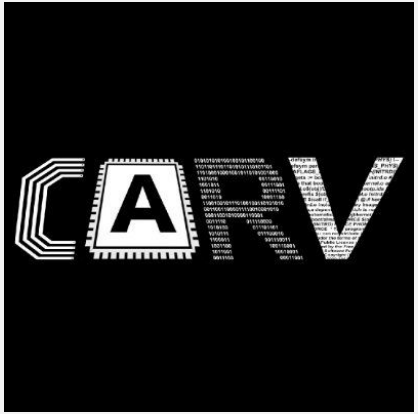
Frisbee

Automated Testing Over Kubernetes

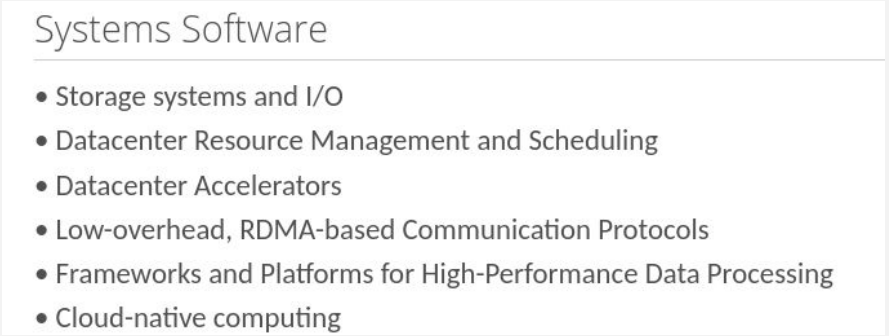
Fotis Nikolaidis
Systems Research Engineer
ICS-FORTH, Greece

A story about a lab …



A story about a lab …

- The fun …



- The misery …
  - Writing proper tests require the same level of engineering effort as the system they test! Much less attractive to write ….
  - As opposed to features, the value of tests is seldom visible in the short term
  - Test automation becomes essential as the projects grow.

Systems Testing Challenges

- Multiple interacting components
- Multiple programming environments
- Multiple libraries
- Multiple applications
- Multiple architectures
- Multiple clusters
- Functionality and performance are both important

Key Requirements

- Allow researchers to focus on their systems, minimizing distractions from testing issues.
- Offers a fully automated and disposable testing environments with the tools that researchers may need.
- Help researchers under the performance and behavior of their systems, under various operating conditions.
- Automatically validates the system for transition into erroneous states or SLA violations.
- Integration with CI/CD pipelines to test early, and test often !

The testing landscape

- 😖 Binary Processes on Physical Infrastructure
- 😖 Binary Processes on VMs
- 😖 Containers on Docker
- 😖 Containers on Docker-Compose
- 😖 Containers on Kubernetes
- 😖 Containers on Kubernetes with CI/CD pipelines
- 😎 Frisbee: A Kubernetes-native testing platform

The testing landscape

- Binary Processes on Physical Infrastructure
  - x Manual scripts to deploy software and synchronize test-steps.
  - x Portability of the test
  - x High maintenance cost
  - ✓ Low execution overheads

The testing landscape

- Binary Processes on VMs
  - x Manual scripts to deploy software and synchronize test-steps.
  - ✓ Portability of the test
  - x High maintenance cost
  - x High execution overheads

The testing landscape

- Containers on Docker
  - x Manual scripts to deploy software and synchronize test-steps.
  - ✓ Portability of the test
  - ✓ Low maintenance cost
  - ✓ Low execution overhead

The testing landscape

- Containers on Docker-Compose
  - ✓ Rich and expressive DSL for writing multi-stage scenarios
  - ✓ Portability of the test
  - ✓ Low maintenance cost
  - ✓ Low execution overhead
  - x Bound to a single node

A Compose file looks like this:

```
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
  redis:
    image: redis
```

```
$ docker-compose up

Creating network "composetest_default" with the default driver
Creating composetest_web_1 ...
Creating composetest_redis_1 ...
Creating composetest_web_1
Creating composetest_redis_1 ... done
```

The testing landscape

- Containers on Kubernetes
  - x DSL for deployment and configuration, not for multi-stage experiments
  - ✓ Portability of the test
  - ✓ Low maintenance cost
  - ✓ Low execution overhead
  - ✓ Multi-node testing environment

