Fitness-aware Brokerage of Hosted Containerized Environments

- Pathikrit Ghosh
  (2014A7PS0105G)

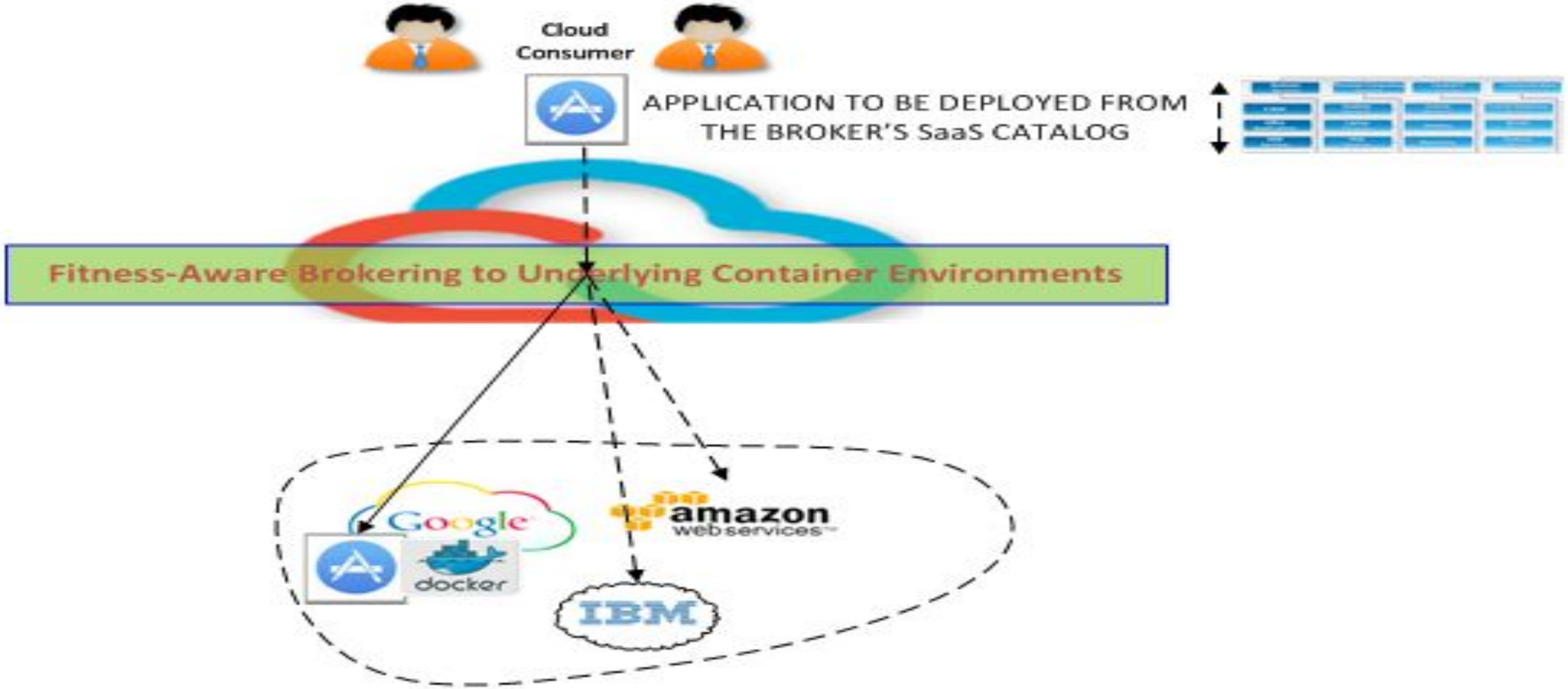
Background

- Containers are increasingly becoming vehicles for deploying production workloads. Google extensively uses containers for Internet services such as its search engine and gmail. Examples of other enterprise container users are Cisco, Makemytrip.com, and VMware.
- However, containerization of mission-critical workloads is not yet mainstream. Virtual Machines are often the environment of choice to host SLA'ed workloads. This is partly because the aspects of resiliency, reliability and dependability around container consumption is not mature enough.
- We propose a method to host SaaS applications in a cost effective manner by engineering their dynamic deployment on best-fit container hosting environments.

