An open, holistic framework to combine metric data, log events and inventory data.

IT infrastructure (no matter the size) is composed of multiple layers, supplied by different commercial vendors as well as more and more open-source projects. This increasing complexity needs to be addressed by defining a common sense framework to consume, process and explore the data produced by any part of the system; no matter if it is metrics data, event information or inventory data.

The pace of innovation is becoming faster and faster and thus the integration of new services has to keep up to speed. Long integration cycles are going to be a handicap in a competitive, ever changing world.

In this paper a system is introduced, which serves as an open framework to consume, process, store and make use of bits and pieces of information originating from each infrastructure layer. The framework uses Docker as a foundation of a microservice architecture and integrates the best-of-breed open-source technologies such as Graphite, Logstash and Neo4J. This approach provides an inter-layer methodology which allows connections between subsystems like network, software inventory and resource scheduler to assist in complex analysis of problems and performance. For example by correlating Infiniband performance metrics with job-related information obtained from SLURM.

I. INTRODUCTION

A data-center is comprised of a variety of different layers. From the physical hardware, support facilities (e.g. cooling), up to end-user applications. Some of them are complex in themselves, and often debugging a particular layer is impossible without taking information from connected parts of the infrastructure into account.

One example is the network interconnect, which provides communication between multiple entities. A view into the layer itself provides information about data flows (bandwidth, latency) and errors on a switch and end-node basis.

As an system operator this angle might look appealing, but without a context in which the data flows occur, the metrics are almost rendered useless, because the correlation with information from other layers is put onto the shoulders of the operator. He/she has to switch different tools with different User Interfaces and aspects to solve inter-layer problems.

A. Model Discrepancy

One aspect why the correlation of inter-layer information is such a hard task is a psychological one. Over time a human operator forms a mental model of the system under his/her supervision. If information is received from somewhere, be it an event that occurred or performance metrics, this information has to be mapped into the mental model which has been build so far. The human is served best by a given tool, if the tool's model and his/her mental model is perfectly aligned. E.g. a hierarchical structure is better displayed as a graph rather then a set of unconnected nodes. This might be one reason why it is very common to recreate existing tooling around infrastructure projects (Not-Invented-Here syndrome[1]), even though the underlying problem is shared and equal among a broad spectrum of people. The driver seems to be that each system engineer has a slightly different mental model and prefers to build something from scratch.

This gets even worse in the complete infrastructure, since a variety of layers has to be covered, worst-case with different models even for common entities. The network model only deals with host channel adapters and does not consider the actual host in his model (as InfiniBand does, the dominant high-speed interconnect in HPC systems) in contrast to a resource scheduler like SLURM (Simple Linux Utility for Resource Management[2]) which only cares about hosts. Even these closely related topics, which are easily fitted in a mental model, are hard to align in practice. An even harder challenge is to model a commodity Ethernet network and align it with a special purpose interconnect like InfiniBand. This leads to mostly distinct tools (even operational teams) for different techniques within the IT infrastructure.

B. Commoditization Problem

While commoditization and specialization of hard- and software was one of the drivers to lower cost and the entry barrier to technology, it introduces an increase in layer count and independent parts within each layer. The task to create an inter-layer framework became even harder and only monitoring tools like NAGIOS (Health check and alerting system widely used within the IT[3]) provided some insight. The financial incentive to create an overarching framework which allows to mix, match and correlate information from different aspects is quite low,

[1] Investigating the Not Invented Here syndrome, 1982, R. Katz

[2] 🌍 http://slurm.schedmd.com/

[3] 🌍 https://www.nagios.org/

since the component vendors do not benefit much from providing hooks to all tools out there.

C. Open, Holistic Framework

To overcome this problem this work proposes an open framework embedded into a microservice architecture. The implementation aims for decoupled back-end, middle ware and front-end. Inspired by the metrics ecosystem Graphite (see II D 4), which had this idea baked into the core from the beginning.

II. METHOD

A. Microservices

The motivation behind this work dates back to the urge to run a complete HPC cluster stack in a visualized fashion on a single laptop in order to be able to gain a better understanding of how a HPC software stack interacts. This first attempt was presented during the OSDC2014 (Open Source Datacenter Conference 2014[4]) .

