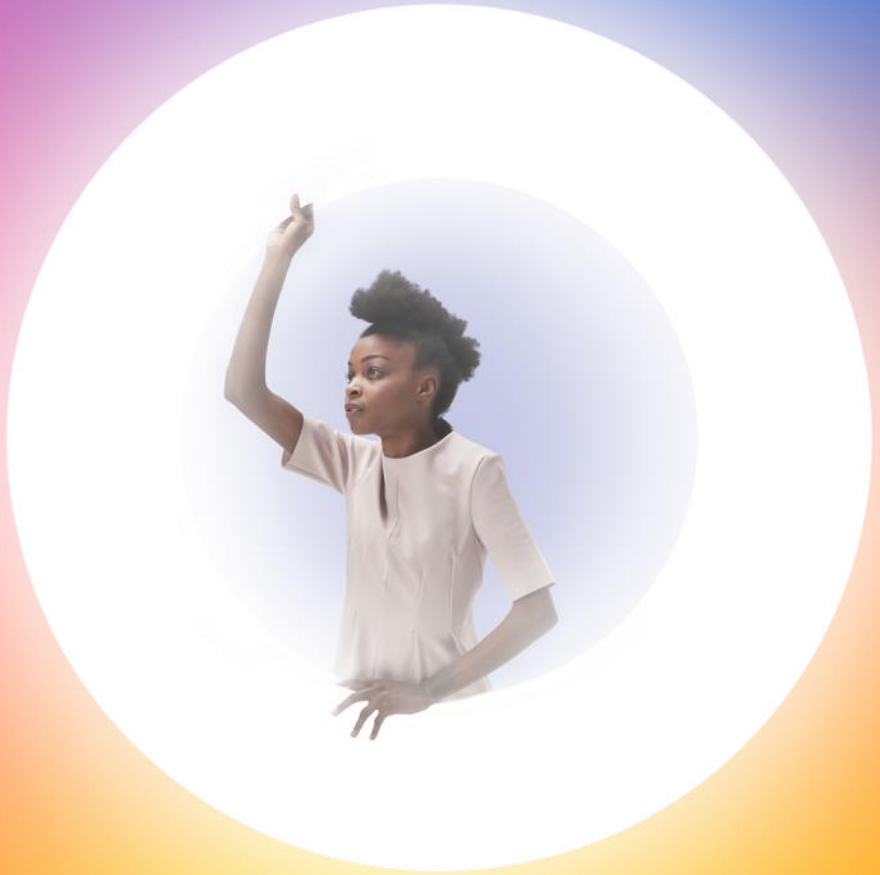
- Coding more advanced logic in Functions with good separation of concerns
- Dealing with those provisioning and reconciliation loops!
- Using state stores, secrets vaults, pub/sub middleware, workflows, encryption and other complex technologies without further increasing composition/pipeline/function complexity
- Writing to common APIs for external services running in different technology stacks, cloud providers and/or using multiple programming languages
- Doing the above while developing and testing locally, followed by a seamless deployment in Kubernetes

What can we do?



