seaclouds

AGILITY AFTER DEPLOYMENT

Modelling Planning Controlling

SeaClouds Project

D4.3 Design of the run-time reconfiguration process

Project Acronym	SeaClouds
Project Title	Seamless adaptive multi-cloud management of service-based applications
Call identifier	FP7-ICT-2012-10
Grant agreement no.	Collaborative Project
Start Date	1st October 2013
Ending Date	31st March 2016
Work Package	WP4. WP SeaClouds run-time environment
Deliverable code	D4.3
Deliverable Title	Design of the run-time reconfiguration process
Nature	Report
Dissemination Level	Public
Due Date:	M16
Submission Date:	16th February 2015
Version:	1.0
Status	Final
Author(s):	Miguel Barrientos (UMA), Jose Carrasco (UMA), Javier Cubo (UMA), Elisabetta Di Nitto (POLIMI), Adrián Nieto (UMA), Diego Pérez (POLIMI), Román Sosa (ATOS), Christian Tismer (NURO), Andrea Turli (CloudSoft), PengWei Wang (UPI)
Reviewer(s)	Javier Cubo (UMA), Antonio Brogi (UPI), Ernesto Pimentel (UMA)

Dissemination Level

Project co-funded by the European Commission within the Seventh Framework Programme		
PU	Public	X
PP	Restricted to other programme participants (including the Commission)	
RE	Restricted to a group specified by the consortium (including the Commission)	
CO	Confidential, only for members of the consortium (including the Commission)	

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	08/01/15	First ToC	Jose Carrasco
0.2	12/01/15	Second ToC and deadlines	Jose Carrasco, Javier Cubo
0.3	23/01/15	Third ToC and assignments of tasks	Jose Carrasco, Adrián Nieto, Miguel Barrientos, Javier Cubo
0.4	29/01/15	First contributions and minor modifications in the ToC	Jose Carrasco, Adrián Nieto, Miguel Barrientos, PengWei Wang, Román Sosa, Andrea Turli, Javier Cubo, Diego Pérez, Elisabetta Di Nitto, Christian Tismer
0.5	11/02/15	Revision of the first contributions and second contributions	Jose Carrasco, Javier Cubo, Adrián Nieto, Miguel Barrientos, Román Sosa, Andrea Turli
0.6	13/02/15	Complete review of a more stable version	Antonio Brogi, Ernesto Pimentel
1.0	16/02/15	Stable version after the reviews	Jose Carrasco, Javier Cubo

Table of Contents

Executive Summary 6
1. Introduction 7
1.1 Structure of the document 7
1.2 Terminology 7
1.3 Glossary of Acronyms 8
2. Problem statement, motivation and approach 9
2.1 Related work 9
2.2 SeaClouds Reconfiguration approach 11
3. SeaClouds Reconfiguration Strategies 14
3.1 SeaClouds components involved in the Reconfiguration process 14
3.2 Main differences between scenarios of Repairing and Replanning 15
3.3 Triggering and Reconfiguration Request 16
4. IaaS and PaaS Challenges in Reconfiguration 17
5. Repairing Strategy 19
5.1 Repairing Scenarios addressed in SeaClouds 19
5.2 SeaClouds Repairing Mechanisms 22
5.3 SeaClouds Use Cases 24
6. Replanning Strategies 31
6.1 Replanning Scenarios addressed in SeaClouds 32
6.2 SeaClouds Replanning Mechanisms 36
6.2.1 Generation of the Replanning Application Model 38
6.2.2 Dependencies Detection and Management 39
6.3 SeaClouds Uses Cases 41
7. Data Migration and Synchronization 44
8. Conclusions 47
References 48

List of Figures

Figure 1. Reconfiguration strategies in the SeaClouds Platform (numbers into circles represent the order of execution of activities for each strategy). 12

Figure 2. Reconfiguration process in the SeaClouds platform considering the functionality and connections of the different components. 15

Figure 3. Deployment of a Plan (DAM) in SeaClouds. 20

Figure 4. Scenario of a violation for repairing. 21

Figure 5. Scenario of a repairing action in SeaClouds. 21

Figure 6. SeaClouds Repairing strategy overview. 23

Figure 7. NURO Cloud game topology overview. 24

Figure 8. Examples of Stateless and Stateful modules. 33

Figure 9. Replanning on a stateless module. 34

Figure 10. User/Alert triggered migration of a stateful module. 36

Figure 11. SeaClouds Replanning strategy overview. 37

Figure 12. Application topology of the running example. 38

Figure 13. Providers searching that will be used in the migrations. 39

Figure 14. Replanning, triggering effectors on the Deployer component. 40

Figure 15. Hegira4Cloud high level architecture. 46

List of Tables

Table 1 Acronyms. 8

Table 2. Increase of requests use case. 27

Table 3. Decrease of requests use case.. 28

Table 4. PHP node failure use case.. 29

Table 5. DB overload use case. 29

Table 6. Follow the user use case. 30

Table 7. Reconfiguration Application Plan example. 37

Table 8. Migration description which is contained in a RAM.......................... 39

Table 9. Stateless Front-end migration use case... 42

Table 10. Stateful Backend migration use case... 43

Executive Summary

In this deliverable, a first design of the run-time reconfiguration process, trying to preserve the soundness when requirements are violated, is documented. How the reconfiguration mechanisms will generate reconfiguration recommendations is detailed here. Thus, the reconfiguration process will suggest to the cloud functionalities to be replaced by re-establishing the soundness of the deployed adaption in case violations of QoS properties and Service Level Agreements occur. Also, the design of the connection with the planning, deployment and monitoring processes is considered in this document. Finally, the data migration and synchronisation process as a value-added to SeaClouds is described.

1. Introduction

This document presents the design of the runtime reconfiguration process in SeaClouds. A reconfiguration process involves several steps, starting from the definition of the concepts around the process to the strategy or algorithm involved in the decisions around the process.

In SeaClouds, a reconfiguration starts when, once the application has been deployed according a plan, the Monitor component detects a violation on a requirement or an user-defined SLA rule and the user confirms the need of a reconfiguration of the application at runtime, also considering other actions which could fix the problem without the need of generating a new plan. Although usually the reconfiguration process will need a user confirmation, the system could be preconfigured to autonomously decide a replanning under some circumstances.

1.1 Structure of the document

After giving the definition of the core concepts of a reconfiguration process in Section 1.2, where we define what a Reconfiguration, both Repair (or Self-healing) and Replanning strategies means, Section 2 presents the problem statement and motivations why and how the SeaClouds platform is tackling reconfiguration, also comparing previous ideas related to cloud reconfiguration.

In Section 3, we describe more in detail the differences between the reconfiguration strategies considered, as well as the explanation of when the reconfiguration is triggered. In Section 4, the challenges and differences of the reconfiguration at IaaS and PaaS level are given. This covers to deal with VM, servers, network, or storage at IaaS level and with operating system, programming language execution environment, database, or web server at PaaS level.

Section 5 and 6 detail the Repairing and Replanning strategies, more in deep, respectively. For both cases, we first explain when the specific strategy is performed exemplifying this with addressed scenarios, and how SeaClouds proposes to implement the strategy using also the concrete NURO Cloud Gaming case study in which SeaClouds' consortium is working (use cases of the Social Networking case study could be also adopted in a similar way, and it will be done in the next steps for the real adoption, although for simplification reasons, in this document we have decided to illustrate the strategies by using only one of the case studies).

In Section 7, we introduce a first attempt to address the specific problems with the reconfiguration of information, storage or databases of cloud applications. Finally, Section 6 presents some conclusions for this document.

1.2 Terminology

In this section, we list the main concepts we will use in the reconfiguration process in SeaClouds.

Reconfiguration: Reconfiguration allows to manage issues in an application deployment in a reactive way. Reconfiguration process detects when application requirements are violated and it tries to fix it using different mechanisms depending on the kind of violation. Each issue could be accomplished using a concrete management methodology in order to ensure the expected application behaviour. In SeaClouds, we consider two different reconfiguration strategies: repairing and replanning.

Repairing: This reconfiguration strategy tries to solve an incident of the application status (like a failure in some modules). In this case, SeaClouds detects the cause of the malfunctioning and then it tries to reach a stable and expected application status using the operations supplied by deployment resources, mainly stop/start/rescale resources. In the SeaClouds context, we consider the technique of **self-healing** is a synonym for repairing.

Replanning: In this reconfiguration strategy, SeaClouds detects that the current deployment status does not work as it was expected, and a repair action cannot resolve the situation, so a replan is needed. Then, SeaClouds finds the resources or application modules that exhibit a bug or fail in its performance and tries to fix the detected issues redeploying (moving or replanning) the resources or modules over different cloud providers, in order to take advantage of the features offered by those new providers. This entails a cost that has to be considered in the SeaClouds replanning strategy.

1.3 Glossary of Acronyms

Here we list the different acronyms which will be used in this document.

Acronym	Definition
SaaS	Software-as-a-Service
PaaS	Platform-as-a-Service
IaaS	Infrastructure-as-a-Service
DaaS	Database-as-a-Service
QoS	Quality of Service
QoB	Quality of Business
SLA	Service Level Agreement
GUI	Graphical User Interface
API	Application Programming Interface
APP	Application
DB	Database
DAM	Deployable Application Model
RAM	Reconfiguration Application Model
VM	Virtual Machine
HA	High Availability

Table 1 Acronyms.

2. Problem statement, motivation and approach

Since its very foundation, cloud computing is a representative of the concepts of dynamism, changes and adaptations. Its proclaimed resource elasticity and on-demand resource acquisition and release are good examples of such dynamism.

Calling *configuration* to a snapshot of the resources allocated to an application at a given moment, a *reconfiguration* is what happens when changing between two configurations. Reconfigurations take place when the application is not executing as expected the current configuration, or when there are devised configurations more suitable than the current one (e.g., in terms of cost of cloud resources).

The reconfiguration process is applied over the system in execution providing mechanisms for rescheduling and re-execution of the modules, with the purpose of foreseeing the compliance of the properties and the soundness of the orchestration among the application modules and their connections.

Thus, SeaClouds will offer different reconfiguration alternatives to be selected by the application provider in an interactive way, allowing the deployment of the modules over the different clouds according the distribution plan generated in the Planning phase. In this line, the monitoring service we propose uses provider-independent metrics (defined by SeaClouds) to provide the status of the application as well as of the single services composing it. This monitoring process, in connection with the planning and deployment processes, can detect the need of load-balancing or distribution of Cloud services on several Cloud providers, and interact with the reconfiguration module in order to determine when it is required to perform an evolution or migration of the services. As a consequence of monitoring, dynamic reconfiguration can be used to evolve the orchestration by considering all the changes required (without suspending the execution of services not affected by those changes). Evolution may imply updating a service, dynamically replacing erroneous services or migrating it to a different Cloud provider to leverage its advantages or avoid the shortcomings of another Cloud provider.

In the following sections, we first present related works doing efforts in cloud reconfiguration, and then, we presents how SeaClouds tries to solve this problem.

2.1 Related work

The ideas around the reconfiguration of the application modules in cloud computing are not something new, so previous works have proposed some ideas.

An important issue is as regards the cost of reconfiguring. Thus, in [1, 2] the authors describe a hierarchy to group the different kinds of reconfiguration that could be done in a cloud application, addressing these issues from the point of view of cost-aware.

An analysis of the different reconfiguration strategies in [3] motivates a guideline to develop cloud applications using a set of techniques which allows to avoid the main Cloud Computing issues, like as vendor lock-in [4, 5]. Moreover, this paper proposes a new classification to achieve the aforementioned kind of reconfigurations. However they do not take into account the possibility of performing a cross-provider migration neither the option to move only one application module instead of the whole application, which is a major focus on SeaClouds.

In many works, authors refer to the possible reconfiguration strategies as dynamic scaling (up/down scale), replication and migration (online or offline), in accordance with the reconfiguration strategies analyzed and proposed by SeaClouds (for example, local resizing is one of the repairing examples studied in Section 5.3). As we will describe along this document, several techniques are used to reach these goals in SeaClouds.

In [6] the authors define a methodology to ensure an efficient autoscaling in a Cloud system, using predictive models for workload forecasting. It provides a set of key parameters to establish the system's behavior focusing on linear equations for the quantification of the constraint violations and the cost of the reconfiguration operations necessary to fix the system status for scaling a system. Like the previous work, [7] proposes to improve the efficiency of the system through the analytical performance (based on a queueing network system model) and using workload information to predict the application behavior and optimize the virtual machines provisioning to the application requirements.

