Complete Dynamic Multi-cloud Application Management

Project no. 644925

Innovation Action

Co-funded by the Horizon 2020 Framework Programme of the European Union

Call identifier: H2020-ICT-2014-1

Topic: ICT-07-2014 – Advanced Cloud Infrastructures and Services

Start date of project: January 1st, 2015 (36 months duration)

Deliverable D6.3

Solutions for Non-functional Aspects of Cloud Computing

Due date:	30/11/2015
Submission date:	19/12/2015
Deliverable leader:	UvA
Editors list:	M. Živković (UvA), C. Loomis (SixSq)

Dissemination Level

☒	PU:	Public
☐	PP:	Restricted to other programme participants (including the Commission Services)
☐	RE:	Restricted to a group specified by the consortium (including the Commission Services)
☐	CO:	Confidential, only for members of the consortium (including the Commission Services)

List of Contributors

Participant	**Short Name**	**Contributor**
Interoute S.P.A.	IRT	
SixSq Sàrl	SIXSQ	C. Loomis, K. Skaburskas, S. Tavera, L. Schaub, K. Basbous
QSC AG	QSC	
Technische Universitaet Berlin	TUB	
Fundacio Privada I2CAT, Internet I Innovacio Digital A Catalunya	I2CAT	J. Aznar
Universiteit Van Amsterdam	UVA	M. Živković
Centre National De La Recherche Scientifique	CNRS	

Change history

Version	Date	Partners	Description/Comments
0.1	07/12/2016	SixSq, UvA	Initial copy into Word format
0.2	12/12/2016	SixSq, UvA	First complete version for review.
0.3	14/12/2016	I2CAT	Incorporate comments from J. Aznar.
1.0	15/12/2016	SixSq, UvA	Improved SoTA, final corrections.

Table of Contents

List of Figures

List of Tables

Executive Summary

Non-functional aspects, such as cloud application cost and performance, have a strong impact on the management of cloud applications. Both the research community and the main cloud providers have already developed auto-scaling tools and solutions. These tools avoid underutilization and overutilization of the cloud resources while still maintaining a good level of quality for the hosted services. This has two advantages: for the customer, this reduces the total costs, while the cloud provider can serve more customers with the same infrastructure. Cloud providers' solutions, however, are limited in terms of prices, availability, reliability, and connectivity because they are locked to a single vendor.

Building on the existing work, this document describes the CYCLONE solution for dealing with non-functional and functional aspects of cloud computing to ensure that an application has the optimal resources allocated to it over its full lifecycle.

The solution relies on SlipStream (with extensions) to handle the provisioning and re-provisioning of resources as the operating conditions for a cloud application change over time. All components of SlipStream contribute to the solution: The Workspace for application policy definition, the deployment engine for (re-)allocating resources, the monitoring to provide performance data, and the Service Catalog for finding appropriate cloud resources. SlipStream also provides an example application to demonstrate how auto-scaling of applications can be accomplished.

In a multi-cloud, multi-user, multi-application environment, there cannot be a unique solution that applies in all cases. The specific decisions on what actions to take (scale-up, scale-down, migration, etc.) are defined by an application-specific policy using input from general algorithms for detecting changes in metrics. We, however, do propose general algorithms to determine when changes in non-functional metrics occur and allow the application developer/operator to define the actions to take when those changes are detected.

Advanced implementations of the auto-scaling SlipStream features and the algorithms to detect changes in application metrics exist. All that remains is to bring these two components together to validate the overall solution and to demonstrate the defined use cases.

The document also reviewed the state of the art for autoscaling, both in the academic and commercial worlds. The comparison of SlipStream with competing products and services has shown that many of them offer comparable features, but only for subsets of SlipStream's full feature set. SlipStream is still the only product to offer a comprehensive multi-cloud solution for all aspects of application management (including scaling) and brokering.

1. Introduction

Cloud computing infrastructures are a recognized alternative to buying and maintaining custom IT infrastructures on premise. The primary economic attraction is the ability to pay for the computational resources only when they are needed—a shift from capital expenditures to operational expenditures. The flexibility to change the allocated resource dynamically, known as elasticity, is the basis of the utility computing model.

The dynamic provisioning of virtualized resources offered by cloud computing infrastructures allows applications deployed in a cloud environment to increase and decrease *automatically* the allocated resources. The main purpose of this capability, known as auto-scaling, is to minimize automatically the resources allocated to an application while satisfying the varying workload and performance constraints. The need for auto-scaling is particularly important during workload peaks, during which applications may need to scale up to extremely large-scale systems. *Non-functional aspects, such as cloud application cost and performance, have a strong impact on the management and auto-scaling of cloud applications.*

Both the research community and the main cloud providers have already developed auto-scaling tools and solutions. These tools avoid underutilization and overutilization of the cloud resources while still maintaining a good level of quality for the hosted services. This has two advantages: for the customer, this reduces the total costs, while the cloud provider can serve more customers with the same infrastructure. Cloud providers' solutions, however, are limited in terms of prices, availability, reliability, and connectivity because they are locked to a single vendor.

These limitations motivate the current research efforts in cloud computing to find efficient ways to exploit multiple cloud infrastructures for the deployment of an application or service. The capability to federate different cloud providers and cloud services becomes particularly useful when smaller companies want to provide unique offers in the competitive cloud market. With a federated or hybrid-cloud system, application operators can make their own trade-offs from cloud service providers with varying capabilities, guarantees, and stability.

The types of application or service operators that can profit from such platforms are varied. Examples include multimedia (audio and video) transcoding, scientific processing, strong encryption/decryption, distributed compilation, translation, and solving NP-hard problems from a model. Many companies already offer such services through the Internet, such as Zencoder [ZEN16] for video file transcoding. These companies have a strategic interest in reducing the costs for instantiating their services while maintaining the satisfaction of their paying customers.

We present here the application management platform for the CYCLONE that takes advantage of cloud federation to avoid the limitations of a single cloud provider, while maintaining a good quality of service (QoS) for the customers and minimizing infrastructure costs for the service provider. The solution is decentralized and self-organizing; none of the cloud providers in the federation have a central role in how resources for the service are placed within the cloud providers. The solution is also self-adapting, reducing the human effort required when redeploying and reconfiguring the system in case of a serious failure in one cloud provider.

The document first describes the motivating use cases and then identifies the requirements of the system. It then describes the components of the CYCLONE application management system (centered on

SlipStream) that contribute to the auto-scaling solution. This system provides the means, but not the knowledge, to scale a system automatically. The knowledge is embedded in the algorithms used to scale the application; the document describes the current state of the art and proposes the algorithms to be used in CYCLONE for scaling applications based on both functional and non-functional requirements.

The CYCLONE solution is centered on SlipStream. However, there are other platforms and products that provide similar functionality. A comparative analysis between SlipStream and the alternatives justifies the project's choice of SlipStream, but also indicates potential markets for exploiting SlipStream and the other CYCLONE components after the project.

The document concludes with a summary of the solution, the main points of the comparative analysis, and points for future work.

2. Motivation

2.1. Use Cases

To motivate the CYCLONE auto-scaling framework, several use cases have been identified. The first three highlight the general functionality (independent of the selected application) required of the system. They rely upon a couple of statistical properties of the non-functional parameters (e.g. costs, response time distributions, etc.) that describe the application and cloud infrastructure performance. The last two use cases are real-world use cases that further illustrate the requirements of the system and demonstrate the utility of the solution.

2.1.1. Generic Use Case 1: Provisioning (UC1)

Through the SlipStream AppStore, users can find a list of available applications and then filter them based on their requirements. In this scenario, three functionally equivalent applications are found: A1, A2, and A3. With the SlipStream Service Catalog, users can find all the functional and non-functional characteristics of each cloud infrastructure. In this scenario, two possible cloud infrastructures are identified: C1 and C2.

The user then specifies the application selection and placement criteria (or "policy"), choosing for example, the cheapest and the fastest application. Within the policy, the user also specifies the relative importance of each criterion: for example, the cost is ten times more significant than the application response time. The user could also specify that there is a maximum budget and/or a deadline.

Based on this input, the SlipStream ranks the applications and cloud infrastructures, presenting the user with a ranked list of choices. The user can choose any option (e.g. A2 on C1) and then tell SlipStream to deploy the application.

2.1.2. Generic Use Case 2: Redeployment (UC2)

Once the optimal deployment solution has been found, the selected application is deployed on the chosen cloud infrastructure(s). Starting virtual machines on cloud infrastructures is not instantaneous; it can take tens of seconds to several minutes, depending on the cloud service provider. The startup latency must be benchmarked to understand when the deployment process has encountered an error or performance degradation on the chosen cloud infrastructure(s). Performance issues or problematic response times can also arise for running applications. In each case, it may be beneficial for the user (with respect to costs and completion time) to interrupt the current deployment process and then re-calculate the placement policy without the problematic cloud infrastructure. The user can then choose a new optimal application/cloud pair from the ranking.

Despite the presumption that redeployment can be beneficial, redeployment within a federated or hybrid cloud infrastructure is something that is not yet well-understood and certainly not automated. The conditions under which redeployment is beneficial must be identified before this can be rolled into an algorithm for application management.

2.1.3. Generic Use Case 3: Monitoring (UC3)

Performance volatility on cloud infrastructures is common and such volatility can lead to changes in application performance that need to be addressed. Typically, the performance deteriorates abruptly and the response time for an application increases significantly.

Monitoring is essential to detect such situations and overall performance management of cloud applications. When the monitoring determines that the performance has changed significantly, the CYCLONE platform may request scaling up the application's resources or redeployment of application components elsewhere. Volatility can also lead to improved performance. Monitoring can also detect these situations and trigger a process to scale down the application.

2.1.4. Scientific Image Processing (UC7)

Publicly funded research creates an immense amount of data that has general social, academic, and commercial value. The taxpayers, through funding agencies, increasing expect these data to be available through "open data" programs. Despite these expectations, finding viable business models that keep the maintenance costs for the public reasonable has been an obstacle to widespread availability of open data. SixSq is exploring solutions with European Space Agency (ESA) in which public data is hosted on European cloud infrastructures and partially or fully monetized to reduce the need for public subsidies.

Technically, a viable solution requires:

- Detailed knowledge of the storage locations of dataset components,
- Means of placing analysis applications near the data of interest, and
- Ranking of multiple providers based on price or other characteristics.

All but the first are already features of the CYCLONE brokering and matchmaking components. Integrated data management is a crucial feature for this use case that is planned for SlipStream in the last year of the project. Advanced networking may also play a role here, if significant bandwidth is required for remote access to datasets.

More information about this use case can be obtained from the CYCLONE use case portal [UC7].

2.1.5. Benchmark Driven Placement (UC8)

A significant part of the design discussion for the CYCLONE brokering and matchmaking components dealt with benchmarks, both general and application-specific benchmarks. At the UCC 2015 conference [UCC15], a bioinformatics group from Cardiff presented custom tooling that collects benchmarks for common bioinformatics tools on many community and public clouds and that uses this information to optimize the placement of their virtual machines in the future [CON15]. The benchmarks are collected continuously from active scientific analyses and are augmented with information from directed probes of each infrastructure.

Demonstrating their workflow with the CYCLONE tools would be an interesting, direct validation of the project's brokering and matchmaking design. Similar research groups would benefit by being able to optimize the placement of their scientific analysis applications without having to invest effort in building up and maintaining their own customized management tools.

Although the project will probably not work directly with this group, an example application will be created to demonstrate how benchmark driven placement can be achieved with the CYCLONE tools. More information about this use case can be obtained from the CYCLONE use case portal [UC8].

2.2. Requirements

Based on the above use cases, a set of requirements for the auto-scaling features was derived. Many of the requirements duplicate those that were previously identified for the matchmaking and brokering features described in earlier deliverables. Overall, these requirements require minor adjustments to existing features of SlipStream, rather than significant additional development.

Table 1: Derived Requirements

1	Triggering of scaling actions of an application based on application metrics using simple, predefined algorithms (e.g. adding node based on machine load).
2	Triggering of scaling actions of an application based on application metrics defined by the developer of the application.
3	Ability to publish application-specific benchmarks of cloud providers into the Service Catalog or Open Service Compendium.
4	Placement based on static characteristics (e.g. geographical location) of a cloud service provider.
5	Placement based on dynamic VM monitoring information from SlipStream itself.
6	Placement based on external information pushed into the SlipStream Service Catalog or Open Service Compendium.
7	Placement based on the join of all information associated with a given cloud service provider.
8	Ranking of selected cloud service providers based on predefined algorithms (e.g. price).
9	Ranking based on algorithms provided by the application developer and/or the application operator.
10	Ability to trigger notifications/alerts through SlipStream.
11	Ability to trigger scaling actions from within the application.
12	Ability to search the Service Catalog and Open Service Compendium manually to see the results from various policies and to ideally then associate those policies with applications.

2.3. State of the Art

Cloud-based application cases have been reviewed and we considered unresolved issues related to cloud platforms [DIA16]. Regarding auto-scaling schemes, related concepts and taxonomy were surveyed [ALI14]. A technical review of auto-scaling for elastic cloud-based applications was provided, and the Gartner group describes auto-scaling as an automatic expansion or contraction of system capacity, and indicated that such a capacity is a commonly desired feature in cloud infrastructure as a service and platform as a service offerings [LOR14]. In other words, auto-scaling refers to the significant capability of a cloud computing environment to utilize virtualized computing resources automatically. In this scheme, virtualized resources can be increased or decreased dynamically by adapting resource utilization to satisfy the given requirements. Auto-scaling contributes to cost control. The key features of auto-scaling are the ability to scale-out, i.e., automatic addition of resources during increased demand, and scale-in, i.e., automatic termination of unused resources when demand decreases. Scale-out and scale-in schemes are referred to as horizontal scaling. Unlike horizontal scaling, vertical scaling increases computation resources in existing nodes. Auto-scaling at the service level is important because services run on a set of connected virtual machines. The optimal model-driven configuration of cloud auto-scaling infrastructure was studied

[DOU12]. It was implemented an open-source cloud environment with auto-scaling to access resources for a flexible period with varying requirements in bioinformatics and biomedical workflows [KRI17], and was implemented an auto-scaling model in simulation experiments using the Amazon Elastic Compute Cloud (EC2) to reduce resource costs and test the quality of service in terms of response time and availability [QU16].

The key features of auto-scaling are:

- The ability to scale out (i.e., the automatic addition of extra resources during increased demand) and scale in (i.e., the automatic termination of extra unused resources when demand decreases, to minimize cost).
- The capability of setting rules for scaling out and in.
- The facility to detect automatically and replace unhealthy or unreachable instances. Auto-scaling is often referred in the context of resource provisioning, scalability, and elasticity. These terms are often used interchangeably, but they are slightly different concepts.

Some challenges for the auto-scaling:

- Insufficient tools for monitoring and aggregating metrics at the platform level and service level to support auto-scaling decisions.
- Auto-scaling in hybrid cloud environments is not well supported. In hybrid scenario, cloud may offer different auto-scaling techniques that are not compatible with each other, so there would be an interoperability issue in auto-scaling resources across the two clouds.
- The efficiency of auto-scaling in terms of the reliability of the auto-scaling process is not well managed. Failure of the auto-scaling process can result in violations of the system's QoS requirements of performance and scalability and even incur unnecessary cost.
- The relationship between auto-scaling and quality attributes such as availability, reliability and security is unknown.

2.3.1. mOSAIC

An FP7 project mOSAIC (http://www.mosaic-cloud.eu/) was an FP7 research project that looked at cloud application management from cloud brokering and interoperability to deployment and execution. The main concepts are:

Component: Represents the basic building block of a cloud application, the atomic deployment and execution unit, which is materialized as one, or a set of tightly coupled, OS processes that run in an isolated environment. There are many types of components, each type mapping to an application tier, but they are treated the same by the platform. In general, they fit in one of the following component categories:

1. The "user" component, which embody the code developed by the user, and implements the needed logic;
2. Resource or middleware component, which provides generic services like data storage (MySQL), message brokering (RabbitMQ), etc.
3. Specialized components, which are of particular use in the mOSAIC platform or in a cloud environment, like the HTTP Gateway serving as a load-balancer.

Controller: The orchestration service that initiates the deployment and controls the execution of the components.

Hub: A bus-like or RPC-like system that allows components to discover each other, or exchange configuration messages.

This project built an open-source prototype, with web applications as main application domain. However, the scalability of the solution is manual, and there is no monitoring. The resources providers are Amazon EC2 and Eucalyptus.

Within mOSAIC, any component is able to listen on ports, provided that it requests access beforehand, or can receive inbound requests from the Internet. The resources allocated to a particular component are configured by the operator. The real limitation is that the component must run on Linux and cannot require root access.

This project has many differences with CYCLONE platform, as it does not provide any monitoring information, nor does it handle the runtime QoS of the deployed applications.

2.3.2. Cloud4SOA

The Cloud4SOA (http://www.cloud4soa.com/) project provides an open semantic interoperable framework for developers and providers, capitalizing on Service Oriented Architecture (SOA), lightweight semantics and user-centric design and development principles. The system supports Cloud-based application developers with multiplatform matchmaking, management, monitoring and migration by interconnecting heterogeneous offerings across different providers that share the same technology.

Cloud4SOA provides four core capabilities implemented by the reference architecture:

Matchmaking. The matchmaking component allows searching among the existing offerings for those that best match the developer’s needs. To succeed in this, the matchmaking algorithm capitalizes on the Semantic layer and, especially, on the PaaS and Application models while it distinguishes the user’s needs into application requirements and user preferences. The degree of relation is computed based on the similarity of the semantic descriptions between offerings and an application profile taking also into account the target user’s preferences. The outcome of the matchmaking algorithm is a list of PaaS offerings that satisfy users’ needs, ranked according to the number of satisfied user preferences.

Management. The module performs an analysis of the application requirements to build a specific application deployment descriptor. It then checks if a valid SLA contract has been previously agreed between the specific offering and the application, finally initiating the deployment process using the Cloud4SOA standard API exposed by every Cloud4SOA platform adapter.

Monitoring. To consider the heterogeneity of different Cloud architectures, Cloud4SOA provides a monitoring functionality based on unified platform-independent metrics, such as latency and application status, to allow application developers to proactively monitor the performance of applications hosted on multiple Clouds environments.

Migration. The Cloud4SOA framework aims to support a seamless migration to tackle semantic interoperability conflicts. Moving an application between offerings consists of two main steps: i) moving the application data and ii) moving the application itself. During the first step, all the application data is retrieved from the offering where the application is running and moved to the new one. To avoid inconsistent states, the application is stopped before data move. Once data has been initialized at the new provider, the application is deployed as well.

2.3.3. OPTIMIS

OPTIMIS aims at optimizing IaaS cloud services by producing an architectural framework and a development toolkit. The optimization covers the full cloud service lifecycle (service construction, cloud deployment and operation). OPTIMIS gives service providers the capability to easily orchestrate cloud services from scratch, run legacy applications on the cloud and make intelligent deployment decisions based on their preference regarding trust, risk, eco efficiency and cost (TREC). It supports end to end security and compliance with data protection and green legislation. It also gives service providers the choice of developing once and deploying services across all types of cloud environments - private, hybrid, federated or multi-clouds.