Quick

Overview



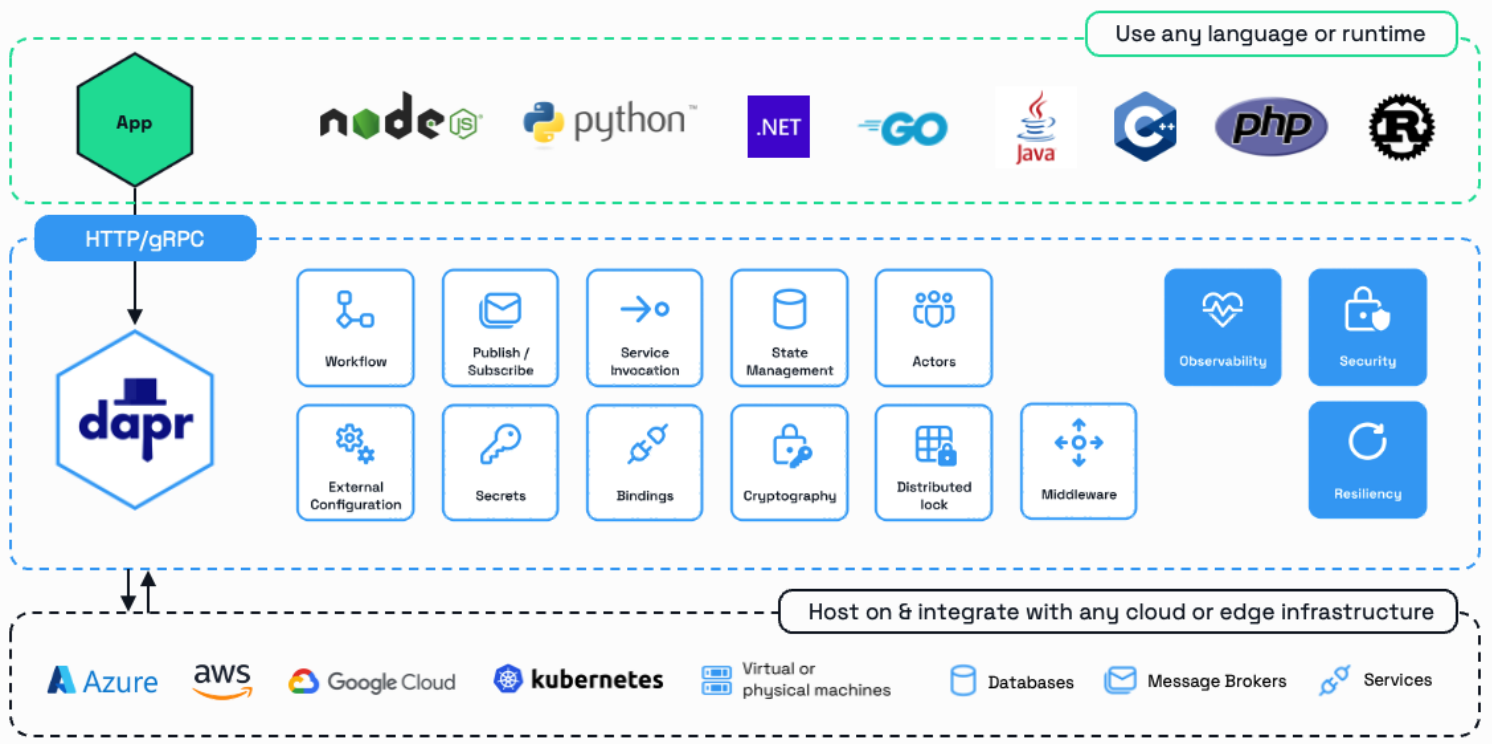
dapr

Distributed Application Runtime

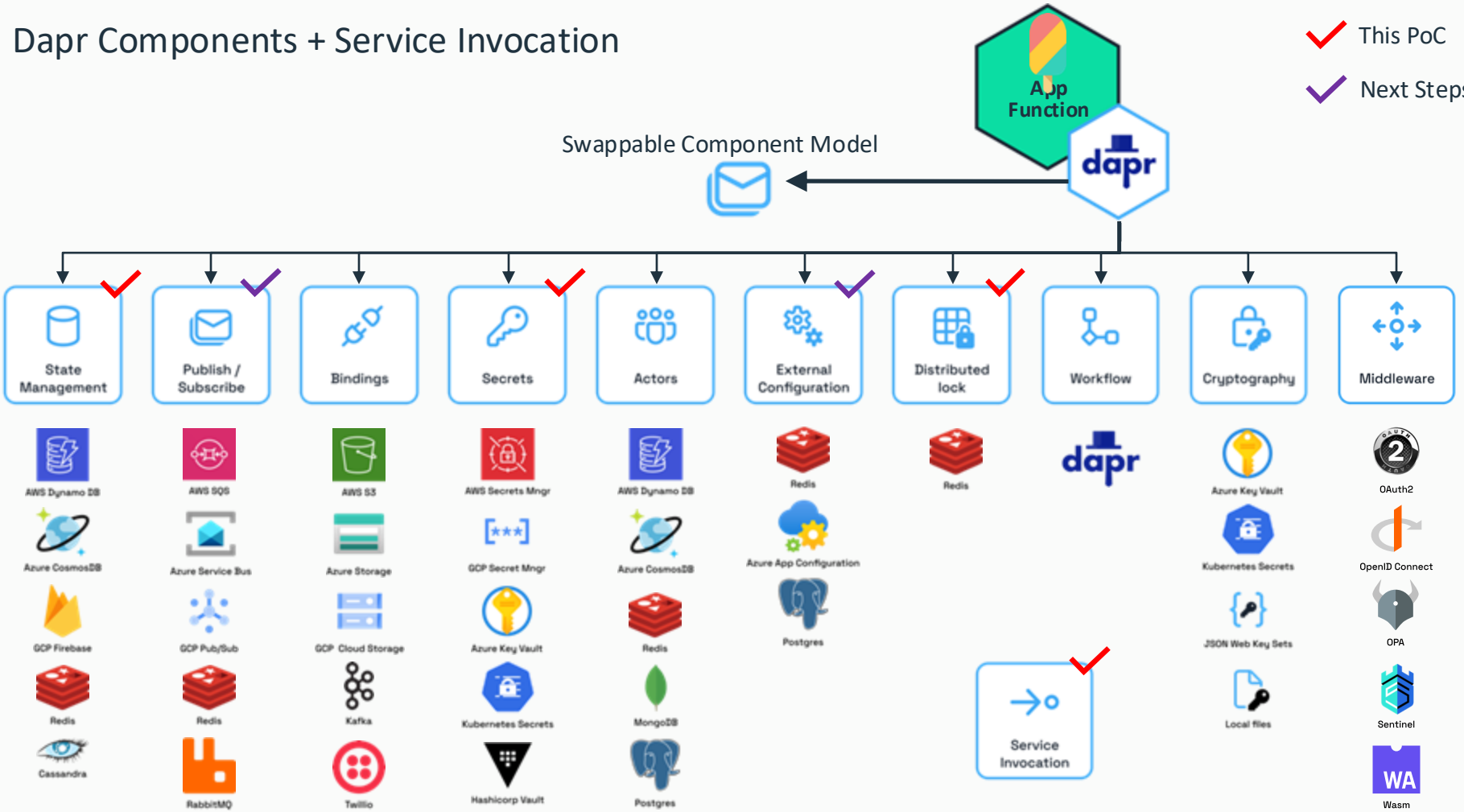
Portable, event-driven, runtime for building distributed applications across cloud and edge

https://dapr.io

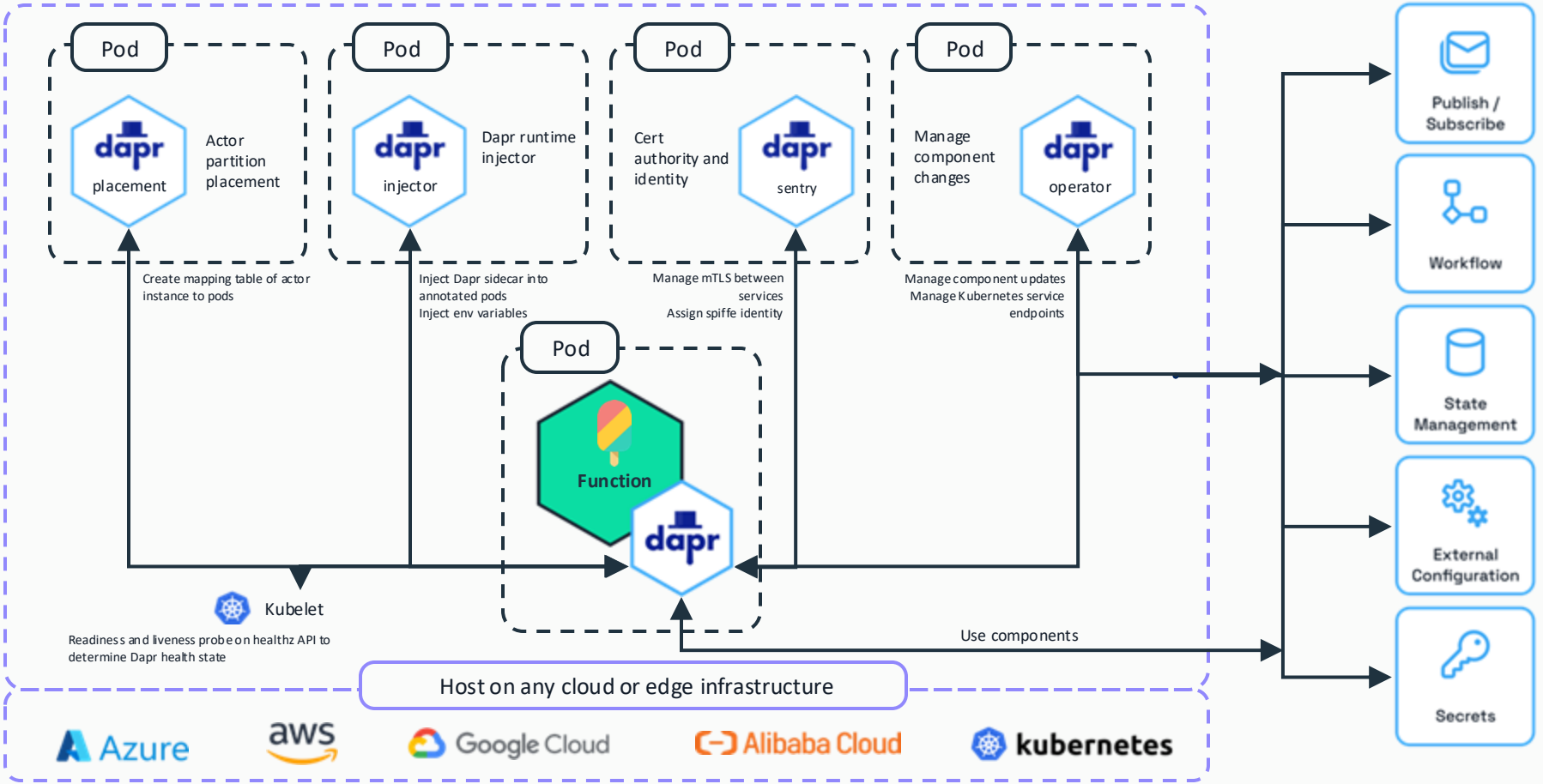
https://docs.dapr.io/contributing/presentations/



