INSTITUTE OF COMPUTER SCIENCE

Frisbee

Streamline non-functional testing on Kubernetes

Fotis Nikolaidis[1]
Antony Chazapis[1]
Manolis Marazakis[1]
Angelos Bilas[1]

[1]Foundation of Research and Technology Hellas (FORTH), Greece

MARIE CURIE ACTIONS

3 Ways to Test

Interactive, or manual testing

A human execute tests one-by-one, without test scripts.

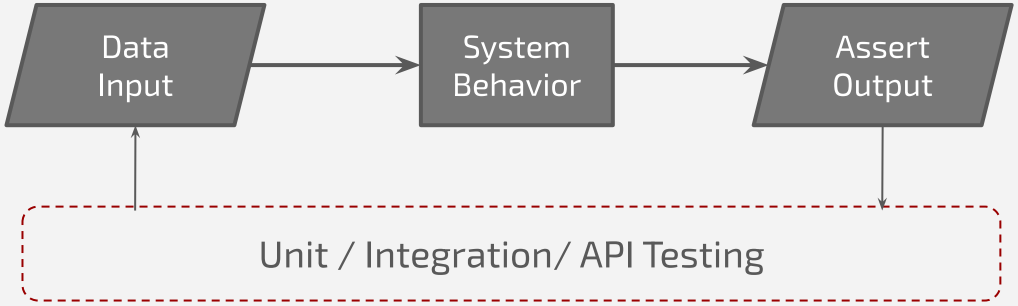
Automated testing

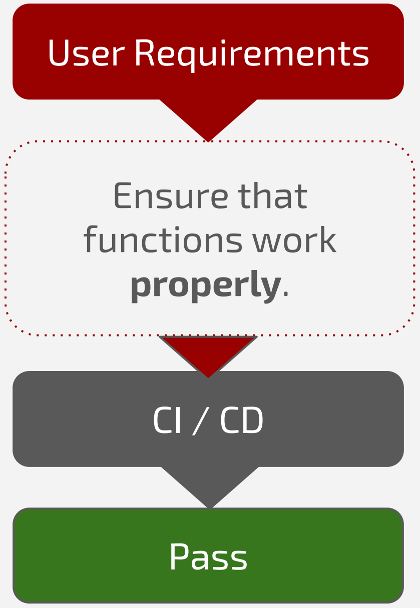
A framework executes test scripts written by a human.

Continuous testing

Applies the principles of automated testing in a scaled, continuous manner, to achieve the most reliable test coverage, at every stage of development lifecycle.

Functional Testing

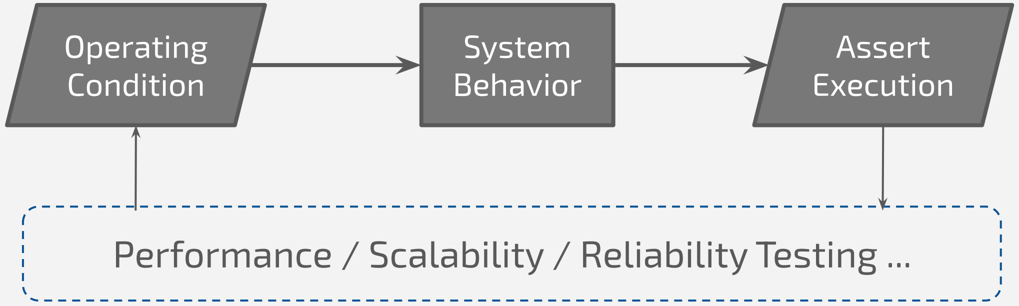




Today's solution … **Continuous Testing**

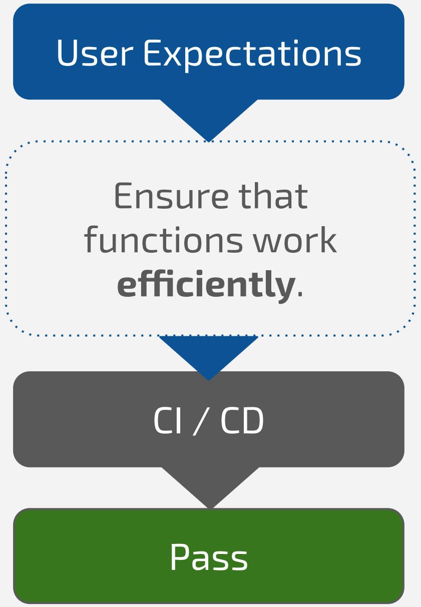
✓ Improved test efficiency
✓ Lower maintenance costs
✓ Minimal manual intervention
✓ Maximum test coverage
✓ Reusability of code
➢ **Early in the software delivery process**

Non-Functional Testing

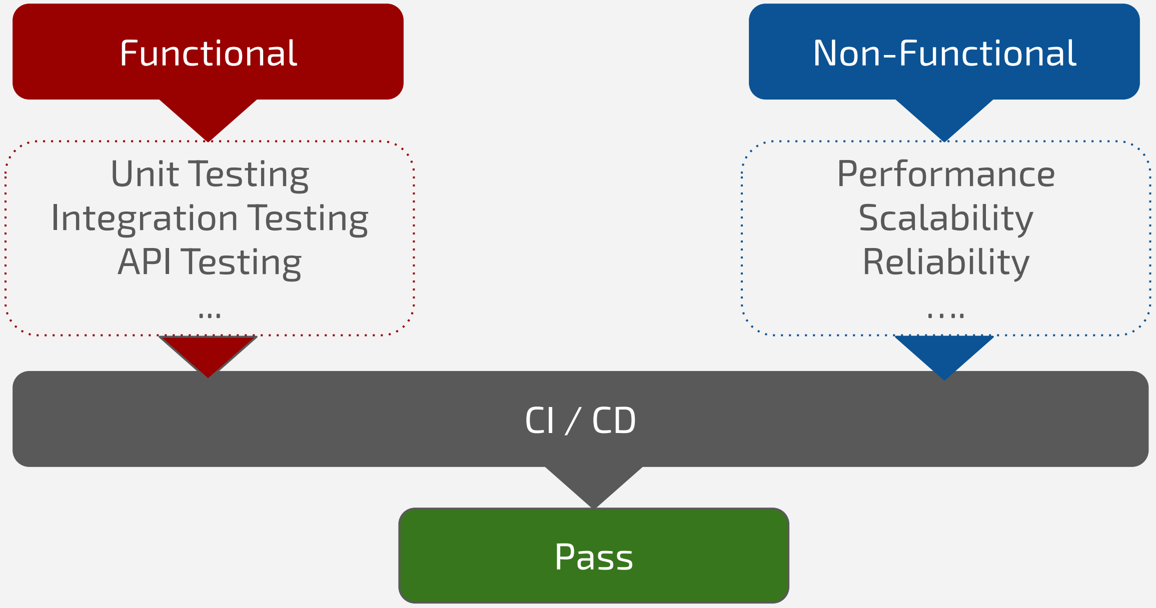


Today's solution ... **Manual testing**

- x Hardcoded APIs / dirs/ nodes /...
- x Biased Testing
- x Dependence on shared environments
- x Minimal support for fault injection
- x Manual collection of analysis results
- ➢ **Late in the software delivery process**



Streamline Testing



What Kubernetes does for testing ...