OPTIMIS simplifies the management of infrastructures by automating most processes while retaining control over the decision-making. TREC solution involves choosing an optimal target platform based on trust, risk, eco and cost data. This dynamic data that is extracted from target platforms is used for the benefit of service providers and end users and can be 'weighted' to fit their needs, ensuring an automated runtime solution. This can be used at runtime to dynamically manage the optimal platform for a service run, providing cross-platform scalability, using platforms with different hypervisors and cloud software, the OPTIMIS tools provide a common approach and methodology to achieve this.

2.3.4. MODAClouds

The MODAClouds project (www.modaclouds.eu) leverages the results of OPTIMIS and Cloud4SOA projects to provide a run-time environment which conceptual architecture is given in Figure 1. The project uses MAPE-K autonomic loop (Monitor, Analyze, Plan, Execute, Knowledge) that represents a blueprint for the design of autonomic systems where a managed element is coordinated by a loop structured in 4 phases and a common knowledge.

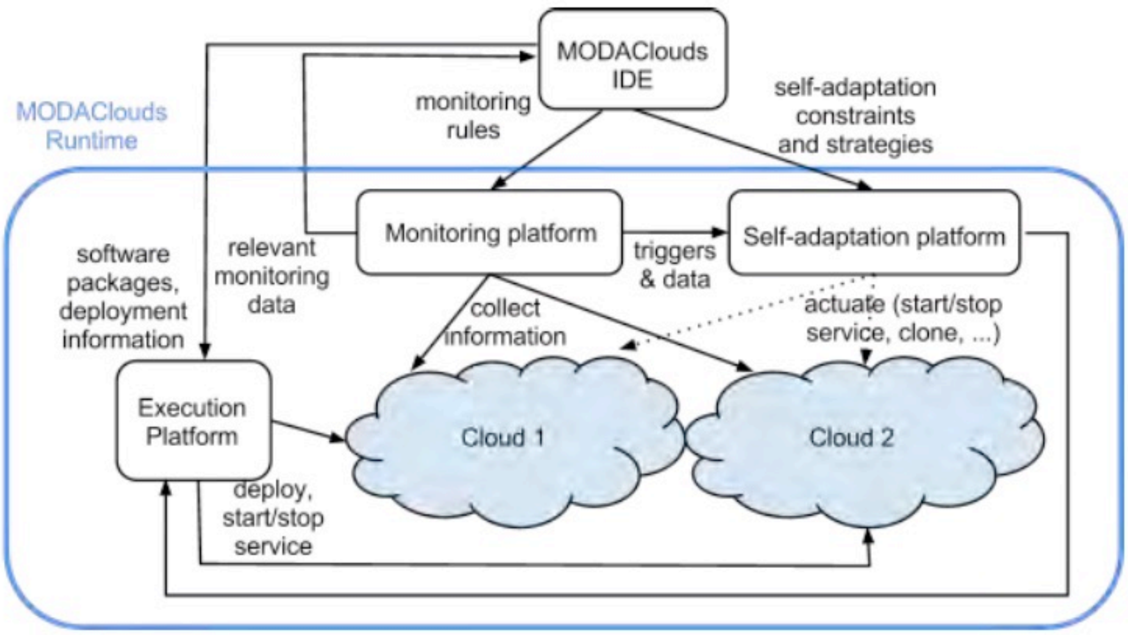


Figure 1: Conceptual Architecture of MODAClouds

In general, MODAClouds specifies:

1. A monitoring platform to characterize the state of applications developed and deployed using MODAClouds.
2. Self-adaptive policies to manage application QoS at runtime. These policies rely on models, shipped with the runtime environment, to perform predictions on application performance and scalability as well as to track or estimate its current status and resource demands.
3. An execution platform for managing application deployment, configuration, and run-time execution. This platform will be utilized by the self-adaptive policies to manage application QoS.

4. Data synchronization and load balancing mechanisms to support the execution of an application that is replicated over multiple clouds to ensure high-availability.

There are however, some important differences between CYCLONE and MODAClouds. CYCLONE applies optimal substitution models that could be used for re-deployment of applications. This is a unique feature of the CYCLONE platform. While MODAClouds investigate in detail the models to perform predictions on application performance, in our opinion, these models are complex, and while these could be potential used, there is always a question of the accuracy for some modelling approaches, in particular the queueing-theory based models. We have therefore chosen to build a library of anomaly-detection based solutions, and will investigate the efficiency of these algorithms. In particular, non-parametric models of anomaly detection require no a priori knowledge of underlying application performance models. This makes them easier to implement and apply within highly-variable cloud environments.

We also consider other optimization parameters than TREC.

3. Proposed Solution

The proposed solution for auto-scaling of cloud application to maintain a given Quality of Service (QoS) is illustrated in Figure 2. There are two core components of the solution: SlipStream and an Analytics Engine. SlipStream handles all aspects related to resource management for the application. The Analytics Engine encapsulates the knowledge necessary to decide what actions are required (if any) given the current state of an application. Both are described in detail in the subsequent chapters.

In the figure, there are three functionally equivalent applications (A, B, C) that are deployed within two different cloud infrastructures (one and two). Based on current conditions, application requirements, and user preferences, SlipStream ranks possible placement solutions to present the user with an ordered list of alternatives. In this case, application B deployed within cloud infrastructure 1 was the optimal choice.

When the deployment begins, SlipStream starts the resource monitoring. Once the application is operational, it too can provide application metrics, such as performance or response time. These metrics create a time-series of data that is streamed to the Analytics Engine for analysis. The Engine currently supports two different mechanisms for data analysis: anomaly detection and application optimization (via resource scaling, component migration, or application substitution).

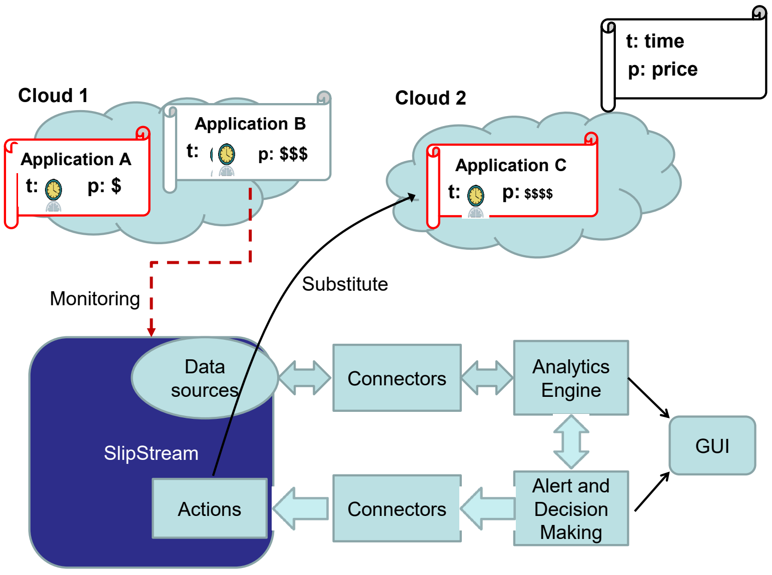


Figure 2: The QoS-control Architecture and Interaction with SlipStream

3.1. Anomaly Detection

Anomalies are items, events or observations that do not conform to an expected pattern or other items in a dataset. For example:

- Significant deviations in time series metrics,
- Significant changes in message rates,

- Rare or unusual log messages, or
- Unusual user behavior.

If the expected or normal patterns of behavior are known, anomalies can be identified. Given a time-series, the Engine models the time-series producing an expected value that may be later consumed by the anomaly detection methods. The anomaly detection methods include both outlier and change-point detection algorithms.

For an outlier detection, the algorithms identify a time-series element for which the observed value is significantly different from the expected value from the rest of the time series.

For change-point detection algorithms, the goal is to identify for a given time-series of observations whether the current point (or one in the near past) represents a change in the series. One desires to detect a potential change as soon as possible. In general, change-point detection methods monitor some test statistic, which is based on the observations, and issue an alarm if this test statistic exceeds a certain threshold, such that the calculated probability of not detecting a change-point is kept below a certain predefined value, for example, $\alpha = 5\%$. Statistically speaking, the change-point detection methods perform a hypothesis test for every time step. Change-point detection methods can be used to detect a change in mean or a change in the distribution of the time series, for example.

3.2. Application Optimization

As for the optimization, these algorithms are used to provide optimal policies that take as input the cost functions, response-time distributions, and (theoretically) reward/penalty functions. These policies further specify when to substitute one application for another and which application to substitute. The value to be optimized may change by application and actor; for example, it may be maximizing the revenue, for the provider), or minimizing the costs, for the application owner.

The optimization algorithms may also be triggered by the anomaly detection, for instance when a performance anomaly has been identified and a new policy must be derived. Once a new policy is available, a decision is then made as to whether to apply the newly derived policy or not. The decision-making process may be completely automated. If changes are necessary, the specified actions are then applied to the deployed application by SlipStream. In Figure 2, a decision is made to substitute Service B in Cloud one by Service C in Cloud two.

4. SlipStream Components for Auto-scaling

4.1. SlipStream Architecture

Non-functional requirements strongly influence the choice of cloud service providers (CSPs) for a given application deployment. These requirements include:

- Security guarantees or certifications of CSPs,
- Historical and current operational quality, and
- Application performance or response under varying loads.

The set of non-functional and operational requirements are often formalized as part of Service Level Agreements (SLAs) between the cloud application owners and cloud service providers. SlipStream, the CYCLONE component that deals with cloud application lifecycle management, provides the means to monitor such agreements and to react in case of violations.

The high-level architecture of SlipStream consists of four functional blocks:

- **Application Description Repository** Allows application developers to describe the functional and non-function requirements, software installation/configuration, and parameters for cloud components and multi-component cloud applications.
- **Service Catalog** Provides information concerning the administrative, financial, and operational characteristics of CSPs. The information is collected into "offers" that can then be selected based on both functional and non-functional requirements.
- **Deployment Engine** Combines application and CSP information to choose and then provision appropriate resources for an application. This engine handles the full lifecycle of the application including the initial deployment, scaling actions, and termination.
- **Monitoring** SlipStream monitors the state of all deployed cloud resources, allowing the deployment engine to control the state of resources, to raise alerts for abnormal conditions, and to provide usage information.

Users can access these functional blocks through a comprehensive API, a web browser interface, and a command line client. The web browser interface is roughly organized around the four functional blocks described above. The API is a REST-based API running over the HTTP(S) protocol. The API is currently migrating from a proprietary REST API [SAPI15] to one based on the CIMI standard from DMTF [CIMI16].

Figure 3 shows the functional blocks of the SlipStream server, access methods, and the underlying CSPs that provide resources for users' cloud applications.

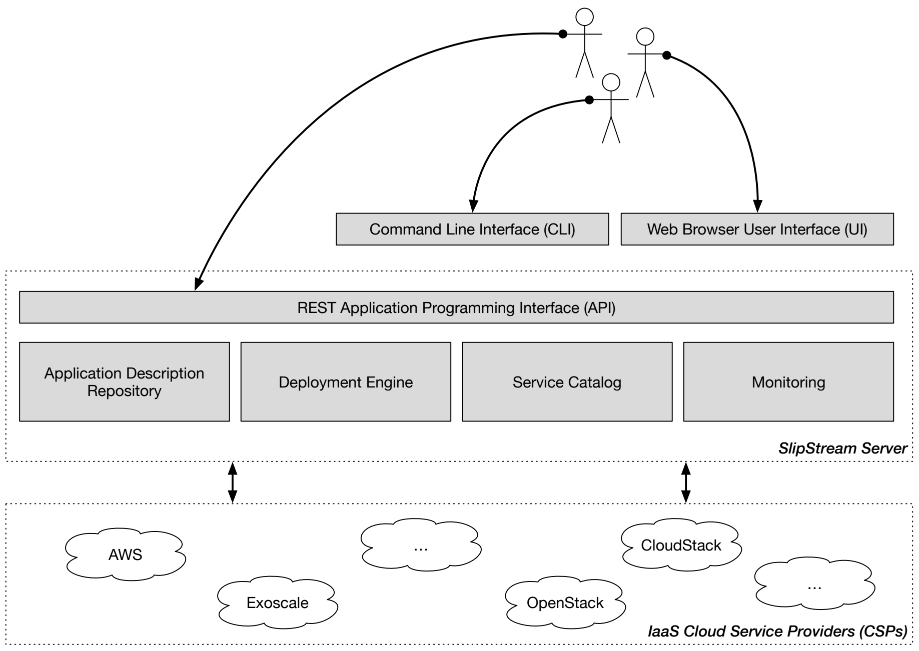

Command Line Interface (CLI)
Web Browser User Interface (UI)
REST Application Programming Interface (API)
Application Description Repository
Deployment Engine
Service Catalog
Monitoring
SlipStream Server
AWS
...
CloudStack
Exoscale
OpenStack
...
IaaS Cloud Service Providers (CSPs)

Figure 3: Functional Blocks of SlipStream

4.2. Application Description Repository

Running applications on any cloud system requires management of virtual machine images. In many cloud management systems, users generate these machine image files and then publish them to make them available on their cloud service provider (CSP). The management overhead associated with the transport, conversion, and evolution of these images discourages the use of multiple CSPs.

SlipStream takes a different approach: users specify the resource requirements, placement constraints, and the software installation and configuration procedures. SlipStream then uses this information to transform existing, minimal images optimized for each cloud provider into the customized VM requested by the user. This has two advantages: 1) the image descriptions are portable and can be used for any cloud supported by SlipStream and 2) all knowledge about the application is captured and managed.

On clouds that support customized user images, binary image files can be produced ("built") by SlipStream to reduce the startup latency. Users must explicitly request the build of these binary image files, but once produced, SlipStream will use them automatically. This maintains cloud portability while allowing users to optimize for particular CSPs.

SlipStream provides a "workspace" in which users manage their application and application component descriptions. These descriptions can be shared with other users. In addition, system administrators can publish vetted applications into an "App Store" to make them visible to all SlipStream users. Applications in the App Store can be found easily and launched with a "single click".

Within the component definitions, the resource requirement and placement constraints directly affect the CSPs that are chosen. The placement constraints can include security, location, availability, and other non-functional requirements to support the definition and enforcement of SLAs.

4.3. **Deployment Engine**

The deployment engine orchestrates the entire process to bring a cloud application into a working state, including choosing appropriate cloud resources and provisioning them. The implementation maintains a finite state machine for each application and manages the transitions between the defined states.

The Placement and Ranking Service (PRS), part of the deployment engine, matches application requirements and constraints to available CSP offers. It first filters the offers to remove those that do not meet the minimum resource requirements and then ranks the acceptable offers by price, from lowest to highest. Automated provisioning choses the lowest price offer; manual provisioning can choose any offer that passed the filtering.

The deployment engine must interact with CSPs to provision the resources required for an application. SlipStream internally uses a uniform, abstract interface to handle interactions with cloud service providers. “Cloud connectors” then implement this interface, allowing SlipStream to seamlessly support a wide range of different cloud service providers.

All applications need to provide information to its users (e.g. the endpoint of the service) and/or take configuration options from the operator (e.g. what database to configure). Moreover, there is often a need to coordinate the configuration of different machines in an application, for example, ensuring that a database client does not start before the database is ready. To meet these needs, SlipStream provides a parameter “database” for each running application instance. All parts of SlipStream, as well as the users and their applications, can access this database to exchange information.

The following sections provide more details on the placement, provisioning, and coordination aspects of the deployment engine. Of the three, the PRS is the most crucial for handling functional and non-functional requirements of cloud applications.

4.3.1. Placement and Ranking Service

The PRS, a micro-service within the SlipStream server, selects acceptable offers from CSPs. It has a simple API, whereby cloud application requirements in JSON format are passed to the PRS via an HTTP PUT and the service responds with a ranked list of clouds, also in JSON format. SlipStream comes with its own implementation of this service but alternate implementation can be used by changing the SlipStream configuration.

To perform its function, the PRS interacts with the service catalog, filtering and ranking the offers it finds there. The PRS obtains information about the application and user cloud configuration through the input document. Consequently, it does not have any direct interaction with the application description repository or the other functional blocks of SlipStream.

The first operation is the filtering of service offers. The filtering only accepts offers for clouds that the user has configured and from those only selects offers based on either: capacity requirements or an exact match of an instance type or “flavor”.

Capacity requirements are expressed in terms of CPU/RAM/Disk at the component level. Only service offers with CPU/RAM/Disk values above the component requirements are selected. As some clouds do not use fixed values for CPU/RAM/Disk, this rule can be overridden with the `schema-org:flexible` attribute set to `true`.

```
{
  "components" :
    [{
       "module" : "moduleURI",
       "cpu.nb" :  1,
       "ram.GB" :  2,
       "disk.GB" : 10,
       "placement-policy" :  "<optional CIMI filter>",
       "connector-instance-types" :
         {
           "connector-name1": "<size1>",
           "connector-name2": "<size2>"
         }
    }],
  "user-connectors" : ["connector-name1", "connector-name2"]
}
```

When specifying an instance type or "flavor" for a component, only offers with the exact instance type are retained. If both the capacity requirements and an instance type are defined, the instance type filtering takes precedence.

A component may also specify a placement policy, such as:

```
{"placement-policy": "schema-org:location='ch'"}
```

which would select only those offers that provide resources in Switzerland. Despite the name "placement", this policy is completely general and can filter on any characteristics of a CSP offer. If the offer provides security certification information or availability, for example, those values can be used to select acceptable services. These policies are applied in addition to (i.e. with a logical "and") to the resource requirements.

For each connector in the filtered list the ranking process calculates a metric. Currently this metric is the estimated price per hour in euros. The list is enriched with this information and then sorted by increasing price. (Offers without prices are put at the end of the list.) The result is then returned.

The detailed JSON input and output formats are shown below. Note that connector-instance-types attribute contains the preferred instance types per connector.

The 'node' attribute is only used for applications (i.e. multi-component deployments); it is not present for deployments of simple components. Ordering is indicated with the 'index' attribute. When price is not available, the value -1 is returned.

```
{
  "components" :
    [{
        "module" : "moduleURI",
        "cpu.nb" :  1,
        "ram.GB" :  2,
        "disk.GB" : 10,
        "placement-policy" :  "<optional CIMI filter>",
        "connector-instance-types" :
          {
            "connector-name1": "<size1>",
            "connector-name2": "<size2>"
          }
      }],
  "user-connectors" : ["connector-name1", "connector-name2"]
}
```

```
{
  "components" :
    [{
        "node":  "<unused-when-not-an-application>",
        "module":  "moduleURI",
        "connectors":
          [{
             "name": "connector-name2",
             "price": 0.0171,
             "currency": "EUR",
             "cpu":  2,
             "ram": 8,
            "disk": 10,
            "instance_type": "micro",
            "index": 0
          },
          {
            "name": "connector-name1",
            "price": 0.0185,
            "currency": "EUR",
            "cpu": 4,
            "ram": 32,
            "disk": 200,
            "instance_type": "medium",
            "index": 1
          }]
     }]
}
```

4.3.2. Provisioning

The provisioning (and freeing) of cloud resources is handled by the SlipStream server. SlipStream manages a cloud application's lifecycle by driving it through the well-defined set of states.