Problem Statement

- Internal IT departments of industries are moving to an Everything-as-a-Service (XaaS) model, wherein a cloud broker (or in general, a digital marketplace) maintains a portfolio of IT services.
- We see a need for a smart fulfillment engine within the cloud broker that automatically deploys a chosen catalogued IT service into a best-available-fit container out of a set of available container engines hosted on various supported underlying clouds.
- More broadly, the problem statement is to supply Best-fit Container Deployment as a Service to applications that are part of a broker's SaaS catalogue, in an on-demand, pay-as-you-go manner.

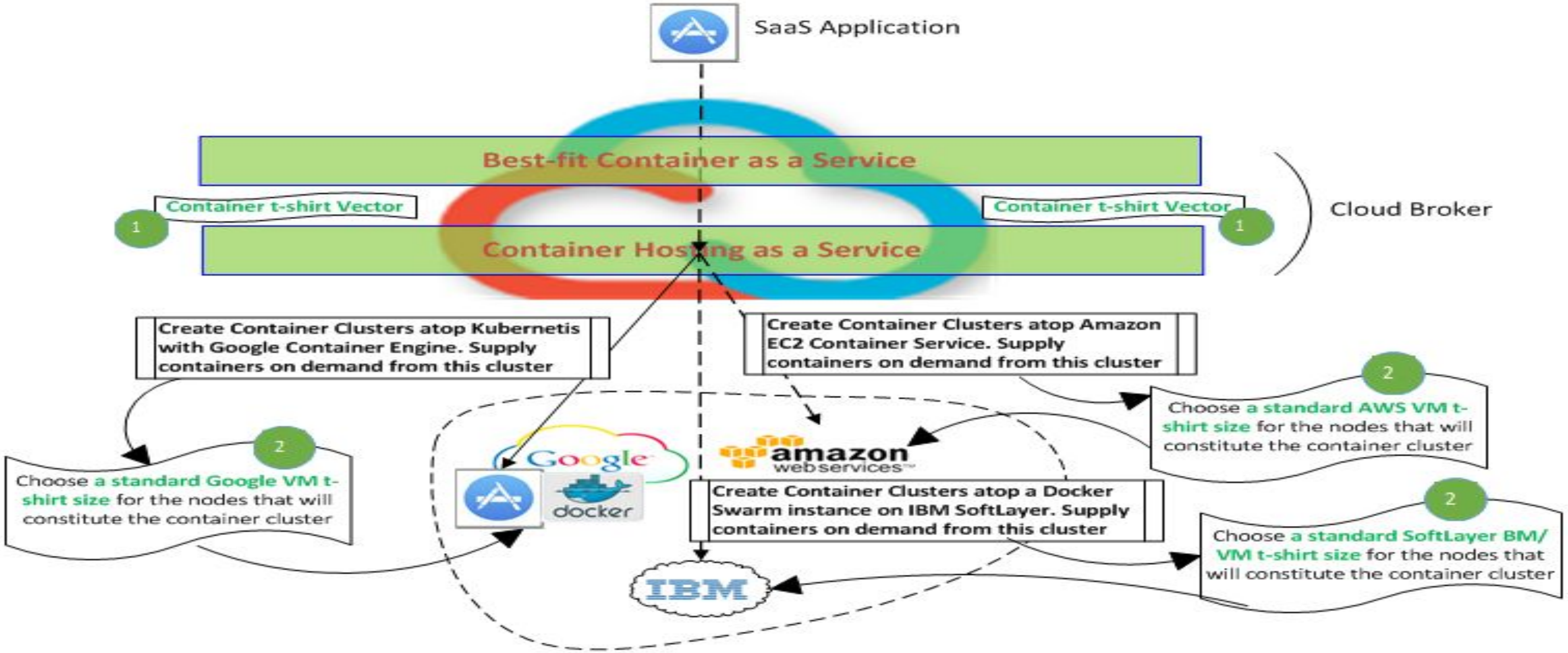
Overview


Cloud Consumer
APPLICATION TO BE DEPLOYED FROM THE BROKER'S SaaS CATALOG
Fitness-Aware Brokering to Underlying Container Environments
Google
docker
amazon web services
IBM