Kubernetes brings five critical things to testers:

- Cheap disposable and portable environments
- Unambiguous communication between testers and developers
- Seamless Integration with CI tools
- Experiments can scale from a desktop to hundreds of machines
- Direct access to distributed logs

➢ Kubernetes is great for running **unbiased non-functional testing**

… But writing tests is complex !

kubernetes

- Orchestrate workflows with logical dependencies.
- Get into a complicated failure state quickly (Chaos Engineering)
- Easily observe the global state of the SUT (system & app metrics)
- Define finite-horizon experiments (when has a test passed or failed ?).

➢ **Testers focus on the testing mechanism rather than the test case !!!**

Frisbee

Frisbee is a Kubernetes platform for exploring, testing, and benchmarking distributed applications.

Frisbee

YAML Language to describe testing workflows.

Write Tests

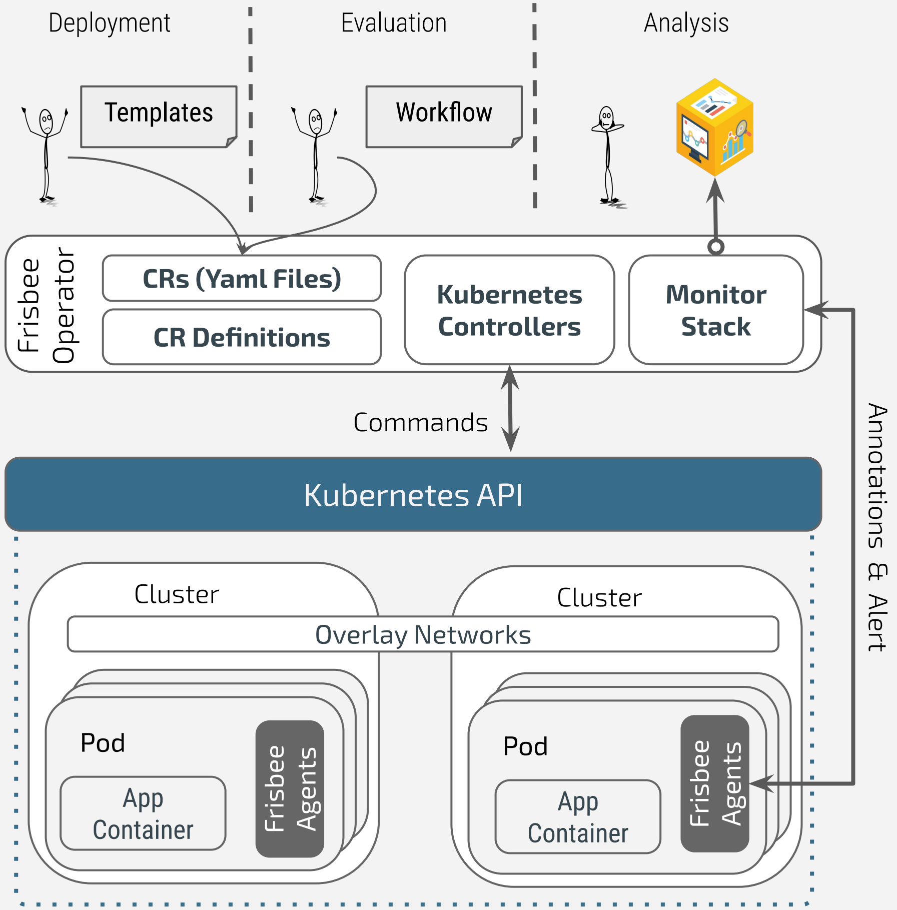
Run Tests

Scalable engine to automate the execution of test-cases

Integration with Real-time dashboards and alerting.

Debug Tests

Architecture



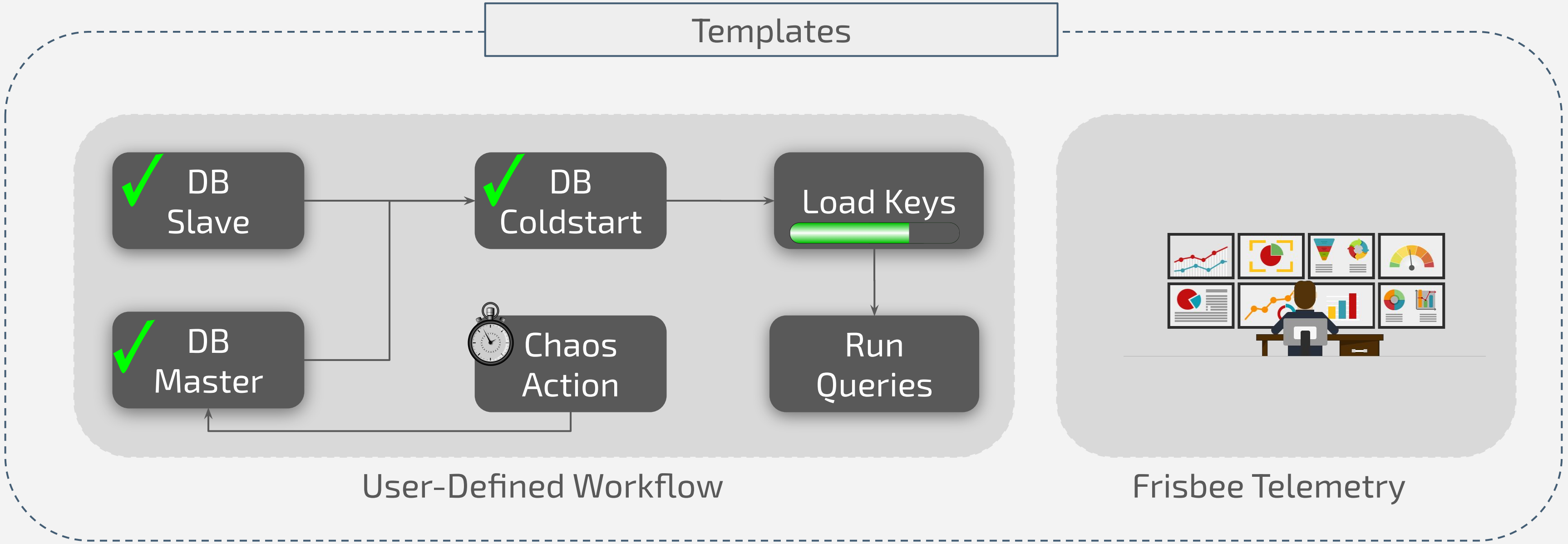
Templates: libraries of frequently-used specifications.