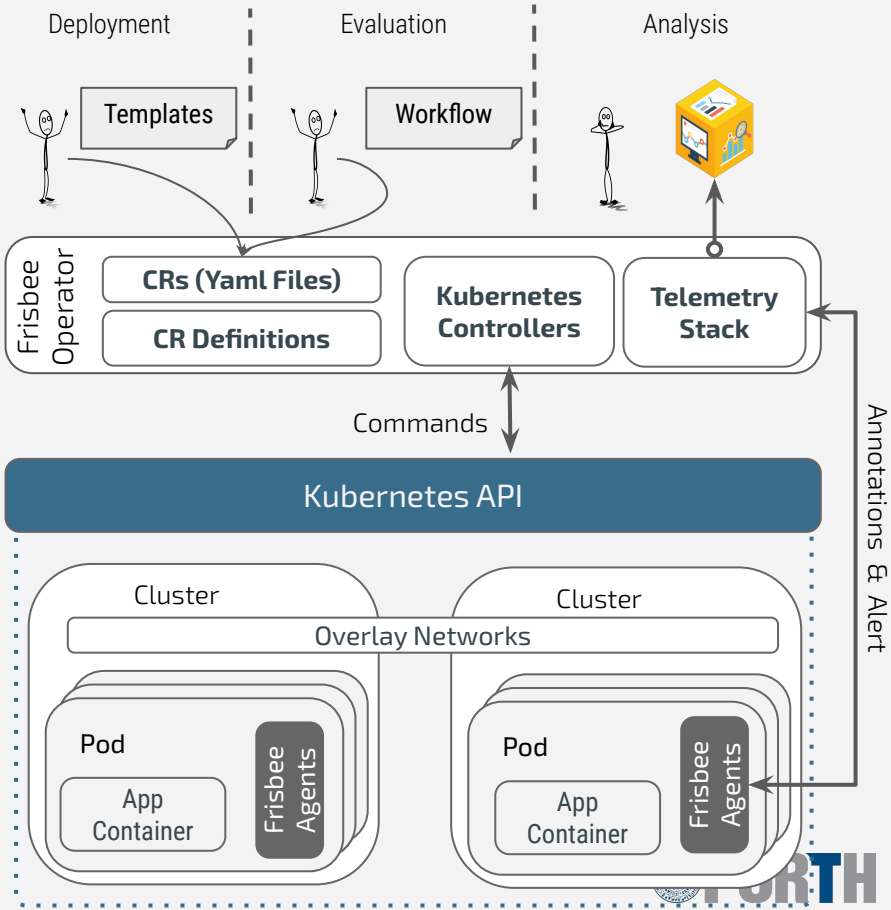
The testing landscape

- Containers on Kubernetes with CI/CD pipelines
  - ✓ Rich and expressive DSL for writing multi-stage scenarios
  - ✓ Portability of the test
  - ✓ Low maintenance cost
  - ✓ Low execution overhead
  - ✓ Multi-node testing environment
  - x Still requires human effort to analyze the results

The Frisbee Testbed

Frisbee is a Kubernetes extension that provides:

- ✓ Rich and expressive DSL for writing complex testing scenarios
- ✓ Portability of the test
- ✓ Low maintenance cost
- ✓ Low execution overhead
- ✓ Multi-node testing environment
- ✓ Programmatically assertable conditions



A “Hello, Network!” Frisbee test


```
spec:
 actions:
   # Create an iperf server
 1 action: Service
   name: server
 2 service:
     templateRef: iperf.server

   # Create a cluster of iperf clients
 3 action: Cluster
   name: clients
   depends: { running: [ server ] }   4
   assert:
 5   state: '{{.state.NumOfFailures()}} >= 1'
     metrics: 'avg() of query(metric, 5m, now) is below(1000)'
   cluster:
 6   templateRef: iperf.client
     instances: 30
     inputs:                          8
 7     - { server: .service.server.one, seconds: "600" }
       - { server: .service.server.one, seconds: "30" }
     schedule:
 9     cron: "@every 1m"
```

Scenario

1. Instantiate a Service
2. Based on the given template
3. Then instantiate a cluster of services
4. When certain conditions are met.
5. Abort on failures or SLA violations
6. Within the cluster, create 30 services.
7. Iterating over the given inputs.
8. Inputs may use addressing macros
9. Schedule 1 service every 1 minute.