All services were put into a distinced machines, which only served an environment needed to run a given service. Without calling it microservices, the idea behind it was quite similar with the approach microservices are associated with. One of the most quoted definitions of the term microservices is the following[5]:

> microservices is defined as a loosely coupled service oriented architecture with bounded context
>
> *Adrian Cockcroft*

In practice the emerging pattern is that small teams get full ownership of a particular service (bounded context). How the service is implemented is of no interest to the clients (most likely microservices themselves) of this service. The interaction is strictly defined by a contract. A common protocol to provide this interaction is a RESTful API based on HTTP and JSON.

A litmus test of such an architecture is an update of services. 'Loosely coupled' should lead to a solution in which an update of one service should not enforce an update to services using the API. If this test is passed successfully the system provides independent update cycles for each service, without breaking the current system as a whole.

B. Service Discovery

When implementing a microservice architecture the amount and scale of services can become a big burden when it comes to coupling the services. If hooking services together implies manual labor to put IP addresses into configuration files, the scalability of the system is going to be fairly limited. Instead, the system should advertise its capabilities and discover other services automatically.

The used implementation started as a set of bash scripts updating `etcd` (distributed key/value store[6]). This key/value store was used by `SkyDNS`, which exposes entries over a DNS interface and thus made the coupling of distributed services relatively easy. To dynamically create configuration files a tool called `confd` (Configuration management tool using templates[7]) emerged, which queries `etcd` and produces configuration files from templates if a watched value changes.

A more holistic tool to deal with this set of functionality is `Consul` (distributed cluster management[8]). It combines a key/value store, service checks, a DNS interface and an RESTful API in one golang binary, which is easy to manage.

Listing 1 shows an example of a service, including attributes and checks to determine the state of the service based on the return code of the script (like NAGIOS does).

```
{
  "service": {
    "name": "nginx",
    "port": 80,
    "check": {
      "script": "nmap 127.0.0.1 -p80|grep open",
      "interval": "10s"
    }
  }
}
```

Listing 1. Service definition within `Consul`. If the keyword `open` is not found within the nmap output the service is set into a warning state.

If the check indicates the service is usable, the service is exposed via an DNS interface. Thus the service can be reached via the FQDN (Fully Qualified Domain Name) <service−name>.service.consul as shown in the example in Listing 2.

```
$ dig SRV +short nginx.service.consul
1 1 80 nginx.node.consul.
```

Listing 2. DNS query of `nginx` service, issued with `dig`, a common DNS resolver.

The ablility to reference a service via DNS is extremely helpful, since name resolving works system-wide. Instead of static IP address, a configuration file contains the FQDN to describe a dependency to a different node.

[4] https://www.netways.de/?id=3132#c27028

[5] http://www.slideshare.net/adriancockcroft/microxchg-microservices

[6] https://github.com/coreos/etcd/

[7] http://www.confd.io/

[8] https://www.consul.io/

Furthermore the complete tooling is going to work as well; checking network connectivity is as simple as using `ping` to resolve the FQDN.

Listing 3 shows an example of the RESTful API to access information as JSON blobs.

```
$ export URL=localhost:8500/v1/catalog/service
$ curl -s $URL/nginx|python -m json.tool
[
    {
        "Address": "172.17.0.6",
        "Node": "nginx",
        "ServiceAddress": "",
        "ServiceID": "nginx",
        "ServiceName": "nginx",
        "ServicePort": 80,
        "ServiceTags": null
    }
]
```

Listing 3. Exposure of services via RESTful API.

This provides the ability to use this information in external scripts or use the rich ecosystem that has already build up around service discovery.

Since this work aims to be modular and flexible in its composition, a service discovery mechanism which is able to dynamically update itself is crucial to the success of the framework.

C. Linux Containers

Linux Containers[9] leverage Kernel Namespaces to isolate different groups of processes. Unlike hypervisor visualization, which emulates the hardware of a physical computer, Linux Containers are spawning processes with certain flags, attaching the process (and its children's processes) to a set of namespaces. Within the Linux Kernel namespaces exist to encapsulate processes, mount points, network stacks, inter-process communication and domain information. Furthermore a container can be granted only a subset of kernel capabilities.

A common way to restrict the use of resources (CPU, memory, I/O) available by a running container is the use of CGroups (Linux Control Grroups[10]).