For ensuring an efficient distribution over potentials cloud providers, Schroeter et al propose in [8] a system based on applications and cloud models using an extended feature of EFM [9, 10]. Using SaaS applications, the users are able to compose their own multi-tenant system to provide a managed service. The provisioning of this system composition could be reconfigure dynamically in order to scale and modify the used resources.

A different technique is used by Haibo et al [11] using a genetic algorithm to carry out the autoscaling in a cloud system. In this case, it focuses on reaching an efficient energy consume in the provider's data centers.

Compared to these approaches, as aforementioned, a key goal of SeaClouds is to allow the migration of application modules between different clouds providers.

In [12] the authors describe mechanisms to allow the dynamics reconfiguration in distributed software system, which also could be considered cloud. This work provides a way to find a first efficient provisioning of a system over a set of resources and services based on key features as cost or time. A proposed component, called Planner, is in charge of composing a provisioning plan to distribute the application over the expected providers. Moreover, it proposes monitoring techniques to monitor the application to know the current status and behavior of a deployed application and the computing resources. Then, depending on the collected data it determines if a reconfiguration is necessary, for example, migration, scaling, etc. Again, the Planner composes any reconfiguration plans to fix the issues of the deployed application. This work, applied mainly to distributed systems, is in the same line that SeaClouds, although in SeaClouds we go beyond with the idea of considering multi-cloud deployment and migration of individual modules of the application.

Leyman et al describe in [13] a distribution process based on top-down and bottom-up analysis techniques for finding the best application provisioning. A predictive model of the workload based on statistical distributions and the real system behaviour allow to predict the performance of the application when it is deployed over a cloud provider. These techniques are integrated in a distribution and reconfiguration proposed process, which describes the necessary steps to carry out the distribution and pos-deployment management, including the migration, of an application. Even though unlike the work above, this is focused on the cloud environments, but it does not devote as many resources for describing the accomplishment the reconfiguration process.

In summary, although the aforementioned works provide several ideas or solutions, our proposal in SeaClouds considers all the proposed concepts and unify them inside a multi-cloud environment where a single application module can be reconfigured with independence whether the goal is to perform a reconfiguration inside the same provider or not. Also we go in two directions as regards the strategies, such as it will be explained below.

2.2 SeaClouds Reconfiguration approach

In the SeaClouds reconfiguration approach, we can distinguish two different triggers for a reconfiguration process: human-triggered reconfigurations and automatic reconfigurations. In human-triggered reconfigurations, a user could define a new plan and order its execution to the Deployer. To avoid both human errors and requiring continuous attention of application's owner, reconfigurations can be also triggered automatically.

Figure 1 depicts the reconfiguration strategies considered in the SeaClouds platform, where we represent the options of reconfiguring with the decision of the user (Human) of in an automatic way (Automatic - Monitor) based on monitoring mechanisms to determine when it is required, interaction with the Planner and the Deployer components.

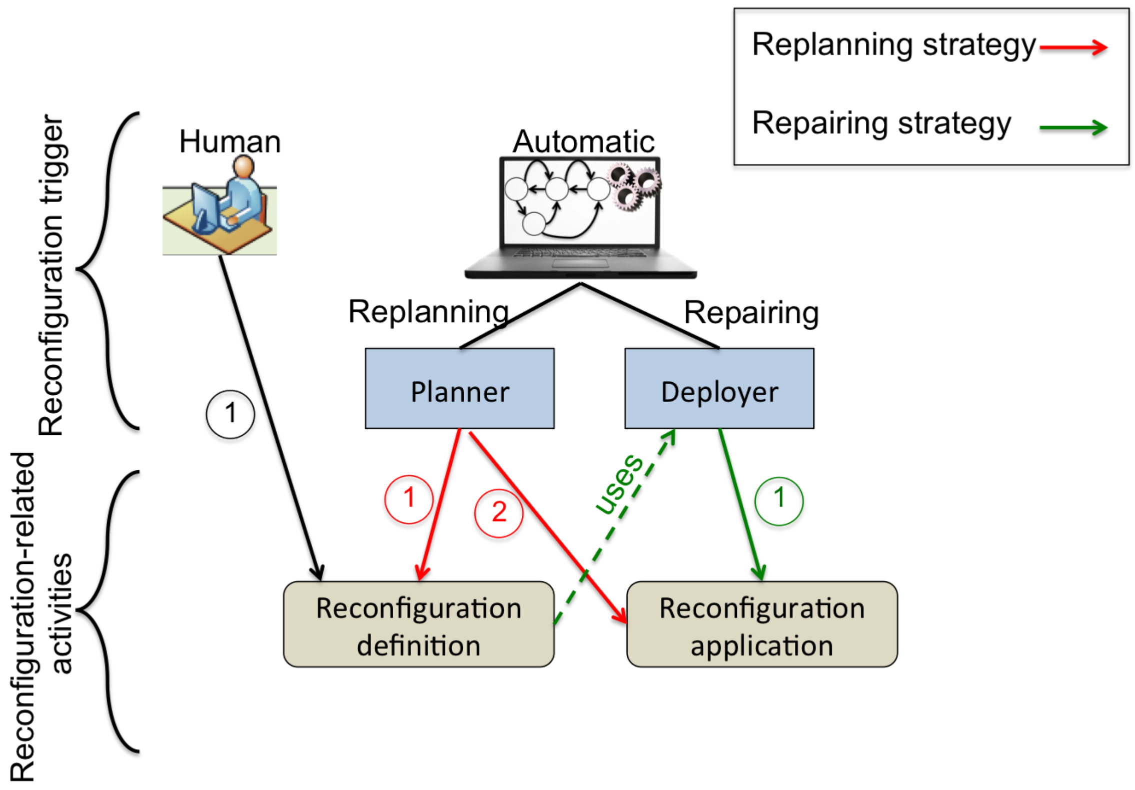

Figure 1. Reconfiguration strategies in the SeaClouds Platform (numbers into circles represent the order of execution of activities for each strategy).

For an automatic reconfiguration, it is needed first to compute how to reconfigure (i.e., to define the reconfiguration process using the available reconfigurations actions, as indicated in the steps in the figure, such as: migrate components, deploy/undeploy components, start/stop components, replicate components, delete component replica, change the maximum number of replicas for a component, etc.), and later apply these reconfiguration actions (see steps in the figure).

Among the automatic reconfiguration processes, we can distinguish again between two types, foreseeable ones that are expected to be executed often and where the state of the environment can be predicted beforehand, and unpredictable or unusual ones where the environment features cannot be predicted reliably.

SeaClouds Planner has the capability to define new reconfiguration processes even if they have never been considered before in the cloud environment, for example, the unreachability of a complete cloud provider. This capability allows to solve some problems that may happen during the application runtime but cannot be predicted the first time that the application is deployed. This capability is implemented by the reconfiguration strategy called **replanning**. For doing this replanning, it requires global information as a snapshot of the state of the environment. This information of the current situation comes from the Monitor, SLA service, Discoverer and Deployer.

Notwithstanding the value of a module that can decide for suitable configurations even under situations that had never seen before, the replanning activity takes time and cost efforts, since it has to deliberate the suitability of a new Deployable

Application Model (DAM) among a large set of possible alternatives, and consider the cost of replanning. Therefore, the time and cost required for its computation are not negligible. This is inconvenient for applications that should reconfigure often because the application spends too much time being deployed in non-suitable configurations.

To improve this situation, SeaClouds implements a complementary reconfiguration strategy called **repairing.** In this case, the Monitor will trigger the available effectors (Deployer effectors) to solve the ongoing problem as soon as some violation situations occur (e.g., when some characteristics are perceived, or when arrives a clock-based event for applying the *"follow-the-sun"* policy).

On the one hand, repairing strategy allows the application to reduce its amount of time in wrong configurations because it skips the deliberation activity that defines the reconfiguration; so it is suitable for often execution at runtime. Moreover, the existence of repairing strategy releases the Planner from computing several times the same reconfiguration definition, by reducing the cost of replanning. On the other hand, repairing strategy is effective to execute reconfigurations for a-priori on user on-demand situations.

It is worth noting that, in the example previously mentioned, regarding the unreachability of a cloud provider, although it could be predictable event, it would be required to also predict the state of the rest of the cloud providers in order to anticipate the reconfiguration definition.

3. SeaClouds Reconfiguration Strategies

In this section, we describe the concepts around the application reconfiguration strategies in order to clarify how the SeaClouds Platform should manage a reconfiguration process.

3.1 SeaClouds components involved in the Reconfiguration process

Monitoring and managing applications is an arduous task. An application management solution should consider multiple runtime metrics to understand effectively and possibly efficiently the status of the application in order to manage it correctly.

In SeaClouds, "the source of truth" is the Deployable Application Model (DAM), the concrete plan that is executed by the Deployer Engine. The Deployer Engine, in fact, executes DAM concrete plans produced by the planning stage. In the planning stage, SeaClouds assembles a plan that contains the modules of the application and the topology, the concrete services that execute the modules, and the policies to fulfill user requirements, like SLA or cost-constraints. Just to give a very basic example, a SeaClouds' user can ask to deploy a 3-tier web application in EU ensuring that its response time will always be less than 300 ms but without spending more than x euros/month. This (simple) example shows the complexity of managing applications: those are reasonable constraints, but the SeaClouds platform will have to translate these high-level constraints into a DAM concrete plan that the platform can run and manage.

As we can see in Figure 2, after an alert is triggered we differentiate two reconfiguration types. Repairing–occurs when the violation can be fixed without the need of generating a new plan, because it was previously considered by the user or even it is dynamically order by the user. In this strategy, the Monitor is directly connected to the Deployer to check the actions which the Deployer could fix. However, if the situation cannot be fixed or it was tried to fix it but the action failed, then a Replanning is performed, calling the Planner.

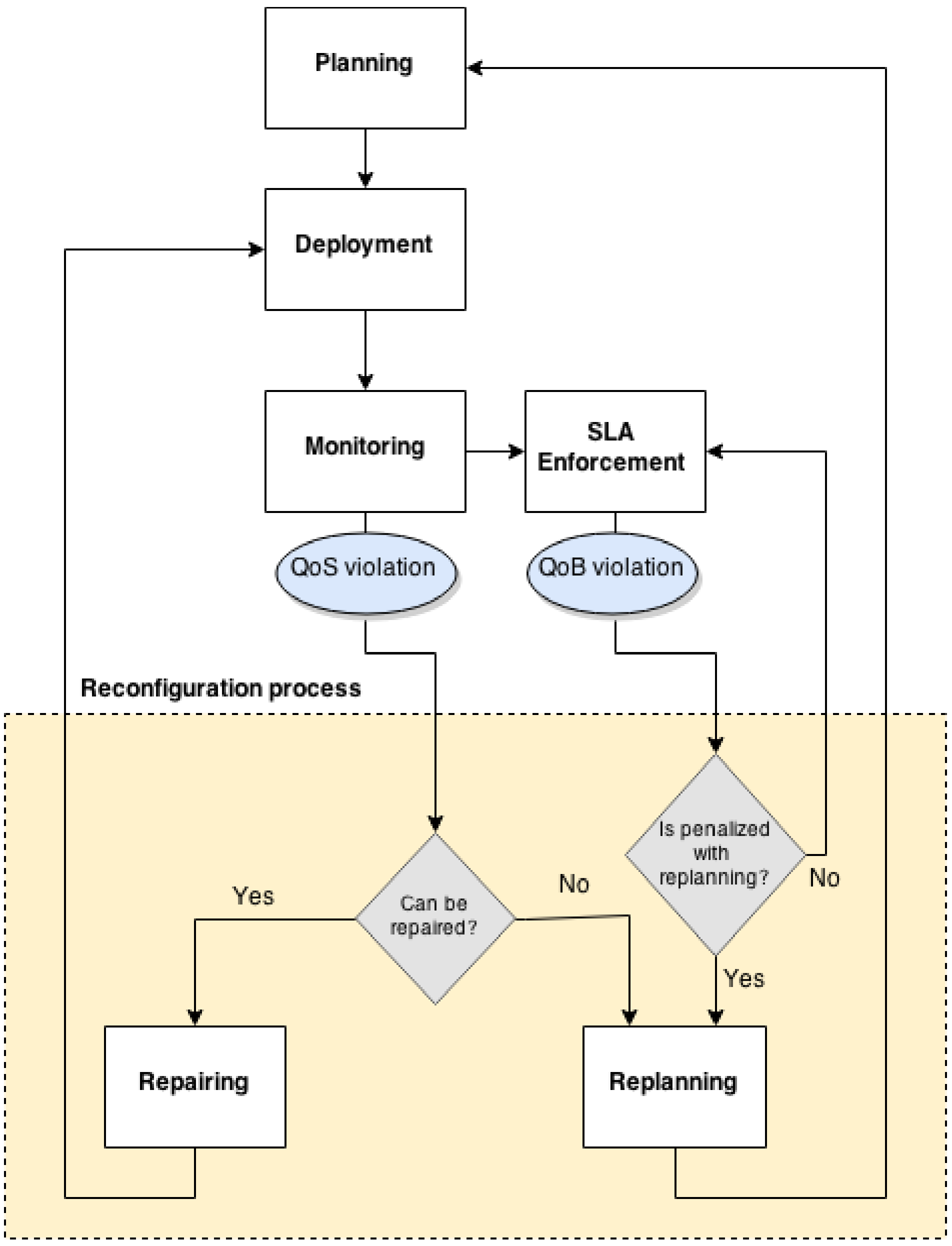

Figure 2. Reconfiguration process in the SeaClouds platform considering the functionality and connections of the different components.