A “Hello, Network!” Frisbee test

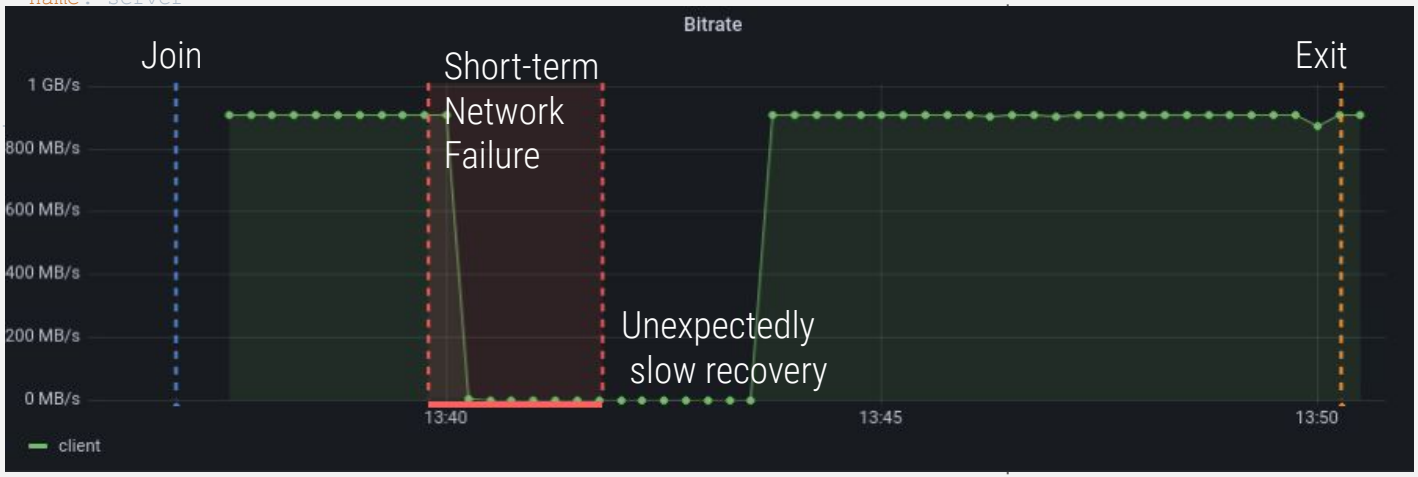

```
spec:
 actions:
   # Create an iperf server
 1 action: Service
   name: server
 2 service:
     templateRef: iperf.server

   # Create a cluster of iperf cl
 3 action: Cluster
   name: clients
   depends: { running: [ server
   assert:
 5   state: '{{.state.NumOfFail
     metrics: 'avg() of query(m
   cluster:
 6   templateRef: iperf.client
     instances: 30
     inputs:                                  8
 7     - { server: .service.server.one, seconds: "600" }
       - { server: .service.server.one, seconds: "30" }
 9   schedule:
       cron: "@every 1m"
```

Scenario

1. Instantiate a Service
7. Iterating over the given inputs.
8. Inputs may use addressing macros
9. Schedule 1 service every 1 minute.

A “Bye Bye, Network!” Frisbee test


```
spec:
 actions:
   # Create an iperf server
   - action: Service
     name: server
```




```
   # After a while, inject a network partition
   - action: Chaos
     name: partition
     depends: { running: [ server ] }
     chaos:
       templateRef: chaos.network.partition
       inputs:
         - { server: .service.server.one, duration: "2m" }
```