1. Docker

The Linux Container implementation that took off within the last couple of month is Docker. It leads the rise of Linux Containers as a commodity technology, accessible for a broad range of users. It provides an easy and intuitive way to build, distribute and run container images. By developing Docker in the open and providing an API, Docker gained a lot of momentum and surrounded Docker with a rich ecosystem.

Linux Containers are a perfect fit for a microservice architecture. They do not spin up a complete operating system but only the process necessary for the service and therefore minimize the overhead. Furthermore Linux Containers are started and stopped within a fraction of a second, which provides a proactive elasticity needed to be agile.

2. Containers versus Virtual Machines

One could argue that virtual machines are best suited to serve monolithic applications, which assume an uninterrupted operation. If the underlying hardware is failing the virtual machine gets migrated to a different host. Ideally just with a slight hiccup, while the network address is transferred. The application itself is designed to life a long and protected life.

In the microservice architecture dealing with failure is a crucial part of the design itself. A system should not have single points of failure and should self-heal if a service fails. How many (and which) concurrent errors a system can withstand depends on the effort put into it and might differ in certain stages of the development, but the ideal goal is a self-healing system.

D. Metrics Engine and Log Pipeline

Unlike metrics, which are data points associated in a time series (most are repetitive, like system performance information) fashion, log data consists of distinct events. Granted, the line is a bit blurry and some events include metric data that might be extracted in a subsequent step. NAGIOS for example provides performance information with most of the alarm events to enrich the event.

In an ideal world the two would be separated pieces of information. A metric does not provide context but pure distilled performance information in the form of a value connected to a metric key and timestamp.

A log event would be pure context without any performance information (e.g. 'Cronjob to sync video data started by user bob').

In practice the producer of an event is most likely to cross this line, since the past favored an enriched (human readable/understandable) event over a pure piece fed into the correlation engines of today. But today this line gets redrawn; in times were big data is a hype topic for the last couple of years, the quality/clarity/focus of incoming data matters.

[9] 🌍 `https://en.wikipedia.org/wiki/LXC`

[10] 🌍 `https://www.kernel.org/doc/Documentation/cgroups/`

1. *Log Pipeline*

In the past Log engines like Splunk (a commercial platform for Operational Intelligence[11]) were high priced data consumers, which proposed to assist in dealing with the flow of incoming unstructured data. Without such a framework it was very hard to normalize all the data and make use of the information gathered.

In the last couple of years open-source tools emerged. The most prominent is Logstash (powerful, modular event pipeline[12]), followed by Graylog (Log management platform[13]). They provide a modular pipeline, which allows to take data from different inputs, normalize it (commonly in some JSON-style format) and alter the information on the way through. Among others the data is then persisted in a data-store, e.g. in Elasticsearch (Open-source text-index engine based on Lucene[14]). Both ecosystems provide visualization tools and connectors to a rich ecosystem.

```

input {
    syslog {
        port => 5514
        type => syslog
    }
}
filter {
    mutate {
        add_tag => [ "special" ]
    }
}

output {
    stdout { codec => rubydebug }
}
```

Listing 4. Basic Logstash configuration which consumes events via syslog endpoint, adds a tag and writes events to stdout.

The power of these pipelines lies in the flexibility they provide. With a couple of lines (an example is shown in Listing 4) of configuration the logs are centralized and accessible via an intuitive dashboard like Kibana (Client-side javascript visualisation framework[15]) as they were never before. Compared to plain text files on each server itself this takes the accessibility of log event information to another level.

2. *Elasticsearch*

Storing text in a way that allows fast searches is a complex task. Elasticsearch tackles that by pre-processing data while it is consumed by its engine. This process is called indexing and it splits the information in various ways to speed up the search. Elasticsearch consumes JSON formatted data and adds additional fields if needed. If the use-case indicates that ranges in steps of hundreds are often used, the date value=15235 might be also stored as hundred_val=15200. This allows a binary check on the additional value instead of the evaluation of 15200 <= 15235 and 15235 <= 15299.

By providing use-case driven pre-processing Elasticsearch is able to provide a powerful search engine for unstructured data. Log events are a primary example in which Elasticsearch provides fast an easy access to data.

3. *Kibana*

Kibana is an open-source javascript framework to visualize data stored in Elasticsearch. This open source project is backed by Elastic[16], the company behind Elasticsearch and integrates seamlessly with their datastore.