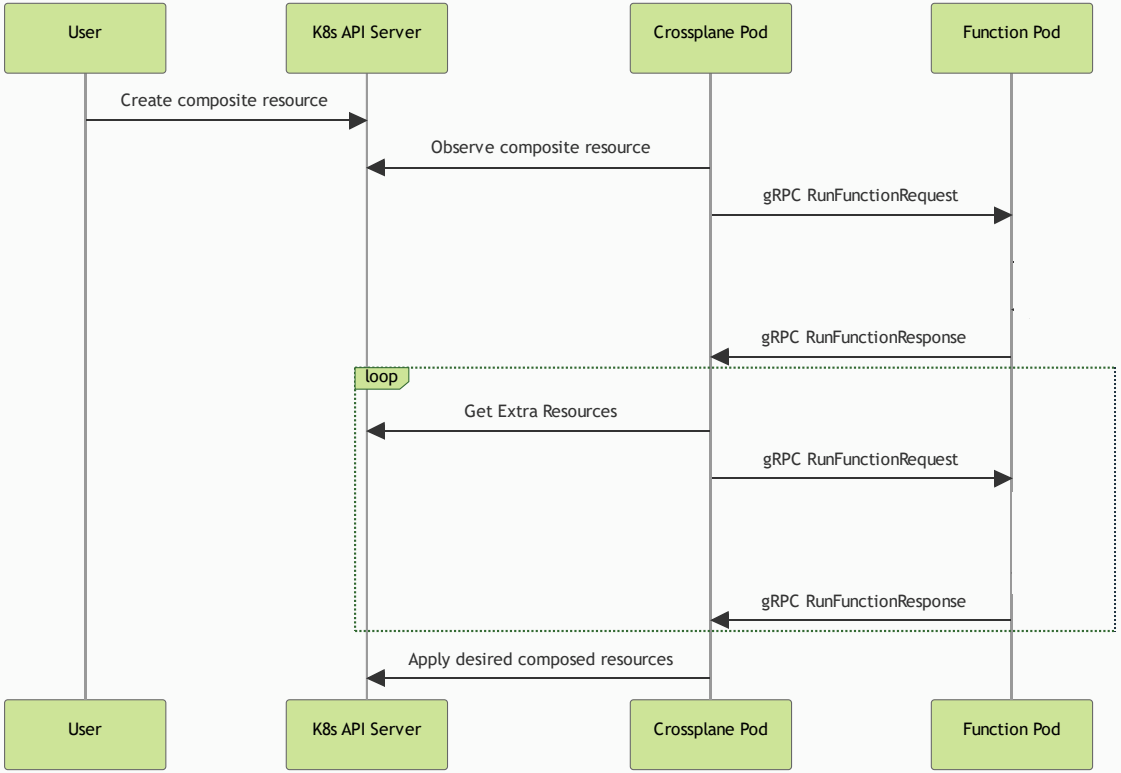
Dapr Components + Service Invocation



Dapr on Kubernetes



How Composition Functions work and how do the work with Dapr

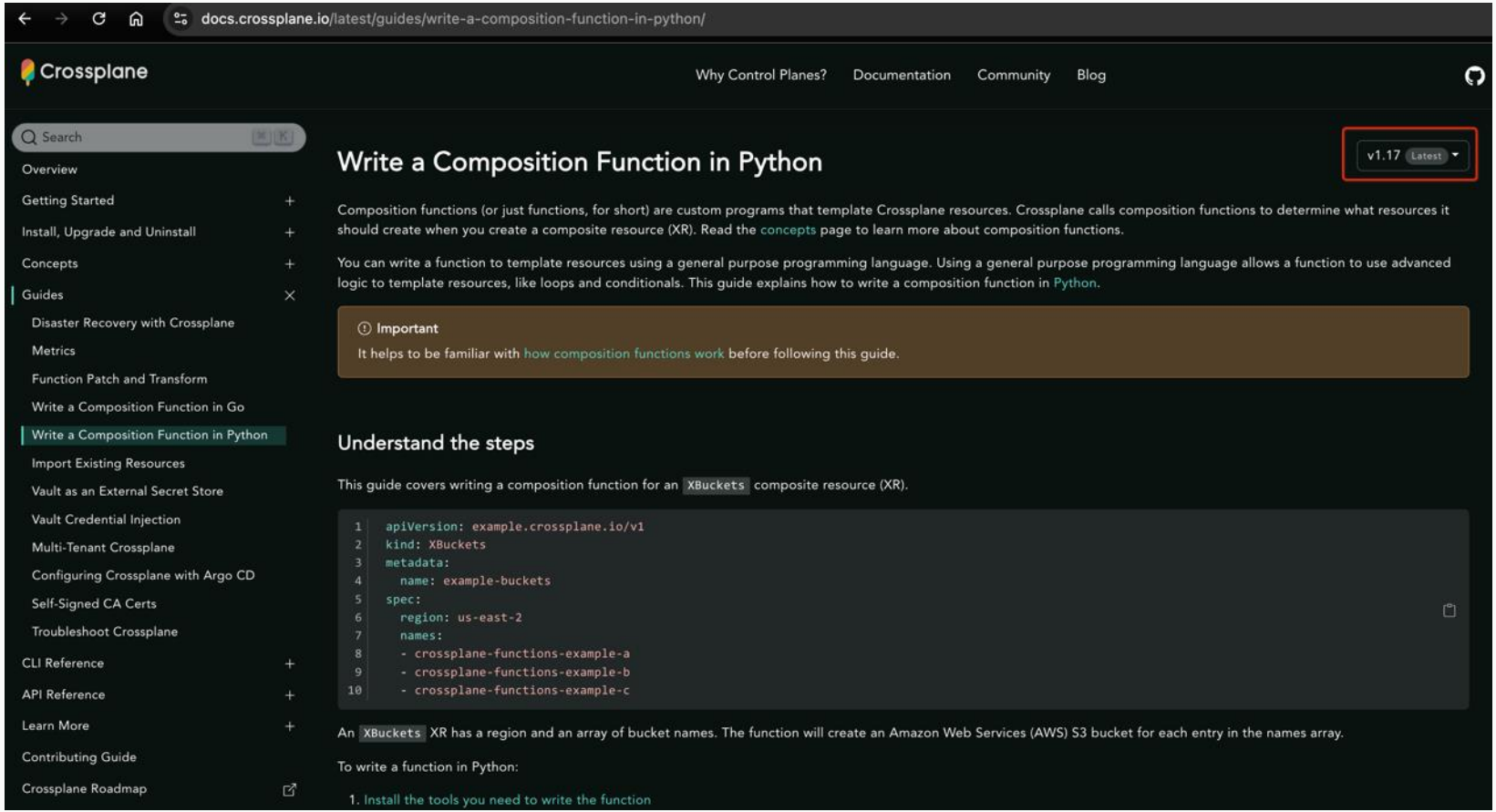


https://docs.crossplane.io/latest/concepts/compositions/#how-composition-functions-work

Prerequisite steps for using Dapr

- Install the `dapr` cli in your laptop and run a `dapr init` command
- Install Dapr in your Kubernetes cluster
  - `dapr init -k` or
  - Run the Dapr installation Helm chart for greater control over configuration: https://docs.dapr.io/operations/hosting/kubernetes/kubernetes-deploy/