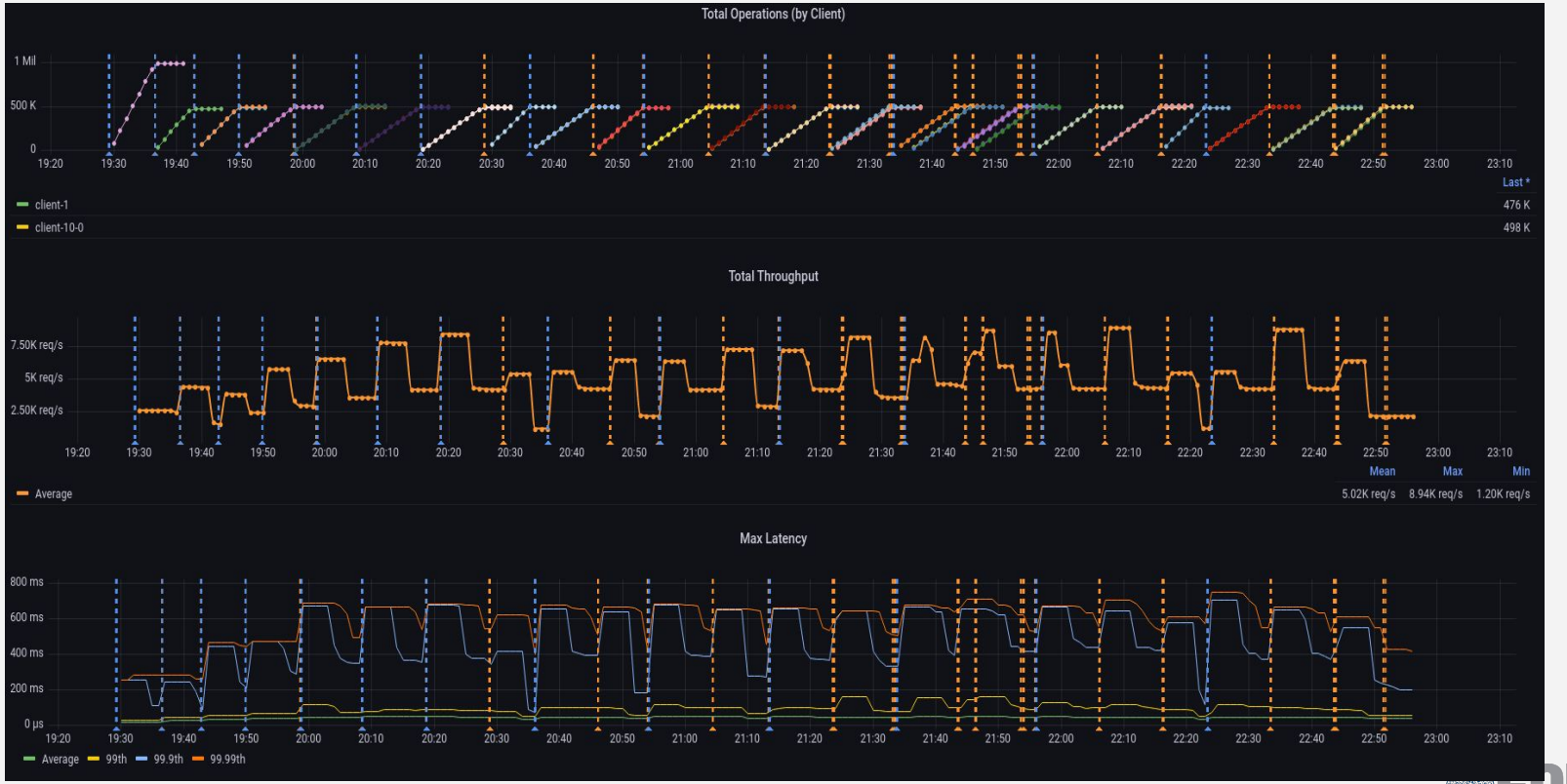
1. Abstract failures as Chaos Jobs
2. Enable execution-driven fault injection