Methodology

- Expose the designed offering as part of a SaaS catalogue in a digital marketplace such as that of a cloud broker
- Implement an ‘integration agent’ (called Best-Fit-Container-as-a Service or BCaaS) that programmatically containerised the catalogue element and places it under the best-available-fit container engine on an underlying cloud such as:
    - Kubernetes on the Google Container Engine
    - EC2 Container Services on the Amazon Cloud
    - Docker Swarm Engine on the IBM SoftLayer Cloud

Best Fit Container as a Service (1 / 4)


SaaS Application
Best-fit Container as a Service
Container t-shirt Vector
1
Container t-shirt Vector
1
Cloud Broker
Container Hosting as a Service
Create Container Clusters atop Kubernetis with Google Container Engine. Supply containers on demand from this cluster
Create Container Clusters atop Amazon EC2 Container Service. Supply containers on demand from this cluster
2
Choose a standard AWS VM t-shirt size for the nodes that will constitute the container cluster
2
Choose a standard Google VM t-shirt size for the nodes that will constitute the container cluster
Google
docker
amazon webservices
Create Container Clusters atop a Docker Swarm instance on IBM SoftLayer. Supply containers on demand from this cluster
2
Choose a standard SoftLayer BM/ VM t-shirt size for the nodes that will constitute the container cluster
IBM

Best Fit Container as a Service (2 / 4)

The cloud broker, in the context of this project, has two parts:

- The Container-Hosting-as-a-Service Provider: This service creates, maintains and capacity-manages container clusters on supported underlying clouds.
  - It creates, for example, (one or more) Kubernetes clusters on the Google Container Engine, EC2 container clusters on the AWS cloud and private Docker Swarm enabled on locally hosted servers.
  - It then creates a set of comparable container cluster sets across clouds by appropriately choosing supported VM/bare-metal node sizes on the underlying cloud.
  - For example, a comparable container cluster set could be built out of m1.small VMs on the ECS engine hosted by the Amazon Cloud, g1-small VMs on the Kubernetes engine hosted by the Google Cloud. The constituency of this set (e.g., {m1.small, g1-small, 1x2Ghz}), can be statically determined either experimentally or via publicly available documentation.

Best Fit Container as a Service (4 / 4)

Mathematically, this set of comparable container cluster node set S can be represented as:

$$S[i] = \{x_j \in P_j \Lambda H(x_j) = K_i\}$$

i = 1 to L, where L is the number of types of comparable container cluster, supported by the Container-Hosting-as-Service Providers.
j = 1 to N, where N is the number of participating public cloud providers.
P_j = Set of VM/BM computer T-shirt sizes published by cloud provider j
X_j = Element of P
K_i = Constant computing power in units of the power independent benchmark
$H(x_j)$ = Provider independent benchmark of the VM/BM computing power specified by X_j.
H() has to be determined by one-time experiment through public documentation.

The costs of each element in S[i] are hence comparable.

Best Fit Container as a Service (3 / 4)

- The Best-Fit-Container-as-a-Service Provider: As described, the Container-Hosting-as-a-Service maintains S[i]. The value of i (the identity of the best comparable cluster set to containerize a specific SaaS application) is determined by a parametrized Non Functional Requirement (NFR) of the application that is sought to be containerized. For example, for an e-commerce SaaS, it could be the number of simultaneous users of the web service.
- The problem confronting the Best Fit-Container-as-a-Service Provider is this: Given i as described above, which element in the S[i] set corresponds to the best fit cluster to host the chosen containerized SaaS application? Even though the constituent nodes of the clusters in the S[i] set are 'comparable', the runtime fitness of hosted containers can differ, given that containers are user space applications executing on the VM/BM nodes that constitute a container cluster.
- The container t-shirt-size vector (see definition of struct t-shirt-vector) is an abstraction in this layer (layer labeled '1' in Figure 2). The compute-units-vCPUs is the compute field in this vector. Some container hosting services support this abstraction, while others don't. For example, in ECS, a container can specify a number between 1 and 1000, where 1000 is the full compute power of a node of the container cluster. In general, there is no comparable fine-grained control at this level.

Algorithmic FLow

- An asynchronous thread that gets periodically scheduled to collect the ‘fitness quotient’ of containers of various ‘t-shirt sizes’ belonging to all supported container hosting environments. Note that unlike fitness of bare metal servers that is stateful (a broker can recognize a server that it has seen before using unique IDs), the fitness of a container is stateless.
- A synchronous thread for run-time deployment of a SaaS application - the best-fit container service is procured based on the available fitness quotients across clouds of the container “closest” in t-shirt size to the request in question.
- A cognitive feedback thread to finetune four key parameters in our method using an ‘exponential back-off and slow fine-tune’ approach.