Workflow: list of actions that specify what will happen throughout the test.

Controllers: parse templates and run workflows.

Testing Workflow

Action	Description
Service:	Create an instance of a templated service.
Cluster:	Create multiple services that run in a shared context.
Chaos:	Inject failures to simulate abnormal behaviors.



Assert: Object State

Object State assertions checks the phase of an object.
Phase: a simple, high-level summary of where the object is in its lifecycle.

Pending: The object has been received by Kubernetes, but one or more of jobs has not been set up and made ready to run.

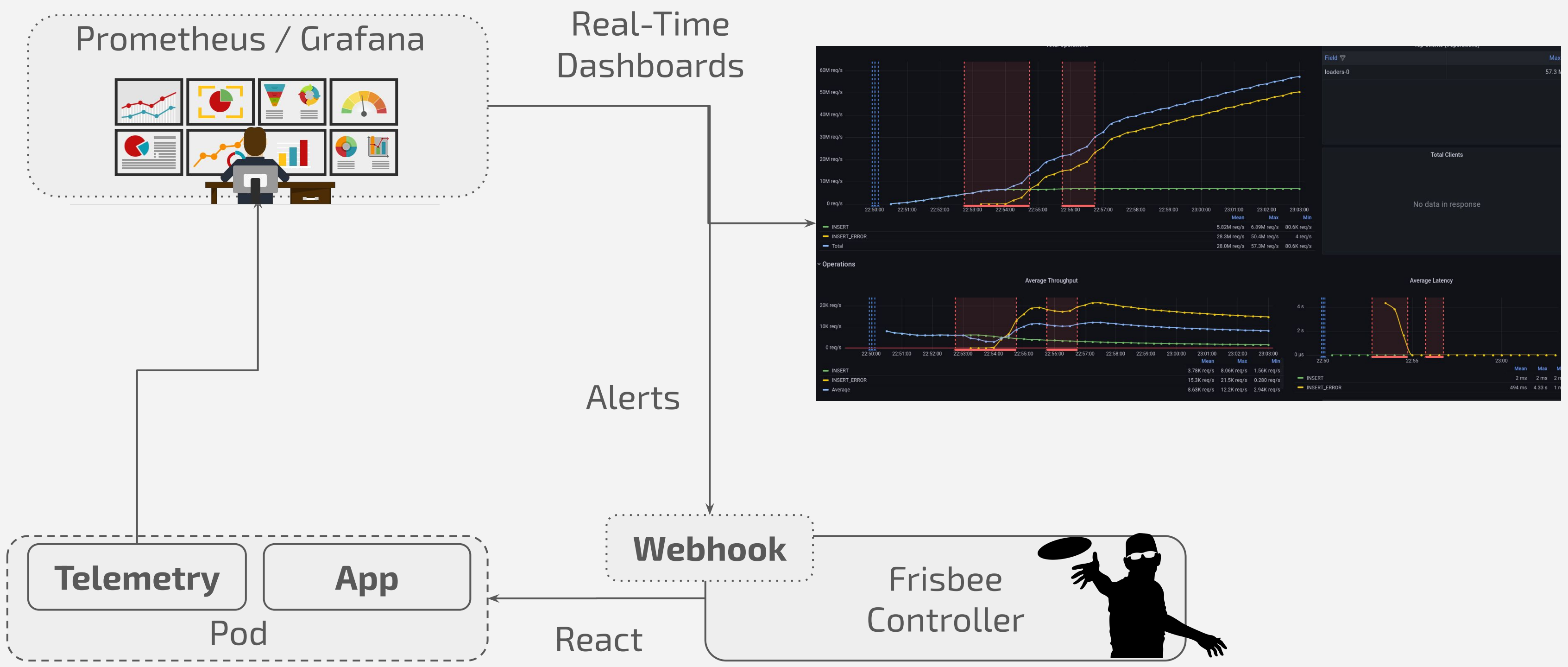
Running: All of the jobs in the object have been created. At least one job is still running, or is in the process of starting or restarting.

Success: All jobs have terminated in success, and will not be restarted.

Failed: All jobs have terminated, and at least one jobs has terminated in failure.