For components (simple applications with only one machine), SlipStream itself will directly start the machine in the cloud. However, for applications with multiple machines, SlipStream delegates some of the lifecycle management to an additional machine called an "orchestrator". The orchestrator is responsible for provisioning all the machines of the application and will also handle scaling actions. For deployments that will not scale, the orchestrator is terminated once the application is ready; for scalable applications, the orchestrator remains to handle provisioning when scaling requests are made.

Throughout the application lifecycle, SlipStream executes and synchronizes the provisioning and configuration actions following a two-level state machine. Figure 4 shows the main state machine with its transitions and synchronization.

The secondary state machine only applies to scalable deployments during a scaling action. When a scaling action is initiated, the main state machine moves to the "Provisioning" state, where the scaling actions will be applied. The state machine then continues until it returns to the "Ready" state. Figure 5 shows the secondary state machine; the elements in blue only apply to vertical scaling actions.

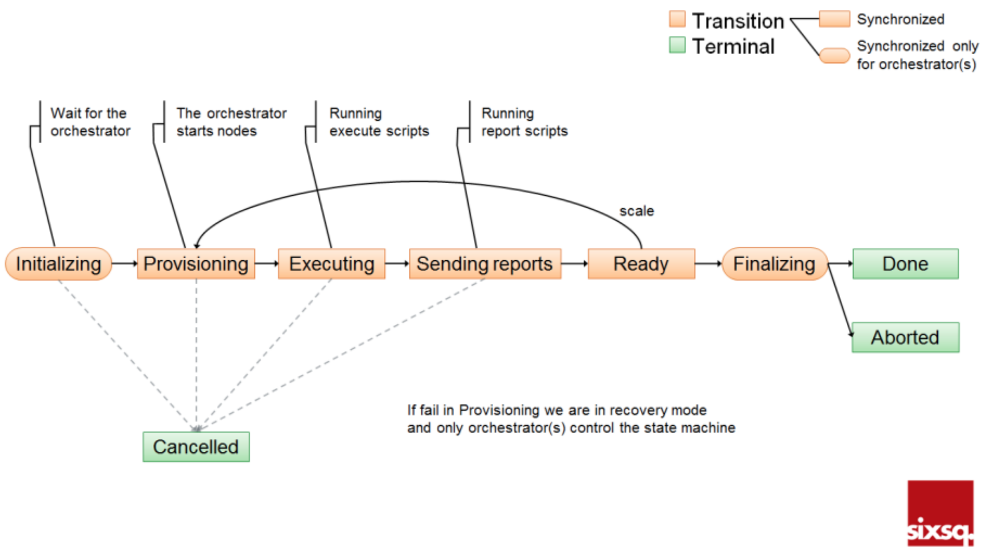

Transition
Terminal
Synchronized
Synchronized only for orchestrator(s)
Wait for the orchestrator
The orchestrator starts nodes
Running execute scripts
Running report scripts
scale
Initializing
Provisioning
Executing
Sending reports
Ready
Finalizing
Done
Aborted
Cancelled
If fail in Provisioning we are in recovery mode and only orchestrator(s) control the state machine
sixsq

Figure 4: Deployment States

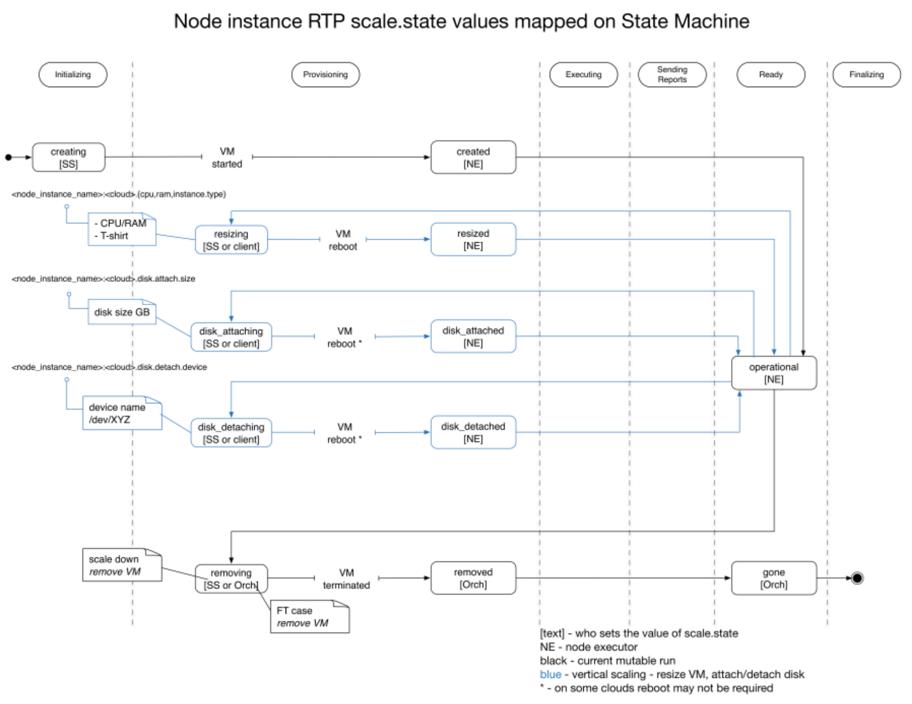

Node instance RTP scale.state values mapped on State Machine
Initializing
Provisioning
Executing
Sending Reports
Ready
Finalizing
creating [SS]
VM started
created [NE]
<node_instance_name>:<cloud>.(cpu,ram,instance.type)
- CPU/RAM
- T-shirt
resizing [SS or client]
VM reboot
resized [NE]
<node_instance_name>:<cloud>.disk.attach.size
disk size GB
disk_attaching [SS or client]
VM reboot *
disk_attached [NE]
operational [NE]
<node_instance_name>:<cloud>.disk.detach.device
device name /dev/XYZ
disk_detaching [SS or client]
VM reboot *
disk_detached [NE]
scale down remove VM
removing [SS or Orch]
VM terminated
removed [Orch]
gone [Orch]
FT case remove VM
[text] - who sets the value of scale.state
NE - node executor
black - current mutable run
blue - vertical scaling - resize VM, attach/detach disk
* - on some clouds reboot may not be required

Figure 5: Scaling States

4.3.3. Coordination

For each running instance of a component or application, SlipStream maintains a simple key-value database, which is used to share information between SlipStream, managed virtual machines, and the operator of the application. This “parameter database” can be accessed via the web browser UI, the SlipStream CLI, and the SlipStream API.

Table 2 describes the most important parameters that are managed by SlipStream or treated specially. The table does not include all parameters, nor parameters defined by specific components.

The global parameters provide information about the complete application. Generally, the parameters like “ss:state” and “ss:abort” are used to control the state machine of the application. Others like “ss:url.service” are treated as URLs within the browser interface to facilitate access to deployed applications. Those that are not read-only, can be set by the user or the application itself through the SlipStream API or the `ss-set` command in the CLI.

SlipStream provides some “automatic”, read-only parameters for the virtual machines within a deployed application. The most important are “hostname” and “instanceid” that provide the machine’s IP address and cloud instance ID, respectively.

A “node” in a SlipStream deployment refers to a collection of machines of the same type (or class) and in the same cloud infrastructure. There are a few parameters set by SlipStream that make it possible to iterate over the machines within the node class.

Application components can define any number of input and output parameters. All parameters, including the user-defined ones, can be accessed via the SlipStream command line client using the commands `ss-get` and `ss-set`. Note that trying to access an undefined parameter will abort the deployment.

The `ss-get` command will, by default, block (subject to a timeout) if the parameter has not yet been given a value. This allows parameters to act as simple semaphores to coordinate the behavior between virtual machines within a given deployment. It is common for components to define "ready" flags to facilitate this cross-component coordination. For example, a database component may provide a "ready" flag to indicate to clients when the database is available.

The parameter database probably would not be used directly in the management of SLAs, but could be used indirectly to provide, for example, the number of failed machines or the current number of jobs in a batch queue. Note, however, that the values would need to be set directly by the application. Similarly, triggering state changes must be done externally via the API.

Table 2: SlipStream Application Parameters

Name	Scope	Read-only	Description
ss:state	Global	Yes	Current state of the application
ss:abort	Global	No	Global abort flag for the application type of deployment
ss:category	Global	Yes	Type of deployment
ss:tags	Global	No	A list of tags set by the user to differentiate deployments
ss:url.service	Global	No	A service URL for the overall application
Ids	Node	Yes	List of IDs for machines in this node class
Multiplicity	Node	Yes	Number of machines in this node class
Hostname	Machine	Yes	Contains the IP address of the VM
Instanceid	Machine	Yes	The cloud-specific identifier for the VM
Abort	Machine	No	Flag indicating if the application has aborted

4.4. Monitoring/Optimization Infrastructure

SlipStream monitors the state of virtual machines deployed as part of a cloud application, both during and after the deployment. During the deployment process, the information helps synchronize the installation and configuration actions; afterwards, it is displayed on the users' dashboards to provide an overview of their cloud activity. As a deployment works its way through the SlipStream state machine, SlipStream generates "events" with information about the progress. Users and users' applications can similarly generate their own specialized events for their applications. All the events can be accessed from the web browser UI, CLI and the various APIs. Eventually, notifications based on these events will be supported, allowing corrective actions to be triggered based on the state of the application.

In parallel, SlipStream periodically polls the cloud infrastructures to obtain information about users' running virtual machines. The information is correlated with the known virtual machines from SlipStream deployments to provide an overall view of resources both managed and not managed by SlipStream. This information is available on the web browser UI dashboard and through the CLI and APIs. Eventually, discrepancies between the expected and actual deployment states will be reported as events, allowing notifications and automated corrective actions. Similarly, availability statistics will be published into the service catalog, influencing the selection of CSPs for future deployments or scaling actions.

The monitoring will play a crucial role for deploying self-adapting and self-healing applications on clouds. This provides the feedback on the current state of the application that will be the basis for triggering scaling, migration, or other actions.

4.5. Service Catalog

The Service Catalog is a repository of characteristics (stable information) and dynamic properties associated with a given cloud infrastructure. It is composed of three REST resources following the CIMI standard. The main one is the Service Offer, the two others (Service Attribute and Service Attribute namespace) provide semantic information and namespace functionalities.

4.5.1. Sources of Information

This information is inserted (or updated) into the Service Catalog both on regular basis ("Information Providers" that feed the prices) and by SlipStream itself continuously (via the monitoring).

- **Application monitoring from SlipStream itself.** SlipStream continuously monitors the state of virtual machines within deployed cloud applications. Currently this information is only available internally, so SlipStream would need to be modified to make this information available through the API. This information could be used to trigger deployments (with optional migration) of application components.
- **Price information for clouds.** "Information Providers", that is, processes that parse web page information and regularly update price values. The typical refresh rate of these data is slow.
- **Application-specific benchmarks.** The applications themselves can publish specific information about the performance of a cloud infrastructure for a given application. This could be used to influence future scheduling decisions for the application. It could also be used to initiate changes in the current application instance as well.
- **Application metrics.** These are application-specific metrics that are taken frequently and used inside the deployment to decide on scaling actions. These metrics typically change too quickly and are too specific to an application instance to push into the service catalog.
- **External information.** Information such as security certifications (e.g. from the Cloud Security Alliance) or generic benchmarks (e.g. Cloud Harmony) can be inserted into the Service Catalog or Open Service Compendium to provide a richer set of information to use for resource selection. This may also be extended to "endorsements" where a third-party would certify certain cloud services.

4.5.2. Service Catalog Resources

The most important resource is the Service Offer that contain actual offers from CSPs. The other resources support Service Offer by providing namespaces (to avoid name collisions across all service offers attributes) and meta-data information (mainly a description of an attribute). Each attribute name for a Service Offer must be of the following form: `prefix:name`. For example, `schema-org:location` is a valid attribute name if a Service Attribute Namespace resource with this prefix exists. Each service attribute namespace prefix is guaranteed to be unique.

4.5.2.1. Offers

Self-contained offers for specific services that are associated with a single, specific price calculation. An example, would be a VM with a specific size and duration. Another would be a persistent storage area with a given size, SLA, etc. The schema for a Service Offer is very flexible. This is basically a key value store (except for the `connector/href` attribute that identifies the name of the cloud).

```
{
    "connector" : {
      "href" : "nuvlabox-christiane-nusslein-volhard"
    },
    "schema-org:last-online" : "Tue Oct 11 08:52:41 UTC 2016",
    "schema-org:state" : "ok",
    "schema-org:flexible" : "true"
}
```

4.5.2.2. *Attributes*

The semantic definition of a single attribute. This associates with a URI with the semantic definition of the attribute. This also allows internationalization of the description of the attribute.

```

    "prefix":            "NonBlankString",
    "attr-name":         "NonBlankString",
    "type":              "type",
    "authority":         "NonBlankString",
    "major-version":     "NonNegInt",
    "minor-version":     "NonNegInt",
    "patch-version":     "NonNegInt",
    "normative":         "true-or-false",
    "en":                {
                             "name":  "Name of the attribute",
                             "description": "Description of the attribute",
                             "categories": ["category-one", "category-two"]
                         },
    "fr":                {
                             "name":  "Nom de l'attribut",
                             "description": "Description de l'attribut",
                             "categories": ["categorie-un", "categorie-deux"]
                         }

}
```

4.5.2.3. *Namespaces*

Associates a URI with a given prefix. Note that the pair prefix/URI must be unique across all Service Attribute resources. The JSON structure for Service Attribute Namespace with the CIMI common attributes stripped out looks like:

```
{
  "prefix":  "a-prefix-without-dot-or-slash",
  "uri":  "NonBlankString"
}
```

4.5.3. Filtering/Query Language

The current implementation uses the CIMI filtering [CIMI16] with the filter query parameter to select suitable service offers. For example, the following expression:

```
((connector/href='connector-name1') or (connector/href='connector-name2'))
and
((schema-org:flexible='true') or
((schema-org:descriptionVector/schema-org:vcpu>=4 and
  schema-org:descriptionVectorschema-org:ram>=16 and
  schema-org:descriptionVectorschema-org:disk>=50)))
```

will select connectors named “connector-name1” or “connector-name2” and with CPU/Ram/Disk above 4/16/50, or with a flexible schema.

The use of native Elastic Search API could bring more flexible ways of querying the Service Offer. For example, it would be possible to use fuzzy matching, phrase or proximity queries, etc. Whether this is required will depend on the current and future use cases.

4.6. Accessing SlipStream

The many features of SlipStream would not be useful unless they can be accessed easily. Fortunately, SlipStream provides three mechanisms for accessing the service:

- **REST API:** A complete API that allows programmatic access to the service. The API follows recent trends and provide a resource-oriented or REST API that works over the HTTP protocol. This allows easy access from all programming languages without having to provide customized libraries for each.
- **Web Browser UI:** A convenient graphical method of accessing the service and allowing people to manage information concerning their cloud applications, notably in the context of this document, managing the placement policies that contain the functional and non-functional requirements for the applications. Uses the REST API of the service.
- **Command Line Interface (CLI):** Access from the command line, either for quick tests or for scripting, is essential for any complete solution. SlipStream provides a CLI written in Python to make its deployment and use nearly universal. The CLI uses the SlipStream REST API and exposes most of the available functionality.

All three of these access methods play roles in the solutions for functional and non-functional aspects of cloud application management. Specifically, the web browser interface allows policies to be easily written and associated with application components. The REST API and CLI facilitate the manual and automated scaling of cloud applications (see following sections) in response to operating conditions, whether it be the changing load on the service or changing characteristics of the CSPs.

4.7. Auto-scaling of User Applications

With SlipStream one can automatically scale up and down the number of instances of an application component of a (possibly) complex application. Currently only one component can be scaled at a time. Work to remove this limitation is ongoing. There are two ways to take advantage of auto-scaling with the existing SlipStream implementation:

- **Black-box autoscaling:** The simplest approach is to use the default implementation of the auto-scaling in SlipStream. Currently, it is suitable for the applications where the only one component will scale based on only one metric. If these requirements are met, adding a special autoscaler component to the user’s application, providing configuration file with the application component scalability constraints, and deploying the application as a scalable deployment allows user to benefit from the automated scalability provided “out of the box” by SlipStream.
- **Do-it-yourself (DIY) autoscaling:** The implementation of the autoscaler in SlipStream allows users to supply their own implementations of the autoscaling logic. In this case, SlipStream takes care of the deployment of the autoscaler platform and running the user supplied auto-scaling logic. What is required from the user is the inclusion of the autoscaler in the user’s application as a component and providing a public URL with the autoscaling implementation.

In the above cases, it is required that chosen components of the user’s application publish metrics on which the scaling actions will be based and to support the automatic scaling.

SlipStream uses Riemann [RIES16] to implement its auto-scaling decision making feature and consequently requires that the metrics be published as Riemann events. To facilitate metrics collection, Riemann has a wide range of metric publishers [RIEC16]; these include a Riemann plugin for “collectd” and a Python CLI and library API.

The implementation of the autoscaler in SlipStream is completely open and flexible. It allows users to write their own decision making logic as Riemann streams and to provide it to the autoscaler component as an input parameter. This can be useful in cases when the default autoscaling implementation, that comes with the SlipStream autoscaler, does not fully satisfy the needs of the user's application.

4.7.1. Configuration of Auto-scale Constraints

Below is the example configuration file (in edn format, see [EDN12]) to be used with the black-box autoscaling approach. The configuration defines scalability constraints for an application component called *webapp*.

```
[
 {
  ;;
  ;; Mandatory parameters.

  ; name of the component in the application
  :comp-name         "webapp"

  ; service tags as sent by Riemann publisher in the event
  :service-tags      ["webapp"]

  ; monitored service metric name (as sent with Riemann event)
  :service-metric    "avg_response_time"

  ; value of the metric after which start adding component instances
  :metric-thold-up   500.0
  ; value of the metric after which start removing component instances
  :metric-thold-down 200.0

  ; max number of component instances allowed
  :vms-max           4   ; "Price" constraint.

  ;;
  ;; Optional parameters (with defaults).

  ; min number of component instances allowed
  ;:vms-min       1

  ; number of instances by which to scale up
  ;:scale-up-by   1
  ; number of instances by which to scale down
  ;:scale-down-by 1
 }
]
```

It reads the following way:

1. For an application component *webapp* the autoscaler expects to receive Riemann events:
    1. with the service name *avg_response_time*,
    2. one of tags being *webapp*;
2. When the value of the metric in the event is
    1. Above *metric-thold-up* (500.0), then the autoscaler should perform a *scale up* action,
    2. Below *metric-thold-down* (200.0), then the autoscaler should perform a *scale down* action;
3. The autoscaler should not perform *scale up* action if there are already *vms-max* (4) component instances running,
4. The minimum number of component instances should not go below *vms-min*. That is, the autoscaler should not attempt a *scale down* action if the number of the currently running component instances is *vms-min*. (1),
5. The autoscaler should scale up by *scale-up-by* number of instances and scale down by *scale-down-by*.