Minimize cost of algorithm

Cost: The algorithm relies on periodically collecting the ‘fitness quotient’ of various t-shirt-sizes of all supported container engines. This is done ‘out of band’ and incurs a cost of purchasing the containers under test. Moreover, this is a continuous cost since stale data has to be purged to maintain currency. To minimize this cost, we:

i. Take only ‘sufficient-samples’ that are chosen randomly. This is the # of t-shirt sizes in each supported hosted container whose fitness quotient is maintained by our algorithm
ii. Cognitively adjust the value of ‘sufficient-samples’
iii. Correspondingly also adjust ‘nearness-threshold’, which is the permissible maximum distance between the requested t-shirt-size and the nearest t-shirt-size whose fitness quotient is available
iv. Cognitively adjust the ‘epoch-of-freshness’, which is the width of the time window beyond which the fitness values are considered stale
v. Cognitively adjust the ‘percentage-real-time-samples’, which is the percentage of SaaS deployment requests for which fitness quotient is calculated in real time

Fitness Quotient of a Hosted Container Service (1 / 2)

We introduce Fitness quotient (FQ) of a container service as a metric that depends on the following:

1. Speed Quotient (SP)
2. Scalability Quotient (SC) and
3. Stability Quotient (ST)

Let $\Omega_{SP,}\ \Omega_{SC}$ and Ω_{ST} be the fractions that specify the relative importance of SP, ST and SC, respectively, in determining container fitment.

$$\Omega_{SP} + \Omega_{SC} + \Omega_{ST} = 1 \qquad \text{-----}\ [2]$$

These constants are user-definable, but we choose their default values as 0.7, 0.2 and 0.1 respectively, reflecting empirically preferred requirements.

We define Fitness Quotient as:

$$FQ = SP_{(0 < SP < \Omega SP)} + SC_{(0 < SC < \Omega SC)} + ST_{(0 < ST < \Omega ST)} \qquad \text{-----}\ [3]$$

Fitness Quotient of a Hosted Container Service (2 / 2)

- **Speed Quotient (SP)** is calculated in Millions of Whetstone Instructions Per Second (MWIPS), truncated at 10,000 MWIPS, and normalized between 0 and Ω_{SP}. To do this, the open source Whetstone performance benchmark code is containerized and executed in all participating container engines

- **Scalability Quotient (SC)** is calculated as the average millisecond response time to fulfil a scaling request normalized between 0 and Ω_{SC}. The response time is the averaged total in seconds (truncated at 1000 seconds) of:
  - Response time of the scaling API supported on the hosted container engine, and
  - Startup time of a newly spawned container

- **Stability Quotient (ST)** is calculated as the current hourly uptime of a containerized application normalized between 0 and Ω_{ST}. For cost reasons, this is executed only on a single container-size in a given container engine. This container executes forever and sends keep-alive messages to the polling thread in units of uptime in hours (capped at 1000 hours). This code is containerized and executed in all participating container engines.

Equation 3 can now be rewritten as:

$$FQ = SP*\Omega_{SP}/10000 + SC*\Omega_{SC}/1000 + ST*\Omega_{ST}/1000 \quad \text{-----} \ [4]$$

Data Collection Epoch

- The algorithm will periodically flush stale fitness data that lies beyond an ‘epoch of freshness’. See invocation of degauss-stale-entries().
- The algorithm will periodically populate fitness data until there are sufficient samples in the current epoch. See poll-for-fitness-quotient().
- Fitness quotients are determined for randomly chosen t-shirt sizes.