Minimum modifications to a Composition Function template project in Python

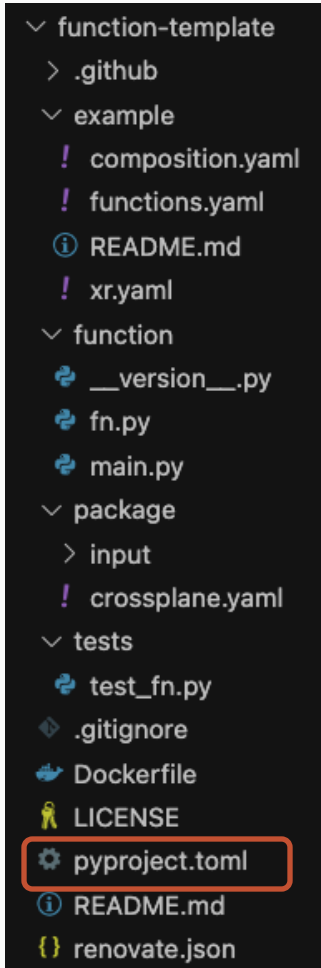


Minimum modifications to a Composition Function template project in Python

- Add Python `requests` package to `pyproject.toml`
- That's all you need to call Dapr via HTTP from a Crossplane Function

Add a current version of the Python 'requests' package

```
dependencies = [
  "requests==2.32.3",
  "crossplane-function-sdk-python==0.5.0",
  "click==8.1.7",
  "grpcio==1.67.0",
]
```



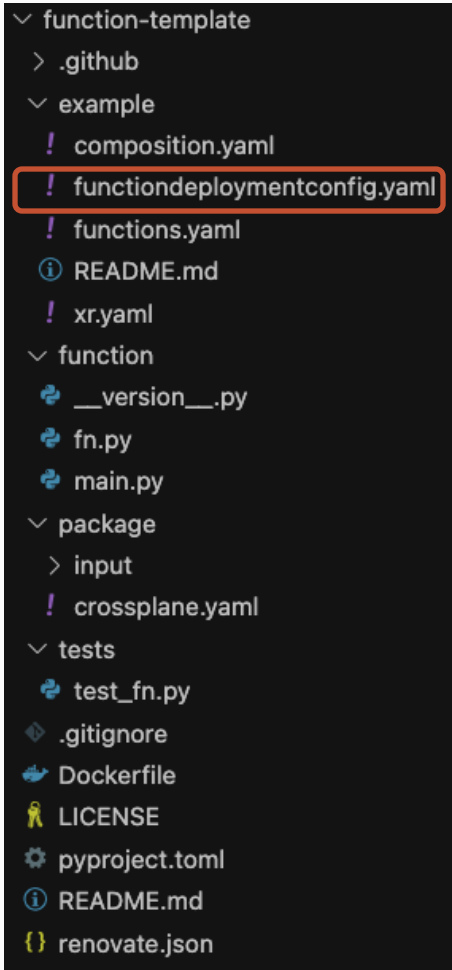
Minimum modifications to a Composition Function template (any language)

- Add a `DeploymentRuntimeConfig` manifest to your project

Add name to function Config Ref parameter

Inject the Dapr sidecar when deploying the function

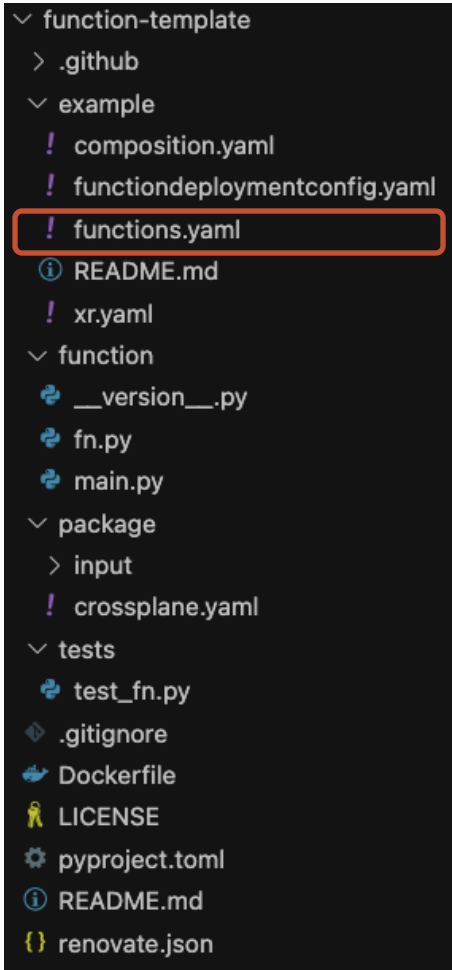
```yaml
# DeploymentRuntimeConfig to inject Dapr to a function
# Change dapr.io/app-id annotation to match your function name
apiVersion: pkg.crossplane.io/v1beta1
kind: DeploymentRuntimeConfig
metadata:
  name: function-policy-validation-deployment-config
spec:
  deploymentTemplate:
    metadata:
      labels:
    spec:
      selector: {}
      template:
        metadata:
          annotations:
            dapr.io/enabled: "true"
            dapr.io/scheduler-host-address: ""
            dapr.io/app-id: "<your-function-name>"
```



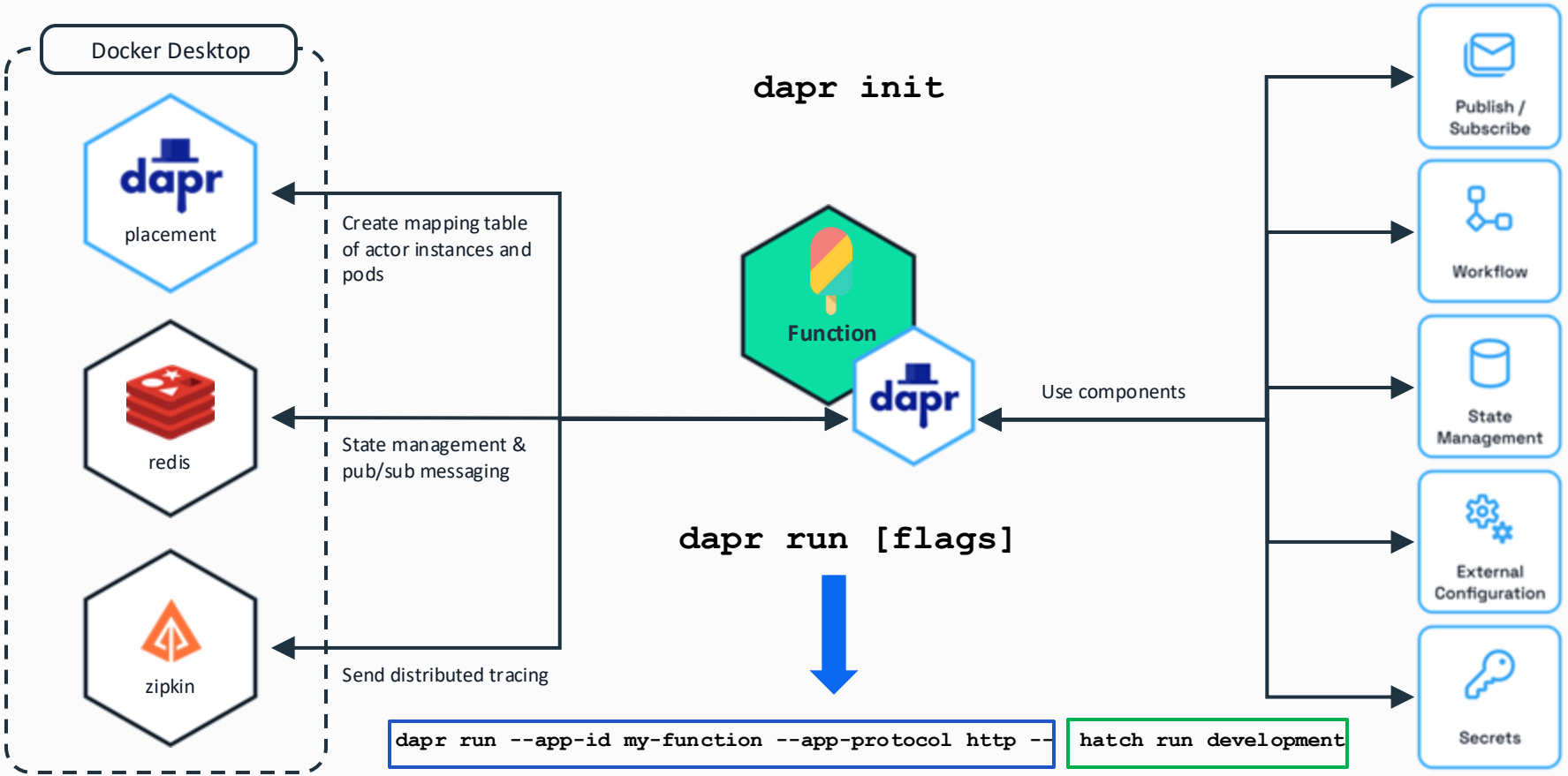
Minimum modifications to a Composition Function template (any language)