4.7.2. Autoscaler Component

The autoscaler component is available in AppStore on Nuvla[1], the SlipStream "as a Service" run by SixSq (see Figure 6). Its source code is published on GitHub in the "slipstream-auto-scale-apps" repository[2] in the "autoscaler" module. The structure of the autoscaler application is:

```
app/
   dashboard.json                # Configuration of the layout of Riemann-dash.
   dashboard.rb                  # Configuration of the Riemann-dash service.
   event-example.json            # An example of Riemann event in JSON.
   project.clj                   # Clojure project file for working with Riemann streams file.
   riemann-ss-streams.clj        # SlipStream scaling logic as Riemann streams.
   scale-constraints-example.edn # Example scale constraints.
deployment/
   deployment.sh                 # Deployment of the autoscaler component.
```

The application uses Riemann to process incoming component metrics as events. The main part of the application is the Riemann configuration file, `app/riemann-ss-streams.clj`, a script is written in Clojure programming language.

The default implementation loads the auxiliary library (as the "sixsq.slipstream.riemann.scale" Clojure namespace) that defines custom event processing streams and helper functions. It then uses those functions to read in the scalability constraints for the component(s), to process/update incoming events (or generate new events), to make the scaling decisions, and to request the scaling actions from SlipStream.

The decision-making algorithm uses Riemann's moving time window stream with a window size of 30s to smooth out spikes in the incoming metrics' time series.

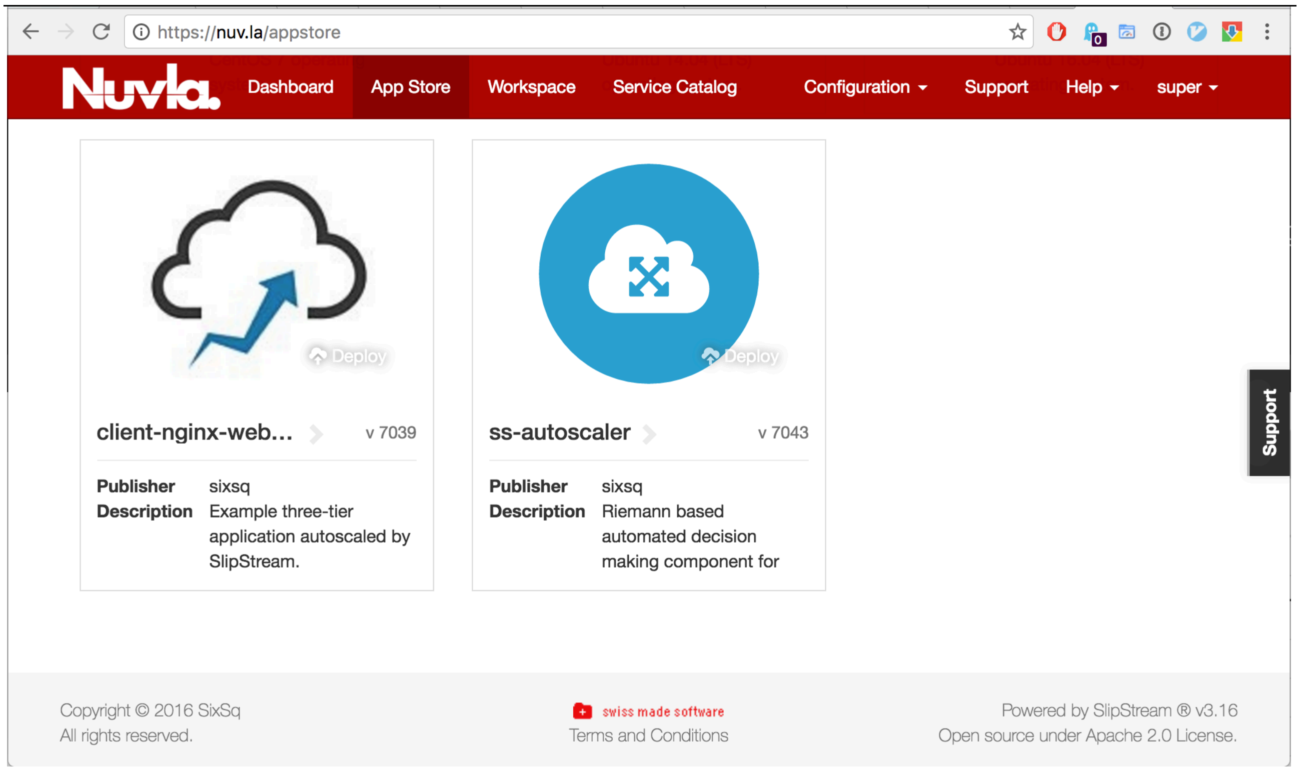
The functions defined in the "sixsq.slipstream.riemann.scale" namespace simplify the main configuration script by providing a number of utility functions that hide the details of interacting with SlipStream to request scaling actions, creating new and/or updating old events, and (re-)indexing and/or publishing them to Graphite. The namespace is defined in a the "run-proxy/api" module in the GitHub repository. All the public functions are documented.

Based on the example in `app/riemann-ss-streams.clj` and taking advantage of the utility functions, application developers can write their own scaling logic and then provide it to the autoscaler component as a public URL via an input parameter. For details, see the autoscaler component in the AppStore.

The deployment script of the autoscaler module deploys Riemann, the Riemann dashboard, and Graphite. After deployment, the Riemann dashboard can be found at the URL http://<autoscaler IP>:6006 and Graphite, at http://<autoscaler IP>.

Riemann acts as a stream processing engine with no or little memory of the events. Graphite is used to store the historical data of the events. Primarily this is used to plot the historical data, but could also be read into the Riemann streams to consider the history when making scalability decisions.

[1] https://nuv.la
[2] https://github.com/slipstream/slipstream-auto-scale-apps



Publisher sixsq
Description Example three-tier application autoscaled by SlipStream.

Publisher sixsq
Description Riemann based automated decision making component for

Copyright © 2016 SixSq
All rights reserved.

Powered by SlipStream ® v3.16
Open source under Apache 2.0 License.

Figure 6: Autoscaler and Scalable Application

4.7.3. Publishing Component Metrics to Autoscaler

The autoscaler makes the scalability decisions based on the metrics coming from the user's application components and the thresholds provided by the user. Because the autoscaler is a Riemann application, the user must use one of the many Riemann clients to push the metrics (as events) to it. (See [GAL07] for the available clients.)

4.8. Example Auto-scale Application

The following section describes the example auto-scale application that uses write Riemann plugin of collectd and a custom publisher written in Python using Riemann client library API.

The example auto-scale application is a web application that uses the Riemann collectd plugin and a custom publisher written in Python and the Riemann Client API. It is schematically depicted in Figure 7 and consists of the following components:

- **webapp:** a stateless web application that takes requests, synchronously performs a moderately intensive computation (calculating Pi up to 100 digits), and returns the result,
- **nginx:** a load balancer based on Nginx [NGIN16] that distributes client requests to the set of stateless web servers,
- **client:** a test client based on Locust [LOC16] that simulates a varying number of clients, and
- **autoscaler:** Standard SlipStream autoscaler component that makes scaling decisions.

The application can be found in the AppStore on Nuvla (see Figure 6); its source code is in the "client-nginx-webapp" module in the GitHub repository.

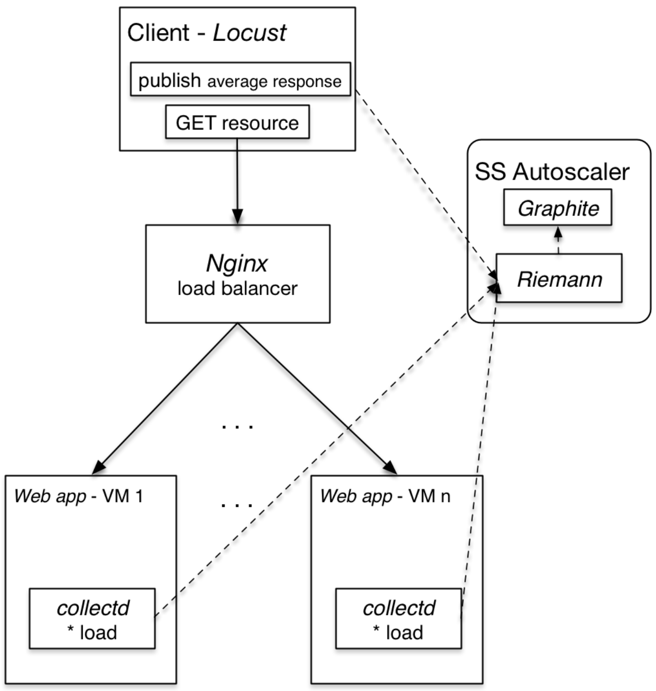


Figure 7: Components of Example Auto-scale Application

4.8.1. Application Configuration and Deployment

Figure 8 shows the definition of the application in SlipStream. Note that values for the autoscaler input parameters, namely, "riemann_config_url" and "scale_constraints_url" must be provided. The first one defines the URL with a resource under which the following files are expected:

- `riemann-ss-streams.clj`: Riemann streams using SlipStream scale up/down actions,
- `dashboard.json`: Riemann dashboard layout and queries, and
- `dashboard.rb`: Riemann dashboard configuration.

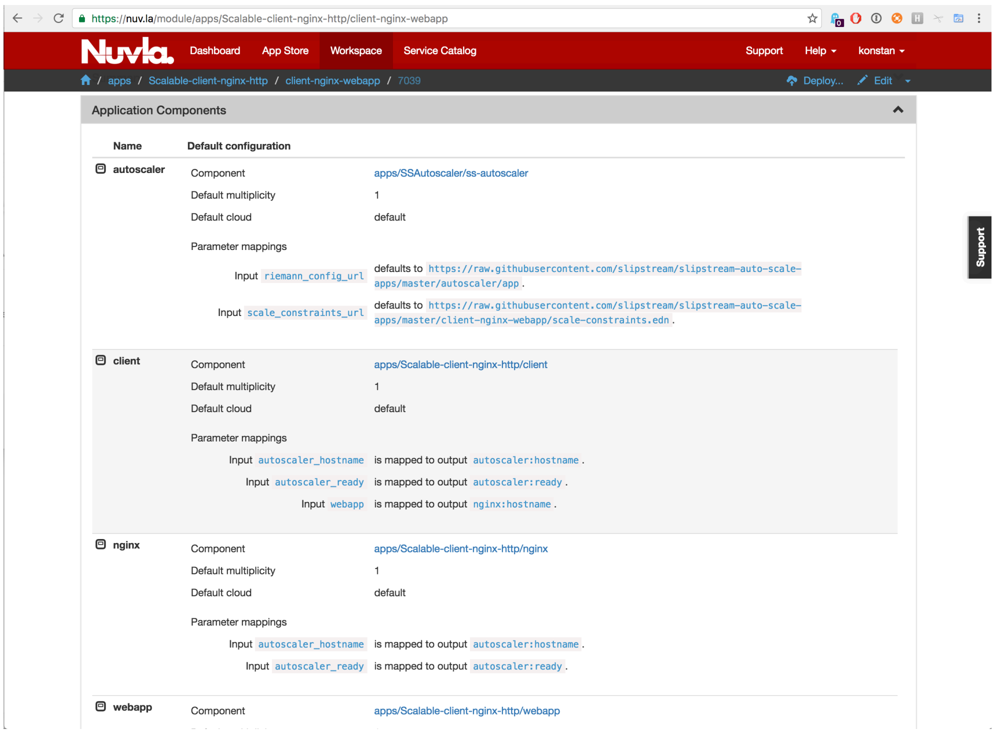
They contain the processing logic for autoscaling actions and the configuration for the Riemann dashboard. This allows users to provide their own implementation of the scaling logic and dashboard configuration. The second input parameter "scale_constraints_url" provides the URL with the application scaling definitions. For this application, they look like:

```
$ cat scale-constraints.edn

[
 {:comp-name          "webapp"
  :service-tags       ["webapp"]
  :service-metric     "avg_response_time"
  :metric-thold-up    7000.0
  :metric-thold-down  4000.0
  :vms-max            4}
 ]
```

This file is application-specific; for the example, it comes from the user application module on GitHub.

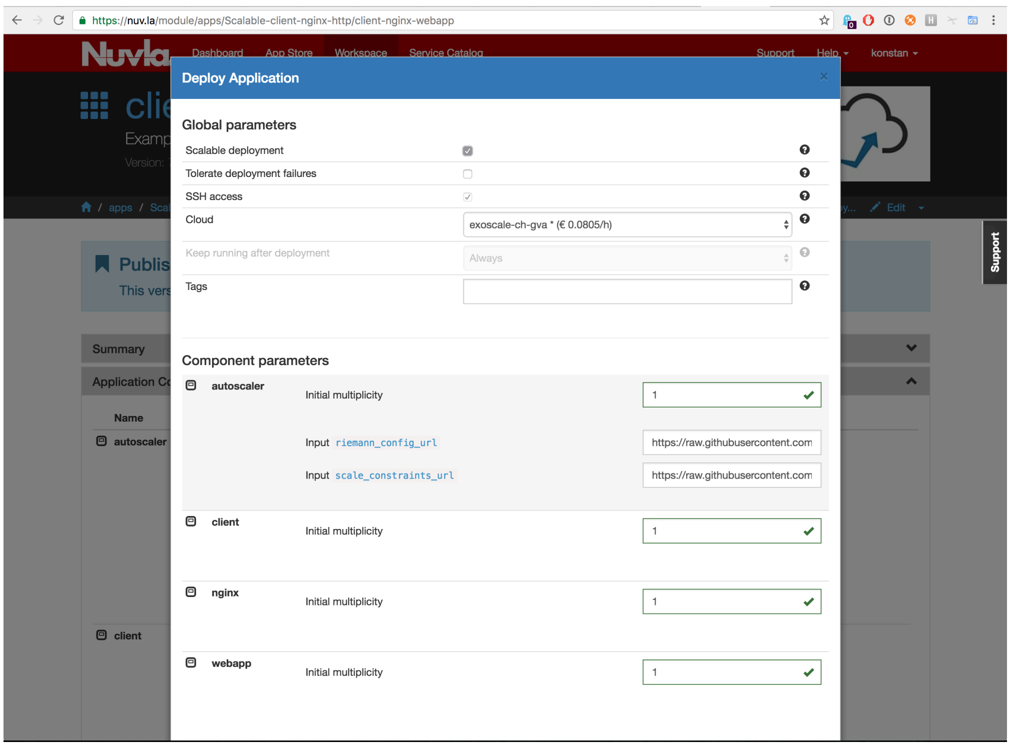
Clicking on Deploy button brings the Application Deployment dialog (see Figure 9). In this dialog, you must check the box to indicate that this is a scalable application; you can optionally change the multiplicity of the *webapp* component. Select the cloud and proceed with the deployment.


https://nuv.la/module/apps/Scalable-client-nginx-http/client-nginx-webapp
Nuvla
Dashboard
App Store
Workspace
Service Catalog
Support
Help
konstan
apps / Scalable-client-nginx-http / client-nginx-webapp / 7039
Deploy...
Edit
Support

Application Components

Name	Default configuration	
autoscaler	Component	apps/SSAutoscaler/ss-autoscaler
	Default multiplicity	1
	Default cloud	default
	Parameter mappings	
	Input `riemann_config_url`	defaults to `https://raw.githubusercontent.com/slipstream/slipstream-auto-scale-apps/master/autoscaler/app`.
	Input `scale_constraints_url`	defaults to `https://raw.githubusercontent.com/slipstream/slipstream-auto-scale-apps/master/client-nginx-webapp/scale-constraints.edn`.
client	Component	apps/Scalable-client-nginx-http/client
	Default multiplicity	1
	Default cloud	default
	Parameter mappings	
	Input `autoscaler_hostname`	is mapped to output `autoscaler:hostname`.
	Input `autoscaler_ready`	is mapped to output `autoscaler:ready`.
	Input `webapp`	is mapped to output `nginx:hostname`.
nginx	Component	apps/Scalable-client-nginx-http/nginx
	Default multiplicity	1
	Default cloud	default
	Parameter mappings	
	Input `autoscaler_hostname`	is mapped to output `autoscaler:hostname`.
	Input `autoscaler_ready`	is mapped to output `autoscaler:ready`.
webapp	Component	apps/Scalable-client-nginx-http/webapp

Figure 8: Example Scalable Application Definition in SlipStream



Deploy Application

Global parameters

Scalable deployment	✓
Tolerate deployment failures	☐
SSH access	✓
Cloud	exoscale-ch-gva * (€ 0.0805/h)
Keep running after deployment	Always
Tags	

Component parameters

autoscaler	Initial multiplicity	1
	Input `riemann_config_url`	https://raw.githubusercontent.com
	Input `scale_constraints_url`	https://raw.githubusercontent.com
client	Initial multiplicity	1
nginx	Initial multiplicity	1
webapp	Initial multiplicity	1

Figure 9: Deployment of Example Scalable Application

4.8.2. Usage of the Application After Deployment

After a successful deployment of the application one should first open the HTTP URL published by the client component. It provides a page with a description of the application, a deployment diagram, and links to the services running on other components (see Figure 10).

The first link to follow is the one pointing to the Locust load generator; this simulates a varying number of clients. Although Locust can be used as an automatic load generator, this application assumes manual configuration of the load parameters. Figure 11 shows the definition of three parallel users that Locust will use to access the web layer and load it with requests. The endpoint to contact and the resource of the web application were already automatically configured during the component deployment.

After Locust starts loading the web application, a custom Riemann events publisher collects the average response time metric from Locust and publishes it as an event to the Riemann service running on the autoscaler. The event looks like:

```
{"service": "avg_response_time",
  "tags": ["webapp"],
  "ttl": 10,
  "host": "httpclient",
  "time": 1479202972,
  "metric_f": 3167.896}
```

Other metrics are also collected and sent from client (Locust) and *webapp* component instances. For example:

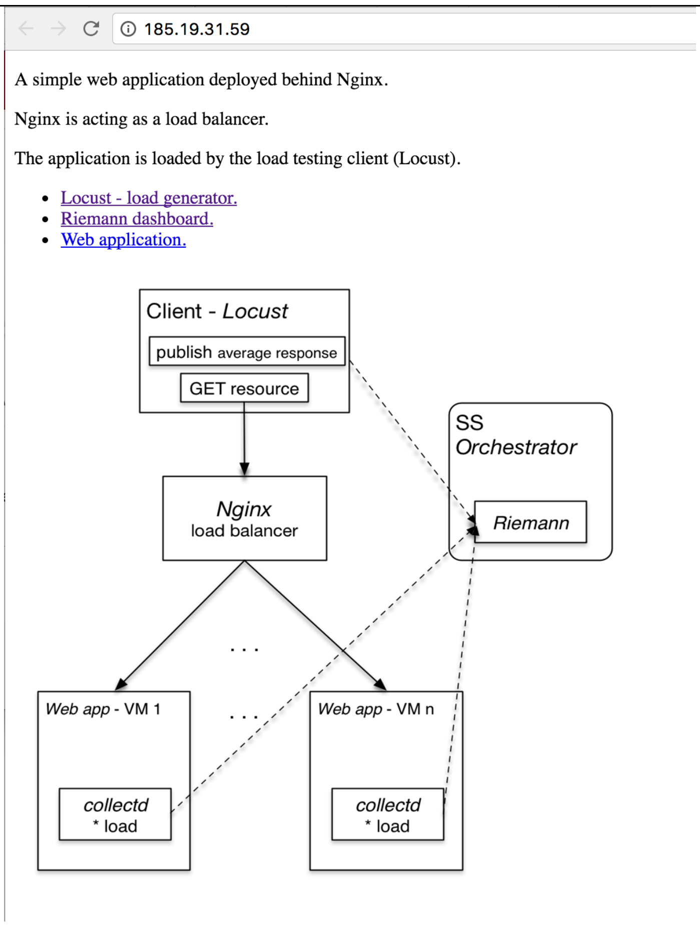
- The *client* publishes number of concurrent clients used by Locust, current requests per second, and the request fail rate;
- The *webapp* instances publish their current 1, 5, and 15 min load. This metric is published through the Riemann collectd plugin. This is deployed and configured automatically on each *webapp* with the `collectd.sh` script available from the repository.