3.2 Main differences between scenarios of Repairing and Replanning

The aim of reconfiguration process is to fix the requirement violations after the application has been deployed by SeaClouds Deployer. Depending on different kinds of violations, repairing and replanning are two different reconfiguration strategies in our SeaClouds platform.

As described in the previous sections, the repairing reconfiguration is based on the management of deployment resource, which allows SeaClouds to adjust the deployed application according to the runtime information and related monitoring rules (previously defined by the user, or even requested at runtime), using the Deployer Engine effectors. Then, it involves dynamic changes or fixed fails of some components or the entire application. The applicable scenarios mainly include replacing/restarting a failed component, scaling to meet the demand, and applying a follow-the-sun policy. By contrast, the replanning reconfiguration will try to handle the cases that cannot be solved by repairing. It needs to modify the plan specified in the DAM, that describes the distribution of the application modules, and do a redeploying. Thus, replanning cannot be completed independently by only the Deployer Engine, but also needs the work of planner to update the DAM which may contain new distribution of the application modules. Replanning reconfiguration will be more complicated than the repairing, since migration may happen in this process.

In conclusion, repairing and replanning are two different reconfiguration strategies that try to handle different reconfiguration scenarios. Repairing can be implemented automatically considering the Monitor and Deployer (Engine), while replanning involves a more complex process (not only considering some failures, but also triggered by some QoB violations) and needs to involve also the Planner to generate a new plan.

3.3 Triggering and Reconfiguration Request

As depicted in Figure 2, the necessary alerts to perform a reconfiguration are triggered by two components: the Monitor and the SLA Service components.

On the one hand, the Monitor component is responsible for the control and enforcement of Quality of Services (QoS) properties, as well as forwarding violations of these properties to the interested subscribed modules.

Each time a violation is detected by the Monitor, it analyzes if SeaClouds can fix the situation. If so, the Deployer Engine performs a repairing reconfiguration through the Deployer effectors. If not, a replanning trigger is sent to the Planner component, where a replanning reconfiguration is performed.

The point is that the Monitor component works at real-time, having a close look at the performance of the application, and reacting immediately.

On the other hand, the SLA Service component is listening for violations that impact on the business of the application, performing a long term analysis. i.e., Quality of Business (QoB). If a QoB policy states that a migration should occur if violated, then the SLA component triggers a replannification alert to the planner.

4. IaaS and PaaS Challenges in Reconfiguration

This sections describes the challenges of the reconfiguration process at both levels, IaaS and PaaS, in a general way and how SeaClouds is proposing to do it.

Reconfiguration at PaaS Level is the continuous process of "managing" the life of a business application where its modules have been distributed over multiple PaaS providers. In this context "managing" means the automatic or semi-automatic capability to react to the change of the environment in term of QoS and QoB. Some aspects should be considered at PaaS level reconfiguration are the operating system, programming language, execution environment, database, or web server.

On one hand, when a QoS or QoB violation occurs, SeaClouds should be able to move part of the business application to another cloud offering and automatically reconfigure the entire system (business application and cloud services) to ensure the correct behavior of the application and its modules.

On the other hand, the reconfiguration at PaaS level might also happen for the involvement of the Application Operator. Actually, the SeaClouds multi-cloud governance and re-configuration facilities give application “developers & operators” the visibility and the necessary control across private and public PaaS clouds to have all the time the business application under control. The ability to check the status and performance of the application and even compare it to other applications in an abstract way can provide a great benefit to developers & operators by providing data that can be used to take manual decisions (with the help of the SeaClouds GUI) such as the scaling or migration actions of the business application.

Also reconfiguration at IaaS level should be tackled. We have identified two types of reconfiguration: repairing and replanning. As regards the replanning, our approach will consider two types: replanning of stateless modules, and replanning of stateful modules (see Section 6.2). Based on our previous experience on IaaS and PaaS providers [14, 15, 16, 17], we can mention the following circumstances as challenges to be addressed by SeaClouds:

- Stateful modules, as explained in Section 6.2,
- PaaS reconfiguration limited by operations provided by PaaS providers,
- Handle IaaS and PaaS reconfigurations in a uniform way, without imposing a certain configuration scheme to the application.

Regarding the migration of stateful modules, the migration of state might be dependent on the type of module and even of the state itself. This leads to a situation where an ad-hoc procedure has to be written in order to perform the migration.

This is even worse in the case of PaaS providers, where the needed primary operations to restore the module status in the new provider may not be available. For example, to migrate a database to Pivotal CloudFoundry, where the provider does not supply a load-database operation, and delegates that action on external providers, such as Flyway [18].

Another aspect related with PaaS providers affects the repairing reconfiguration, where the lack of some scaling operations limits the efficiency of the reconfiguration.

Returning to the database example, PaaS providers have different mechanisms to create services and inject the needed credentials to dependant modules. This is a problem in an initial deployment, and of course, in a redeployment or migration. For example, CloudFoundry assigns an environment variable VCAP_SERVICES [19] to each module where all credentials for binded services are stored in JSON format, while OpenShift stores the credentials in several environment variables [20]. Other providers provide the service's credentials to the user on creation time, so they must be configured in the application by the user or by the deployer. This mechanism is the usual case in IaaS deployments.

These challenges exposed are hard to overcome, so the SeaClouds' consortium will need to set a list of priorities and solving appropriately some of them, avoiding the situation where SeaClouds solves all of them, but in a wrong way.

5. Repairing Strategy

In this section, the repairing strategy is described more in detail. To do this, next, a general introduction is given. Then, the supported repairing scenarios in SeaClouds are described, and the mechanisms SeaClouds uses to solve these scenarios are also presented. Finally, the NURO Cloud Gaming case study is used to illustrate some use cases which could be solved by using the SeaClouds repairing mechanisms.

Today's increasingly complex systems, composed of a variety of components, operating in large-scale distributed heterogeneous environments, require more and more skills to install, configure, tune, and maintain. Determining the root cause of software runtime failures in such complex systems can be problematic and an automated support is clearly beneficial. Ideally, such complex systems would be able to recognize and solve a large portion of these errors on their own.

To this purpose, these systems would need to know when and where an error state occurs, to have adequate knowledge to stabilize themselves, to be able to analyze the problem situation, to make repairing plans, to suggest various solutions to the system administrator and/or to heal themselves without human intervention.

Autonomic computing has been proposed as a way to reduce the cost and complexity of systems, to control their manageability and to achieve the above desired situation.

Self-management is central to autonomic computing and addresses four tasks: self-configuring, self-healing or repairing, self-optimizing, and self-protecting. Next, we present the scenarios related to repairing

5.1 Repairing Scenarios addressed in SeaClouds

This section describes the repairing scenarios covered currently by SeaClouds using its Monitor and Deployer Engine.

The ultimate goal of a repairing system is to equip current distributed applications with strategies with which they can determine by themselves the root cause of their software runtime failures and make plans or suggestions for repairing these failures. To reach this goal, the following design criteria are considered:

- No application source code is required.
- Performance overhead should be acceptable.
- For those failures that cannot be healed, the cause should be diagnosed.
- All repairing actions should be reported to system administrators.

Then, sensors can be instrumented at the strategic positions in the application to report different information, like application failures or application content, that are instrumented in all strategic components (objects) of the application. Thus, it allows to provide the knowledge about the application behavior as we can describe above.

In this context, repairing is not only a property of an autonomic system, but it is also one of the reconfiguration strategies supported by the SeaClouds platform. Repairing tries to restore the status of an application without asking for a replan. In fact, during the planning stage, SeaClouds platform prepares a DAM which contains the topology of the application that is going to be deployed, and also a set of rules or policies are specified containing strategies to respond to runtime failures. Therefore, the inputs to the Planner are the result of a combination of inputs (user-requirements environment constraints) that will be translated into a pre-defined set of actions that can be used to mitigate runtime troubles. This peculiar SeaClouds feature is described into details in the following.

As depicted in Figure 3, after deploying a DAM, and during all the lifecycle of the application, SeaClouds monitors the status of crucial sensors of the application itself and it is able to react to particular situations using the pre-defined actions which already are defined by the user.

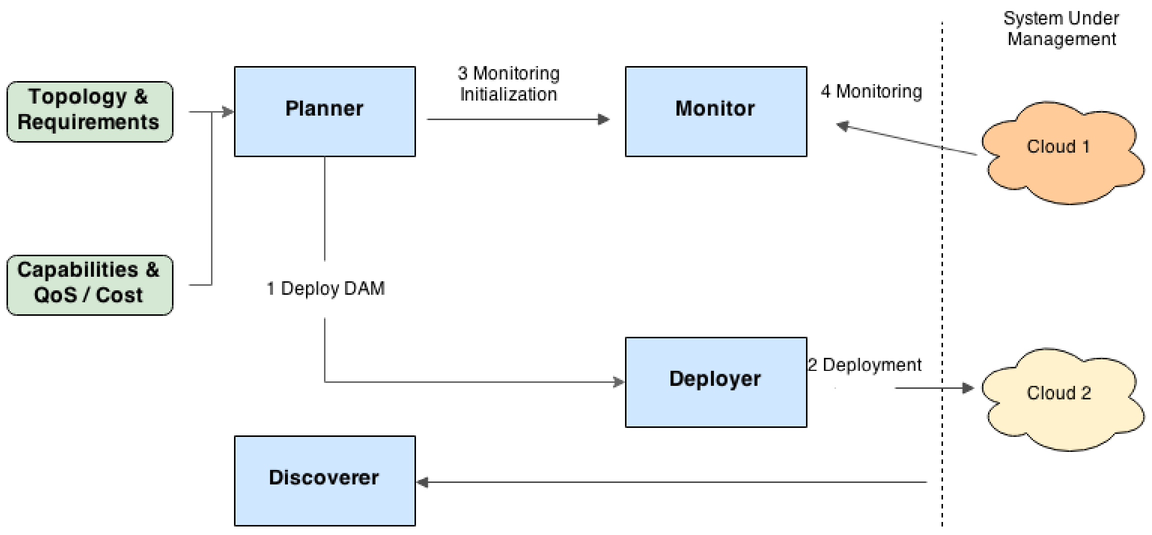

Figure 3. Deployment of a Plan (DAM) in SeaClouds.

In the scenario depicted in Figure 4, SeaClouds detects an underperforming application, according to predefined constraints and SLAs: *"response time to long"*.