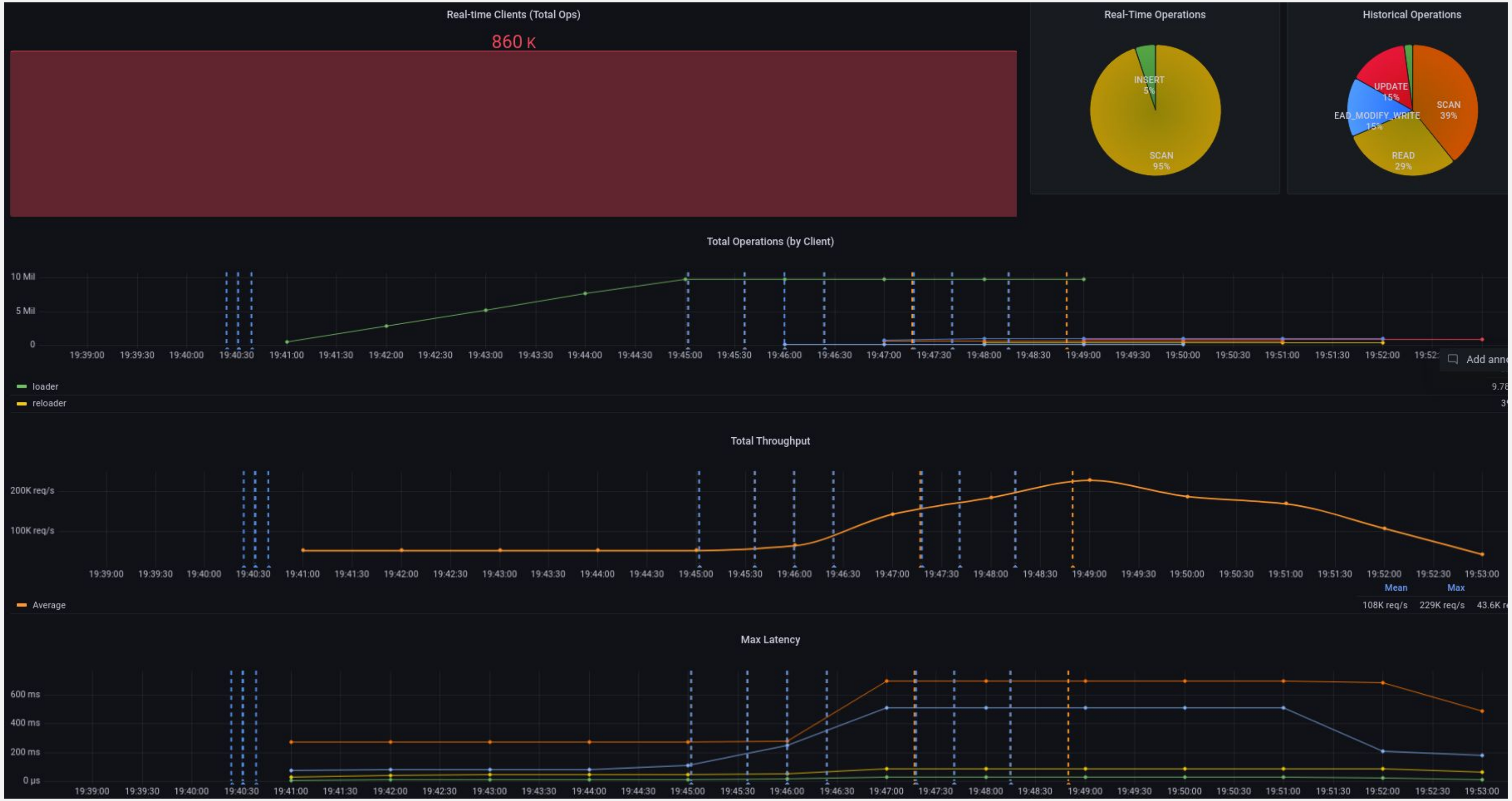
Assert: SLA

SLA assertions check whether KPI metrics are within expected limits.

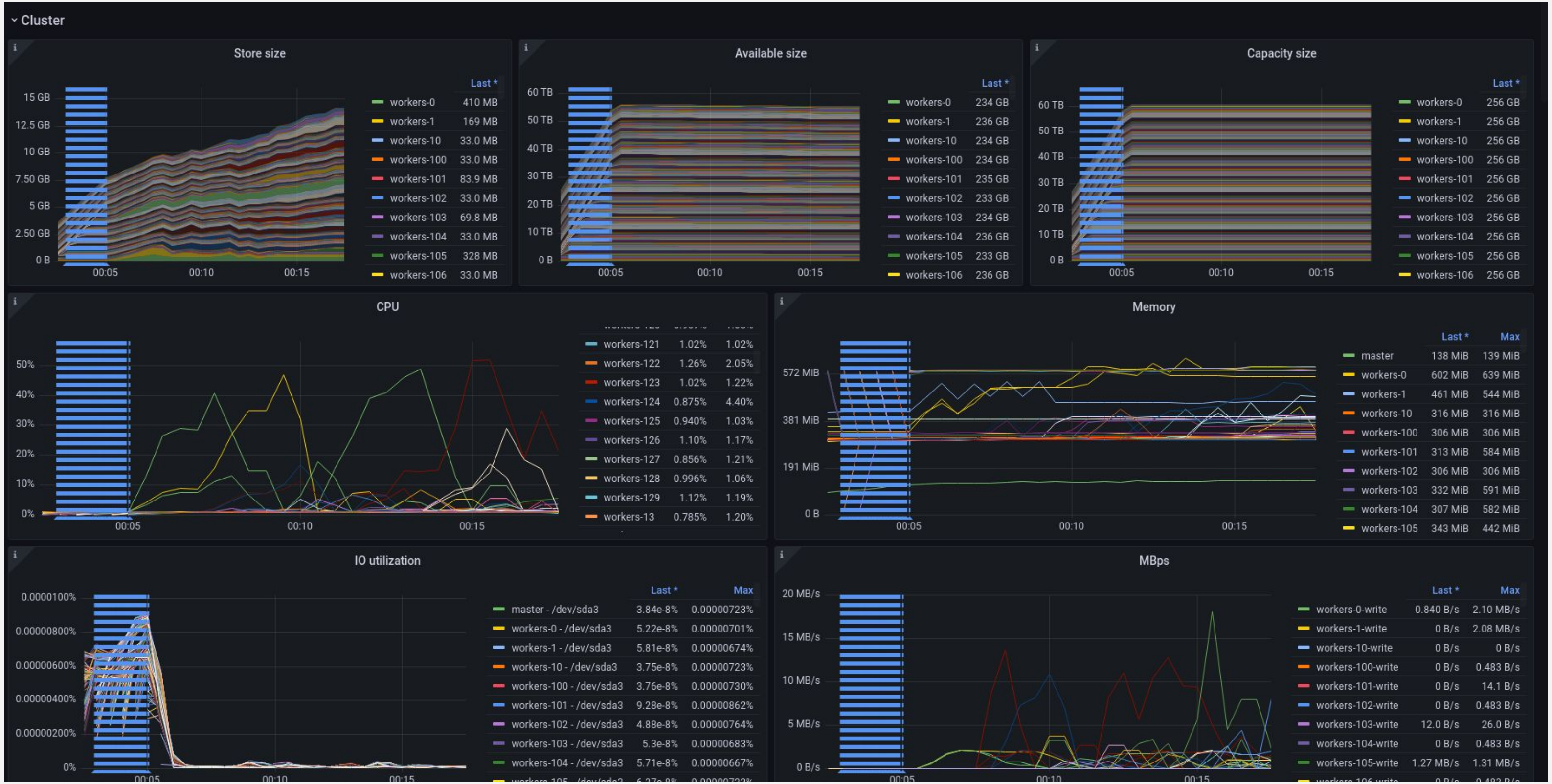


Testplans

Performance



Scalability



Store size

	Last *
workers-0	410 MB
workers-1	169 MB
workers-10	33.0 MB
workers-100	33.0 MB
workers-101	83.9 MB
workers-102	33.0 MB
workers-103	69.8 MB
workers-104	33.0 MB
workers-105	328 MB
workers-106	33.0 MB

Available size

	Last *
workers-0	234 GB
workers-1	236 GB
workers-10	234 GB
workers-100	234 GB
workers-101	235 GB
workers-102	233 GB
workers-103	234 GB
workers-104	236 GB
workers-105	233 GB
workers-106	236 GB

Capacity size

	Last *
workers-0	256 GB
workers-1	256 GB
workers-10	256 GB
workers-100	256 GB
workers-101	256 GB
workers-102	256 GB
workers-103	256 GB
workers-104	256 GB
workers-105	256 GB
workers-106	256 GB

CPU

workers-121	1.02%	1.02%
workers-122	1.26%	2.05%
workers-123	1.02%	1.22%
workers-124	0.875%	4.40%
workers-125	0.940%	1.03%
workers-126	1.10%	1.17%
workers-127	0.856%	1.21%
workers-128	0.996%	1.06%
workers-129	1.12%	1.19%
workers-13	0.785%	1.20%

Memory

	Last *	Max
master	138 MiB	139 MiB
workers-0	602 MiB	639 MiB
workers-1	461 MiB	544 MiB
workers-10	316 MiB	316 MiB
workers-100	306 MiB	306 MiB
workers-101	313 MiB	584 MiB
workers-102	306 MiB	306 MiB
workers-103	332 MiB	591 MiB
workers-104	307 MiB	582 MiB
workers-105	343 MiB	442 MiB