T - shirt size of a container

- We generically define the t-shirt specification supported by a container engine as the following vector:

```
struct t-shirt-vector
{
        Region-Zone            /* Data Center ID */
        Compute-units-vCPUs    /* Compute Spec */
        Memory-in-GBs          /* Memory Spec */
        Disk-type              /* Ephemeral or Persistent */
        Cluster-size           /* Number of VM nodes that make up the cluster */
        Network                /* VLAN that the container cluster is in */
};
```

Algorithm to compute fitness quotient (1 / 3)

```
/* Invoked Periodically from the fitness-aware broker engine*/
void poll-for-fitness-quotient ()
{
 struct t-shirt-vector t-shirt-vector-instance;

 if (epoch-of-freshness does not contain sufficient-samples)  {
     /* Generate random t-shirt vector within range. Only first 3 fields are varied */
  /* Multiply Compute-Units-vCPU by 1000 for ECS. Need to adjust by trial and error! */
  t-shirt-vector-instance = generate-random-t-shirt-vector (supported-t-shirt-range);

  for each supported cloud (cloud-id) under the digital marketplace do {
    /* Store determined FQs */
    fitness-quotient [cloud-id][t-shirt-vector-instance] =
                                   speed-test (cloud-id, t-shirt-vector-instance) +
                                   scale-test (cloud-id, t-shirt-vector-instance) +
                 stable-test (cloud-id);
                 }
          }
          degauss-stale-entries (fitness-quotient[][]);
}
```

Algorithm to compute fitness quotient (2 / 3)

```
/* speed-test to return speed quotient: Invoked from poll-for-fitness-quotient */
int speed-test (int cloud-id, struct t-shirt-vector container-size)
{
   /* Containerize and run the Whetstone Benchmark.  Get the speed quotient in MWIPS (Millions of Whetstone Inst per Second */

     speed-q = containerize-benchmark-code-and-deploy (cloud-id, container-size);

      /* Normalize the speed quotient to between 0 and $\Omega_{SP}$ */

     normalized-speed-q = speed-q* $\Omega_{SP}$ /10000;

     return (normalized-speed-q);
}
```

Algorithm to compute fitness quotient (3 / 3)

```
/* scale-test to return scale quotient: Invoked from poll-for-fitness-quotient */
int scale-test (int cloud-id, struct t-shirt-vector container-size)
{
  /* Code to test scale for this t-shirt-sized container:
     This code will measure (in milliseconds) and add
     (1) Responsiveness of the scaling API and (2) Container startup time
     Truncate at 1000 seconds */
     scale-q = containerize-scale-test-code-and-deploy (cloud-id, container-size);
     normalized-scale-q = scale-q* $\Omega_{SC}$ /1000;
     return (normalized-scale-q);
}

/* stable-test to return stable quotient: Invoked from poll-for-fitness-quotient */
int stable-test (int cloud-id)
{
  /* Code to test stability for this t-shirt-sized container:
     Containerize & Deploy only if it is not already running.
     This code will return its uptime in hours. Don't increment beyond 1000 */
     stable-q = containerize-stable-test-code-and-deploy (cloud-id);
     normalized-stable-q = stable-q* $\Omega_{ST}$ /1000;
     return (normalized-stable-q);
}
```

Algorithm: Synchronous thread

```
/* Integration code Invoked by the broker to realize a SaaS Deployment */
cloud-id find-best-fit-container (struct t-shirt-vector container-size)
{
   /* Get ‘closest’ t-shirt size present in the fitness-quotient table
      in terms of Euclidean distance */
   closest-t-shirt-size = get-nearest-t-shirt (container-size);

   /* If there are no close enough samples, return error */
   /* Nearness threshold is fine-tuned via feedback loop */
   if (closes-t-shirt-size > nearness-threshold) return (ERROR);

   cloud-id = find-best-fit-container-engine (closest-t-shirt-size);

   /* Increase window size of the epoch-of-freshness and the nearness-threshold
      Also see the cognitive feedback loop
      Do NOT change if value is outside of defined threshold */
   epoch-of-freshness++;  nearness-threshold++;
   sufficient-samples--; percentage-real-time-samples--;

   cognitive-backoff-and-finetune (cloud-id, container-size);
   return (cloud-id);
}
```

Cognitive Feedback Loop

1. False Positive: The placement algorithm wrongly recommends a container hosting system as the best fit for the job. False Positive Probability (FPP) is the probability of a False Positive event.
2. False Negative: The placement algorithm wrongly determines that a container hosting system is not the best fit for the job. False Negative Probability (FNP) is the probability of a false negative event.

$$\text{FPP}[C_i] = \sum_{j=1 \text{ to } N,\ j <> i} \text{FNP}[C_j]$$

For a predetermined percentage of customers (say, 5%), the fitness engine performs the fitness test in real time on the same t-shirt size as the just received SaaS deployment request. If the observed fitness does not match the calculated fitness, the latter has resulted in a false positive and at least one false negative. In this scenario, the feedback thread exponentially backs off & slowly recovers:

i. Halves epoch-of-freshness and nearness-threshold and
ii. Doubles sufficient-samples and percentage-real-time-samples

It also calculates the confidence level of the proposed “Best-fit Container Deployment as a Service” as the ratio of the number of times when observed and calculated fitness match out of the total number of observations.