Based on the load metrics reported by the *webapp* instances, a Riemann stream dynamically calculates the current number of the instances and indexes it to allow the Riemann dashboard to query and display it.

The current multiplicity of the *webapp* component is queried by a Riemann stream directly from SlipStream and indexed. As one can see in Figure 12, there is a delay between the provision request (multiplicity as reported by SlipStream) and availability of the virtual machine (as reported by the load metrics). This is almost entirely due to the provisioning latency on IaaS level; SlipStream's control flow contributes negligibly to the latency.

Based on the given constraints, the autoscaler attempted to satisfy the scalability constraints provided for the *webapp* component by keeping the average response time metric within the requested bounds.

All the application level metrics (native or generated) are published to Graphite, which is deployed on the autoscaler and runs alongside Riemann. See Figure 13, which shows historical evolution of the average response time in Graphite.


185.19.31.59

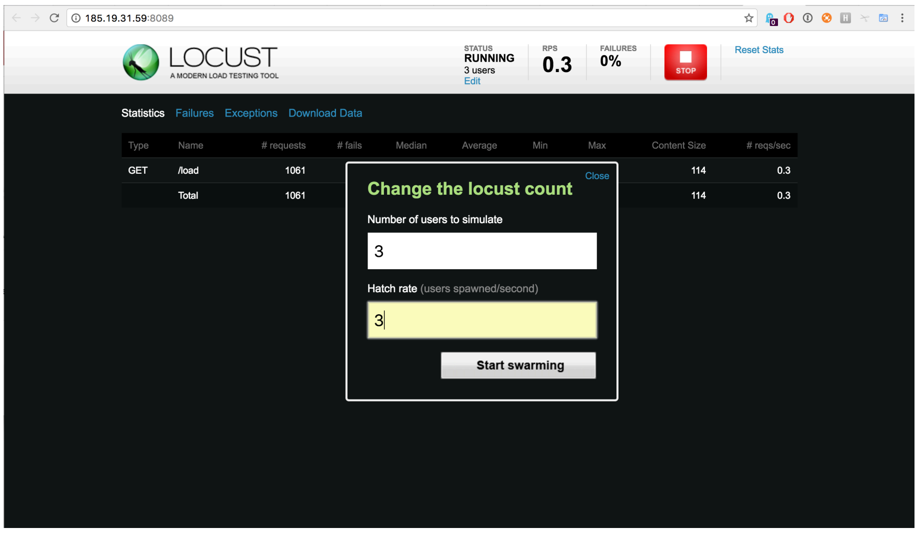
A simple web application deployed behind Nginx.

Nginx is acting as a load balancer.

The application is loaded by the load testing client (Locust).

- Locust - load generator.
- Riemann dashboard.
- Web application.

Figure 10: Application Entry Point


185.19.31.59:8089

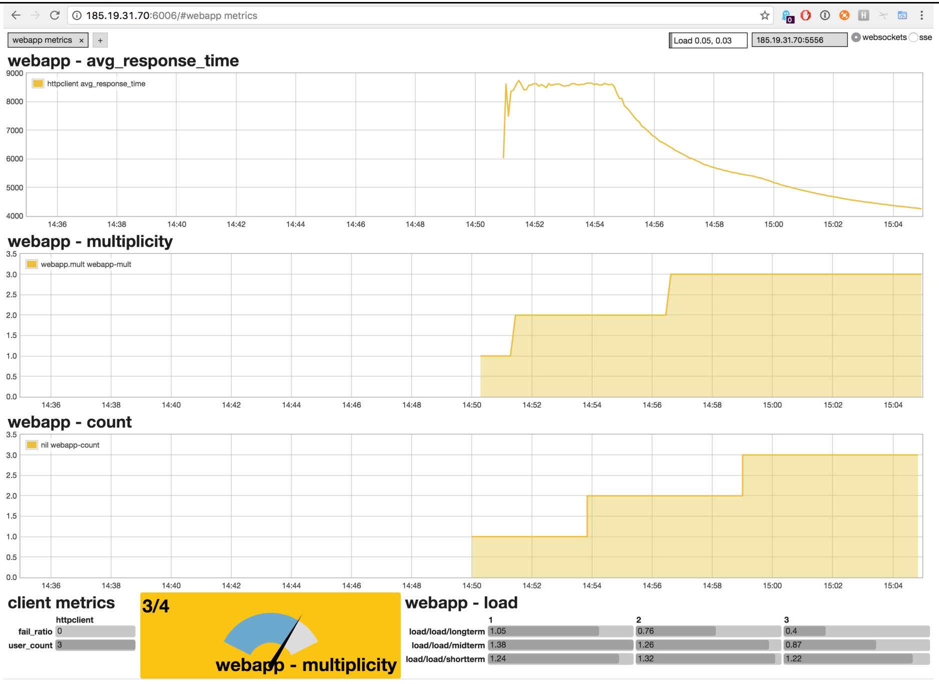
Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
GET	/load	1061						114	0.3
	Total	1061						114	0.3

Change the locust count

Number of users to simulate: 3

Hatch rate (users spawned/second): 3

Figure 11: Locust: Defining Work (as 3 Users) to Load the Application



client metrics

	httpclient
fail_ratio	0
user_count	3

webapp - load

	1	2	3
load/load/longterm	1.05	0.76	0.4
load/load/midterm	1.38	1.26	0.87
load/load/shortterm	1.24	1.32	1.22

Figure 12: Riemann-dash: High Load on the Web Layer

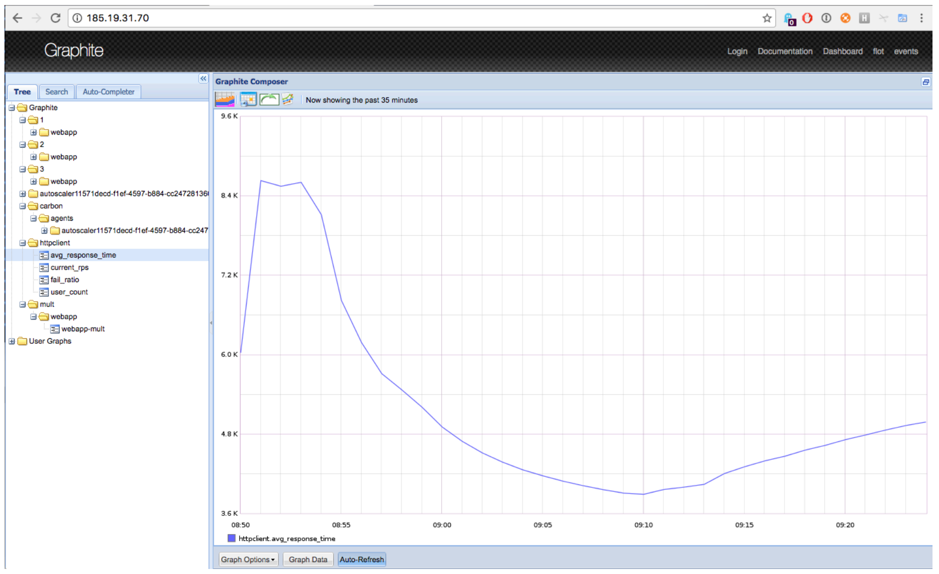


Figure 13: Average Response Time as a Function of Time in Graphite

5. Analytics Engine

5.1. Theoretical Background

One of the key features of cloud computing is the capability of acquiring and releasing resources on-demand. The objective of a cloud application operator in this case is to allocate and deallocate resources from the cloud to satisfy its Service Level Objectives (SLOs), while minimizing its operational cost. To achieve high agility responding to rapid performance fluctuations, the resource provisioning decisions must be made automatically. An application performance model can be constructed using various techniques, including queuing theory, control theory and statistics.

The resource control decisions can be either proactive or reactive. The proactive approach uses predicted demand to allocate resources based on the anticipated load. This could be used, for example, for day/night load fluctuations. The reactive approach makes decisions based on the current load and performance. Both approaches are necessary for effective resource control in dynamic operating environments.

Within CYCLONE, we use a statistical (stochastic) model for application response-time (latency) in the form of probability density functions (PDF). Because the PDF of response time is a continuous function, we need to store a representation of it in an acceptable format within our system. We use histograms, where two parameters are of importance: the number of bins (*b*), and the bin width (*m*). The values for these determine what the resulting histogram looks like. Choosing a small bin width will result in a chaotic representation with many isolated peaks. Conversely, choosing the large bin width will obliterate the details of the histogram that are necessary for the good representation of the data. Formulas that may be used for the selection of *b* [SCO79] and *m* [STU26] are:

$$m = 1 + 3.3 \cdot \log_{10}(n)$$

$$b = 3.49 \cdot sn^{-1/3}$$

where *n* is the sample size. The response time (RT) may be specified by the application/cloud providers as an SLOs within the Service Level Agreement (SLA). Note that this may not be the only way to specify the response time; for example, the PDF could be obtained from benchmarking *before* a service based on the cloud application is offered.

The model that we considered while implementing algorithms is specified as the following:

Each one of *N* applications has a response time represented by the random variable $D_i \geq 0$. For each random variable D_i a respective probability density function (PDF) is given. The response-time distributions may be obtained by benchmarking or specified by the provider. Next, the execution costs c_i (in money units) are specified for each application.

Once the application is selected, the performance may deteriorate with a deadline approaching, while the jobs submitted have not yet finished. The Analytic Engine is constantly updated with the measured application performance (via a monitoring connector). In such a case, another application is used. Naturally, this comes at additional costs. This situation may occur when choices are made between relatively slow and cheap applications and applications that are in general faster but more expensive.

The questions that need to be resolved are:

1. When does the performance of the deployed application deviate from what has been promised (in an SLA or through benchmarking)?
2. How often does one need to collect the performance updates from the applications that were not initially selected?
3. What is the optimal moment to take an action? These actions may include rescaling, or substituting the initially selected application with another one.

The answers to these questions are given in the following sections.

5.1.1. Optimal Application Selection

The specified model is used as the input for the analytic engine component. Let us assume there are two functionally equivalent applications, *AP1* and *AP2*. These applications are deployed on their respective cloud infrastructures and the application operator must select one. When application operator wants to select one of the applications, he specifies constraints covering functional requirements as well as non-functional requirements like cost and response-time. Based solely on functional requirements, *AP1* and *AP2* will be listed.

To decide between them, non-functional properties need to be compared. There are many possible choices: deadlines (which implicitly convert to total costs), availability, etc. These parameters are related to one another in an objective function, for which a maximum (or minimum) is found.

Therefore, the application operator needs to specify how these parameters are related to one another; he does this by weighting them. For example, it is possible to specify that relative weights for cost and response time is 4:1, meaning that cost is 4 times "more important" for him. Other application operators may have different these weights for their objective function.

The maximum for the objective function is then found subject to certain optimization criteria. The optimization criterion may be, for example, the highest expected reward. We will typically have a mono-objective function, in which we want to optimize a trade-off between all the objectives. This means that one should define the weighted-sum objective:

$$w_1 \cdot obj_1 + w_2 \cdot obj_2 + w_3 \cdot obj_3 \text{ where } w_i > 0 \text{ and } w_1 + w_2 + w_3 = 1$$

Depending on the ratio between the weighted coefficients, one objective could be made more "important" than the rest. This is expected to result in a unique solution.

However, for multi-objective functions, each objective could be optimized "separately". There are many techniques which transform the multi-objective problem into several mono-objective problems that one can solve sequentially (for example, the epsilon-Constraint, two step method of Aneja & Nair). The expected result here is a set of non-dominated (Pareto) solutions ("Pareto front"); that is, there is no solution that is better than other solutions with respect to all objectives; otherwise, this solution would be an optimal one.

5.2. Optimal Application Substitution and Optimal Redeployment of Applications

The main points arising from this section are:

- The optimal application substitution and optimal redeployment of applications can be utilized only if observed distributions (response time for applications and deployment times) exhibit the so-called "decreasing hazard rate". Therefore, they are not universally applicable.
- Optimization only makes sense if multiple alternatives are available: functionally equivalent applications on different cloud providers or the same application available on multiple cloud providers.
- We derive the policies for optimal deployment or substitution before deployment takes place or before applications are used. Once the deployment has not been finished until given deadline, all it takes is to evaluate the calculated policies and determine at which cloud the application should be redeployed, or which application/cloud combination should be used instead of the current one.

5.2.1. Introduction

In many situations, restarting system components, re-issuing a request, or re-establishing a network connection improves the performance or availability of the system under consideration significantly. It is not always known precisely *why* this is necessary or beneficial. Most Internet users are familiar with the fact that clicking the reload button often helps in speeding up the download of a page, although one typically understands only to a limited extent why the download was slow in the first place. Another example in which restart is beneficial is in the case of software "aging"; the re-initialization of the software environment may help in preventing application failures and speeding up processing. In many cases, very little is known about the causes of delay or aging and consequently cannot correct the root problem. In practical situations, therefore, we want to determine the optimal time to restart, without knowing or modelling the details of the system.

What characteristics do tasks that benefit from restarts exhibit? In general terms, the completion time when starting fresh must be less than the completion time when not restarting. We can formalize this by letting the random variable *T* denote the completion time of a task. Assume we are interested in the mean completion time. Under the assumption of independent identically distributed completion time of successive tries, one would restart at time τ when

$$\mathbb{E}[T] < \mathbb{E}[T - \tau | T > \tau]$$

holds.

The question then becomes, what distributions of *T* fulfil this requirement for at least one value of τ. Distributions with heavy tails have the required behavior. For such distributions, as the tail decreases at a polynomial rate, leaving considerable probability mass at high values of *T*. Heavy-tailed and similar distributions commonly arise when studying Internet applications. Distributions with exponentially decaying tails demonstrate the required behavior quite often as well [BOU98].

Mathematically, the key to the analysis of restarts is the hazard rate *h(t)* of a probability distribution, defined as

$$h(t) = \frac{f(t)}{1 - F(t)}$$

where *f(t)* is the probability density function and *F(t)* is the cumulative distribution of the task completion time *T*. The hazard rate at time *t* can be interpreted as the potential to complete the task, irrespective of what may have happened before. If the hazard rate is monotonically decreasing, the highest completion potential is at time zero and a restart always helps. These distributions (functions) are called decreasing hazard rate (DHR) functions. This is the case for certain heavy-tailed distributions, for example, the Pareto distribution. In contrast, if the hazard rate monotonically increases, the highest completion potential is at infinity and restart never helps. The hazard rate for the exponential distribution is a constant and any restart time is therefore equally effective as is not restarting at all.

5.2.2. Problem Definition, Model, and Analysis

We give a short analysis of the optimal application selection and substitution should one of the applications fail to perform as expected. Let us assume there are two functionally equivalent applications, *AP1* and *AP2*. These applications are deployed on their associated cloud infrastructures and we must choose between them. The choice is made using an optimization criterion, which in our case is the highest expected reward for the application operator. When application completes execution within given deadline δ, provider is rewarded with *R* (money units), otherwise, provider pays a penalty *V*.

In general, for an application *AP_i* we have:

$$\begin{aligned}\mathbb{E}[W_i] &= -c_i + R \cdot \mathbb{P}[X_i \le \delta] - V \cdot \mathbb{P}[X_i > \delta] \\ &= -c_i + r \cdot F_i(\delta) - v \cdot \bar{F}_i(\delta) \\ &= r - c_i - (r+v)\bar{F}_i(\delta)\end{aligned}$$

We should simply choose the service provider *i* for which $\mathbb{E}[W_i]$ is maximal,

$$\begin{gathered}\mathbb{E}[W_i] > \mathbb{E}[W_j] \Leftrightarrow \\ r - c_i - (r+v)\bar{F}_i(\delta) > r - c_j - (r+v)\bar{F}_j(\delta) \Leftrightarrow \\ -c_i - (r+v)\bar{F}_i(\delta) > -c_j - (r+v)\bar{F}_j(\delta) \Leftrightarrow \\ c_i + (r+v)\bar{F}_i(\delta) < c_j + (r+v)\bar{F}_j(\delta) \Leftrightarrow \\ \frac{c_i}{r+v} + \bar{F}_i(\delta) < \frac{c_j}{r+v} + \bar{F}_j(\delta)\end{gathered}$$

Application substitution. Once the application is selected, the performance may deteriorate with a deadline approaching, while the jobs submitted have not yet finished. The derived policies may specify that, in such a case, another application is used. Naturally, this comes at additional cost. The question is at which time this substitution takes place? This situation may occur when choices are made between relatively slow and cheap applications and applications that are in general faster but more expensive. This general situation is illustrated in Figure 14; an application AP_j (in this case *j* = 2) is initially selected. The policy specifies whether it is better to wait till it generates a response or to select service AP_k, perform a retry at a given moment $0 < \theta_{j\to k}$, and wait for service AP_k to complete (in this case *k* = *N*). The expected reward without retry is

$$\begin{aligned}\mathbb{E}[W_j(\delta)] &= -c_j + R \cdot F_j(\delta) - V \cdot (1 - F_j(\delta)) \\ &= -c_j - V + (R+V)F_j(\delta)\end{aligned}$$

The expected reward when a retry is made after $\theta_{j\to k}$is given as:

$$\begin{gathered}\mathbb{E}[W_{j\to k}(\delta, \theta_{j\to k})] = \underbrace{-c_j}_{\text{term 1}} + \underbrace{R \cdot F_j(\theta_{j\to k})}_{\text{term 2}} + \\ \underbrace{(1 - F_j(\theta_{j\to k}))}_{\text{term 3}} \cdot \underbrace{\{-c_k - V + (R+V)F_k(\delta - \theta_{j\to k})\}}_{\text{term 4}}\end{gathered}$$

Term 1 represents the costs of executing the application AP_j. Term 2 represents the reward *R* that provider incurs when application AP_j responds before the retry moment $\theta_{j\to k}$, with probability $F_j(\theta_{j\to k})$. Term 3 represents the probability there is a retry at $\theta_{j\to k}$, and Term 4 represents the expected reward when retry is made at $\theta_{j\to k}$. The retry implies that service costs c_k are paid, and the remaining time to deadline is $\delta - \theta_{j\to k}$.