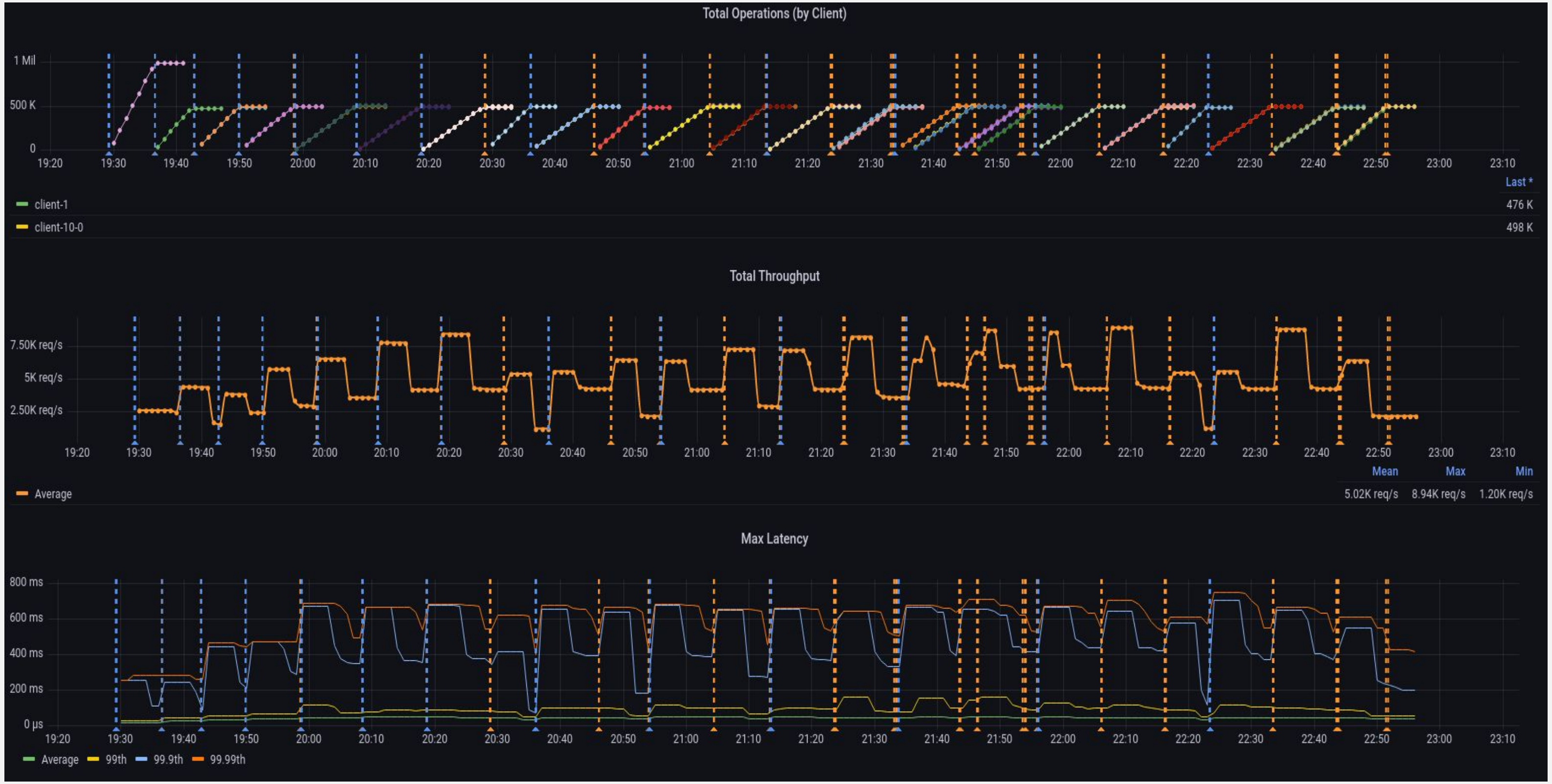
IO utilization

	Last *	Max
master - /dev/sda3	3.84e-8%	0.00000723%
workers-0 - /dev/sda3	5.22e-8%	0.00000701%
workers-1 - /dev/sda3	5.81e-8%	0.00000674%
workers-10 - /dev/sda3	3.75e-8%	0.00000723%
workers-100 - /dev/sda3	3.76e-8%	0.00000730%
workers-101 - /dev/sda3	9.28e-8%	0.00000862%
workers-102 - /dev/sda3	4.88e-8%	0.00000764%
workers-103 - /dev/sda3	5.3e-8%	0.00000683%
workers-104 - /dev/sda3	5.71e-8%	0.00000667%

MBps

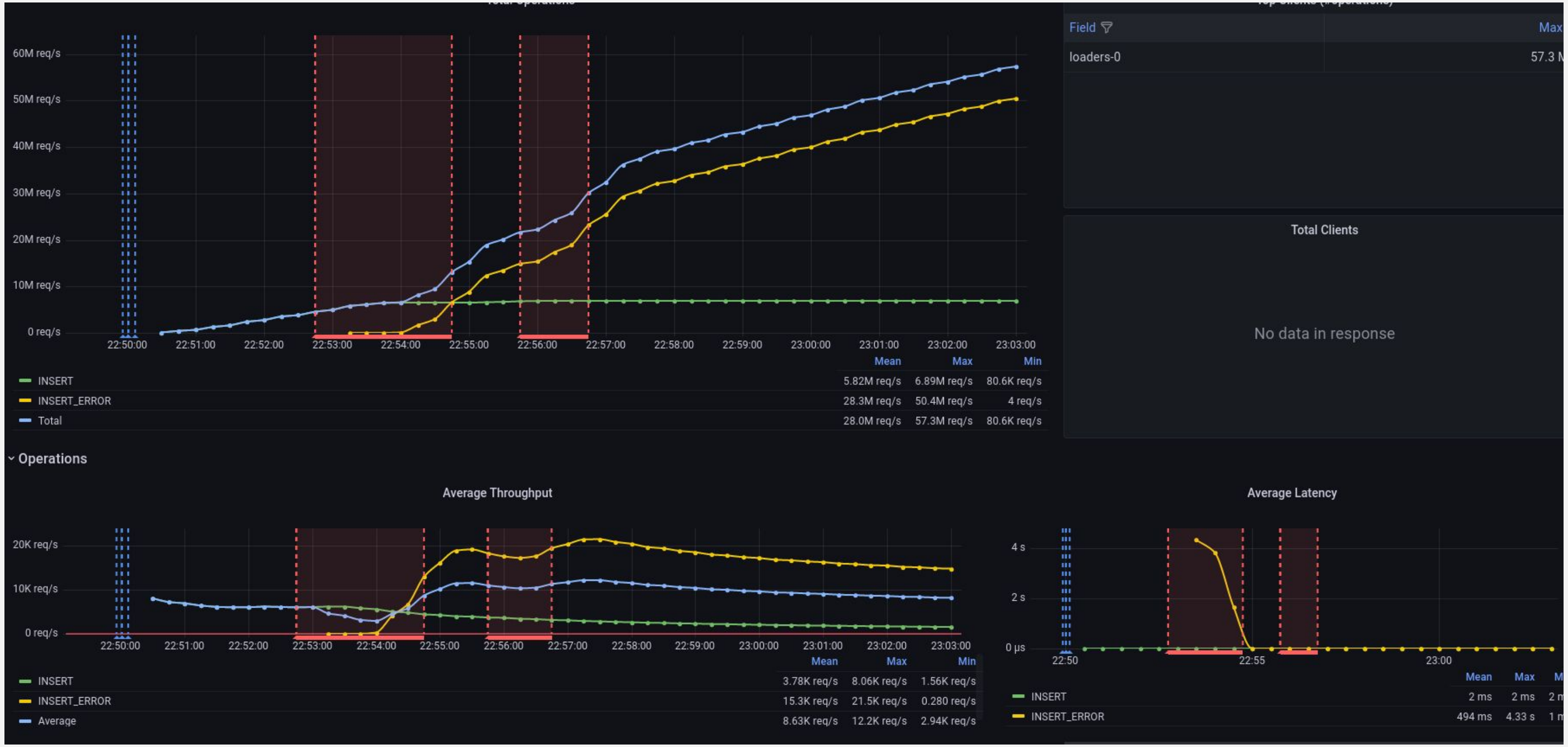
	Last *	Max
workers-0-write	0.840 B/s	2.10 MB/s
workers-1-write	0 B/s	2.08 MB/s
workers-10-write	0 B/s	0 B/s
workers-100-write	0 B/s	0.483 B/s
workers-101-write	0 B/s	14.1 B/s
workers-102-write	0 B/s	0.483 B/s
workers-103-write	12.0 B/s	26.0 B/s
workers-104-write	0 B/s	0.483 B/s
workers-105-write	1.27 MB/s	1.31 MB/s