- Add reference to your `DeploymentRuntimeConfig` manifest for your `functions.yaml` manifest

```yaml
apiVersion: pkg.crossplane.io/v1
kind: Function
metadata:
  name: function-policy-validation
  annotations:
    # This tells crossplane beta render to connect to the function locally.
    render.crossplane.io/runtime: Development
spec:
  # This is ignored when using the Development runtime.
  package: <function docker image name>:<tag>
  runtimeConfigRef:
    name: function-policy-validation-deployment-config
```



Local development with the Dapr CLI



https://docs.dapr.io/getting-started/install-dapr-cli/

Putting it all together

Crossplane

+

dapr

PoC: Kyverno Policy Validator for Crossplane

Problem we're trying to solve in this PoC

Current Status:

1. The Kyverno project provides a comprehensive set of tools to manage the complete Policy-as-Code (PaC) for Kubernetes and other cloud native environments.
2. Kyverno CLI can be used to apply and test policies off-cluster e.g. in IaC and CI/CD pipelines.
3. Policy enforcement happens at the "last mile" in Kubernetes admission controllers.
4. We want to make broad use of Kyverno policies for "Everything-as-Code".
5. We don't want to mutate in the admission controllers and clash with Crossplane.

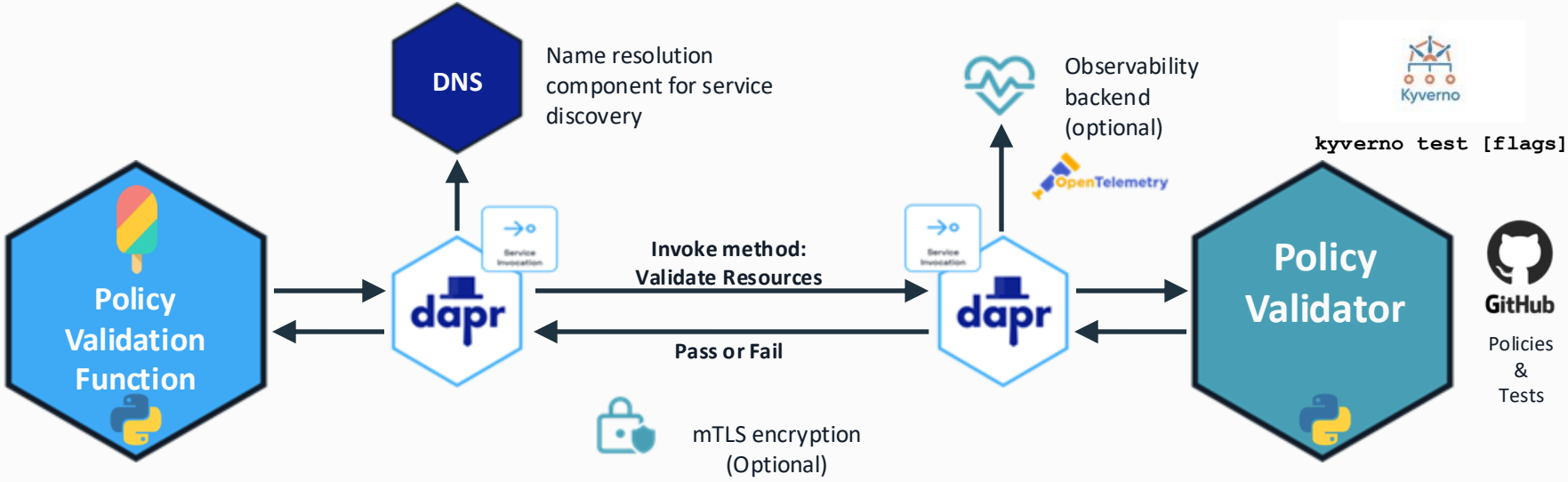
Gap:

- With Crossplane Functions, resources could be generated **dynamically:**
  - Bullet #2 Happens too early. Not all resources to be generated may exist.
  - Bullet #3 Happens too late to validate Compositions prior to deployment

Proposed Solution:

- A Policy Validation Function to be called towards the end of a Composition. It will wrap a `kyverno test` command against the top Composite resource generated in-flight and the applicable policies and tests for the top Composite resource stored in a GitHub repo.

Policy Validation Function and PolicyValidator Microservice



POST
http://localhost:3500/v1.0/**invoke**/policyvalidator/method/validate

POST
http://localhost:5100/pass

POST
http://localhost:5100/fail

POST
http://localhost:5100/locked

Some of PoC challenges and solutions

Challenges: Complexity writing a monolithic function! The mission:

a) **Gather all the desired state resources and render in YAML for the kyverno CLI**