FORTH INSTITUTE OF COMPUTER SCIENCE

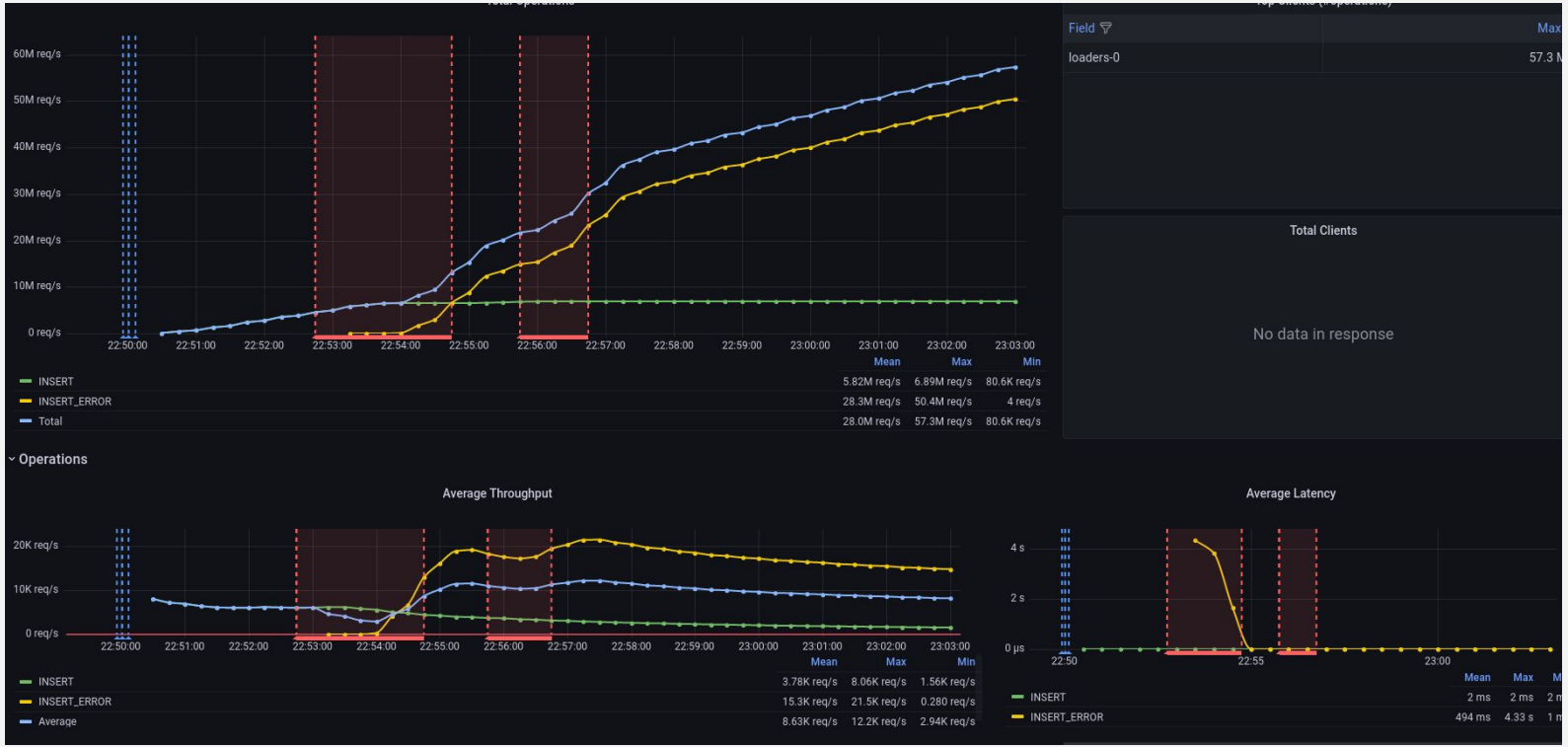
Performance

Scalability

Elasticity

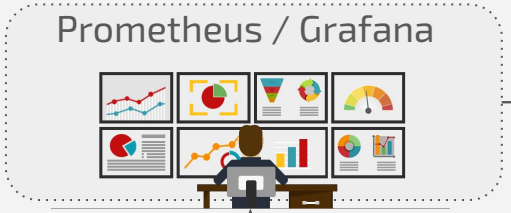

Total Operations (by Client)
1 Mil
500 K
0
19:20 19:30 19:40 19:50 20:00 20:10 20:20 20:30 20:40 20:50 21:00 21:10 21:20 21:30 21:40 21:50 22:00 22:10 22:20 22:30 22:40 22:50 23:00 23:10
Last *
client-1 476 K
client-10-0 498 K
Total Throughput
7.50K req/s
5K req/s
2.50K req/s
19:20 19:30 19:40 19:50 20:00 20:10 20:20 20:30 20:40 20:50 21:00 21:10 21:20 21:30 21:40 21:50 22:00 22:10 22:20 22:30 22:40 22:50 23:00 23:10
Mean Max Min
Average 5.02K req/s 8.94K req/s 1.20K req/s
Max Latency
800 ms
600 ms
400 ms
200 ms
0 µs
19:20 19:30 19:40 19:50 20:00 20:10 20:20 20:30 20:40 20:50 21:00 21:10 21:20 21:30 21:40 21:50 22:00 22:10 22:20 22:30 22:40 22:50 23:00 23:10
Average 99th 99.9th 99.99th