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
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
Average
99th
99.9th
99.99th

Availability

Frisbee



Field	Max
loaders-0	57.3

Total Clients

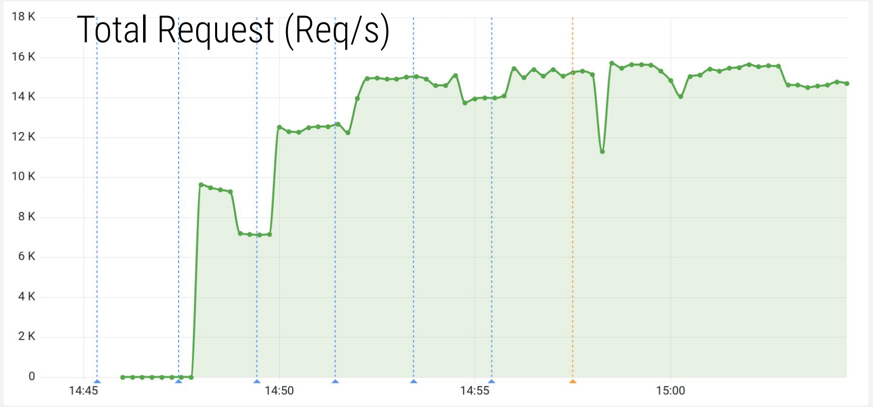
No data in response

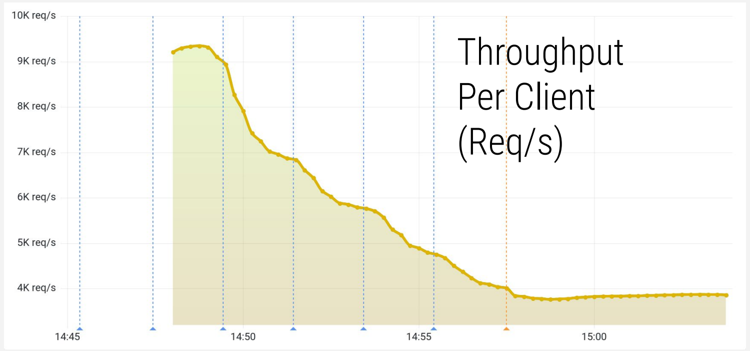
	Mean	Max	Min
INSERT	5.82M req/s	6.89M req/s	80.6K req/s
INSERT_ERROR	28.3M req/s	50.4M req/s	4 req/s
Total	28.0M req/s	57.3M req/s	80.6K req/s

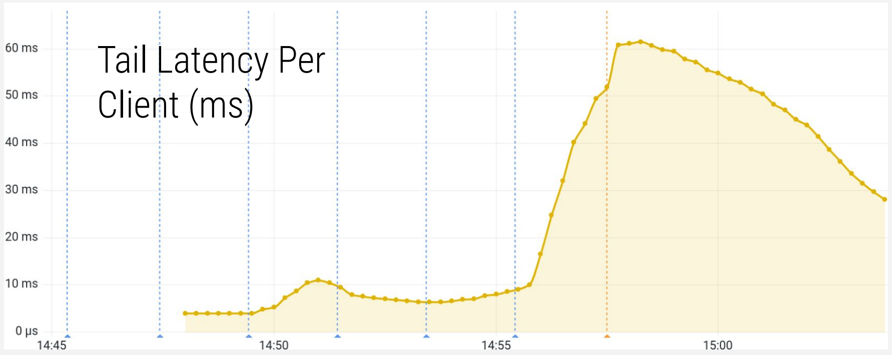
	Mean	Max	Min
INSERT	3.78K req/s	8.06K req/s	1.56K req/s
INSERT_ERROR	15.3K req/s	21.5K req/s	0.280 req/s
Average	8.63K req/s	12.2K req/s	2.94K req/s