4. *Metrics Engine*

The first pillar of the open framework is the ecosystem Graphite (Metrics engine[17]), which captures performance values over time. Started in 2006 it provides a modular system to consume, process, store and access metric information in the form 'key value time-stamp'.

The metrics are persisted in round-robin databases called whisper by a set of daemons called ‘carbon‘. Listing 5 shows an example of how performance metrics are processed and stored.

```
$ for x in {1..5};do
> echo "test.metric ${x} $(date +%s)" |\
    nc -w 1 127.0.0.1 2003
> sleep 5
> done
$ export CDIR=/var/lib/carbon/whisper
$ whisper-fetch $CDIR/test/metric.wsp |\
    grep -v None
1438513460 1.000000
1438513465 2.000000
1438513470 3.000000
1438513475 4.000000
1438513480 5.000000
$
```

Listing 5. Sending and receiving metrics using carbon in combination with a whisper file back-end.

The Graphite framework provides a rich ecosystem of visualizations, plugins, metric sources and alternative daemon implementation to fit almost all use-cases around

[11] http://www.splunk.com/
[12] http://logstash.net/
[13] https://www.graylog.org/
[14] https://www.elastic.co/products/elasticsearch
[15] https://www.elastic.co/products/kibana
[16] https://www.elastic.co
[17] http://graphite.readthedocs.org/en/latest/

metrics gathering, processing, storing and serving. Due to its open-source development and stable API, missing pieces can be added if needed.

This enables each part of the infrastructure layer to submit metrics without much effort. Even application developers can just start sending metrics down the pipeline and let them pop up at the access layer.

5. Metric Key Scheme

If the group of metric generators grows to a certain complexity, the metric keys become a problem, since naming schemes vary depending on the context of the producer. An engineer overseeing the network in a data-center might organize his/her metrics hierarchically (datacenter01.rack01.compute01.eth0.send_bytes), while from the resource scheduler perspective a grouping per job (jobid01.compute01.send_bytes) is a scheme which might fit the use-case best.

To overcome this issue a scheme should be chosen which abstracts from a metric key as single string. The Metrics2.0[18] idea (an example is shown in Listing 6) annotate metrics with metadata.

```
{
    server: dfs1
    what: diskspace
    mountpoint: srv/node/dfs10
    unit: B
    type: used
    metric_type: gauge
    value: 1024
}
meta: {
    agent: diamond,
    processed_by: statsd2
}
```

Listing 6. Metrics2.0 formatted metric

6. Grafana

As the name might suggest Grafana uses the framework of Kibana (see II D 3), but intended to visualize metrics using the Graphite API, hence the name which combines the two words (**Gra**phite + **f** + Ki**bana**). Even though it originated as pure metric visualization tool, it allows to annotate syslog events stored in an Elasticsearch instance (due to its origins a simple task). Furthermore Grafana is able to query other data back-ends then Graphite, namely OpenTSDB[19] and InfluxDB[20].

E. Inventory

Existing inventory systems face a twofold problem. First they are mostly designed for one particular layer of the infrastructure cake. This burdens certain constraints to the model of the system, which are hard to overcome. A holistic approach is very rare, some might provide a global character, but most likely as an historically grown extension.

The second problem comes down to the technique used. Most systems use some sort of Rational Database, which has a tendency to normalize the schema involved to eliminate redundant information. Once a schema is set up, it is hard to shoehorn other aspects of the data-center into the established schema formed by the first consideration.

To overcome this, the system should be backed by a graph database, which allows a schema-free model to evolve over time. Different aspects use different models. Inter-layer dependencies are modeled by connecting the related nodes. This allows a rich query to span across multiple layers.

F. OpenSM

The management of InfiniBand is centralized and coped with by one daemon per subnet. This service holds the inventory, computes the routing tables for each switch, reacts on events within the network et cetera. The OpenFabrics Alliance[21] provides an open-source implementations called OpenSM[22].

Fabriscale[23] has extended OpenSM, to provide better routing capabilities alongside with enhancements in operational aspects. By providing a RESTful API the daemon `fabriscale`, is a good fit to be integrated in a microservice architecture. Furthermore it provides a plugin to extract performance information and send these off to the metrics engine.

G. Holistic Framework

To provide a holistic, multi-layered approach to infrastructure management the proposed framework combines all of the presented techniques. An open metric engine, log event pipeline and inventory system based on a graph database. All of this held together using `Consul` as a service discovery tool. The microservice architecture comprises of the services shown in Figure 1.