Availability

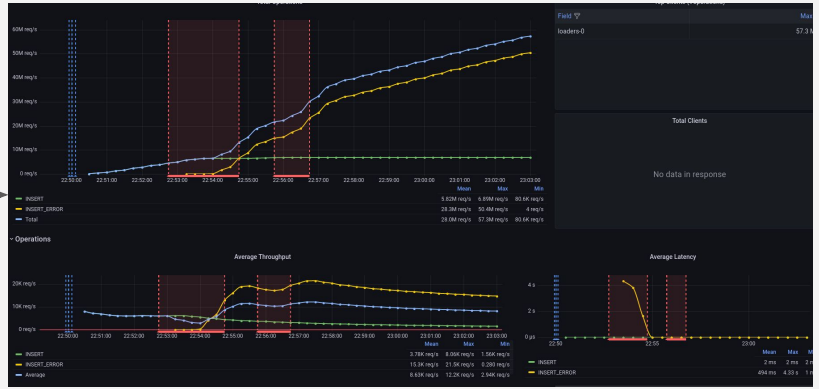


Assert: Metrics-Driven

Metrics-Driven assertions check whether the system operates within expected limits.

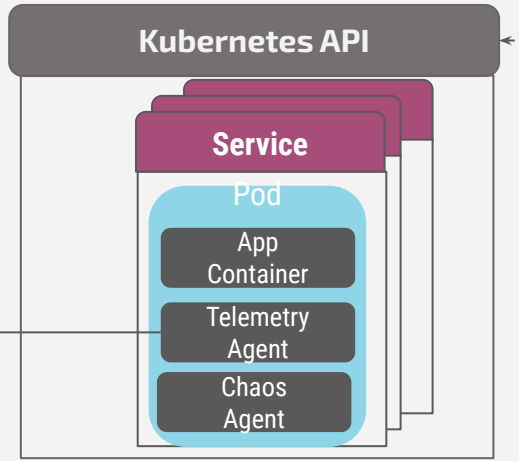


Real-Time Dashboards



Alerts

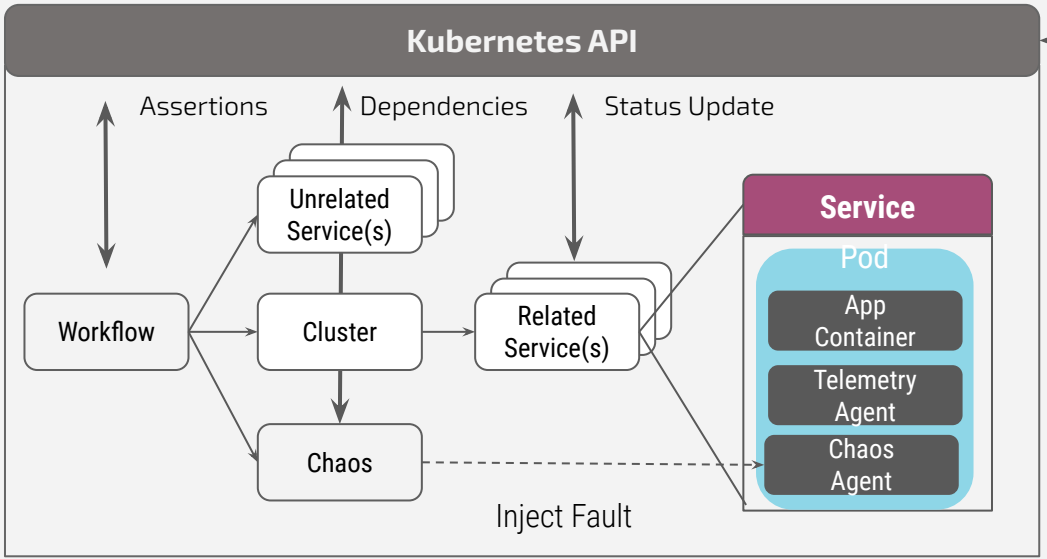
```
metrics:
avg()
of query(metric, 5m, now)
is below(1000)
```



React

Assert: Object State

State-Driven assertions check whether the system transits into erroneous states.