	Mean	Max	M
INSERT	2 ms	2 ms	2 m
INSERT_ERROR	494 ms	4.33 s	1 m

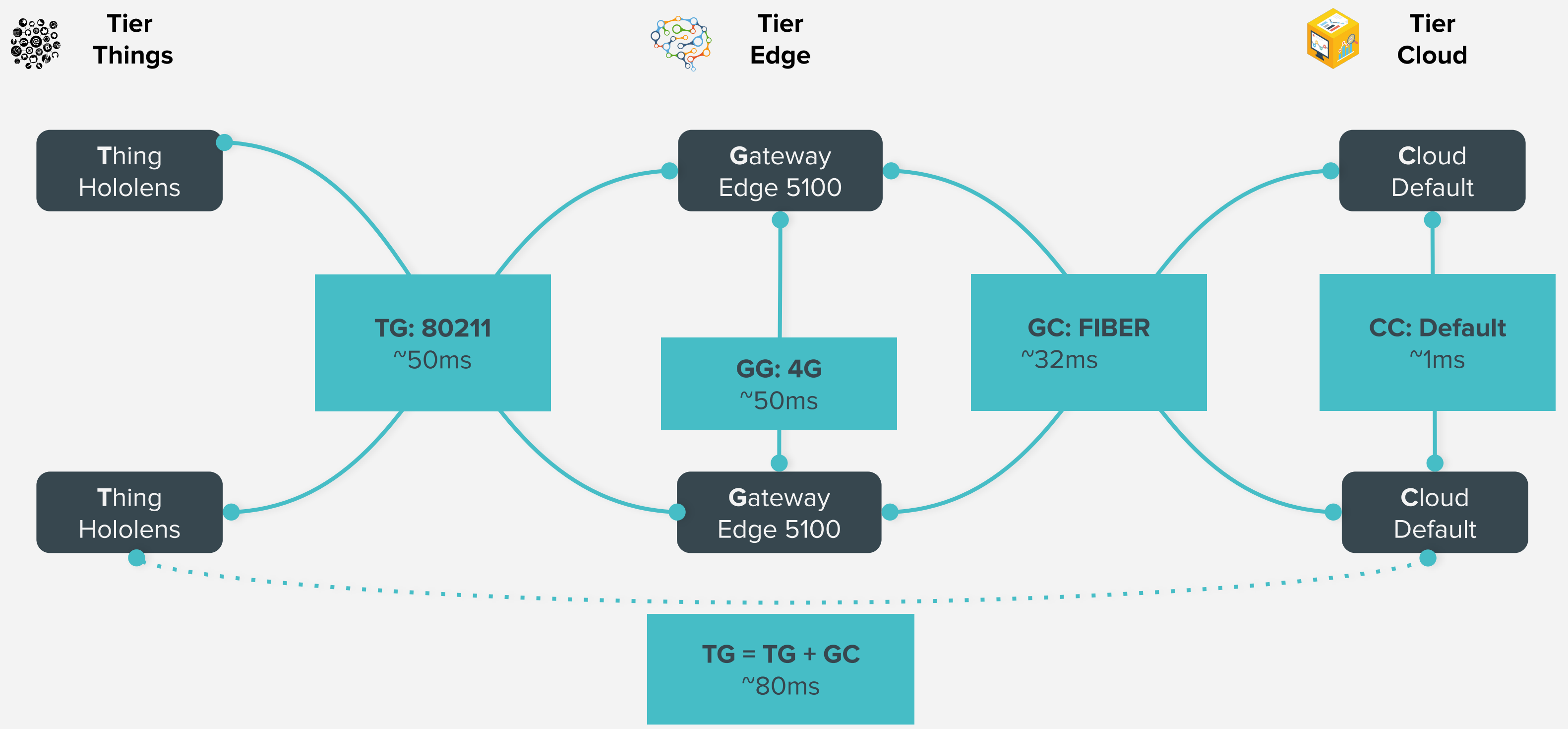
Saturation


Total Request (Req/s)
18 K
16 K
14 K
12 K
10 K
8 K
6 K
4 K
2 K
0
14:45
14:50
14:55
15:00


Throughput Per Client (Req/s)
10K req/s
9K req/s
8K req/s
7K req/s
6K req/s
5K req/s
4K req/s
14:45
14:50
14:55
15:00


Tail Latency Per Client (ms)
60 ms
50 ms
40 ms
30 ms
20 ms
10 ms
0 µs
14:45
14:50
14:55
15:00

Emulation of IoT Environments



Looking for Collaborators

Devops

- Testing workflows
- Systems for testing
- Tutorials

Developers

- Controllers
- Helm Installation

Researchers

- Many ideas floating around

Source available at
https://github.com/CARV-ICS-FORTH/frisbee

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

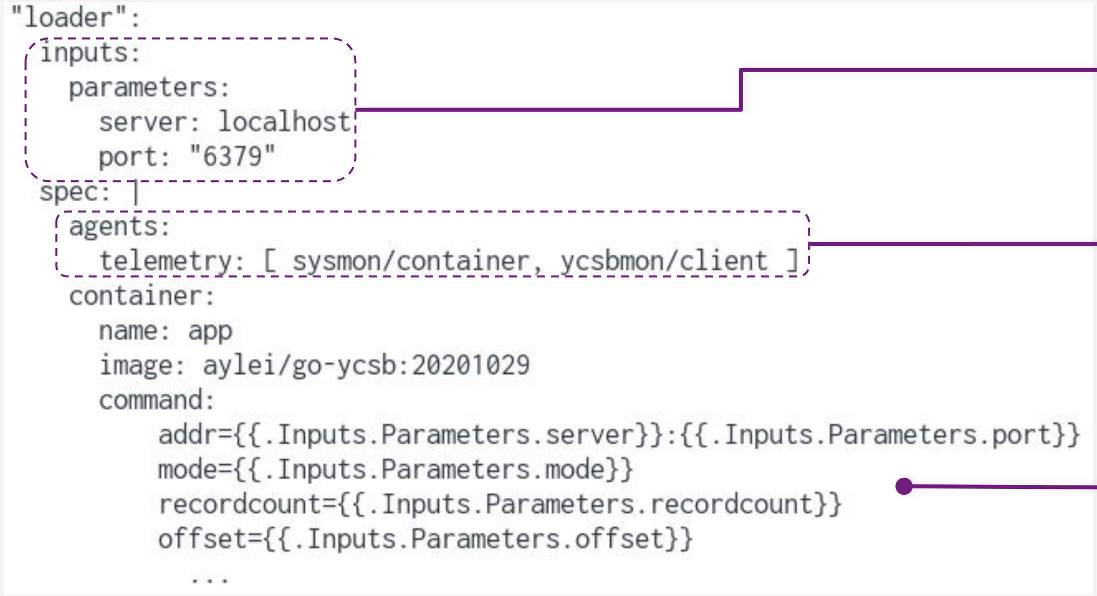
INSTITUTE OF COMPUTER SCIENCE

Backup Slides

Frisbee Primitives

Templates

Implement skeletons, exposing a bunch of parameters where dynamic data will be injected to create multiple variants of a specification.



```
"loader":
  inputs:
    parameters:
      server: localhost
      port: "6379"
  spec:
    agents:
      telemetry: [ sysmon/container, ycsbmon/client ]
    container:
      name: app
      image: aylei/go-ycsb:20201029
      command:
          addr={{.Inputs.Parameters.server}}:{{.Inputs.Parameters.port}}
          mode={{.Inputs.Parameters.mode}}
          recordcount={{.Inputs.Parameters.recordcount}}
          offset={{.Inputs.Parameters.offset}}
            ...
```