Let θ^* be the optimal substitution time instance when there are two applications to choose from. As $\mathbb{E}[W|\theta]$ is continuous in θ, the optimal value θ^* on the open interval (0, δ) satisfies

$$\left.\frac{d\mathbb{E}[W|\theta]}{d\theta}\right|_{\theta=\theta^*} = 0$$

giving

$$\left.\frac{d\mathbb{E}[W|\theta]}{d\theta}\right|_{\theta=\theta^*} = c_2 \cdot f_1(\theta) + (R+V)(f_1(\theta)\cdot \overline{F}_2(\delta-\theta)) \\ -\overline{F}_1(\theta)\cdot f_2(\delta-\theta) = 0$$

Therefore, the optimal substitution moment θ^* can be determined from the following equation:

$$h_i(\theta^*)\cdot\left[1+\frac{c_2}{(R+V)\big(1-F_2(\delta-\theta^*)\big)}\right] = h_2(\delta-\theta^*)$$

where

$$h_i(t) = \frac{f_i(t)}{1-F_i(t)}$$

Naturally, solving the previous equations is not practical. Hence, the policy $\pi(\delta_p)$ for given deadline δ_p is determined either from Equation (1)

$$\pi(\delta) = \underset{i=1,2,\dots,N}{\operatorname{argmax}}\{\mathbb{E}[W_i(\delta)]\}$$

or from Equation (2)

$$\pi(\delta) = \underset{\substack{j,k=1,2,\dots,M \\ \theta_{j\to k}\in(0,\delta)}}{\operatorname{argmax}}\{\mathbb{E}[W_{j\to k}(\delta,\theta_{j\to k})]\}$$

The former policy, specified by Equation 1 is used for those values of deadline δ when the expected reward without substitution is bigger than the expected reward with it, that is, when

$$\mathbb{E}[W_j(\delta)] > \mathbb{E}[W_{j\to k}(\delta,\theta_{j\to k})], \forall j,k,\theta_{j\to k}\in(0,\delta)$$

In such a case the policy represents indices of applications resulting in maximum revenue for given deadline δ. Equation 2 is used for those values of deadline δ when substitutions are useful. It contains indices of applications selected for given deadline δ, as well as indices of applications to be used for substitution, as well as the optimal substitution moment.

To evaluate numerically $\mathbb{E}[W_j(\delta)]$ and $\mathbb{E}[W_{j\to k}(\delta,\theta_{j\to k})]$, a discretization of both deadline $\delta \in (0,\delta_p)$ and retry moment $\theta \in (0,\delta)$ is necessary.

The discretization steps are denoted by $\Delta\tau$ for the deadline, and by $\Delta\theta$ for the retry moment. The equal number m of discretization steps for both δ and θ is a natural and convenient choice. In such a case, the discretization steps are $\Delta\tau = \Delta\theta = \delta_p/m$.

The evaluation grid of $\mathbb{E}[W_{j\to k}(\delta,\theta_{j\to k})]$ is illustrated in Figure 15. For each point within such a grid, the value of the function is calculated based on the presented equations. The number of evaluation points is therefore $m\cdot(m+1)/2$. The grid points where function $\mathbb{E}[W_{j\to k}(\delta,\theta_{j\to k})]$ reaches a maximum are represented by symbol ⋆ in this figure. Therefore, these points also specify optimal retry moments for given deadline value δ.

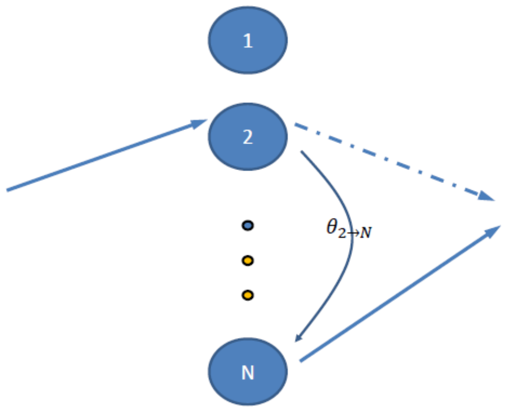


Figure 14: Application Substitution

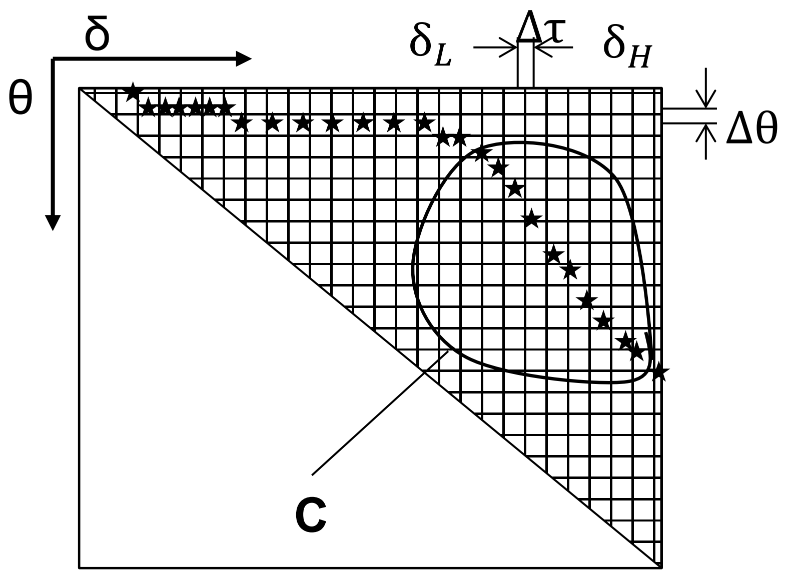


Figure 15: Evaluation Grid for Expected Revenue in Case of Application Substitution

5.3. Frequency of Performance Updates from Applications

We have seen that response time distributions are an example of probabilistic Quality of Service (QoS) guarantees. These guarantees are used to derive optimal selection and substitution policies. Due to the volatility of the cloud environment(s), these distributions are not time-invariant; they change over time. Because of this, previously determined policies may no longer be accurate.

Our approach for realizing the response-time distributions considers these time-varying changes. The current response-time distribution is compared to the one used for the previous policy update. Using well known statistical tests, we determine if a significant change occurred and thus if the policy must be recalculated. By tracking response times, the actual response-time behavior can be captured in empirical distributions.

We illustrate our approach using Figure 16. After each execution of a request in (1) the empirical distribution is updated at (2). Once an application is selected, other applications may not be used for a while. In that case, we do not receive any information about these applications. These could become attractive when an application is updated or when it is deployed on a more powerful cloud instance. Therefore, in (3), when an application is not used for a certain time, a probe request will be sent at (4) and the corresponding empirical distribution will be updated at (5a). After each calculation of the policies, the current set of empirical response-time distributions will be stored. These are the empirical distributions that were used for policy calculations and form a reference response time distribution. Calculating the policies for every new sample is expensive and undesired. Therefore, a strategy is used where the policies will be updated when a significant change in one of the applications is detected. For this purpose, the reference distribution is used for detection of response time distribution changes. In (4) and (5a), the reference distribution and current distribution are retrieved and a statistical test is applied for detecting change in the response time distribution. If no change is detected, the policy remains unchanged; otherwise the policy is recalculated. In (6) the lookup–table is updated with the current empirical distributions and these distributions are stored as new reference distribution. By using empirical distributions, we can directly learn and adapt to (temporary) changes in behavior of applications.

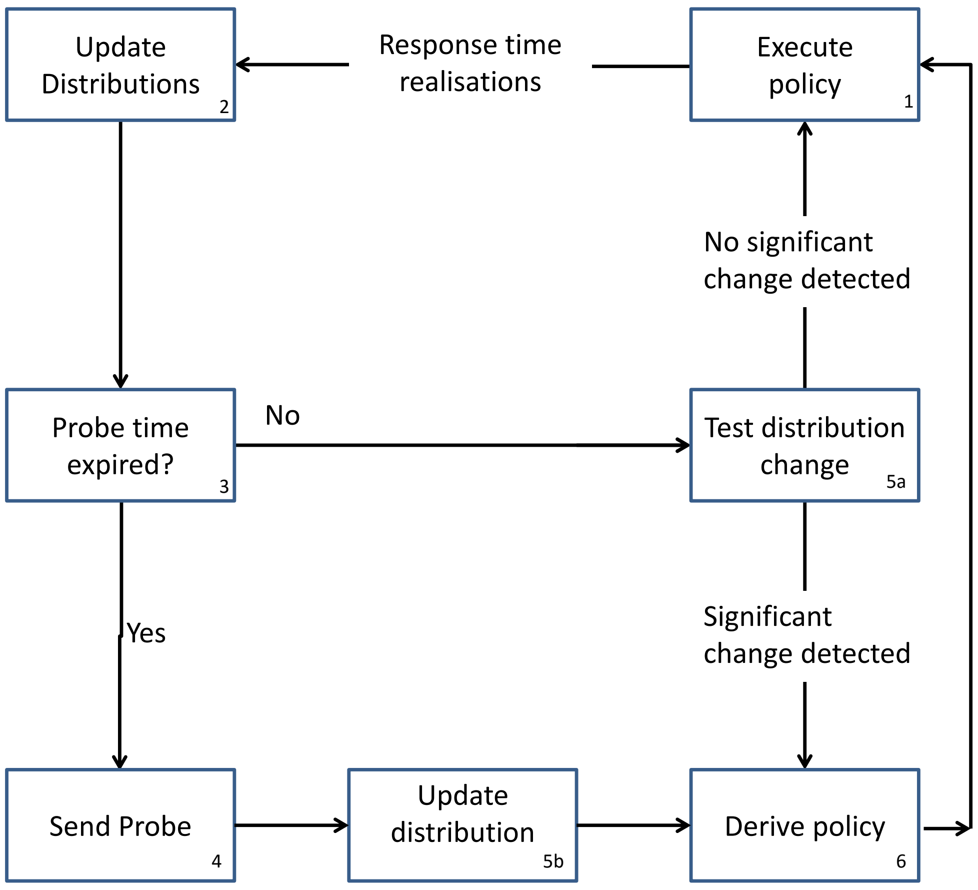


Figure 16: Closed Loop Control Approach

5.4. Performance Deviation

Main points:

- There is no “one size fits all” when these algorithms are applied.
- The Engine needs to have a collection of multiple algorithms.
- We need to verify and test these algorithms in actual deployments to identify those that are most suitable.
- There is a typical trade-off between speed of detection and accuracy. Typically, the longer we wait, the better detection would be.

5.4.1. Background

Monitoring applications/infrastructure in multi-cloud settings implies that measurements arrive in data streams. A data stream has necessarily a temporal dimension, and the underlying process (e.g. application response time) that generates the data stream can change over time [HUL01, WID96]. It is therefore crucial to be able to detect points in time in which such changes occur as these may indicate potential failures, performance deterioration or attacks.

The observed statistical processes are typically modelled as a time-series. The point in a time-series when the statistical properties of an underlying process change is known as a change-point. In practice, these changes may manifest as

- A shift in mean,
- A shift in variance, or
- A change in a distribution of the process.

The change of mean is illustrated in Figure 17. As we can see from this figure, the underlying statistical distribution of the data suddenly changes. Given the statistical randomness in the data, we cannot simply draw a line, say at 9, and declare a change point when it is crossed. Taking such a simple approach would lead to many false alarms. Only when the data is structurally, on average, lower than 10 can we say that a change is occurred.

The change-point detection methods derive from the sequential analysis. Both sequential analysis and change-point detection make use of a test statistic (some function of the observations) and issue an alarm when this test statistic exceeds a certain predefined threshold. The goal is to choose the threshold in such a way that the detection delay is minimized for a given probability of a false alarm (called type I error in statistics). The main difference is that change-point detection tests for a change in the underlying distribution at a point in the sequence whereas sequential analysis tests the whole sequence. Change-point detection methods are more suitable for the CYCLONE project.

There are many algorithms that could be used for the change-point detection. The efforts could be divided in two broad groups, namely

- The offline setting, where inference regarding the detection of a change occurs retrospectively, after all relevant data have been received
- The online setting, features methods in which analysis is performed sequentially - as every new observation is received, the detection method is implemented to locate possible change-points in previous observations.

Change-point detection methods can be further categorized into two classes:

- Parametric: Incorporates distributional knowledge of the data into the detection scheme, and

- Non-parametric: Makes no such distributional assumptions regarding the data.

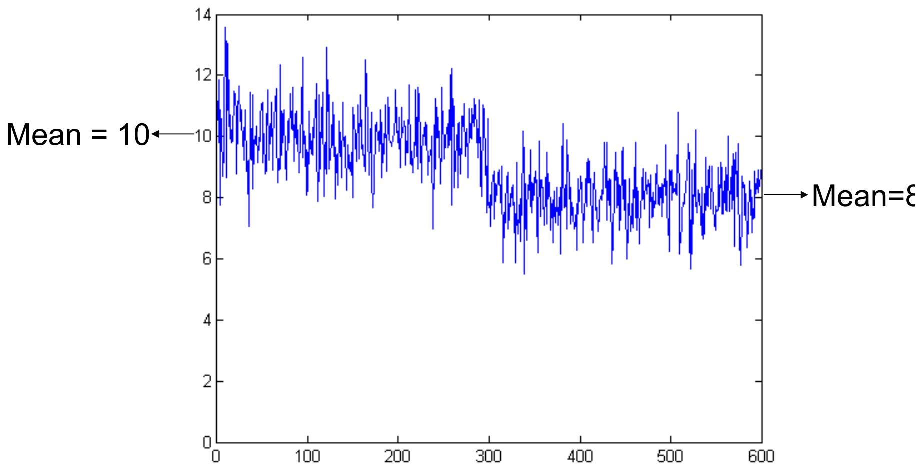


Figure 17: Occurrence of Change of Mean in Time Series

5.4.2. Problem Description

We assume that the data are observed sequentially; every time step adds one more observation. Given the set of observations up to the current time, we want to know whether a change-point has occurred. In general, change-point detection methods monitor some test statistic, which is based on the observations, and issues an alarm if this test statistic exceeds a certain threshold, such that the calculated probability of not detecting a change-point is kept below a certain predefined value α, for example α = 5%. These methods perform a hypothesis test for each time step. We define this hypothesis test below.

More formally, consider a sequence of observations, $X = (X_1, X_2, X_3, \dots)$, which may contain a change-point. In probabilistic terms, the procedure to identity the change-point is:

Suppose there is a $k > 0$ such that the X_i are independent identically distributed (i.i.d) realizations of a random variable with probability density function f for $i = 1, \dots, k-1$ while the X_i are i.i.d. with a different density g for $i \geq k$. In this case, we call k a change-point.

At a point in time $n = 1, 2, \dots$, we check whether a change-point has occurred at some time $k \leq n$, by evaluating $X_n \coloneqq (X_1, \dots, X_n)$, if not we continue by evaluating $X_{n+1} \coloneqq (X_1, \dots, X_{n+1})$ etc. In terms of hypothesis testing, at time n we want to decide between two hypotheses:

- Under the null-hypothesis (H_0) the X_i $(i = 1, \dots, n)$ are i.i.d. realizations of a random variable with density f.
- Under the alternative hypothesis (H_1) there is a $1 \leq k \leq n$ such that up to $k - 1$ the observations are i.i.d. samples from a distribution with density f, while from observation k on they are i.i.d. with a different density g.

Therefore, under the null-hypothesis there has not been a change-point, while under the alternative hypothesis the process changes at some time k.

5.5. Evaluating Change-point Detection Methods

Typical metrics to evaluate a hypothesis test consider the probability of so-called type I and type II errors; the performance of the algorithm at time n can be quantified by these error probabilities. They are defined as follows:

- A type I error (false positive) occurs if we detect a change-point that has not occurred. The type I error probability is therefore the probability of false positives.
- A type II error (false negative) occurs if we miss a change-point. The type II error probability is the probability that we accept H_0 over H_1.

Let us now consider a change-point detection procedure for the entire sequence X. That is the procedure where a stopping rule issues an alarm at time $\tau \geq 1$, defined as the first time that we decide to reject H_0. On the one hand, τ should occur soon after the change-point k; on the other hand, the rate of false alarms should be low. Mathematically, this can be formulated as keeping the distribution of $\tau - k$ stochastically small, given that the change-point takes place at k (i.e., under $H_1(k)$), whereas the distribution of τ should be stochastically large in case there is no change-point (i.e., under H_0).

We will now describe just one of the non-parametric change-point methods used to detect change in mean. The details of the algorithm and a rigorous analysis are given in [BRO93].

5.5.1. The Algorithm of Brodsky and Darkhovsky

This algorithm has no *a priori* assumption on data following a defined distribution and is used to identify a change in mean of the underlying time-series.

At time n this algorithm considers the observations $X_{n-N+1}, X_{n-N+2}, \ldots, X_n$, where N is the window size. To check whether a change in mean has occurred at time $n - N + k + 1$, the average over $X_{n-N+1}, \ldots, X_{n-N+k}$ is compared with the average over $X_{n-N+k+1}, \ldots, X_n$. In other words, the window (of size N) is split in two (unequal) parts, one of size k and the other one of size $N - k$, and respective averages are calculated. When there is no change-point within the window, the difference tends to be close to zero. An alarm is raised when there is a k for which the difference exceeds some predefined threshold $c > 0$. Note that c should be smaller than the mean change, as otherwise issuing an alarm remains rare, even if there is a change-point. For k close to 1 or $N - 1$, one of the averages contains too few values, and the algorithm may produce errors.

To prevent this, a parameter $\gamma \in (0, \frac{1}{2})$ is chosen and only a subset of k is considered namely, $k \in \{\lceil \gamma N \rceil, \ldots, \lfloor (1-\gamma)N \rfloor\}$. The algorithm can be summarized as the following:

Fix the window width $N, \gamma \in (0, \frac{1}{2})$ and threshold $c > 0$. Let:

$$U_n(k, N) = \frac{1}{k} \sum_{i=n-N+1}^{n-N+k} X_i - \frac{1}{N-k} \sum_{i=n-N+k+1}^{n} X_i$$

The algorithm raises an alarm when the test statistic

$$U_n(N) = \max_{k \in \{\lceil \gamma N \rceil, \ldots, \lfloor (1-\gamma)N \rfloor\}} |U_n(k, N)|$$

exceeds the threshold c.

We illustrate the algorithm of Brodsky-Darkhovsky in Figure 18. In this example, the algorithm is used to detect a change in mean from μ_1 to μ_2 at time instance $k = 51$. The settings are $N = 40, \gamma = 0.1$ and threshold $c = 0.5$. A change-point at $k^* = 50$ is detected at time $n^* = 53$. The sample averages are $m_1 = \frac{1}{36}\sum_{i=14}^{49} X_i = 1.65$ and $m_2 = \frac{1}{4}\sum_{i=50}^{53} X_i = 0.98$; their difference is 0.67 which is larger than the threshold value.