Managed States

- Pending
- Running
- Success
- Failed
- Suspended

FORTH INSTITUTE OF COMPUTER SCIENCE

Test factories - Reusability

Scenario:
netfailure.yaml

```
spec:
 actions:
   # Create an iperf server
   - action: Service
     name: server
     service:
       templateRef: iperf.server

   # Create a cluster of iperf clients
   - action: Cluster
     name: clients
     depends: { running: [ server ] }
     assert:
       state: '{{.state.NumOfFailures()}} >= 1'
       metrics: 'avg() of query(metric, 5m, now) is
below(1000)'
     cluster:
       templateRef: iperf.client
       instances: 30
       inputs:
         - { server: .service.server.one, seconds: "600" }
         - { server: .service.server.one, seconds: "30" }
       schedule:
         cron: "@every 1m"

   # After a while, inject a network partition
   - action: Chaos
     name: partition
     depends: { running: [ server ] }
     chaos:
      templateRef: chaos.network.partition
      inputs:
        - { server: .service.server.one, duration: "2m" }
```

Template:
iperf-client.yaml

exposed

```
spec:
 inputs:
  parameters:
    server: localhost
    seconds: "60"
```

hidden

```
service:
 decorators:
   Telemetry:
  -    platform.telemetry.container
  -    iperfmon.client
 containers:
 - name: app
   image: someimage
   Command:...
    # … blah blah

     server={{"{{.Inputs.Parameters. server}}"}}
     seconds={{"{{.Inputs.Parameters. seconds}}"}}

     iperf3 -c ${server} -t ${seconds}
```

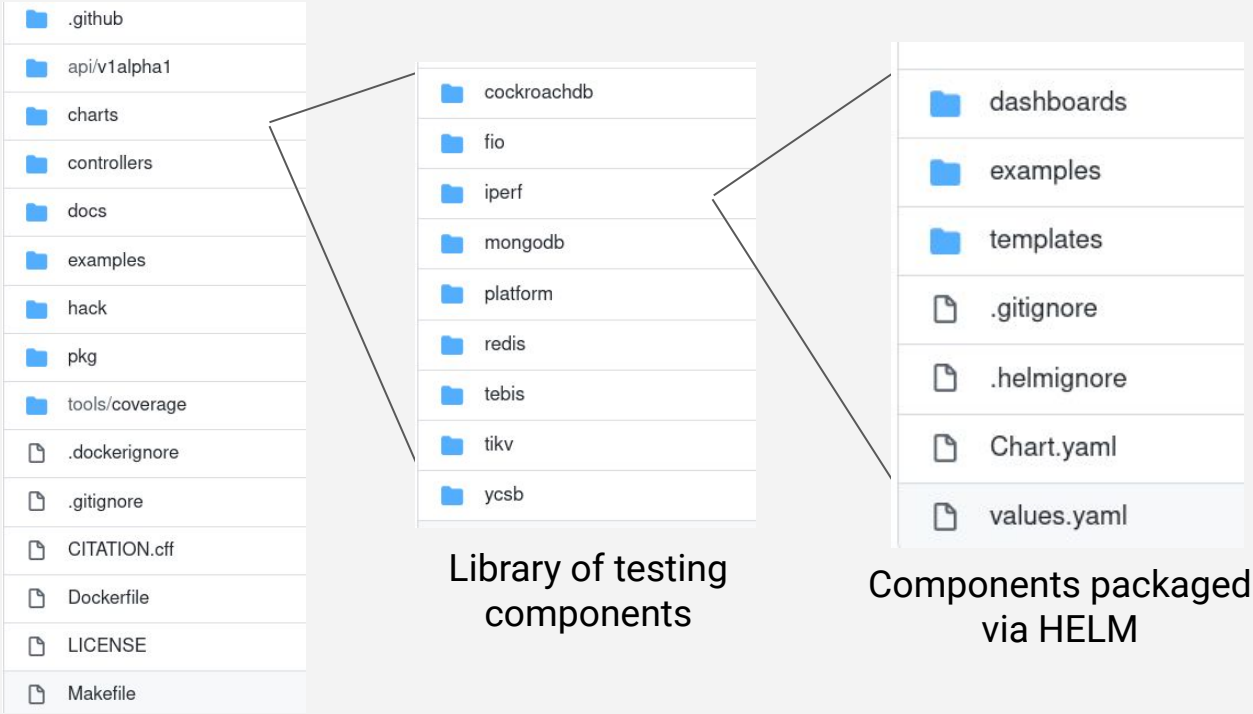
A Frisbee Template


```
spec:
 inputs:
1 parameters:
    server: localhost
    seconds: "60"
 service:
  decorators:
2   telemetry: [platform.telemetry.container, iperfmon.client]
  requirements:
3   persistentVolumeClaim:
     name: datastore
     spec: …
  containers:
  - name: app
    image: someimage
    volumeMounts:
4   - name: datastore
      mountPath: /store
    Command:...
     # … blah blah

5   server={{"{{.Inputs.Parameters.server}}"}}
    seconds={{"{{.Inputs.Parameters.seconds}}"}}

      iperf3 -c ${server} -t ${seconds}
```