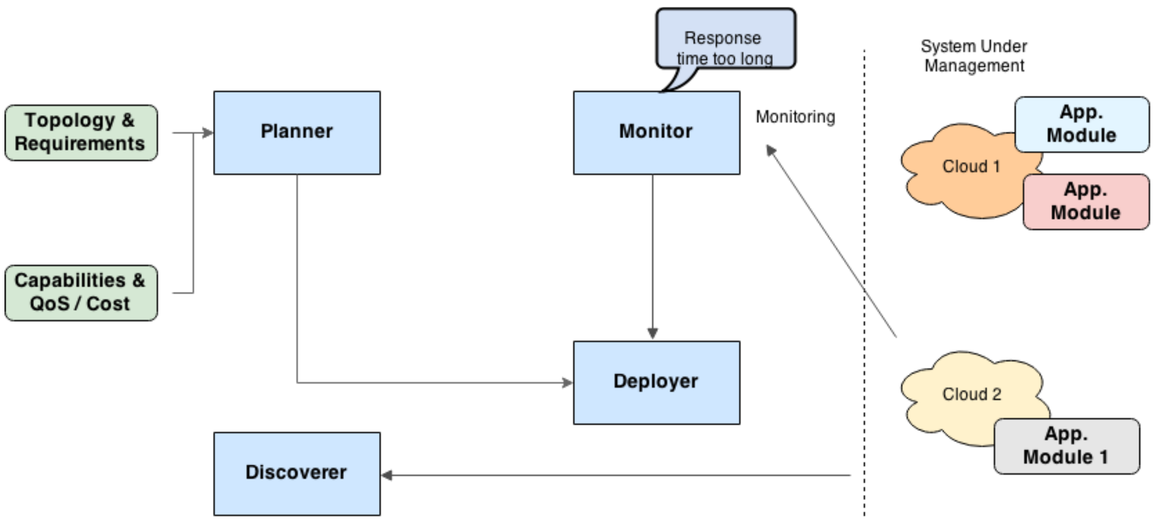

Figure 4. Scenario of a violation for repairing.

Then, SeaClouds Monitor looks among the monitoring rules available and applies the sensible one, in order to stabilize the application behavior. This is accomplished using the operations offered by each resource of the application. Specifically, how illustrated in Figure 5, the Deployer deploys an additional instance into the Cloud 1, performing a repairing action.

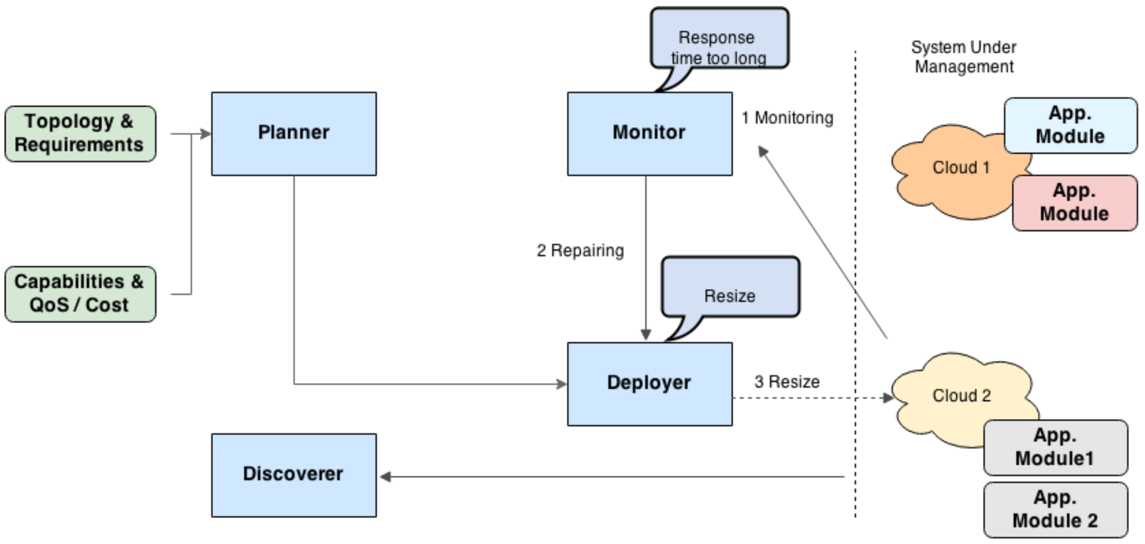

Figure 5. Scenario of a repairing action in SeaClouds.

SeaClouds implements several repairing strategies (based on the ones in [21]) to adapt to the workload and reduce the associated cost to the applications, where some of them are callable by the user or must have been planned by the user at design time:

- High availability (HA)
  - ServiceReplacer: it can be attached to a DynamicCluster and replaces a failed member in response a configurable sensor; if this fails, it sets the

Cluster state to on-fire, which is the state that indicates a problem (e.g., a Database).

- ServiceRestarter: it is attached to a SoftwareProcess and invokes restart on failure; if there is a subsequent failure within a configurable time interval, or if the restart fails, this gives up and emits an entity restart failure signal (e.g., a PHP module).

- Autoscaling
  - AutoScalerPolicy: it is attached to a Resizable entity and dynamically adjusts its size in response to specific emitted events. Alternatively, the policy can be configured to keep a given metric within a required range (e.g., a Database).
- Loadbalancing
  - LoadBalancingPolicy: it can be attached to a pool of containers that can host one or more migratable items. The policy monitors the workrates of the items and effects migrations in an attempt to ensure that the containers are all sufficiently utilized without any of them being overloaded. In addition to balancing items among the available containers, this policy causes the pool Entity to emit events when it is determined that there is a surplus or shortfall of container resource in the pool respectively. These events may be consumed by a separate policy that is capable of resizing the container pool (e.g., a PHP).
- **Follow-the-*** (a scheduled task)
  - Follow the Sun (Moon/Wind/Kilowat) (Inter-Geography Latency Optimization): Policy for moving work around to follow the demand; the work can be any Movable entity, where a Movable entity represents an item that can be migrated between balanceable containers (e.g., region information sensor, or proxy).

5.2 SeaClouds Repairing Mechanisms

In SeaClouds, the Deployer Engine is an application management system, so it is in charge of two crucial tasks: deploy the plan which can consist of multiple components that need to be configured and integrated across multiple machines, and interacts with the Monitor to monitoring key application metrics; scale to meet demand; and restart and/or replace failed components. We refer this second task as `repairing`.

The Deployer Engine is allowed to modify the deployed application whenever it is required by following the user or the Monitor orders. According to the runtime inputs and the policies defined at the level of the entire application and/or components, Deployer Engine owns a number of tools to adjust the application using the instructions (rules or policies). For example, the Deployer Engine can add more resources to a deployed application to meet the growing demand.

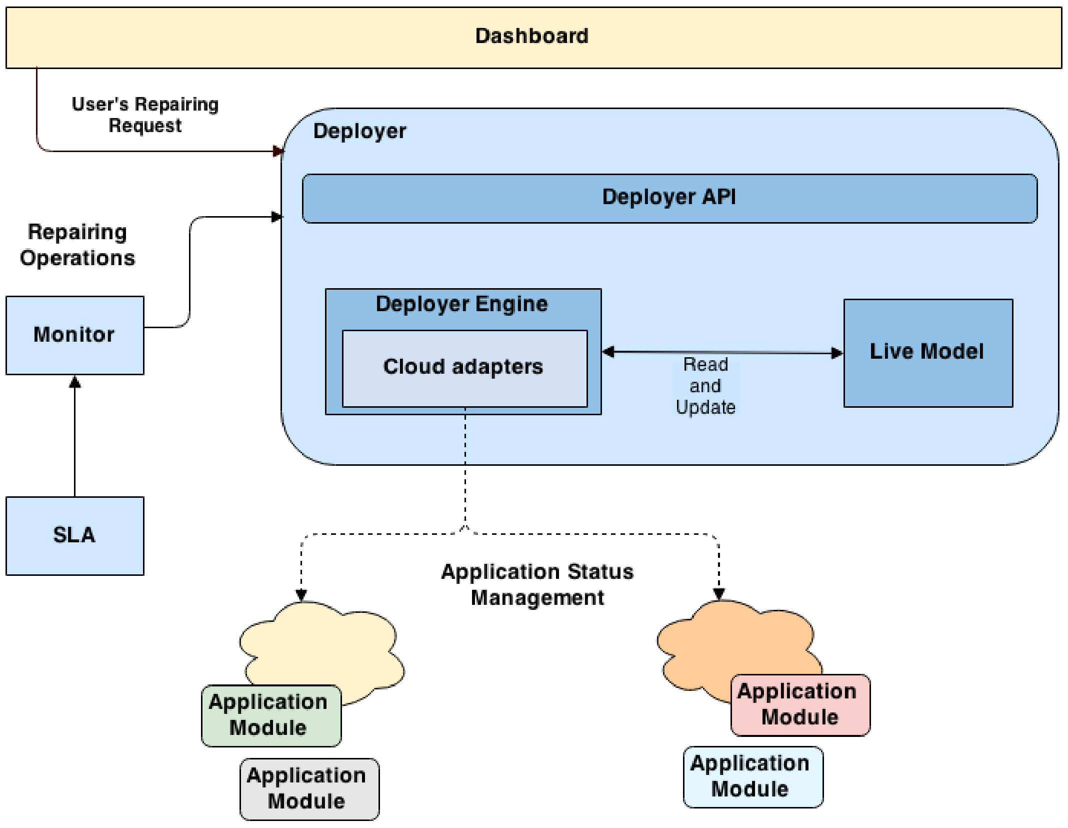

Figure 6. SeaClouds Repairing strategy overview.

Using the previous example of the 3-tier web application and its constraints, a potential DAM may look like the following:

- MyApplication
    - MySqlNode
    - ControlledDynamicWebAppCluster
        - DynamicWebAppCluster Cluster of JBoss7 Servers
        - NginxController

The app server cluster has an *AutoScalerPolicy*. The *AutoScalerPolicy* can be configured to respond to the sensor reporting requests per second per node, invoking the default resize effector, which is the actions accepted by the app server cluster to scale up/down the number of app server instances.

When the requests per node per second will be above the threshold desired by the user (300 requests per second), the Deployer Engine will scale up the number of app servers inside the webApp cluster to automatically scales the cluster up or down to be the right size for the cluster’s current load.

This illustrates a *repairing* strategy, because the *AutoScalerPolicy* is inside the original DAM. Thus, through repairing, the SeaClouds deployer could modify automatically the deployed application and related configuration according to the runtime information and related policies defined in the DAM generated by the SeaClouds planner, so as to meet the dynamic demand of the application during runtime, without replanning or migration.

In the next section we will describe how this process happens with a real use case.

5.3 SeaClouds Use Cases

In this section, we will illustrate the repairing strategy performed by the SeaClouds platform using one of the SeaClouds case study, the NURO Cloud Gaming Case Study, whose architecture is presented in Figure 7.

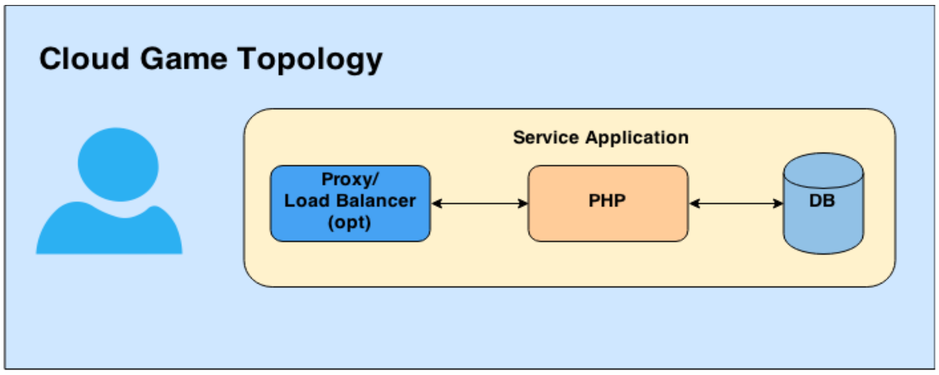

Figure 7. NURO Cloud game topology overview.

Below, we describe in more detail the different components which compose the aforementioned use case.

Game client/testing module

Game clients from all over the world connect to server application to perform game actions and synchronize the game data. In real life typical “follow-the-sun” behavior can be spotted.

NURO company develops cross platform clients with focus on mobile devices. For this case study the game client is out of focus and will be simulated to the server by scripts. The script based testing module for boom scenarios could be also deployed to cloud resources.

Front end: Proxy/Loadbalancer

This represents the front end. It handles the connection between the client and the PHP layer. It can be a separated cloud module or represented by apache. The front end should be in the region where the pile of players reside. This improves the performance by minimizing network latency and furthermore handle optimized connections (keep alive, encryption, etc.) between the client and the server application.