[18] http://metrics20.org/
[19] http://opentsdb.net/
[20] https://influxdb.com/
[21] http://www.openfabrics.org/
[22] http://git.openfabrics.org/
[23] http://fabriscale.com/

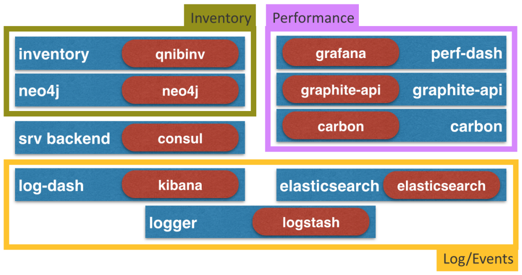


FIG. 1. Composition from a container and service view

III. CURRENT STATUS OF THE FRAMEWORK

By combining the presented technology stacks, the framework provides deep insights intothe complete cluster stack.

A. Domain Modeling

By using a schema-less graph database approach, each domain can be modeled independently. Common entities are connected at any time after they are modeled.

1. *Resource Scheduler*

SLURM provides information about resource allocations: Which jobs was started where, by whom at what time? How long did it last? What was the outcome? and alike. The following SLURM entities are modeled as nodes within the graph.

- **host** The SLURM node, which represents a compute node within the cluster, identified by the hostname.
- **partition** A logical group of nodes which represent the target of a job submission.
- **job** A demand for resources submitted by a user. Including metadata (user, submission/start/end time, job script, etc) and information about the state of the job. If resources are allocated, the job metadata is updated accordingly.

Relationships are added to connect certain nodes. PART_OF declares host memberships to partitions. JOBCLIENT connects a host to a certain running job.

2. *InfiniBand Interconnect*

An InfiniBand representation comprises of a set of physical and logical representations.

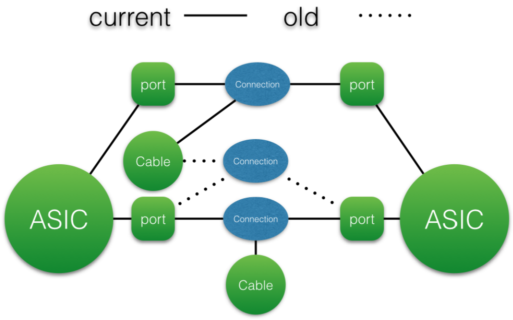


FIG. 2. Modeling of an InfiniBand network.

- **HCA** Network adapter plugged into a host
- **SW** InfiniBand switch
- **PORT** Each HCA and SW are composed of a set of ports which provide connectivity.
- **CABLE** Physical connection between different ports.
- **CONNECTION** Logical entity which connects two PORTS and a CABLE for a given period in time.

Relationships combine two PORTS and a CABLE alongside with meta-information about the time when it was created, if the connection is still valid and a like. Furthermore the routing information is added by providing a ROUTE_TO relationship between a port and a host.

B. Holistic Inventory

The presented two models alone provide combined insight into the cluster. By connecting the HCA (within the InfiniBand model) to a host (within the SLURM model) the inventory allows inter-layer queries.

1. *Job Placements*

By showing the layout of a job within the 'InfiniBand' (IB) topology, an imbalanced job is easily detectable. Figure 3 shows an example in which the hop-count within a job differs. A slightly more complex case would include different port-characteristics like speed or width. By caching information of the metrics system within the graph database an even more sophisticated query might include error or performance counters.

2. *InfiniBand Debugging*

A common debug case within the InfiniBand network is to figure out if errors associated with a connection are

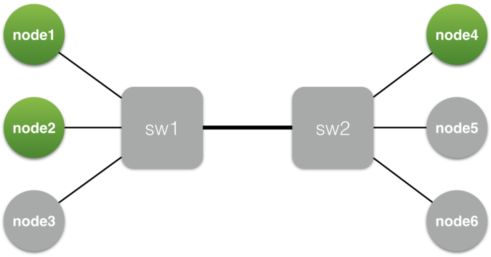


FIG. 3. A job runs on three nodes (green). By using node3 instead of node4 the job could run evenly distributed on sw1

caused by one of the ports or the cable itself. The approach to fix this error is to carefully unplug one connector of a given cable, connect it to a different switch-port and check the error counters of the just established connection. If the error sticks with the cable the opposite side of the cable has to be changed as well, to truly pin down the problem.