b) **Securely `git clone` the repo with the applicable policies and tests**

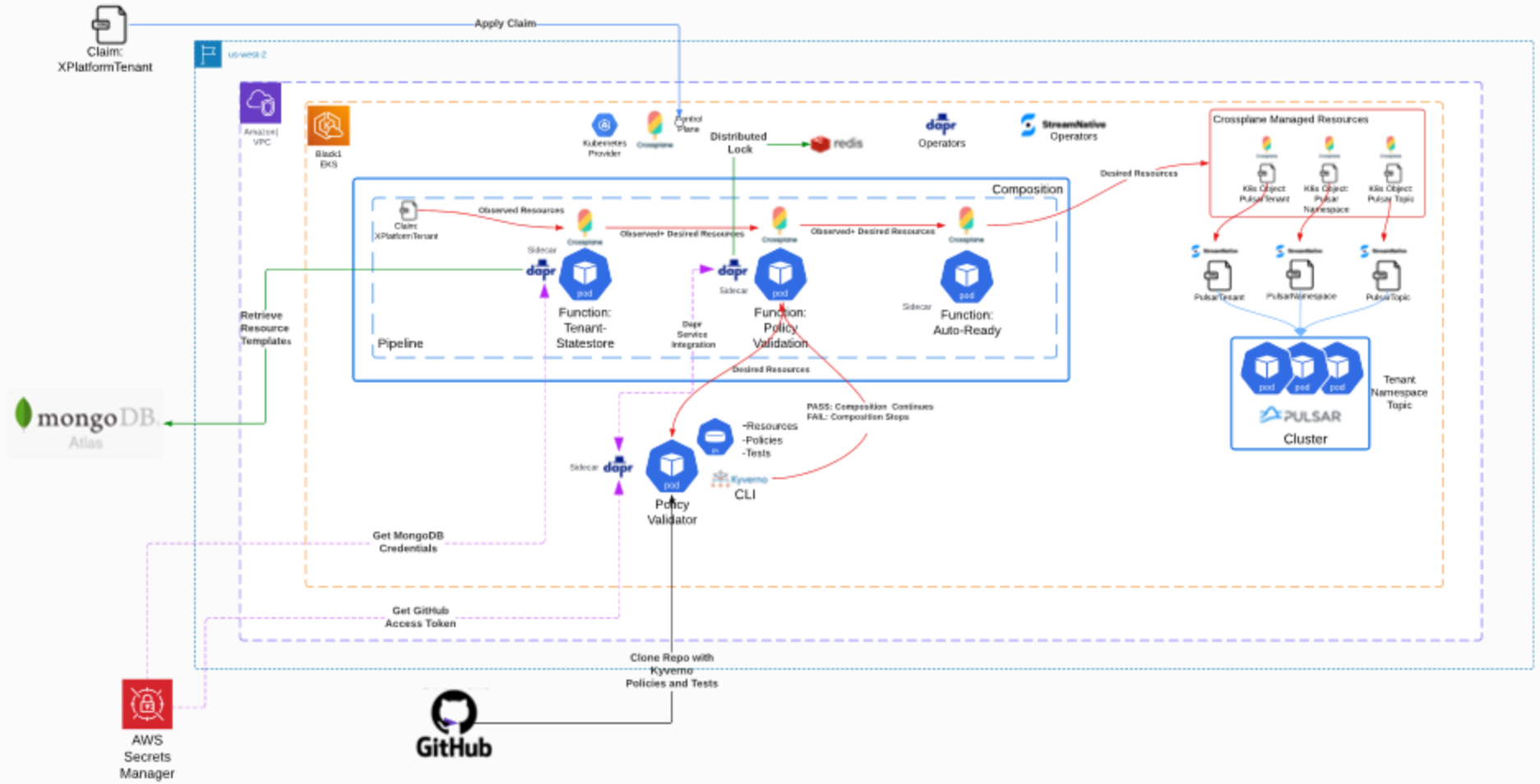
c) **Store the data in a file system in the right format and structure**

d) **Call the `kyverno test` command, format the output, act based on pass/fail and clean up file system**

e) **Handle the Crossplane the creation loop mechanics and don't break the Composition**

Solutions:

- A function to call a PolicyValidator microservice using Dapr Service Invocation (HTTP). Offload the heavy lifting
- Handle transformations in Python: Crossplane proto → JSON strings → Python dict → YAML (Booleans are tricky)
- Dapr Secrets Management to retrieve access token to policies/tests GitHub repo
- Dapr Distributed Lock to lock PolicyValidator (PV) microservice for X minutes after first call until it can return a `pass`, `fail or locked` back to the calling function. Three outcomes:
  1. PV returns `pass`: Calling function returns desired state resources unaltered from the prior function.
  2. PV returns `fail`: Calling function fails the composition returning an abnormal termination log entry.
  3. PV returns `locked`: Similar to a `pass`, but with a warning log entry to look for `pass` or `fail` log entry
- PolicyValidator file system and repo naming convention: kind-apiVersion. Concurrency out of scope for PoC.


Claim:
XPlatformTenant
Apply Claim
us-west-2
Amazon
VPC
Black
EKS
Kubernetes
Provider
Control
Plane
Distributed
Lock
redis
dapr
Operators
StreamNative
Operators
Crossplane Managed Resources
Kiki Object:
Pulsar Tenant
Kiki Object:
Pulsar
Namespace
Kiki Object:
Pulsar Topic
Desired Resources
Composition
Claim:
XPlatformTenant
Observed Resources
Observed+ Desired Resources
Observed+ Desired Resources
Sidecar
Function:
Tenant-
Statestore
Dapr
Service
Integration
Function:
Policy
Validation
Function:
Auto-Ready
Pipeline
Desired Resources
PulsarTenant
PulsarNamespace
PulsarTopic
Retrieve
Resource
Templates
mongoDB.
Atlas
PASS: Composition Continues
FAIL: Composition Stops
-Resources
-Policies
-Tests
Kyverno
CLI
Policy
Validator
PULSAR
Cluster
Tenant
Namespace
Topic
Get MongoDB
Credentials
Get GitHub
Access Token
Clone Repo with
Kyverno
Policies and Tests
AWS
Secrets
Manager
GitHub

Next Steps

- Create a function to update a state store and send an event to an event service using the Transactional Outbox pattern (*). Make reporting Kyverno cli test results a bit friendlier
- Test the Dapr Configuration building block to facilitate configuring function parameters
- Upload project: functions, scripts, configuration file, etc. to a GitHub repository
- Submit PR to the Dapr community and Diagrid to add a “create” method to the Secrets Management building block to create new secrets.
    - This is not typical application developer function, but a necessary one for platform engineering
- Socialize Crossplane/Dapr integration with Upbound and the Crossplane community

(*) https://docs.dapr.io/developing-applications/building-blocks/state-management/howto-outbox/

Thank You!

Hugo Smitter

Platform Architect

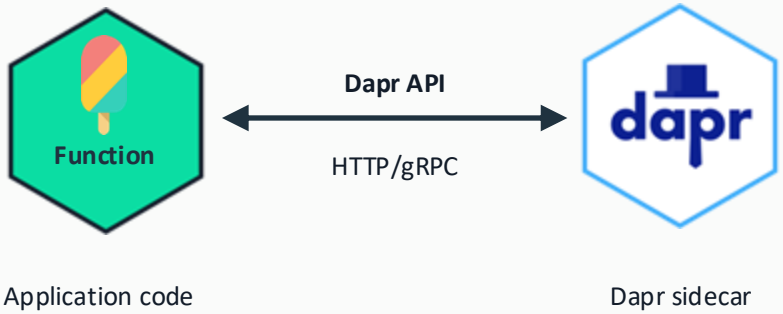
hugosmitter@fico.com

(+1) 214 534 5373

http://github.com/smitterh/Crossplane-Dapr-Integration

Appendix

Sidecar pattern and the Dapr API



POST http://localhost:3500/v1.0/**invoke**/cart/method/order

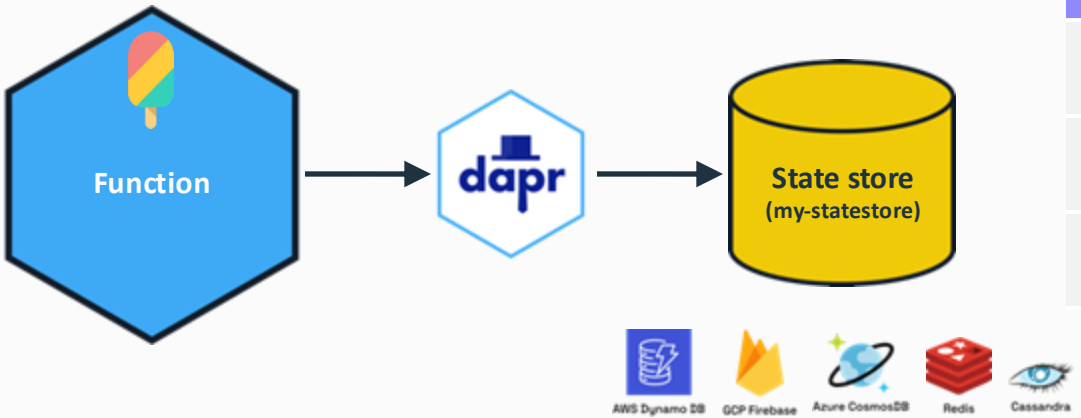
GET http://localhost:3500/v1.0/**state**/inventory/item50