It makes simple to combine a mix of cloud resources on the PHP layer. For example, having a cheap private PHP node for normal/idle situation and adding easily additional expensive nodes from cloud providers in peak or boom situations. It is also helpful for reconfiguration and scaling the PHP module. It can be used to validate the follow the sun strategy of sea clouds and any reconfiguration of the PHP layer.

Webservice: PHP

Main part of the application logic is realized with this module. It processes the client requests, it gets and stores the game data within the database. Depending of costs and database load there is a maximum limit of PHP workers. The PHP module is the perfect candidate for a “load balancer” and “high availability by restart” strategies of sea clouds.

Database: MySQL

This module stores the game data and some parts of the game logic are executed by sql statements. For example data depending mass updates. Database is the perfect candidate for “autoscaling policies” and “high availability by replacement” strategies of sea clouds.

The NURO Cloud Gaming application can be highly simplified as a 3-tier web application, where a number of OSS products are used to compose the final application. The main components are, in fact, the following:

- MySql database, to store the business data.
- PHP runtime to execute the business logic.
- Apache HTTP server, that represents the web front-end where authentication and load-balancing happen.

This apparently simple classic application contains a lot of challenges when an autonomic application management tool needs to look after it. How can the management tool enable zero down-time? How does the application elastically react to external inputs due to gamers that potentially come from different timezones?

Those questions are effectively complex requirements that can be modelled in SeaClouds, passing them to the Planner which will generate a DAM that not only contains the topology and the locations for the deployment of the application, but also a set of policies that are runtime tools.

Using SeaClouds platform, NURO administrators can have out-of-the-box *ServiceRestarter* policy enabled for all the SoftwareProcess that compose their application, db and loadbalancing. This is a basic but powerful repairing strategy, especially in business critical production applications.

NURO devops team could easily set a request/second threshold on the PHP module throughput: if the application is underperforming, the AutoScaler policy will trigger the creation of a new instance of PHP node that will be automatically attached to the Apache HTTP loadbalanced proxy. Of course, if the load on the application would be low, SeaClouds will be able to shrink the cluster of instances, to have a cost-effective usage of the clouds.

In order to answer all of these questions, we describe each scenarios with the following structure:

- **Unique ID**: a unique id for the use case which will be used to point to the use case.
- **Use Case Name**: a name for the use case.
- **Description**: a brief description for detailing the use case.
- **Solution**: the goal of the use case, by describing the final desired status that has to be reached.
- **Issues**: a more detailed description that show the concrets incidents or issues which motivate the use case.
- **Triggering**: this point concerns the agent which triggers the reconfiguration event. It could be a user or a system's component, (e.g., the Monitor).
- **Actions**: the necessary reconfiguration operations.
- **Exception:** it describes the procedure when the target behavior is not expected.

As mentioned in Section 5.3, for each use case we describe Unique ID, Use Case Name, Description, Solution, Issues, Triggering, Actions and Exceptions.

Increase of players

If there is a rise of players, more PHP resources are needed to handle the requests. In this case the application is not performing with the required SLA. Then, SeaClouds starts the repairing process to scale up the number of web server instances and attach them to the loadbalancer (Table 2).

Field	**Description**
Unique ID	RequestsIncrease

Use Case Name	Increase of requests
Description	The sensor detects a violation on the application contains that affects to the Front-end (PHP modules are overcapacity).
Solution	Autoscaling policy.
Issues	This is an early warning indicator - The Front-end performance could be affected due to the increment of the users' number. - The Front-end provisioning should be managed to address the maintenance of the application's performance.
Triggering	request_analytics.minute.requests_delta > n
Actions	Scale up to the limit - Scale PHP up - Scale MySQL up
Exceptions	Cost maximum is reached => inform administrator.

Table 2. Increase of requests use case.

Decrease of players

If there is a decrease of players, less PHP resources are needed to handle the requests. In this case the application is over performing with the required SLA. Then, SeaClouds starts the repairing process to scale down the number of web server instances and detach those from the loadbalancer (Table 3).

Field	**Description**
Unique ID	RequestsDecrease
Use Case Name	Decrease of requests
Description	The sensor detects a violation on the application contains that affects to the Front-end (too many PHP modules are deployed)

Solution	Autoscaling policy
Issues	- The number of the users has decreased, and too resourcer are used according to the current application runtime requirements. - It could be necessary to adapt the deployment context to ensure an efficient used of the cloud resourcer.
Triggering	request_analytics.minute.requests_delta < n
Actions	Scale down to the limit - PHP
Exceptions	Minimum is reached => inform administrator

Table 3. Decrease of requests use case.

PHP node restart

If PHP node does not answer, a restart of the node could help (Table 4).

Field	**Description**
Unique ID	PHPNodeFailure
Use Case Name	PHP node failure
Description	A PHP node does not respond, a node restart could help.
Solution	ServiceRestarter policy
Issues	Maybe the node has crashed or has been hacked.
Triggering	HTTP too long timeout
Actions	Restart node
Exceptions	Node is not responding => inform administrator and restart

Table 4. PHP node failure use case.

Database overload

There are not enough resources to handle an increasing load on one of the database nodes. The node needs to be vertically scaled (Table 5).

Field	Description
Unique ID	DBOverload
Use Case Name	DB overload
Description	A node of the clustered DB is overloaded, increase the resources available to this node (vertical scaling).
Solution	AutoScaler policy
Issues	--
Triggering	Query response time too long.
Actions	Scale DB node up to the limit.
Exceptions	limit reached => inform administrator and restart

Table 5. DB overload use case.

Follow the user

The application receives gamer request from different geographical zones. A geo loadbalancer redirects the traffic to the application replica closer to the user, in order to minimize the latency and the resource dispersion in the cloud distribution (Table 6).

Field	Description
Unique ID	FollowTheUser

Use Case Name	Follow the user
Description	A user request to the application is redirected to the closer application cluster available
Solution	Follow-the-sun policy
Issues	--
Triggering	AutoScaler policy of each application cluster or db cluster insider the different geographical zone covered.
Actions	Add a node to the cluster/s in different regions
Exceptions	N/A

Table 6. Follow the user use case.

6. Replanning Strategies

In this section, we describe the replanning strategies in a general way, as well as the scenarios SeaClouds can tackle and the mechanisms used to do it. Also, we present some use cases related to the mechanisms as regards the Cloud Gaming case study.

As described in previous sections, when an application is deployed by the Deployer component, the Deployer Engine follows the application distribution that is described in the DAM. In this plan, the cloud services and resources were chosen by the Planner according to the application topology and requirements.

Although it is clear that the Planner selects providers whose features are the best for deploying the application, once the application is running in a (multi-)cloud environment, its behavior may not be as expected. Previously, we have described the repairing reconfiguration methodology which tries to deal with a lot of the issues of the application runtime. It is based on the management of the deployment resources to accomplish the expected performance according to the application requirement. However, several scenarios can be found where such kind of reconfiguration is not able to address the challenge of fixing the malfunctioning application and it becomes necessary to modify the distribution of the application modules. In other words, it modifies the plan that described the application module's distribution over the selected cloud providers, ie, replanning. The goal of this task is to provide the application of a deployment context whose features are enough to fulfil the user application requirements.

Thus, it is necessary to analyze the profile of the other cloud providers and select the best alternatives again. Once the new cloud resources have been selected, the next step is the generation of the **reconfiguration plan** describing a **replanning model** of how the reconfiguration process will be carried out. The aforementioned plan is composed by a sequence of the operations which enable detailing the management of each application module and each cloud resource in order to integrate the replanning changes in the current application deployment. For example, these operations could be e.g., stopping a VM, creating a new VM in a new cloud provider, redeploying an application module, connect two applications components, etc migrate an application module to a new cloud provider; instructions/operations that, as we have mentioned, permit existing resources to be managed and others to be created using new cloud providers.

In order to ensure the correct **reconfiguration plan generation**, it is necessary to know the deployment status, e.g., where each application module is deployed. The modules and the cloud resources are configured, in order to provide an updated reconfiguration management of both application modules and applications resources. Therefore, as we have mentioned in others documents [22] the topology is an immutable element over the application's lifecycle, so a replanning process should not modify it. The knowledge about the deployment status will be very useful to maintain the dependencies between the application modules during the replanning without affecting the application topology.

It is also necessary to ensure that the application will not be led to a wrong state by the reconfiguration process. During a replanning, any application module could be stopped or even deleted, which may affect the expected behavior of the other application modules. So, it is very important the reconfiguration plan or replanning model manages all application modules, thereby ensuring the stability of the system. For example, if a database is going to be managed, the application modules that use it must be stopped to avoid data integrity issues.

In relation to what has been stated above, during the replanning process it is necessary to maintain the functional connection with the deployment according to the topology, e.g. between a database and an application module. Thus, if the replanning has affected any module, the reestablishment of their dependencies and connections will be detailed in the reconfiguration plan. For example, if a database has been redeployed on a new provider (reconfigured), the modules that use this database should be reconnected to ensure the performance of the application.

Follow these guidelines will be useful to build a reconfiguration plan which allows *(i)* re-adapting the deployment configuration to the defined requirements, and *(ii)* keeping the expected application behavior without altering either the topology or dependencies. Once the reconfiguration plan has been composed, the operations, that it contains, will be executed to reconfigure the application deployment.

Of course, there are many kinds of use cases that can be applied to the re-planning process. However, as we have already mentioned, we have chosen the examples that best represent the more common scenarios addressed by SeaClouds.

6.1 Replanning Scenarios addressed in SeaClouds

In this section, we address two different scenarios where an unexpected behavior of the cloud resources used to deploy and application modules are found. This not desired performance would take due to SLA violation (QoB) or if the user decides that an application module does not work as intended. As we describe below, it defines two distinct scenarios according to the kind of module whose inaccurate behavior has to be managed:

- **Stateless** module.
- **Stateful** module.

Therefore, we have been added to clarify this **contrast** a brief overview of any application modules classification depending on their nature, **stateless** or **stateful** in Figure 8.

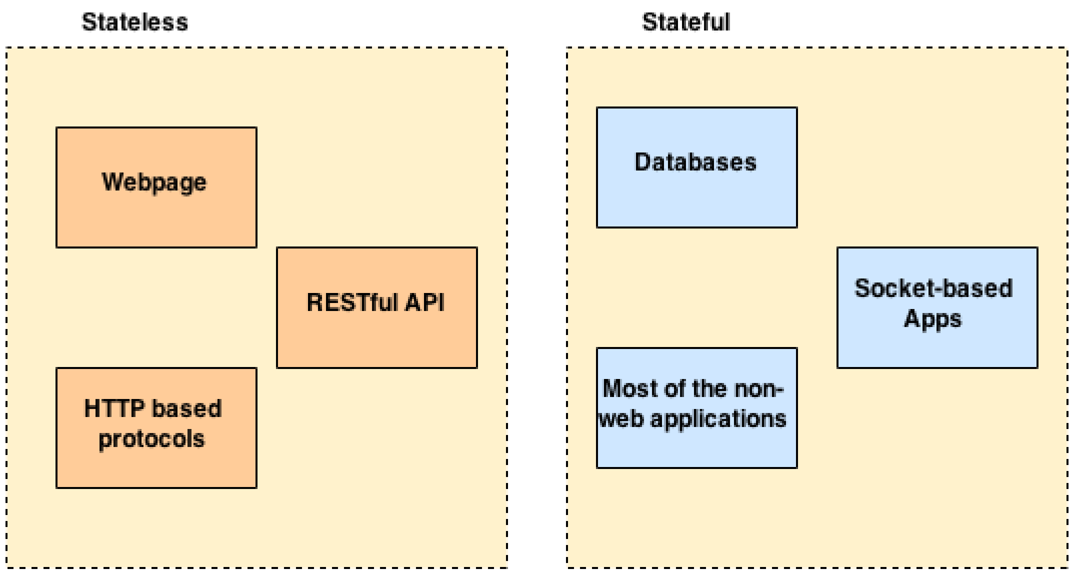

Figure 8. Examples of Stateless and Stateful modules.

The need of maintaining the context on a running application is crucial when performing any replanning action. As we detail below, this simple distinction marks the biggest difference between a stateless module migration and a stateful one. An example of a stateless module can be a web application, which does not store any information about the context of the user, so moving it to another location will not lead to consistency problems. On the other hand, a stateful module is, for example, a database. The migration of stateful modules is a very complex task, and there is no global way to address.