This process is extremely error prone; an inventory to deal with an InfiniBand fabric barely exists. Therefore each step is manual labor with the risk of miscommunication between the on-site engineer changing the cables and the engineer matching the changes to performance counter changes.

By using an holistic inventory (as indicated in Figure 2) the changes are tracked automatically and since they include the timestamps the correlation with the metrics systems is trivial.

C. Open Log and Metric Framework

Determining inter-layer error triggers is an extremely hard task. If e.g. a specific job triggers an error within the interconnects one has to keep track of the job placement, the routes within the interconnect and the time to correlate metrics with this events.

Figure 4 shows the performance metrics and the topology of the InifiniBand network, as a first overview. Hosts which have vanished from the network are going to be marked, so that such errors are easy to spot.

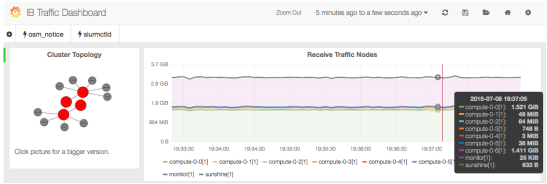
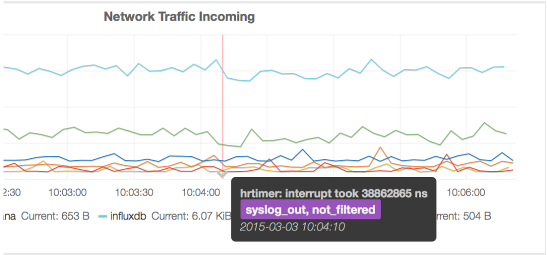


FIG. 4. Dashboard to visualize the topology (left) and the live traffic metrics within the InfiniBand fabric.

Overlaying events and metrics allows to correlate both data sources. Figure 5 shows an example of an event annotated to network metrics. The slight decrease in the upper most metric might indicate causality, which is hard to spot without the overlay.


hrtimer: interrupt took 38862865 ns

syslog_out, not_filtered

2015-03-03 10:04:10

FIG. 5. Performance metrics annotated with syslog events to correlate events and metrics.

IV. FUTURE WORK

The coupling of metrics, log events and inventory data is still an area which is researched and therefore it is not battletested.

A. Data Back-end

When talking to different camps within the space it seems that each back-end is a good fit to cover all needs. Some say, it is safe to store metric information in Elasticsearch. On the other hand InfluxDB supports storing events and so on.

In the future the right balance, back-end and connectivity has to be found.

B. Metrics 2.0

Aligning the metrics2.0 concept to allow tighter and more specific reference of tags might help connect information. Tags might implicitly describe to which datapoint (inventory or event) they are referring to.

C. Inventory Model

More domain models have to be included into the Inventory system to gain more experience in regards of possible friction points and limitation to this approach.

D. Scale Test

The use of this framework is so far limited to small clusters and a laptop. How well it is going to scale and what the limitations are is to be discovered. The scaling of the libraries and scripts to integrate the different aspects are going to be the majority of the work to scale the complete system.

V. CONCLUSION

Overall, this approach provides a framework which allows rapid prototyping and integration of services. By leveraging the service discovery the configuration overhead is minimized and the software stack configures itself for the most part.

A. Modularity

Due to its modularity the framework is an optimal contender to integrate new services without interfering with existing parts. The microservice architecture implicitly forces well defined interactions in between the subparts, which allows to remove, add or duplicate functionality if needed. Incomming metrics for example might be mirrored and send to a backend system for evaluation without breaking existing queries.

B. Graph Database Inventory

By using a schema-less graph database as inventory system, domains can be modeled separatley and connected as a subsequent step. This allows inter-layer correlation without the need to define a least common denominatior before hand.

C. Visualization

Kibana as the main log event dashboard and Grafana as the initial performance metric visualization provide state-of-the-art flexible and intuitive access to the information stored in the complete infrastruture stack. By leveraging the inventory information dashboards are generated automatically to provide up to date views.

D. Performance

While not tested at scale the system works nicely as an testing environment. All subsystems are known to scale horizontally.