POST http://localhost:3500/v1.0/**publish**/mybroker/order-messages

GET http://localhost:3500/v1.0/**secrets**/vault/dbaccess

POST http://localhost:3500/v1.0-beta1/**workflows**/dapr/businessprocess/start

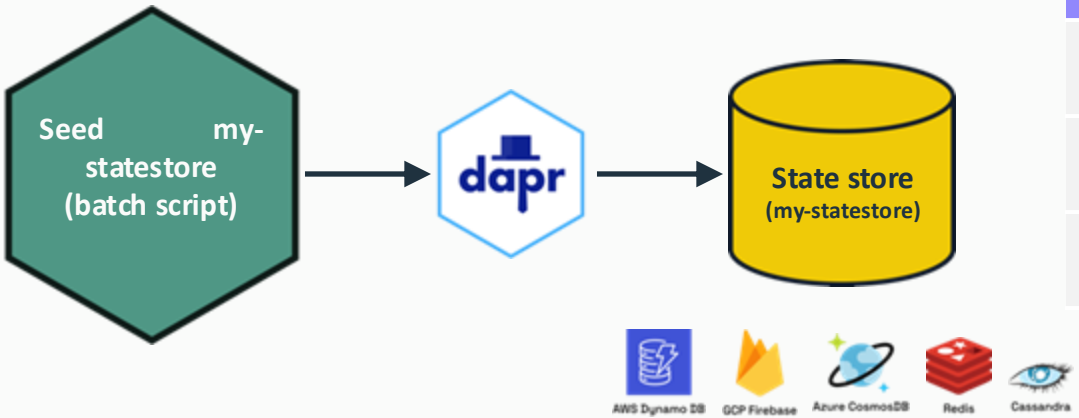
State Management (Key/Value)



key	field	value
template-pulsar-tenant	data	"{apiVersion: ...}"
template-pulsar-namespace	data	"{apiVersion: ...}"
template-pulsar-topic	data	"{apiVersion: ...}"

GET
http://localhost:3500/v1.0/state/my-statestore/template-pulsar-tenant

State Management (Key/Value) (cont.)

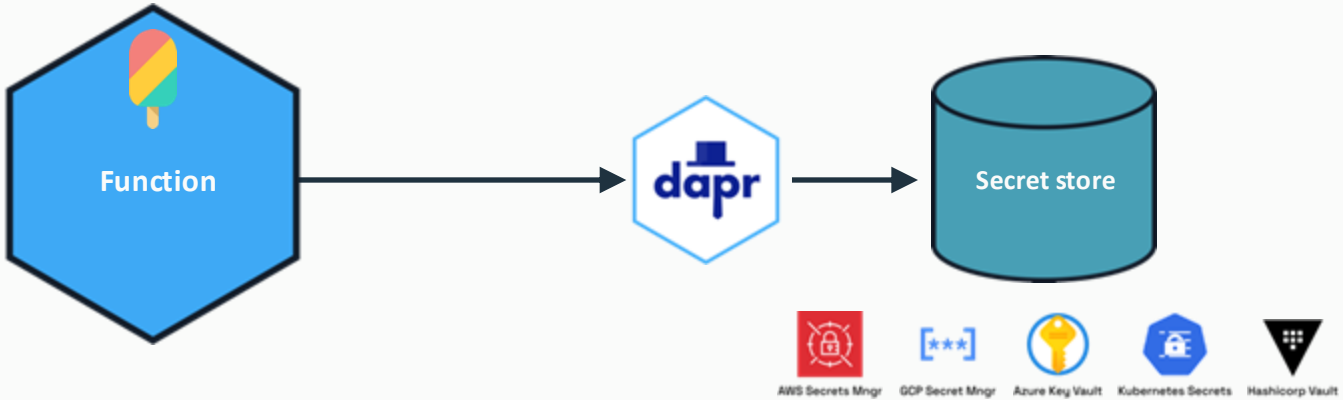


key	field	value
template-pulsar-tenant	data	"{apiVersion: ...}"
template-pulsar-namespace	data	"{apiVersion: ...}"
template-pulsar-topic	data	"{apiVersion: ...}"

POST
http://localhost:3500/v1.0/state/my-statestore

```
[{
  “key”: “template-pulsar-tenant”,
  “value”: “{apiVersion: x, kind: PulsarTenant, metadata: ...}”
}]
```

Secrets Management (use case 1)