Next, taking into account the need of each we detail each aforementioned scenario and the needed actions to accomplish the replanning goals. We have added these specifications because it could be useful to understand the necessary generic process and operations to manage each application module type. Furthermore, we will introduce some concrete use cases of a stateless and stateful replanning in Section 6.3.

Management Actions in scenario 1. Replanning on a stateless module (e.g. web application)

According to the **scenario** whose performance of a **stateless** modules does not expected, with independence of if the user started the migration or it was suggested by the Monitor component, we describe the necessary actions to carry out this task in Figure 9.

First, the reconfiguration process is started (1). Then, the application status is used (2) to compose a new Reconfiguration Plan (that specifically we will call Replanning Model, since it is focused to solve problems by using replanning actions) which describes the necessary operations to carry out the reconfiguration process (3). Finally, the generated plan is executed to accomplish the deployment reconfiguration (4). This latter step is the one that integrates the real modifications in the current deployment and it could be formed by the following sub-steps or sub-routines the migration of a stateless application consists of the following steps:

4. Reconfiguration Execution.

   4.1. Provision the new resources in the target provider.

   4.2. Deploy the application module into the new resources.

   4.3. Redirect the traffic to the new resource.

   4.4. Stop the old application.

   4.5. Free the old resources.

All of this previous steps, which are could be grouped a "Reconfiguration execution" that follows the reconfiguration reconfirmation (Figure 9), ensure that during this process the application topology remains untouched.

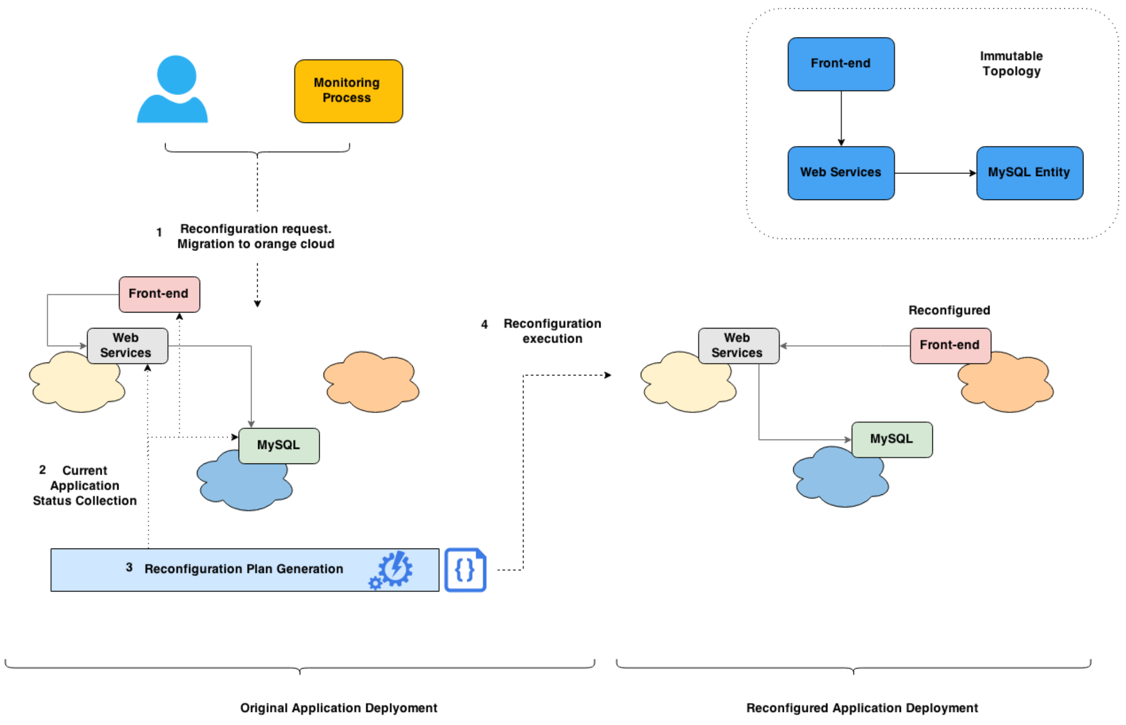
Monitoring Process
Front-end
Immutable Topology
Web Services
MySQL Entity
1 Reconfiguration request. Migration to orange cloud
Front-end
Web Services
MySQL
2 Current Application Status Collection
3 Reconfiguration Plan Generation
4 Reconfiguration execution
Reconfigured
Web Services
Front-end
MySQL
Original Application Deplyoment
Reconfigured Application Deployment

Figure 9. Replanning on a stateless module.

Management Actions in scenario 2. Replanning on a stateful module (e.g. database)

Following the steps in the previous description, we detail the generic process to address the **stateful scenario** where a module of the aforementioned type does not present the desired or expected perform.

Again, a user or the monitoring process could start the management for modifying the location of some application module (or the whole application because of migrating a whole application involves the migration of stateless and stateful modules).

A user or the monitor process may wish to move any application module to a new provider, or the as we can see in Figure 10. For example, he may have found a better agreement, or other most appropriate features, or because the providers agreement was changed. In any case, the replanning (reconfiguration) process needs a new target provider which will use to redeploy any application module.

In Figure 10, the previous process is described in more detail, focusing on a stateful component reconfiguration. A user or a monitor process wants to migrate a stateful module, a database, to a new provider. Thus, one of them initializes a reconfiguration process (1). In this point, we can aware that reconfiguration process is much similar to the stateless reconfiguration mentioned in the latter section.

Next, we can see how the knowledge about current status of the application deployment (2) is necessary to compose the reconfiguration plan (3). Following, (4) the plan is executed to reconfigure (replan) the original deployment. Please note that, when a stateful application module is moved through any cloud provider, it is essential to ensure the maintaining the correct (current and expected) module's state and the application performance. Again, the step 4 is composed by a sequence of sub-steps:

4. Reconfiguration Execution.

    4.1. Provision the new resources in the target provider.

    4.2. Keep the current status of the application module which will be migrated.

    4.3. Deploy the application module into the new resources.

    4.4. Restore the module status in the new.

    4.5. Redirect the traffic to the new resource.

    4.6. Stop the old application.

    4.7. Free the old resources.

For this scenario, two new tasks are included (4.2, 4.4) to permit the modules status maintenance between the old and new component. Probably, these sub-steps are the most complicated to achieve, because each kind of module will need an isolate way to maintain and manage the status and, depending on the module, it could be an error prone or even an unreachable task.

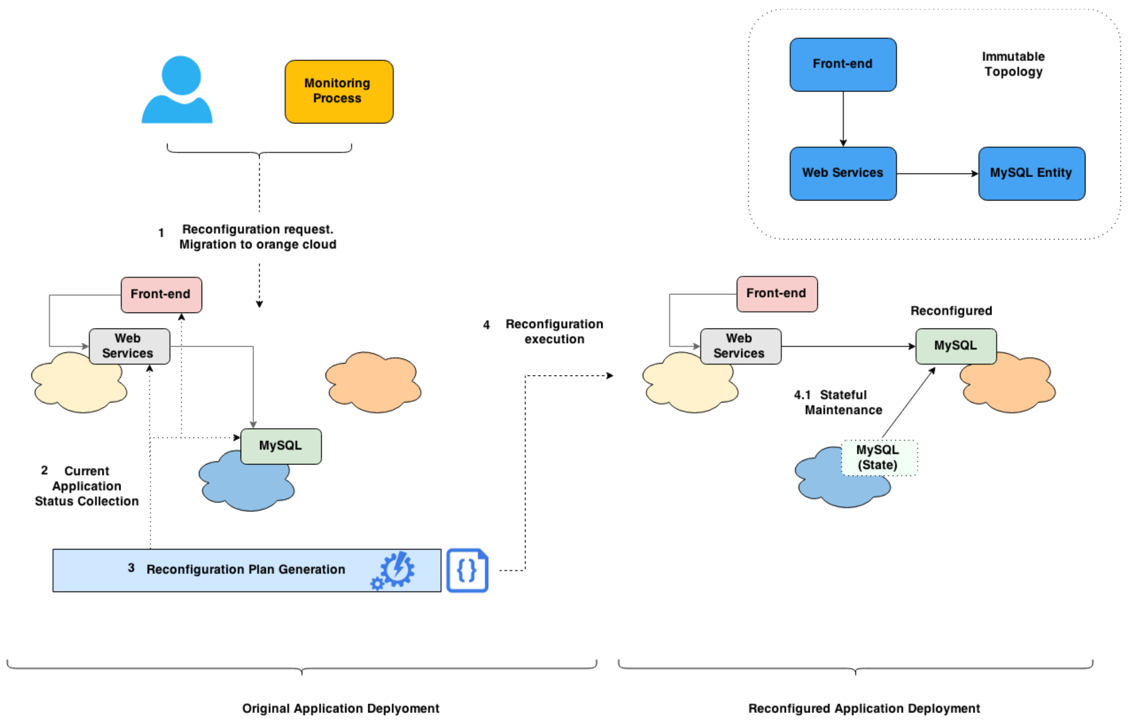
Monitoring Process
1 Reconfiguration request. Migration to orange cloud
Front-end
Web Services
MySQL
2 Current Application Status Collection
3 Reconfiguration Plan Generation
4 Reconfiguration execution
Front-end
Immutable Topology
Web Services
MySQL Entity
Front-end
Web Services
Reconfigured
MySQL
4.1 Stateful Maintenance
MySQL (State)
Original Application Deplyoment
Reconfigured Application Deployment

Figure 10. User/Alert triggered migration of a stateful module.

Please note that, as in the previous scenario, the application topology is not altered while the reconfiguration activities occur.

In the next section, it is described as the aforementioned scenarios will be achieved by SeaClouds.

6.2 SeaClouds Replanning Mechanisms

In this section, we detail how the described replanning scenarios previously will be accomplished into the context of SeaClouds. Therefore, we also specify how its components intersect with each other and the used operations to carry out the reconfiguration.

As we can see in the overview diagram in Figure 11, when the application is deployed (using one or several providers) the Monitor will detect the violations on the applications requisites. If a violation on the application constraints is found out, the Monitor or the SLA Service triggers a Reconfiguration Event which is sent to the Planner (step 1). The Planner request confirmation to the user in order to start the reconfiguration process (step 2 and 3).

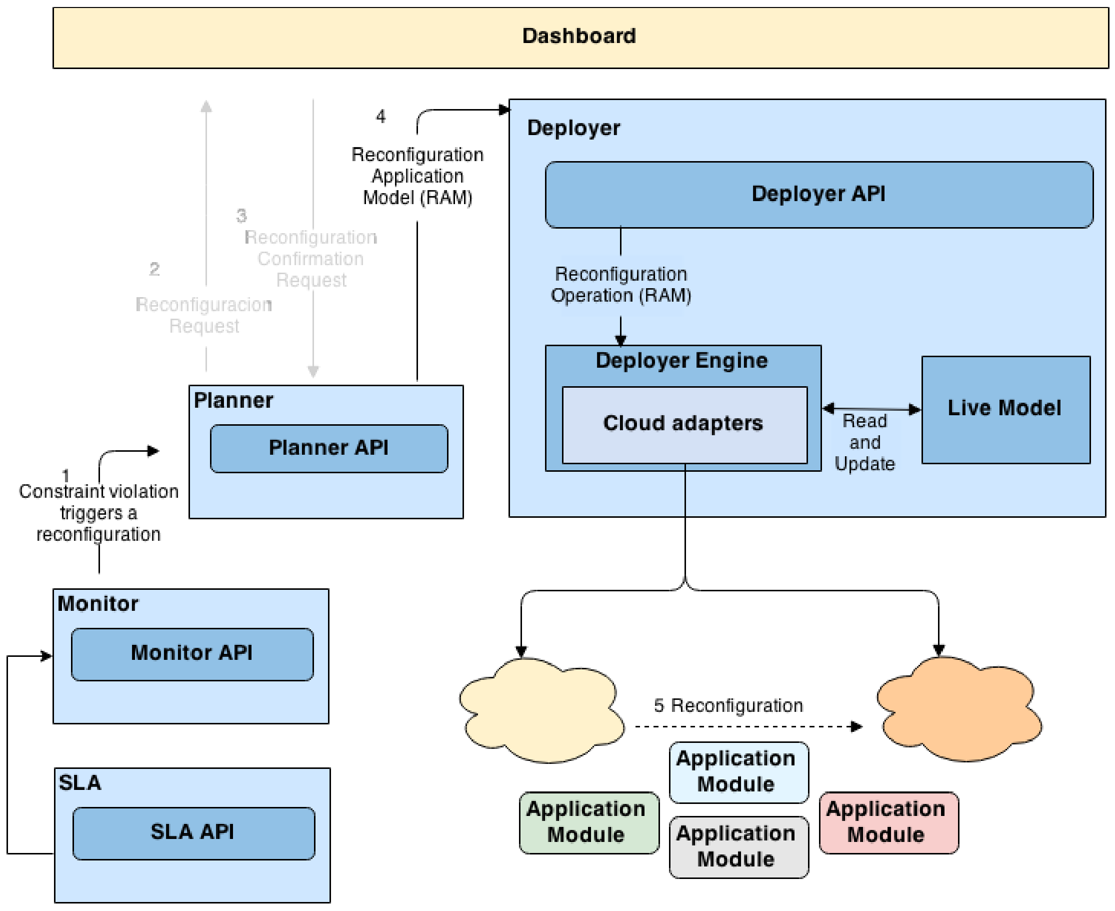

Figure 11. SeaClouds Replanning strategy overview.