Cognitive Feedback Loop : Thresholds

The following parameters are tuned cognitively by our algorithm. However, we set user-definable minimum and maximum values for the control knob. These were our default settings:

Metric	What?	Min Threshold	Max Threshold	Initial Value
sufficient-samples	# of t-shirt sizes in each supported hosted container whose fitness quotient is maintained	10	30	15
nearness-threshold	the permissible maximum distance between the requested t-shirt-size and the nearest t-shirt-size whose fitness quotient is available	0 units	2 units	2 units
epoch-of-freshness	the width of the time window beyond which the fitness values are considered stale	5 days	15 days	5 days
percentage-real-time-samples	the percentage of SaaS deployment requests for which fitness quotient is calculated in real time	5%	10%	5%

Cognitive Backoff Algorithm

```
cognitive-backoff-and-finetune (int calc-cloud-id, struct t-shirt-vector container-size)
{
   /* We will do this only for 5% (configurable #) of requests */
   if (requests++ % 100/percentage-real-time-samples) return;

   /* Do fitness test in real time for the exact container-size requested */
   for each supported cloud (cloud-id) under the digital marketplace do {
          fitness[cloud-id] =        speed-test (cloud-id, container-size) +
                                     scale-test (cloud-id, container-size) +
                         stable-test (cloud-id);
   }

   /* Find the best-fit cloud – the index of the largest fitness[] element */
   cloud-id = index-of-largest-element (fitness);

   /* Does this match calculated fitness from the fitness-quotient table? */
   if (cloud-id != calc-cloud-id) {
          sensitivity-recorder (0); /* Data for sensitivity analysis of ‘cognitive’ params*/
          /* Multiplicatively decrease & gradually increase to find acceptable value
            Do NOT change if current value is outside defined lower/upper thresholds */
          nearness-threshold /= 2; sufficient-samples *= 2;
     epoch-of-freshness /= 2; percentage-real-time-sample *= 2;
   } else sensitivity-recorder (1);
}
```

Sensitivity measures

- We generate a report of the success rate of various combinations of the feedback metrics over time, by performing analytics on the success-rate[] data structure.

```
/* scale-test to return scale quotient: Invoked from poll-for-fitness-quotient */
int sensitivity-recorder (int is-success)
{
   int timestamp = current-time-in-seconds;
   success-rate[i++].sufficient-samples = sufficient-samples;
   success-rate[i].nearness-threshold = nearness-threshold;
   success-rate[i].epoch-of-freshness = epoch-of-freshness;
   success-rate[i].percentage-real-time-samples = percentage-real-time-samples;
   success-rate[i].timestamp = current-time-in-seconds;
   success-rate[i].success = is-success;
}
```

References

We have not encountered direct intersection of any prior art with our proposal, but the following online articles relate to some aspects referred to in this paper:

1. https://www.opvizor.com/docker-performance-on-top-of-vmware-vsphere/
2. https://www.percona.com/blog/2016/08/03/testing-docker-multi-host-network-performance/
3. https://www.percona.com/blog/2016/02/11/measuring-docker-io-overhead/
4. http://mysqlserverteam.com/mysql-with-docker-performance-characteristics/
5. https://forums.docker.com/t/file-access-in-mounted-volumes-extremely-slow-cpu-bound/8076
6. https://github.com/docker/docker/issues/21485
7. https://www.docker.com/

[1] refers to a methodology to execute performance tests on containers using the LINPACK tool
[2] tests if there is I/O overhead associated with running Docker containers
[3] tests if there is overhead if communicating across Docker Network over multiple hosts
[4] publishes performance test results when running MSSQL inside Docker containers, vis-à-vis running a stock MSSQL instance on a server
[5] and [6] refer to recent performance-related bugs reported with Docker containers
[7] is a general reference to Docker related documentation