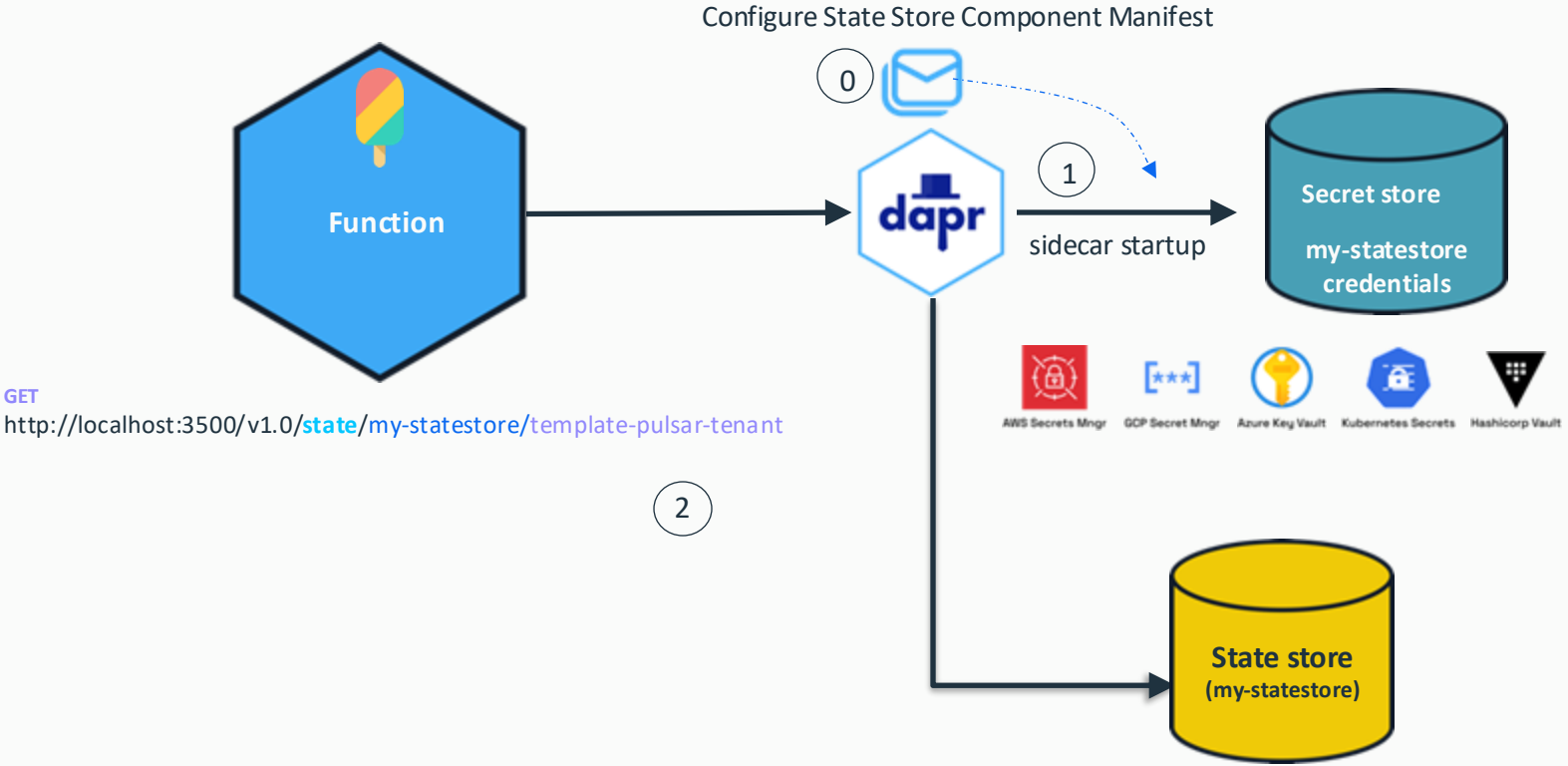


GET http://localhost:3500/v1.0/secrets/myvault/mysecret

RESPONSE

```
{
  “mysecret”: “secretvalue”
}
```

Secrets Management (use case 2)



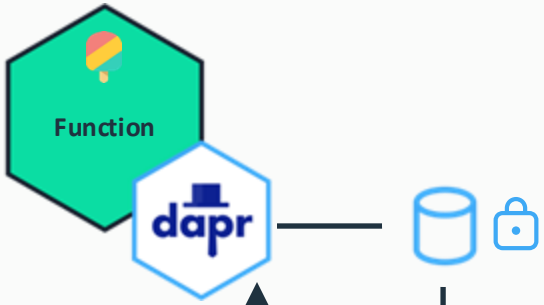
Distributed Lock

ALPHA

The distributed lock API provides mutually exclusive access to resources.

- Only a single instance of an application can hold a lock
- Locks are scoped to a Dapr app-id
- Uses a lease-based locking mechanism to prevent deadlocks

Distributed Lock



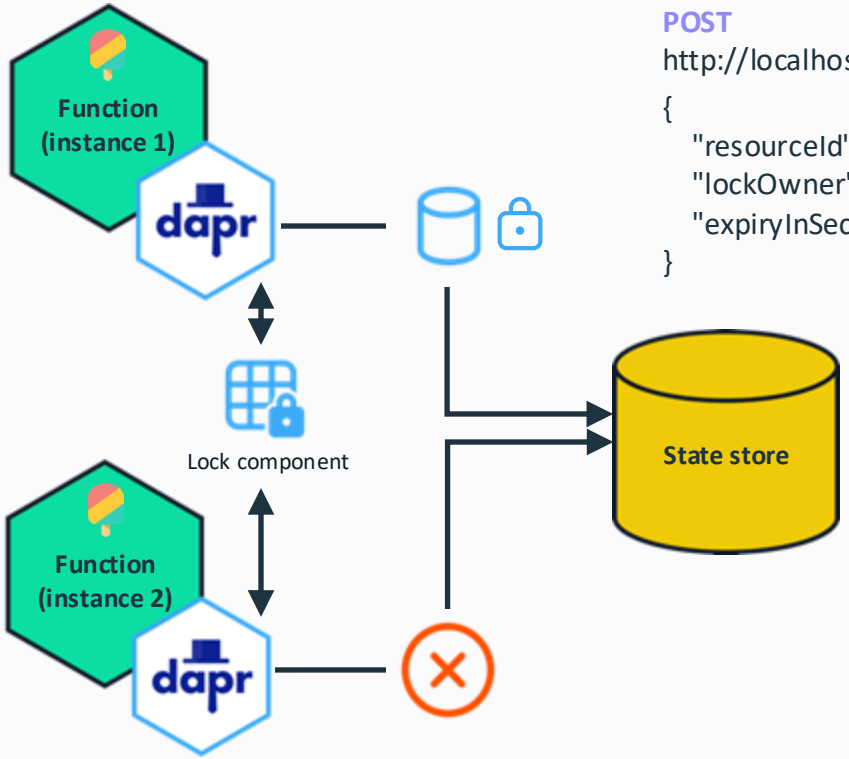
Lock component

POST

http://localhost:3500/v1.0-alpha1/**lock**/statestore

```
{
   "resourceId" : “lock1",
   "lockOwner" : “my-function",
   "expiryInSeconds": 60
}
```

Distributed Lock

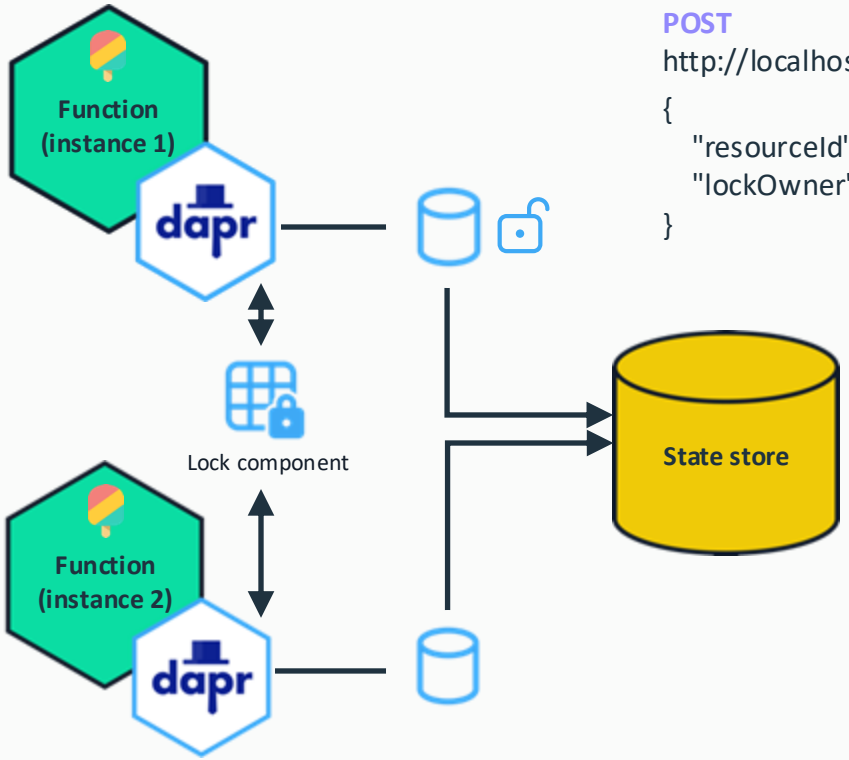


POST

http://localhost:3500/v1.0-alpha1/**lock**/statestore

```
{
   "resourceId" : “lock1",
   "lockOwner" : “my-function",
   "expiryInSeconds": 60
}
```

Distributed Lock



POST

http://localhost:3500/v1.0-alpha1/**unlock**/statestore

```
{
   "resourceId" : “lock1",
   "lockOwner" : “my-function“
}
```

Service Invocation

The service invocation API allows synchronous communication between services.

- Service discovery via name resolution components
- Invoke HTTP and gRPC services consistently
- Configurable resiliency policies
- Built-in distributed tracing & metrics
- Access control policies & mTLS
- Chain pluggable middleware components