The Planner receives this notification and, if necessary, generates a Replanning Application Model (RAM) (step 4). The RAM describes a reconfiguration plan of how the modules should be reconfigured and the features necessary to accomplish it, for example the new target providers. Next, a replanning model is shown (Table 7), which details the required parameters to redeploy an application module in a new cloud provider.

```
        application: application-ID
        module: entity-ID
        targetProvider: new Location (for example aws-ec2:us-west)
```

Table 7. Reconfiguration Application Plan example.

Then, the RAM is received by the Deployer which starts the replanning process (if the Entity can do a migration), doing the necessary operations in order to reconfigure the specified application modules (step 5). For example, redistribute any modules over the new providers, as it has been described in the previous section (Replanning on a stateful/stateless module scenarios), where the necessary basic actions to accomplish a replanning are mentioned.

Thus, this process allows to address both **stateless** and **stateful** management scenarios. It is worth noting that in the **necessary management actions** (described in the previous section), the process to address both of them (stateless and stateful) is very similar, and the difference between them is that in a stateful case it is necessary a tier to maintain the status integrity. SeaClouds assumes a single routine to manage the event triggering and the RAM composition. And finally, during the migration steps (5) they are carried out the necessary operations to carry out the replanning according to the kind of application module that will be being managed.

In the next sections, we detail the necessary steps to replan an application (we do not analyze scaling in/out an application as it is a subset of a migration). In this case, as running example along this document, we have used a simple application composed by three modules: a Front-end module, a Web Services module and a MySQL module, where Front-end depends on the Web Services and this in turn depends on the database (Figure 12).

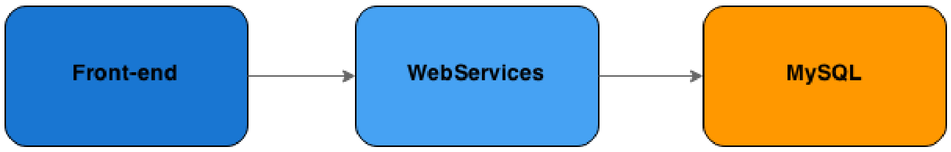

Figure 12. Application topology of the running example.

6.2.1 Generation of the Replanning Application Model

The Replanning Application Model (RAM), is generated by the Planner component. For this task, the Planner can need the current application status for maintaining the dependencies and knowing the current resources which are used for the deployment.

In this case, the Planner will use the Live Model information which contains all the details about the application deployment status, as we can see in Figure 13. As well as the Planner has to maintain unchanged the application topology during the reconfiguration. So, enabling the Planner to know about the Live Model will allow to know the relations between the modules making sure that they will not be altered.

Therefore, for the reconfiguration new cloud resources could be demanded, for example a new VM in new provider. Thus, for ensuring the best option the Planner will use the Discoverer to find the services that fits better with the application (and user) requirements.

Once the plan is composed, where all parameters, constraints and reconfiguration goals are described, it is sent to the Deployer which gets and processes the RAM, launching the replanning process.

For example, seeing the application deployment which is shown in Figure 13, if we wanted to migrate the Front-end from Yellow Cloud to Blue Cloud, a plausible RAM would be (see Table 8):

```
application: Example
module: Front-end
targetProvider: blue cloud
```

Table 8. Migration description which is contained in a RAM.

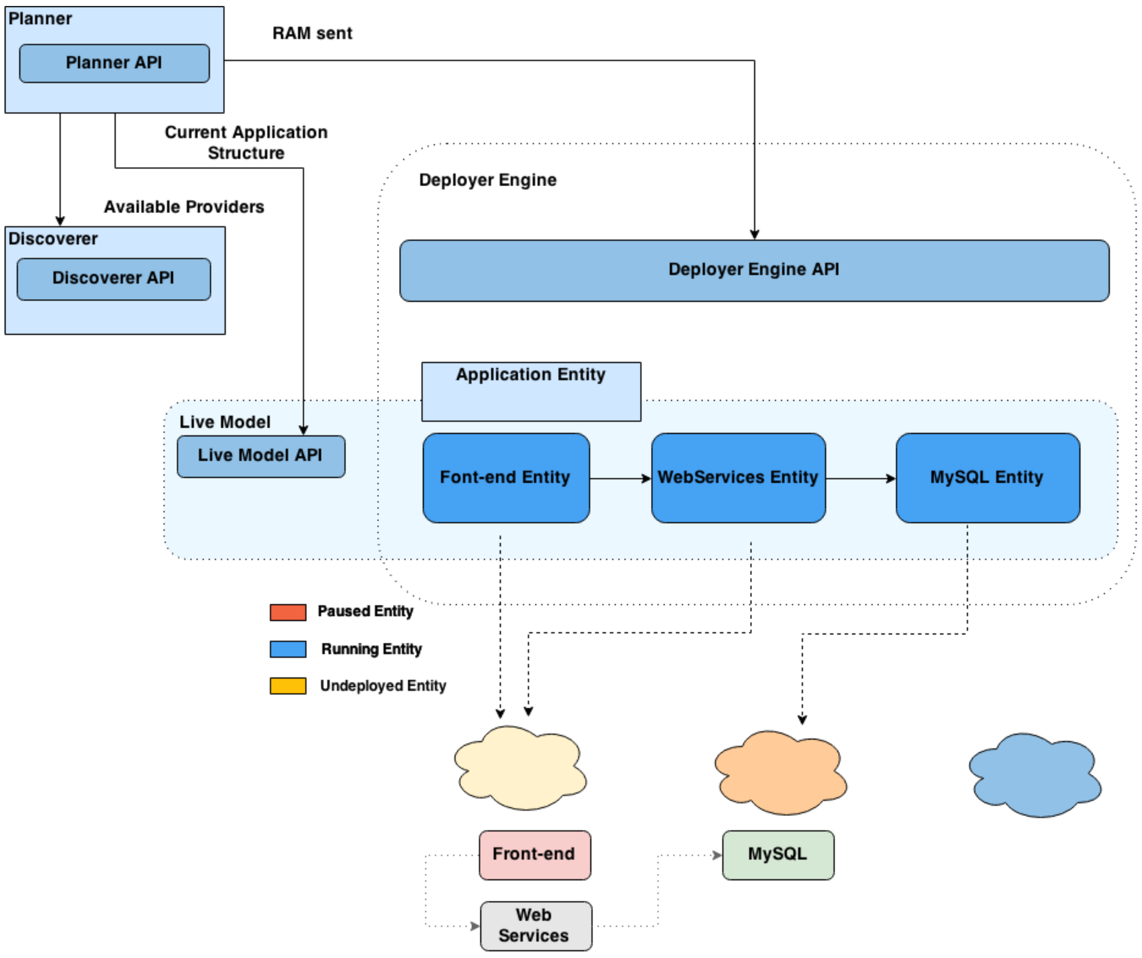

Figure 13. Providers searching that will be used in the migrations.

6.2.2 Dependencies Detection and Management

In the first phase of the replanning process, after the Deployer receives the RAM, it communicates with the Deployer Engine to retrieve the Live Model in order to detect existing dependencies within different Application Modules (see Figure 14). Once the dependencies are detected the deployer performs a sequence of calls to effectors according to the constraints (for example the database must be in read only mode before moving the data to another provider).

In Figure 14, we can see an example of the dependencies management. In this case, it is necessary to stop all entities (representation used by the Deployer for the applications modules [22]) for reconfiguring any application module, for example the database. The dependencies management ensures that the application behavior does not rache a failure state during the reconfiguration process. For example, the dependencies management should avoid that a Front-end tries to persist any data while the database is being reconfigured.

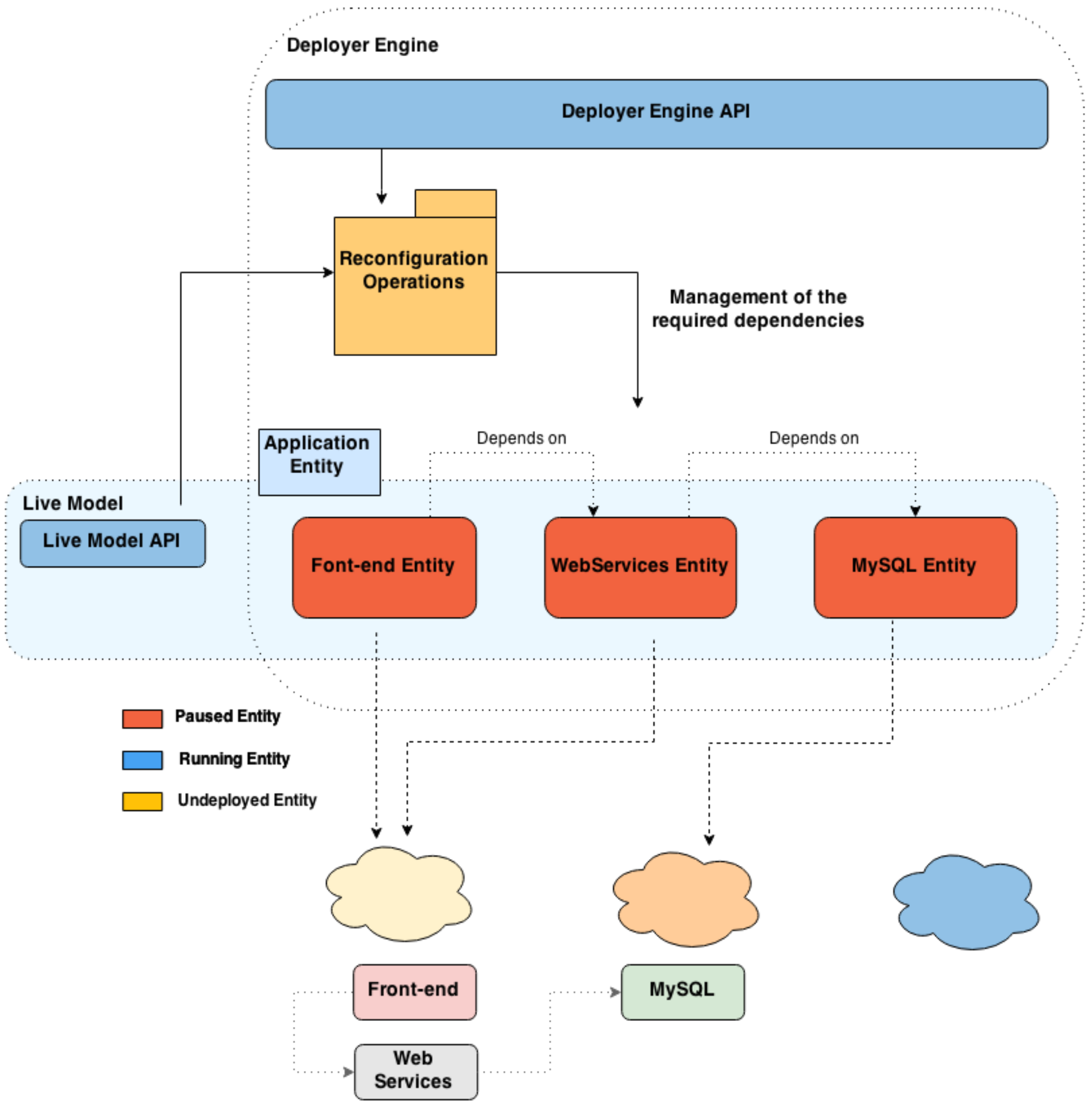

Figure 14. Replanning, triggering effectors on the Deployer component.