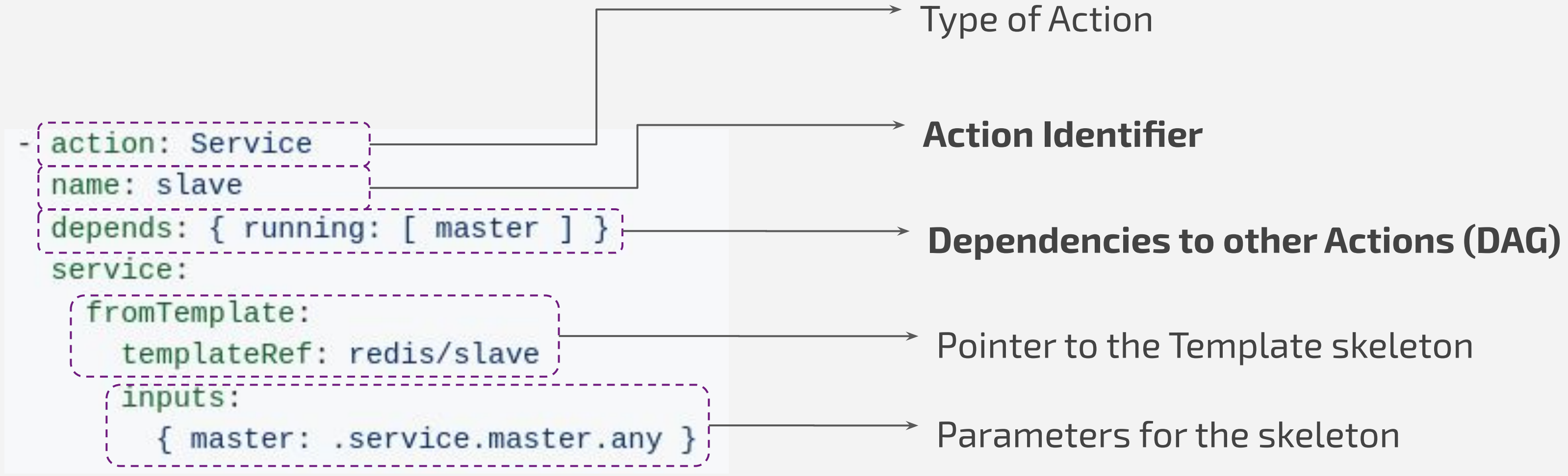
- Sane defaults
- Placeholders for user data

- Sidecar containers.
- Out-of-the-box monitoring

Embedded scripting:
- Run functions on Inputs

Service

Create an instance of a templated service.

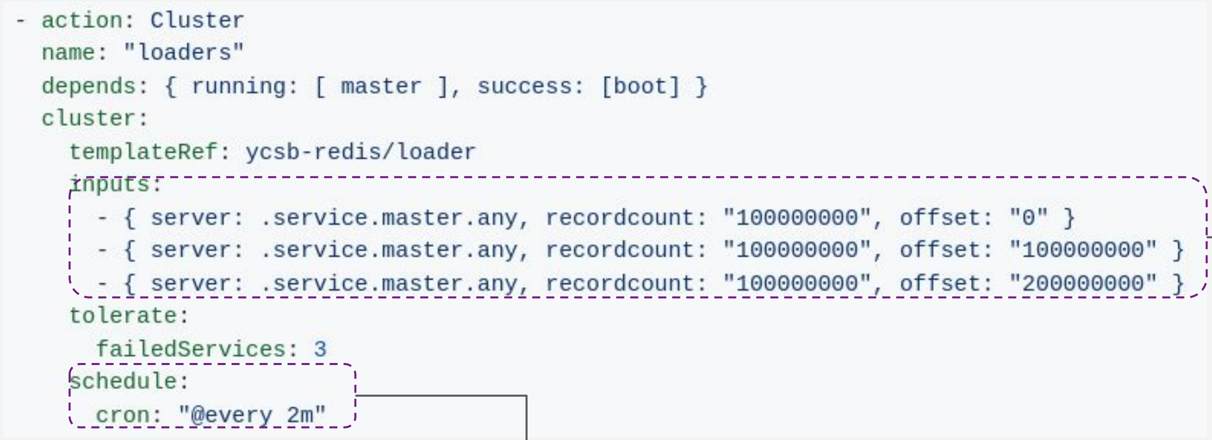


```
- action: Service
  name: slave
  depends: { running: [ master ] }
  service:
    fromTemplate:
      templateRef: redis/slave
      inputs:
        { master: .service.master.any }
```

Type of Action
Action Identifier
Dependencies to other Actions (DAG)
Pointer to the Template skeleton
Parameters for the skeleton

Cluster

Create multiple services that run in a shared context.



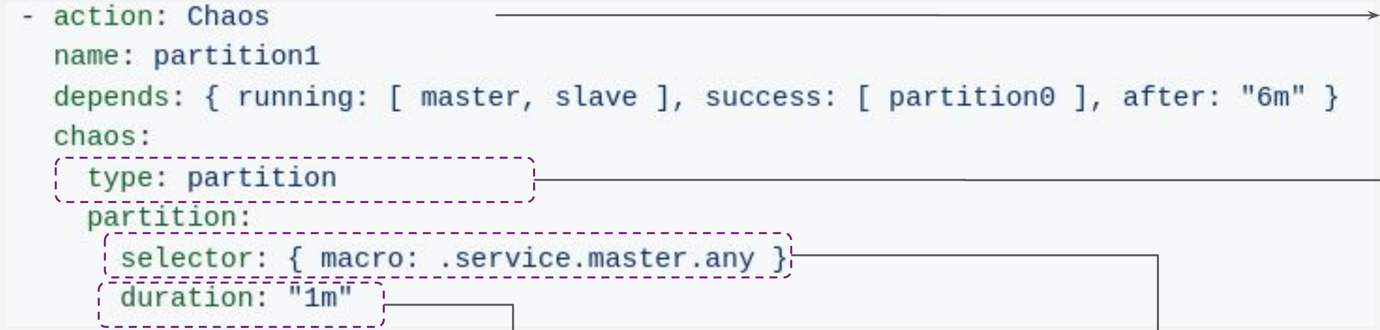
```yaml
- action: Cluster
  name: "loaders"
  depends: { running: [ master ], success: [boot] }
  cluster:
    templateRef: ycsb-redis/loader
    inputs:
      - { server: .service.master.any, recordcount: "100000000", offset: "0" }
      - { server: .service.master.any, recordcount: "100000000", offset: "100000000" }
      - { server: .service.master.any, recordcount: "100000000", offset: "200000000" }
    tolerate:
      failedServices: 3
    schedule:
      cron: "@every 2m"
```

Multiple services with Different parameters.

Interval between the creation of services. (see changing patterns)

Chaos

Inject failures to simulate abnormal behaviors.



```
- action: Chaos
  name: partition1
  depends: { running: [ master, slave ], success: [ partition0 ], after: "6m" }
  chaos:
    type: partition
    partition:
      selector: { macro: .service.master.any }
      duration: "1m"
```

Specify the fault.
Current: kill, partition

Select the Chaos target

Select the Chaos duration.
After this time, the injected fault will be retracted.