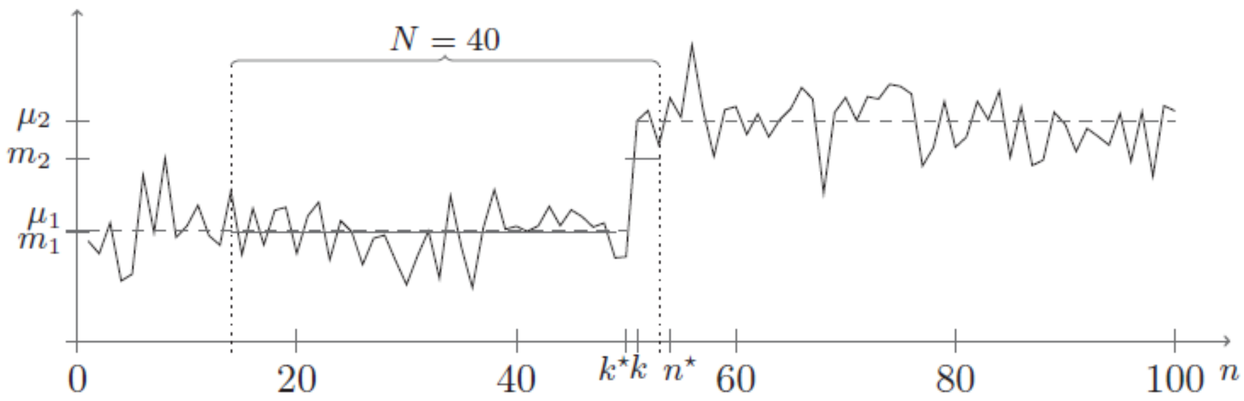


Figure 18: An Example of Brodsky-Darkhovsky Algorithm

5.5.1.1. *Window Size*

We use the concept of a window to divide up a continuous stream of data into batches for processing. For example, if we monitor the average response time of a system and receive a data point every 10 minutes, using a window size of one hour means that at the end of each hour we would calculate the average (mean) value of the last hour's data (6 data points) and compute the anomalousness of that average value compared to previous hours.

The window size has two purposes: it dictates over what time span to look for anomalous features in data and determines how quickly anomalies can be detected. Choosing a shorter bucket span allows anomalies to be detected more quickly but at the risk of being too sensitive to natural variations or noise in the input data. Choosing too long a bucket span however can mean that interesting anomalies are averaged away.

The method requires the user to set three parameters. First, one must decide on the window size, and second, on a parameter γ. A window of observations is divided into two intervals such that both contain at least a fraction γ of the number of observations in the window. Third, a threshold on the size of the mean change must be chosen; this threshold determines when an alarm is raised.

5.5.2. Change in Variance

It is apparent there is much less literature about detection of change in variance than change in mean. Under assumption that mean does not change, the problem of testing for change in variance can be converted to the one of testing for change in mean with simple transformation.

Let us suppose that initial data is X_i and we define $Y_i = (X_i - \mu)^2$, where μ is the mean. Then the change in mean of Y_i will be a change of variance in X_i, and any given algorithm for change of mean could be applied after this simple transformation. This results from the equation $\mathbb{E}Y_i = \mathbb{E}(X_i - \mu)^2 = \mathrm{Var}(X_i)$.

5.5.3. Multidimensional Change-point Detection

Performance monitoring (e.g. networks and applications) often involves multidimensionality (more than one sensor) with dependence between sensors and time. Many of the well-known anomaly detection methods, however, assume one-dimensional, independent input. These assumptions are of course too unrealistic in many cases, and straightforward application to more complex problems frequently leads to erroneous results. Dependence is thus a common characteristic of multidimensional data (measurements).

There exist several generic models known to have good capabilities to model data originating from multidimensional, dependent measurements, especially multidimensional autoregressive, moving average (ARMA) time series models, which are popular in econometrics research. Existing literature on machine learning techniques show that violated model assumptions do not necessarily obviate the efficacy of a given algorithm. As an example, we present a parametric method that makes assumptions of the underlying data. We focus on multidimensional data that exhibit "abrupt" change-points and can include multiple change points over time.

5.5.3.1. Likelihood Ratio Test

In Galeano and Peña's work on detecting covariance changes in multivariate data, they proposed two methods for calculating test statistics from which change points could be identified [GAL07]. These methods model the given data as a vector autoregressive integrated moving average (vARIMA), extracting the errors (or innovations) from this data, and applying these methods on this error data.

The first such statistic, on which we focus here, uses a likelihood-ratio test (LRT) to compare two hypotheses: the null hypothesis H_n that the covariance of this error data is best characterized by a single covariance matrix Σ versus the alternative hypothesis H_a that, at some time point h, the data is best characterized by two separate covariances matrices Σ_1 before h and Σ_2 after h. The logarithm of a modified form of the ratio H_n/H_a then generates a test statistic LR_h that existing literature shows is governed by a chi-squared distribution with degrees of freedom proportional to the dimensionality k of the data. From simulations of this distribution, we can generate a critical value given some α against which to compare this test statistic to determine whether a change point exists at some time h.

5.5.3.2. Algorithm

Given some time-series data $\tilde{y}_t$ and confidence α, we use the algorithm described in Figure 19 to identify points of change in covariance.

To implement LRT, we used Python and Scikit's statsmodels package for fitting data to VAR() models. One should note this restriction to VAR() models is a result of an existing constraint in the statsmodels package.

We also implemented a version of the LRT algorithm that does not rely on calculating the W transformation matrix. Rather than evaluating W, we leveraged statsmodels and its maximum likelihood estimation to fit the data to two new VAR() models for each regime. The above algorithm performs better than this secondary implementation because it obviates the need for separate rounds of maximum likelihood estimation for each level of recursion.

5.5.4. General Conclusions

From the above discussion, there are many algorithms that could be used for detection of changes in the measurements of, for example, application performance. This means that several algorithms need to be implemented and deployed within our Analytics Engine component. The verification of the algorithms requires a good knowledge of assumptions used for these algorithms and estimation of type I and type II errors, which are the main performance indicators of their applicability.

```
Function LRT($\tilde{y}_t, \alpha$) Algorithm by Galeano and Peña [9]
    fit VARIMA($p, d', q$) model to $\tilde{y}_t$ ;
    compute residuals $\hat{e}_t$ ;
    $k \leftarrow$ dimension($\tilde{y}_t$) ;
    $d \leftarrow k(p+q+1) + \frac{k(k+1)}{2} + 1$ ;          /* minimum points needed */
    $n \leftarrow$ len($\tilde{y}_t$) ;
    $df \leftarrow \frac{k(k+1)}{2}$ ;                         /* degrees of freedom for $\chi^2$ */
    $C \leftarrow$ simulateChiSquareMax($df, \alpha$) ;        /* obtain the critical value
    */
    $LR \leftarrow$ zeros($n$) ;
    $S \leftarrow \frac{1}{n}\Sigma_{i=1}^{n} e_i \cdot e_i'$ ;
    for $h \in [d, n-d-1]$ do
        $v \leftarrow h/n$ ;
        $S_1 \leftarrow \frac{1}{h}\Sigma_{i=1}^{h} e_i \cdot e_i'$ ;
        $S_2 \leftarrow \frac{1}{n-h}\Sigma_{i=h+1}^{n} e_i \cdot e_i'$ ;
        $LR[h] \leftarrow n \ln \frac{|S|}{|S_1|^v |S_2|^{1-v}}$ ;
    end
    $h_{max} \leftarrow argmax_h(LR)$ ;
    $\Lambda_{max} \leftarrow LR[h_{max}]$ ;
    changePoints $\leftarrow$ [ ] ;
    if $\Lambda_{max} > C$ then
        changePoints += $h_{max}$ ;
        $W \leftarrow$ transformation governing new data regime (see [9]);
        changePoints += apply LRT to $\hat{e}_t[0 : h_{max}]$ ;
        changePoints += apply LRT to $W \cdot \hat{e}_t[h_{max}+1 : n]$ ;
    end
    return changePoints
```

Figure 19: Algorithm by Galeano and Peña

6. SlipStream Comparative Analysis

SlipStream provides the bulk of the application management functionality within the CYCLONE toolkit. When developing the CYCLONE project, SlipStream was chosen because of its unique feature set in the multi-cloud domain and SixSq was brought into the project because of this. However, this sector has evolved rapidly and there are now other products and services that offer similar functionality. It is important to understand how SlipStream compares to those other products and services to understand how the CYCLONE toolkit can be exploited in the future. This section provides a comparative analysis of SlipStream (and Nuvla which is a "Software as a Service" offering of SlipStream) to other products and services in this sector.

SlipStream, by SixSq, forms the technology foundation of its portfolio of products and services:

1. **Nuvla**: SlipStream as a Service, managed by SixSq to offer brokerage and multi-cloud application automation
2. **SlipStream on premise**: software product that enables multi-cloud and hybrid cloud strategies for customers
3. **NuvlaBox**: instant private cloud, delivered as a fan-less appliance, connected to the SlipStream eco-system for remote management

Nuvla is used in a growing range of industries and fields of application. Similarly, NuvlaBox is targeting "Internet of Things" (IoT) and Smart City markets. And SlipStream on premise is generic and can apply to most application domains.

This short comparative study focuses on SlipStream, in the context of large-scale application management in multi-cloud and hybrid environments. When comparing the SlipStream ecosystem with competition and alternatives on the market, we must carefully compare like and like, even in this narrowed context. Products or services can sometimes seem to compete with SlipStream-based products and services, where, in reality, they can be companions, with little overlap and great complementarity.

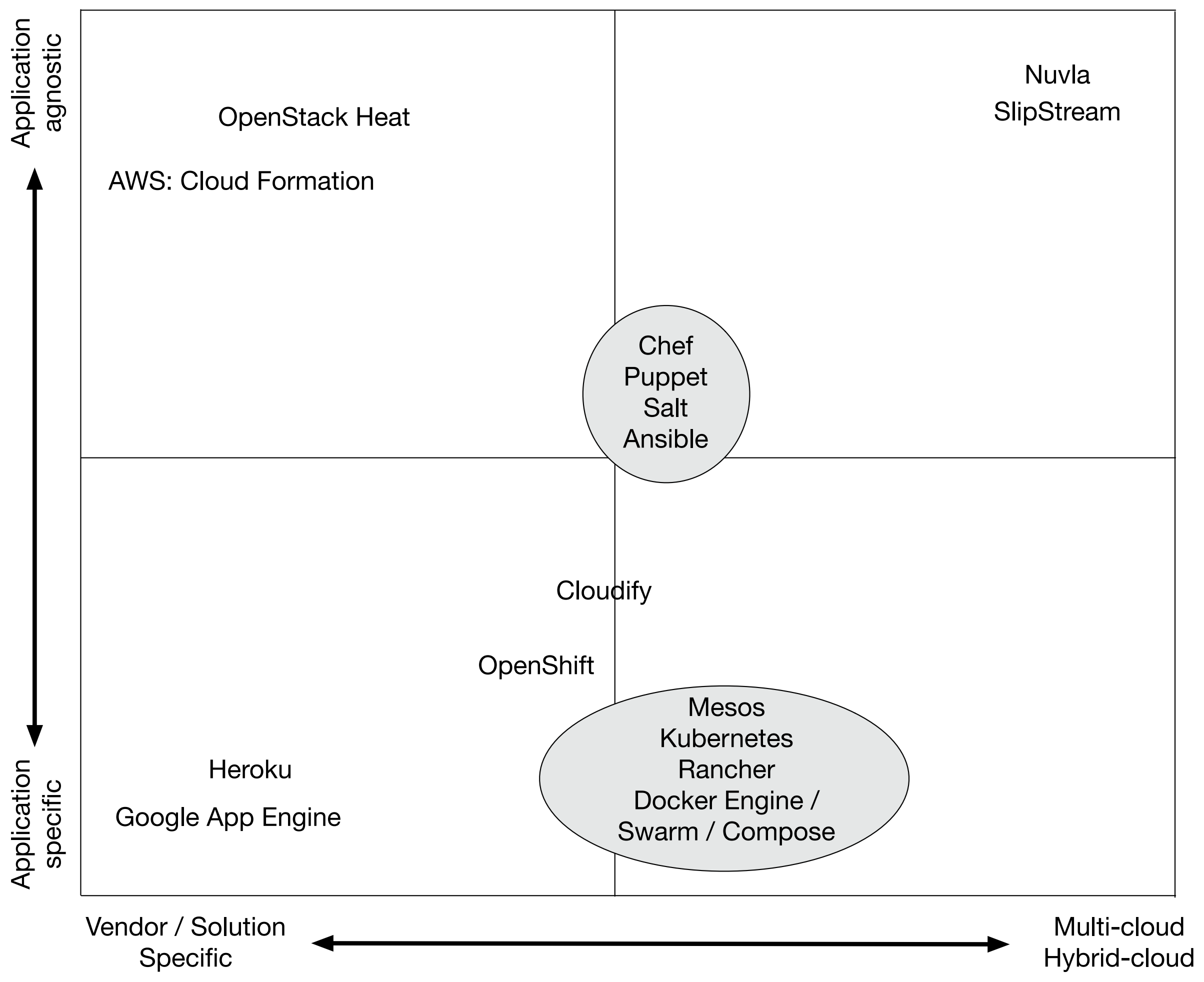
SlipStream provides a wide range of features that span many different markets. We have segmented the analysis into the following areas, showing where SlipStream competes or complements other solutions:

1. Cloud management solutions
2. Configuration management solutions
3. Tools and libraries
4. Container management solutions
5. PaaS
6. Brokerage solutions or services

We chose these categories, since they represent the categories most cloud related products and services use when communicating their value proposition. As you will see throughout this analysis, there are several categories with overlapping features and benefits, or even alternative ways of delivering a given benefit.

For each of these categories, we highlight the support for the open source movement. This is key as customers, both in industry and academia, increasingly value the open source model, because it creates a different dynamic between customers and providers. Indeed, it allows customers to influence the evolution of products and perhaps even contribute to them.

The following figure compares the products and services mentioned in this analysis based on how application-specific or infrastructure-specific they are. As SixSq values vendor-neutral solutions that apply to a broad range of applications, Nuvla and SlipStream sit in the upper-right hand corner of the diagram.



6.1. Cloud Management Solutions

Cloud management solutions are numerous and varied. Several of these solutions attempt, like SlipStream, to provide a multi-cloud or hybrid cloud solution, with a wide coverage of existing IaaS providers. These management solutions typically sit on top of IaaS APIs.

But even within this category, we can distinguish sub-categories:

1. **Cloud Resources Management**: management of cloud resources (e.g. compute, storage, network)
2. **Application management, in the cloud**: application deployment, sometimes referred to as application *orchestration*

Cloud Resources Management can be delivered either as services, such as SlipStream or Nuvla, but also as IaaS solutions, for on premise installations:

1. OpenStack
2. CloudStack
3. OpenNebula

4. vCloud Director

Table 3: IaaS vendors (public and private) vs SlipStream / Nuvla

Product/solution/vendor	Open Source?	Interoperable?	Comment
SlipStream / Nuvla	●	●	Apache 2.0 and support most IaaS solutions and services
OpenStack	●		
CloudStack	●		
OpenNebula	●		
vCloud Director			Commercial product by VMware

These IaaS solutions offer a range of features to manage cloud resources. Several of these solutions also support or include related projects or products able to deliver features that provide higher-level abstractions, towards for example the PaaS layer (see Section 6.5 for details).

There are also a growing number of IaaS delivered by providers:

1. Amazon Web Services
2. Microsoft Azure
3. Google Compute
4. IBM SoftLayer
5. Digital Ocean
6. Exoscale
7. T-System's Open Telekom Cloud
8. Cloud28+ providers

While some providers offer multi-region or multi-zone services, these services are provided by a single vendor, with the risk of lock-in for customers. There are also attempts to provide both client and server-side abstraction, with DeltaCloud as an example of server-side solution. See Section 6.3 for further details on client-side multi-cloud support.

Table 4: Cloud Resources Management vs SlipStream / Nuvla

Product/solution/vendor	Open Source?	Multi-cloud / hybrid?	Available as a Service?	Comment
SlipStream / Nuvla	●	●	●	Open source: Apache 2.0
Amazon Web Services			●	Offers several regions/zones
Microsoft Azure			●	Offers several regions/zones
Google Compute			●	Offers several regions/zones

IBM SoftLayer			●	Offers several regions/zones
Digital Ocean			●	Offers several regions/zones
Exoscale	●		●	Based on CloudStack
T-System's Open Telekom Cloud	●			Based on OpenStack
Cloud28+ providers	●			Based, mostly, on OpenStack
DeltaCloud	●	●		Apache Software Foundation project

On the *Application management, in the cloud*, product and services range, recent years have seen significant movement, with several start-ups being acquired by large system integrators or operators. For example, ServiceMesh was acquired by CSC (October 2013), Enstratius by Dell (May 2016), CliQr by Cisco, and Gravitant by IBM. Once acquired, these solutions are frequently either blended in existing portfolios or integrated with existing offerings. It is therefore often difficult to disentangle these products for comparison.

On the service side, ComputeNext launched a few years ago a marketplace for single virtual machine (VM) applications available on several IaaS platforms. BitNami supports a wealth of standard apps, which typically deploy on a single VM. But these solutions do not support custom made deployment recipes, which is key for many customers, developing their own software, or mixing intimately software components, services and micro-services.

We also have most 'fabric management' systems offering their own 'forge' for facilitating deployment of common, often open source, software (see Section 6.3 for details).

Finally, several projects also attempt to provide application management. These can come from providers, such as Amazon with *Cloud Formation*, or related open source projects, such as OpenStack *Heat*.

Table 5: Application management, in the cloud vs SlipStream / Nuvla

Product/solution/vendor	Open Source?	Support custom apps?	Hybrid cloud support?	Comment
SlipStream / Nuvla	●	●	●	Apache 2.0
ServiceMesh		●	●	Sold to CSC
Enstratius		●		Sold to Dell (and since abandoned?)
CliQr		●		Sold to Cisco
Gravitant		●		Sold to IBM
ComputeNext				Now focusing on on-premise
BitNami				Public Apps
OpenStack Heat	●	●		OpenStack specific and command-line tool

6.1.1. Competitor or companion?

Are the products and solutions identified in the previous section competitors or companions to SlipStream and Nuvla? It depends.

SlipStream, and therefore Nuvla, doesn't provide the IaaS layer. On the contrary, it leverages the IaaS layer, via an architectural layer called connectors. Therefore, SlipStream / Nuvla delivers its value on top of IaaS APIs. And this connector architecture is the reason why SlipStream is truly multi-cloud and supports rich hybrid cloud scenarios (e.g. public-public, public-private, private-private).

Regarding *application management, in the cloud*, SlipStream mostly competes with the solutions listed. Simplicity, multi-cloud, hybrid-cloud and open source are the key differentiator in this segment in favor of SlipStream and Nuvla.

6.2. Configuration Management Solutions

Configuration management or 'fabric management' solutions help to configure servers and, to a certain extent, monitor servers to maintain or evolve the specified configuration. They work by actively altering the server state by, for example, updating software packages or running specific scripts.

The main players in this field are:

1. Chef
2. Puppet
3. Ansible
4. Salt

These solutions have historically targeted system administrators, allowing them to manage large operational systems with fewer manual interventions. More recently they have added functionality to better work with cloud based infrastructures. This applies to both administrators of cloud infrastructures (e.g. managing the IaaS layer on physical servers), as well as applications running in the cloud.