Functionality

1. Declarate parameters
2. Re-use other templates
3. Deployment requirements
4. Automation are for free
5. Use inputs to manipulate the container

Project Structure



Library of testing components

Components packaged via HELM

Visualizations & Alerts

Usage

Test Factory

How to use Frisbee

TL;DR

- Quick Tutorial

1. Make sure that kubectl and Helm are installed on your system.
2. Update Helm repo.

```
>> helm repo add frisbee https://carv-ics-forth.github.io/frisbee/charts
```

3. Install Helm Packages.

```
# Install the platform
>> helm upgrade --install --wait my-frisbee frisbee/platform
# Install the package for monitoring YCSB output
>> helm upgrade --install --wait my-ycsb frisbee/ycsb
# Install TiKV store
>> helm upgrade --install --wait my-tikv frisbee/tikv
```

4. Create/Destroy the test plan.

```
# Create
>> curl -sSL https://raw.githubusercontent.com/CARV-ICS-FORTH/frisbee/main/charts/tikv/examples/

# Destroy
>> curl -sSL https://raw.githubusercontent.com/CARV-ICS-FORTH/frisbee/main/charts/tikv/examples/
```

TiKV

- Charts are self-descriptive

> TiKV provides both raw and ACID-compliant transactional key-value API, which is widely used in online serving services, such as the metadata storage system for object storage service, the storage system for recommendation systems, the online feature store, etc.

TL;DR

Install the platform and dependent charts.

```
>> helm repo add frisbee https://carv-ics-forth.github.io/frisbee/charts
>> helm install my-frisbee frisbee/platform
>> helm install my-ycsb frisbee/ycsbmon
>> helm install my-tikv frisbee/tikv
```

- With dependencies

Run any of the testing plans.

```
>> kubectl apply -f examples/plan.baseline.yml
```

- And usage examples

Summary

- Frisbee: A platform for Kubernetes-native Testing
  - ✓ Multi-node testing environment
  - ✓ Similar environment for dev, test, and production
  - ✓ Controllers run within Kubernetes cluster. Batteries-includes
  - ✓ Experiments written in YAML -> Write once / Run anywhere.
  - ✓ System Spinup -> Testing Actions -> System Validation

Looking for Collaborators

Frisbee

Devops

- Testing workflows
- Systems for testing

Developers

- Kubernetes Controllers
- Testing Resources

Researchers

- Many ideas floating around

https://github.com/CARV-ICS-FORTH/frisbee

fnikol@ics.forth.gr

THANKS

Do you have any questions?

fnikol@ics.forth.gr

FORTH, Crete, Greece

Acknowledgement:
This work is supported by the
European Commission within the scope of:

ETHER (H2020-MSCA-IF-2019)
Grant Agreement ID: 894204