Their cloud management features are specific to their own tooling. Their support for multi-cloud is limited and they offer no interoperability with other fabric management systems. They also require significant investment in time and effort, as users must understand and master new domain specific languages (DSL) and concepts.

Some of the vendors of these above listed solutions have tried to deliver an all-inclusive solution, adding for example ancillary projects to create high-level portals. However, since these solutions primarily aim at system administrators and developers, often preferring command-line interfaces, they generally have poor support for rich web user interface experience, compared to SlipStream AppStore and Dashboard.

The community behind these solutions also maintains rich and interesting repositories of recipes (also known as cookbooks, playbooks or manifests). While the quality and stability of these recipes can vary significantly, they can speed up software installation and configuration.

The support for scaling and auto-scaling of these solutions, if available, is generally complex. In comparison, SlipStream's scaling and auto-scaling solution is simple and multi-cloud.

Support for AppStore and self-provisioning services, to facilitate application deployment for a wider range of users requires extra software and services, while it is a core feature of SlipStream.

Table 6: Application management, in the cloud vs SlipStream / Nuvla

Product/solution/vendor	Open Source?	Language agnostic?	Web UI anf high-level function?	Interoperable?	Comment
SlipStream / Nuvla	●	●	●	●	
Chef	●				
Puppet	●				
Ansible	●				
Salt	●				

6.2.1. Competitor or companion?

We see fabric management solution mostly as companions. If and once the customer has invested time and effort in learning the tool and its language, they shine is their ability to manage the installation and configuration of services. They can also participate to the maintenance and update of services, as software packages are released.

They also require significant investment in time and effort, as users must understand and master new domain specific languages (DSL) and concepts. We argue that SlipStream provides similar benefits, in a simpler manner, with a much lower entry barrier. However, configuration management solutions can easily be integrated in a SlipStream recipe. This can also be a path for integration of diverse configuration management solutions, for customers to harness existing investments.

Finally, as more system administrators and DevOps teams opt for architectures with mostly stateless services and explicit, isolated state management, it is often easier to re-deploy services on new resources (aka VMs or containers), rather than updating or migrating existing services. This can be performed more simply and flexibly using SlipStream, on any cloud.

We therefore recommend that users and customers with investment in these solutions simply use these recipes inside SlipStream application component definitions. See related best practices on how to achieve this.

6.3. Tools and Libraries

To facilitate the management of infrastructure as a Service (IaaS), several client tools and libraries are available. Currently, the most popular are:

1. Terraform: open source project by HashiCorp, is a command-line tool, aiming at simplifying the creation and management of IaaS resources
2. Libcloud: Apache Software Foundation library, written in Python, providing abstractions for managing IaaS resources
3. Jclouds: Apache Software Foundation library, written in Java, providing abstractions for managing IaaS resources

Jclouds and libcloud are both libraries, allowing respectively Java and Python developers to interface with IaaS API to manage virtual resources. Terraform is more a command-line tool, offering an abstract configuration file based system to define and manage IaaS resources.

SlipStream also ships an open source command-line client. The latest version includes an abstraction of the basic cloud resource requirements (e.g. number of cores, memory, root disk size). This means that users do not have to specify 't-shirt' sizes (i.e. predefined bundles of resource sizes, such as cores, memory and disk) for the virtual machines. Having to provide these, breaks interoperability, since users must track and manage the different, often incompatible, t-shirt size definitions. SlipStream allows users to directly specify these underlying resource sizes. It then searches and matches the right offer for each available cloud, using its service catalogue. This ensures uniform and interoperable provisioning of cloud resources.

Table 7: Tools and libraries vs SlipStream / Nuvla

Product/solution/vendor	Open Source?	Interoperable?	Abstract 't-shirt' size?	Support filters?	Command-line?	Support native features?	Comment
SlipStream / Nuvla command-line client	●	●	●	●	●		
Terraform	●			●	●	●	
libcloud	●					●	Apache Software Foundation project
Jclouds	●					●	Apache Software Foundation project

6.3.1. Competitor or companion?

SlipStream interfaces with the IaaS via a system of connectors. Several of the connectors are implemented using libcloud. Earlier versions of SlipStream used Jclouds, but the Jclouds project grew in complexity and the SixSq team found libcloud simpler and more effective. However, libcloud is a developer tool, which requires that developers know some details of each cloud that they want to use.

These tools therefore compete, but can also coexist, as developers and system administrators learn and evolve their tooling.

One key difference is that the SlipStream client provides a higher level and simpler abstraction to IaaS. Terraform, for example, exposing more of the intricacies of each IaaS provider native features, at the cost of portability.

Finally, the SlipStream client provides a truly interoperable client, where application deployments, simple and complex, can be automatically deployed, without requiring any IaaS provider prior knowledge. This also includes default matching of the cheapest offer and cloud, and advanced policy placements.

6.4. Container Management Solutions

Currently, the container world is in ebullition. Beyond the hype and the promises, there is a realization that changes required to embrace the container revolution is intrusive to the way IT is operated and applications crafted. Adding to that the uncertainty between the champions fighting to deliver container management solutions, the market is understandably skeptical and worried.

Some of the confusion comes from the fact that containers per say are not new, but their management solutions, and underlying philosophy, is.

One clear benefit of containers themselves is the packaging model they propose. Up to now, software developers when creating a server, had to assemble binary packages (e.g. RPM, DEB), mixed with configuration scripts. This could be organized using fabric management systems (see Section 6.2 for details). With containers, software developers now have another option to create custom servers. However, creating a container also requires acquiring new knowledge and working within the constraints it imposes.

Currently, container management solutions include:

1. Mesos
2. Kubernetes
3. Docker Engine / Docker Swarm / Docker Compose
4. Rancher
5. CloudFoundry

CloudFoundry was initially open sourced by VMware, with start-ups building solutions on top and system integrators investing in the solution (e.g. Pivotal). But with the recent development of container management systems, the CloudFoundry project is finding itself challenged by new comers.

Container management solutions, such as Mesos and Kubernetes, offer higher level management functions. These solutions are themselves complex systems. Their benefit is in providing a framework for managing workloads and containers, at scale, inside a pool of VMs.

Docker, which is the current favorite container solution, is also entering the management chase, with a few fast evolving projects, such as Swarm, Engine and Compose.

Table 8: Container management solutions vs SlipStream / Nuvla

Product/solution/vendor	Open Source?	Support cloud resource auto-scaling?	Support containers?	Simple to use?	Automatic deployment in cloud?	Comment
SlipStream / Nuvla	●	●	●	●	●	
Mesos	●		●			
Kubernetes	●		●			
Docker Engine / Docker Swarm / Docker Compose	●		●			
Rancher	●		●		●	
CloudFoundry	●		●			

6.4.1. Competitor or companion?

Compared to container management solutions, SlipStream takes a different approach, which is in line with its agnostic philosophy. Using simple scripts, SlipStream packages together binary package installation, as well as configuration and parameterization. It can also leverage containers, if they exist. Therefore, it can take advantage of containers, without forcing users to take the *red pill*. This means users see the benefit of automation earlier, with less cost or disruption to its infrastructure and running applications.

Further, SlipStream can be interfaced with for example the queue of container management solutions, to grow and shrink the available cloud resources, based on the key performance indicators they provide. While this integration is a roadmap item, the possibility to deliver auto-scaled container management solutions to any cloud is interesting and would nicely complement the container management solutions listed above.

Finally, container management solutions target very large scale container deployments. In this age of assembly of services, mixing SaaS and PaaS offering with customer software, it is not clear that this race for hyper scale addresses the need of most organizations and software developers. SlipStream's agnostic and independent approach is a simpler and safer path, while enabling many of the benefits containers offer, while providing simplification in deploying these management solutions, therefore reducing complexity and risk.

As the container world remains a fast-moving target, investing too heavily and too deeply in one of these solution is potentially risky. Instead, SlipStream offers a safer route, while providing access to some of the benefits, as well as potentially providing simpler migration from one container management solution to another, thanks to its ability to automate their deployment.

6.5. PaaS

The Platform as a Service landscape is also very dynamic. With the latest consolidation of cloud management solutions (see Section 6.1 for details) and the advent of container management solutions, the Platform as a Service is being challenged. Here are a few well-known PaaS:

1. OpenShift
2. Cloudify
3. Google App Engine
4. Heroku

According to PaaSfinder[3], SlipStream, or more specifically Nuvla (SlipStream as a Service, managed by SixSq), is a *IaaS-centric PaaS*. This taxonomy might seem a little complex, but it is useful when comparing PaaS solutions. Some PaaS, such as Heroku or Google App Engine, are *generic PaaS* types, where they hide the IaaS layer. This is significant, since it means they do not support hybrid deployment (e.g. you can't connect your private IaaS to the PaaS). Further, they tend to support specific deployments and application architectures – e.g. request / response architectures, such as web applications. They also tend to provide specific state or database management services (e.g. Big Table), pre-integrated with the overall solution. Finally, they have a pricing model specific to them – e.g. based on the number of request/response pairs or resources deployed; such that providers deploy resources to match user application demands as loads vary over time.

These *generic* type PaaS are well adapted for web applications, where deploying the applications only to the PaaS provider is acceptable to users and when the pricing model is also adequate.

Table 9: PaaS vs SlipStream / Nuvla

Product/solution/vendor	Open Source?	Support on premise installations?	Application architecture agnostic?	Available as a Service?	Comment
SlipStream / Nuvla	●	●	●	●	

[3] https://paasfinder.org

OpenShift	●	●	●		
Cloudify	●	●	●		
Google App Engine				●	
Heroku				●	

6.5.1. Competitor or companion?

In this category, SlipStream is a competitor. It boils down to a trade-off between the balance of constraints / benefits of the generic PaaS vs flexibility / portability / multi-cloud benefits of SlipStream.

6.6. Brokerage Solutions or Services

This last category is less mature than the previous, yet offers many benefits. Our aim in brokering is to provide a platform (as a service) where users can self-provision resources and deploy simple and complex applications to a wide range of cloud services. Our *cloud brokerage* definition, therefore, does not include consultancy, search or relational services where humans are performing the brokering between customers and cloud providers.

While we do negotiate regularly with cloud providers, this is done behind the scene and transparently to the users, such they only see the result of these negotiations in our service catalogue (e.g. pricing).

Our vision is to automate this process, where users are provided with relevant, complete and rich information, such that they can select the right *offer* (i.e. combination of cloud services provided by a given provider). Selection criteria can include, for example:

1. SLA
2. Pricing
3. Location
4. Certification

Taking this definition of brokerage, there is little competition on the market. Several IaaS providers offer the possibility to purchase in bulk (e.g. reserved instances) to benefit from cheaper resources. However, the risk and management of these decisions and transactions are with the customers. The broker, as we define it, can decide to bulk purchase, on behalf of its customers, and deliver part of the corresponding savings to the customers. This means the broker performs the analysis and takes the decision, and the risk, to purchase bulk resources.

The benefit for the customer is guaranteed access to the cheapest cloud resources, with the possibility to access even cheaper prices if the broker can purchase in bulk.

In return, the broker takes a minimum brokerage fee, with the possibility to increase this profit if it can successfully sell bulk purchased resources to customers, thus creating a win-win situation.

We know of few companies attempting to deliver such a service. For example, DBCE (Deutsche Börse Cloud Exchange) tried something similar, but with a much more basic offer compared to the richness of SlipStream (or Nuvla), and failed since they stopped operating in March 2016. ComputeNext also proposed a brokerage service, where users can select applications.

As a summary, the following list includes providers that offer partial brokerage services:

1. Amazon Web Services
2. Exoscale
3. T-Systems
4. DBCE

The following table illustrates some of the benefits and shortcomings of these above services, compared to Nuvla's brokerage offer:

Table 10: PaaS vs SlipStream / Nuvla

Product/solution/vendor	Based on Open Source?	Support policy placement?	Support multi-cloud?	Support bulk purchase?	Managed bulk purchase?	Comment
SlipStream / Nuvla	●	●	●	●	●	
Amazon Web Services				●		
Exoscale	●					
T-Systems	●			●		
DBCE			●			

6.6.1. Competitor or companion?

As mentioned above, in the brokerage space as defined in this analysis, Nuvla currently has no clear competitor. Further, its growing support for a wide-ranging number of public clouds, and its ability to support private clouds, makes Nuvla's brokerage benefit highly valuable.

Building on the enabling foundations provided by the service catalogue feature of SlipStream, the brokerage benefits will grow even further. Indeed, with roadmap items including data driven placement, for example in the field of Earth Observation data processing, as well as IaaS provider certifications, the brokerage feature of Nuvla will offer richer attributes to use when choosing the right IaaS provider(s) and offers.

7. Conclusions

This document describes the CYCLONE solutions for dealing with functional and non-functional aspects of cloud computing. The solution relies on SlipStream (with extensions) to handle the provisioning and re-provisioning of resources as the operating conditions for a cloud application change over time. The specific decisions on what actions to take (scale-up, scale-down, migration, etc.) are defined by an application-specific policy. The general algorithms for detecting changes in metrics (used to trigger scaling actions) have been defined. These algorithms now must be brought together with SlipStream to validate the overall solution and to demonstrate the defined use cases.

The implementations of the new features in SlipStream and of the change detection algorithms are very advanced. Reviewing the status of the requirements for SlipStream (see Table 11) shows that most of them are already satisfied; we expect that all of them will be satisfied by the final solution.

The document also reviewed the state of the art for autoscaling, both in the academic and commercial worlds. The comparison of SlipStream with competing products and services has shown that many of them offer comparable features, but only for subsets of SlipStream's full feature set. SlipStream is still the only product to offer a comprehensive multi-cloud solution for all aspects of application management (including scaling) and brokering.

Table 11: Status of Requirements

1	✔	Triggering of scaling actions of an application based on application metrics using simple, predefined algorithms (e.g. adding node based on machine load).
2	✔	Triggering of scaling actions of an application based on application metrics defined by the developer of the application.
3		Ability to publish application-specific benchmarks of cloud providers into the Service Catalog or Open Service Compendium.
4	✔	Placement based on static characteristics (e.g. geographical location) of a cloud service provider.
5		Placement based on dynamic VM monitoring information from SlipStream itself.
6	✔	Placement based on external information pushed into the SlipStream Service Catalog or Open Service Compendium.
7		Placement based on the join of all information associated with a given cloud service provider.
8	✔	Ranking of selected cloud service providers based on predefined algorithms (e.g. price).
9		Ranking based on algorithms provided by the application developer and/or the application operator.
10		Ability to trigger notifications/alerts through SlipStream.
11	✔	Ability to trigger scaling actions from within the application.
12		Ability to search the Service Catalog and Open Service Compendium manually to see the results from various policies and to ideally then associate those policies with applications.

References

[ALI14]	Hanieh Alipour, Yan Liu, and Abdelwahab Hamou-Lhadj. “Analyzing auto-scaling issues in cloud environments”. In Proceedings of 24th Annual International Conference on Computer Science and Software Engineering, CASCON ’14, pages 75–89, 2014.
[BOU98]	Boudewijn R. Haverkort. “Performance of Computer Communication Systems: A Model-Based Approach”. John Wiley & Sons, Inc., 1998.
[BRO93)	B. Brodsky and B. Darkhovsky. Nonparametric methods in change-point problems. Springer, 1993.
[CIMI16]	Distributed Management Task Force, Inc., Cloud Infrastructure Management Interface (CIMI) Model and RESTful HTTP-based Protocol, 2016. https://www.dmtf.org/sites/default/files/standards/documents/DSP0263_2.0.0.pdf
[CON15]	Thomas Richard Connor and Joel Southgate, Automated Cloud Brokerage Based Upon Continuous Real-Time Benchmarking, 2015. http://ieeexplore.ieee.org/document/7431434
[DIA16]	Manuel Díaz, Cristian Martín, and Bartolomé Rubio. “State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing”. Journal of Network and Computer Applications, 67:99–117, 2016.
[DOU12]	Brian Dougherty, Jules White, and Douglas C. Schmidt. “Model-driven auto-scaling of green cloud computing infrastructure”. Future Generation Computer Systems, 28(2):371–378, 2012.
[EDN12]	Rich Hicky, EDN Format, 2012. https://github.com/edn-format/edn
[GAL07]	Pedro Galeano and Daniel Peña. “Covariance changes detection in multivariate time series”. Journal of Statistical Planning and Inference, 137(1):194– 211, January 2007.
[HUL01]	Geoff Hulten, Laurie Spencer, and Pedro Domingos. “Mining time-changing data streams”. In Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD ’01, pages 97–106, 2001.
[KRI17]	Michael T. Krieger, Oscar Torreno, Oswaldo Trelles, and Dieter Kranzlmüller. “Building an open source cloud environment with auto-scaling resources for executing bioinformatics and biomedical workflows”. Future Generation Computer Systems, 67:329–340, 2017.
[LOC16]	Locust, Locust, 2016. http://locust.io/

[LOR14]	Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. “A review of auto-scaling techniques for elastic applications in cloud environments”. Journal of Grid Computing, 12(4):559–592, 2014.
[NGI16]	Nginx, Nginx, 2016. https://www.nginx.com/
[QU16]	Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. “A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances”. Journal of Network and Computer Applications, 65:167– 180, 2016.
[RIEC16]	Riemann, Clients, 2016. http://riemann.io/clients.html
[RIES16]	Riemann, Riemann, 2016. http://riemann.io
[SAPI16]	SixSq Sàrl, SlipStream API Reference, 2016. http://ssapi.sixsq.com
[SCO79]	David W. Scott. “On optimal and data-based histograms”. Biometrika, 66(3):605–610, 1979.
[STU26]	H.A. Sturges. “The choice of a class interval”. Journal of the American Statistical Association, (21):65–66, 1926.
[UC7]	CYCLONE, UC7: Open Scientific Data, 2015. https://cyclone.france-bioinformatique.fr/usecases/view/160
[UC8]	CYCLONE, UC8: Benchmark Driven Placement, 2015. https://cyclone.france-bioinformatique.fr/usecases/view/161
[UCC15]	IEEE, 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC), 2015. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7430473
[WID96]	Gerhard Widmer and Miroslav Kubat. Learning in the presence of concept drift and hidden contexts. Mach. Learn., 23(1):69–101, 1996.
[ZEN16]	Zencoder. Zencoder. 2016. https://zencoder.com/en/

Glossary

B2B	Business to Business
CSP	Cloud Service Provider
DC	Data Center
E2E	End to End
IaaS	Infrastructure-as-a-Service
IPR	Intellectual Property Rights
IT	Information Technology
MaaS	Metal as a Service
NaaS	Network-as-a-Service
Net-HAL	Network Hardware Abstraction Layer
NFV	Network Function Virtualization
PaaS	Platform-as-a-Service
PC	Project Coordinator
PMB	Project Management Board
PoP	Point of Presence
QoS	Quality of Service
SaaS	Software-as-a-Service
SCI	Smart Core Interworks
SDN	Software Defined Networks
SP	Service Provider
TC	Technical Coordinator
TCTP	Trusted Cloud Transfer Protocol
TMB	Technical Management Board
WP	Work Package
WPL	Work Package Leader