Once the dependencies are resolved, now the deployer has the right sequence of operations that will to achieve the target configuration described in the RAM.

Finally, the application element will be provisioned over the target cloud, taking into account the requirements to allow redistribution, and the necessary operations to maintain the application status.

These reconfiguration operations will depend on several factors, the kind and amount of the component that should be reconfigured, status of the application, and, of course, relationships among components (topology).

An important issue during an application (or any any system) provisioning is the artifact management. They represents content needed to realize a deployment such as an executable, a configuration file or data file, or something that might be needed so that another executable can run (e.g. a library). During a replanning reconfiguration, SeaClouds has to ensure the artifact integrity over the final application plan because they are necessary to maintain the desired application behavior.

We saw above as SeaClouds maintains a mapping representation, an entity, for every application component and cloud resource needed to deploy and manage and application. In order to accomplish the artifact management issues, these elements point (or contains) to the artifacts used by each application too. Thus, SeaClouds can make using of the available artifacts in the replanning reconfiguration operations.

In the next section we will describe how this process happens with a real use case, detail the reconfiguration operation needed.

6.3 SeaClouds Uses Cases

In this section, we illustrate the replanning strategy performed by the SeaClouds platform using the NURO Cloud Gaming Case Study previously presented (see Section 5.3). Specifically, we exemplify each previous scenario (Section 6.1) using a use case showing as SeaClouds carries out the replanning process depending on the application component natures that should be reconfigured, **stateless** and **stateful**.

As mentioned in Section 5.3, for each use case we describe Unique ID, Use Case Name, Description, Solution, Issues, Triggering, Actions and Exceptions.

Stateless Scenario. Front-end migration

It has been detected that stateless module of the NURO's application, in this case the Front-end, does not work as it was expected. As follow, the performance (QoS) has worsened, because the SLA constraint was violated. Then, SeaClouds starts the replanning process to find a new provider that will use to redeploy (**migrate**) the Front-end (Table 9).

Field	Description
Unique ID	StatelessMigration
Use Case Name	Stateless Front-end migration

Description	The SLA detects a violation on the application contains that affects to the Front-end (for example the Apache server is overcapacity)
Solution	Migrating the Front-end module to a new cloud provider whose features match with the module's constraints.
Issues	- The SLA changes affect to the performance of the Front-end module, due to the QoS will be altered. - The original cloud context conditions, cloud resources which are used to provisioning the application, have been modified. - The application performance and behavior could be affected so it is necessary adjust the deployment to the new conditions.
Triggering	- The SLA SeaClouds component detects this issue and evaluates the cloud resources and the application status. - The SLA component determine if is necessary to trigger a replanning event to start the reconfiguration process to **migrate** the Front-end module to a new target provides.
Actions	- The Planner receives the reconfiguration (migration) event and generates the Replanning Application Model (RAM). Then, it is sent to the Deployer. - The Deployer receives and process the RAM and carries out the necessary operation to manage the dependencies and migrate the Front-end module.
Exceptions	N/A

Table 9. Stateless Front-end migration use case.

Stateful Scenario. Database Migration

Following the preceding use case, any application components have to be replanning, but in this case, we refer to a stateful module, the database, according to the scenario that has not been explained. In this case, the SeaClouds user consider indicate in an active way the NURO's database has to be migrated to a new provider (Table 10).

Field	**Description**

Unique ID	Stateful migration
Use Case Name	Stateful Backend migration
Description	A SeaClouds user decides to migrate the NURO's database to a new cloud provider.
Solution	Migrating the Database module to a new cloud provider whose features match with the module's constraints.
Issues	- The original cloud context conditions, cloud resources which are used to provisioning the application, may have been modified. - The application performance and behavior could be affected so it is necessary adjust the deployment to the new conditions. - The module context among with the state must be preserved.
Triggering	- The user starts the migration process.
Actions	- The Planner receives the reconfiguration (migration) event and generates the Replanning Application Model (RAM). Then, it is sent to the Deployer. - The Deployer receives and process the RAM and carries out the necessary operation to manage the dependencies and migrate the Front-end module. - As the Database is a stateful element, the Deployer has to ensure the maintenance of the database status during the migration to the new target provider.
Exceptions	N/A

Table 10. Stateful Backend migration use case.

7. Data Migration and Synchronization

In the previous sections we have presented the two SeaClouds reconfiguration strategies, repair and replan. Both of them can result in the fact that the application is moved on a different location, either in the same cloud or even in a different one.

In both cases, if the application intensively uses data stored in some DBMS, it may be convenient to move such data together with the application itself. *Data migration* is then the process needed to move data from a source to a target DBMS, ensuring that data are not lost nor damaged in the process and that the application exploiting the data is not negatively affected by this movement.

Thanks to the standardization occurred at the data model level (with DDL), data migration is a well-established topic for relational databases (see, e.g., [23, 24, 25, 26]. In the NoSQL database field, to the best of our knowledge, the support to data migration across different NoSQLs is quite limited. Some databases provide tools to extract data from them (e.g., Google Bulkloader [27]), but in the end, it is up to the programmer to actually map those data to the target database data model and perform the migration.

Hegira4Cloud [28] is a research effort being developed under the MODAClouds project (www.modaclouds.eu) that focuses on data migration between NoSQL Database as a Service (DaaS), trying to preserve their peculiar characteristics. More specifically, this framework is based on the idea of extracting data from the source DaaS, transforming them into an intermediate format and then into the target DaaS. The intermediate format is defined by an intermediate meta-model described in detail in [28]. It takes into account the features of the most widely used NoSQL and it has been shown to be sufficiently general for dealing with the features of so-called columnar and key-value NoSQL databases [29,30].

The adoption of a new NoSQL system in Hegira4Cloud requires only the development of the translator from this new NoSQL into the intermediate format and vice versa. Furthermore, thanks to this intermediate meta-model, Hegira4Cloud is able to preserve the data types, read consistency policies, and secondary indexes supported by the source DaaS.

In particular, Hegira4Cloud preserve data types by keeping track of the type of each migrated data explicitly, even though that type is not available in the destination DaaS. This is accomplished by performing the following procedure: data converted into the intermediate format are always serialized into a *property value field* and the original data type is stored as a string into a *property type field*. When data are converted from the intermediate format into the target one, if the destination DaaS supports that particular data type, the value is deserialized. Otherwise, the value is kept serialized and it is up to the application level to correctly interpret (deserialize) the value according to the type field.

As extensively detailed in [28] and [31], read consistency policies are handled through the concept of *Partition Group*. Entities that require strong consistency on read operations will be assigned, in the intermediate format, to the same Partition Group value. Entities managed according to an eventual consistency policy will be assigned to different Partition Group values. When entities share the same Partition Group, if the target database supports strongly consistent read operations, then Hegira4Cloud adapts data accordingly (depending on the target database data-model). Otherwise, Hegira4Cloud simply persists the data so as that they will be read in an eventual consistent way, and creates an auxiliary data structure to preserve the consistency information.

Finally, secondary indexes are preserved across different DaaS by means of the property *indexable field*. More specifically, during the conversion into the intermediate format, if a certain property needs to be indexed, it is marked as indexable. When converting into the target format, if the target DaaS supports secondary indexes, the property is mapped consequently according to the specific interfaces provided by the target database. Otherwise, Hegira4Cloud creates an auxiliary data structure on the target DaaS which stores the references to the indexed properties, so that, when migrating again these data to another database supporting secondary indexes, they can be properly reconstructed.

The high level architecture of Hegira4Cloud is shown in Figure 15. A *Source Reading Thread* extracts data from the source database, one entity at a time or in batch (if the source database supports batch operations) and performs the conversion, by means of the respective *direct translator*, into the intermediate format and puts the data in a queue. The queue is used as a buffer, in order to decouple the reading and writing processes and manage the different throughput of the source and target DaaS. When data are in the queue, a *Target Writing Thread* extracts and converts them into the target DaaS data-model, thanks to an *inverse translato*r (specific for each supported database), and stores these data into the target DaaS. Hence, translators are in charge of mapping data back and forth between the source/target DaaS and the intermediate format, performing the (de)serializations, checking for data types support, properly mapping indexes and adapting the data to preserve different read consistency policies.

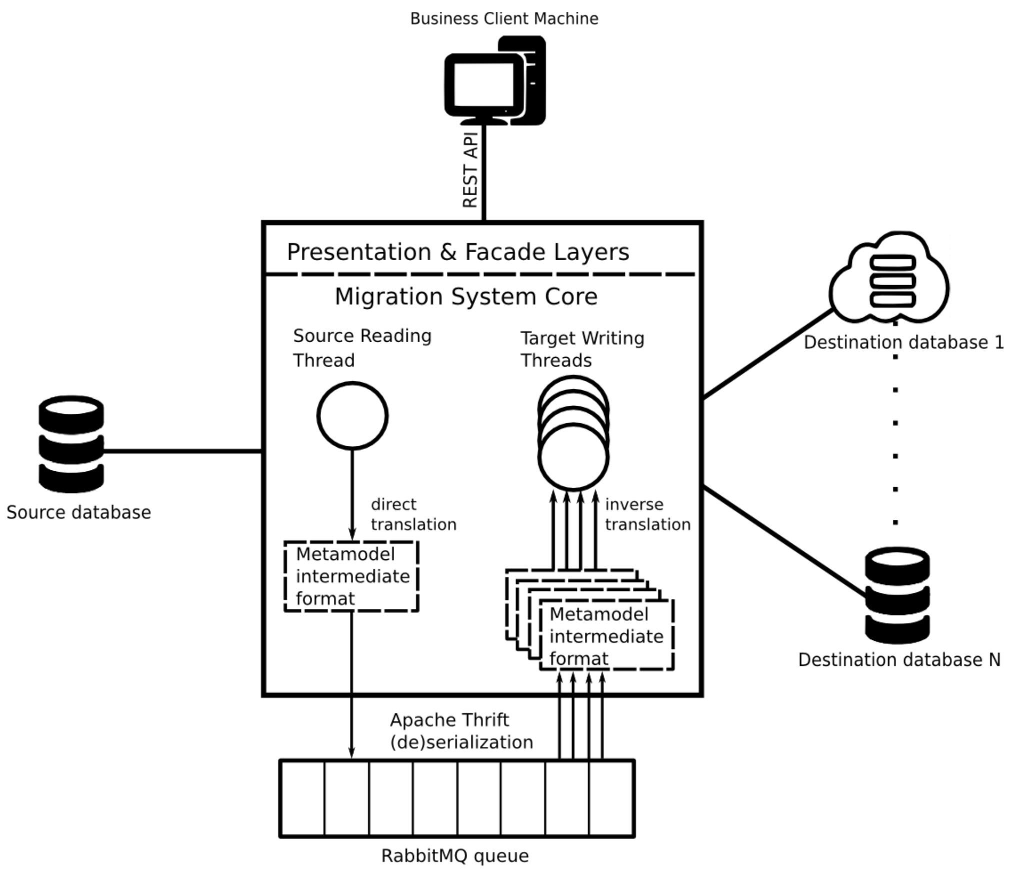

Figure 15. Hegira4Cloud high level architecture.

8. Conclusions

In this document, we have presented a first design of the run-time reconfiguration process. Firstly, we have introduced the reconfiguration strategies in a general way, with the corresponding motivations, and the SeaClouds approach in comparison with other works. Secondly, we describe more in detail how SeaClouds applies the strategies: repairing and replanning, mapping the situations with some scenarios considered in the use cases used in SeaClouds. Finally, we describe the data migration and synchronisation process in SeaClouds.

The options we studied in order to solve the reconfiguration of a cloud application are well known by previous researches. The key of our approach is to remove the restrictions around the reconfiguration scenarios showed in previous work. In particular we avoid focusing on inter-provider migration restrictions, and, instead, performing migration between providers.

In terms of SeaClouds platform, the proposed reconfiguration process is orchestrated by the Planner in connection with the Monitor and the Deployer, based on a strong monitoring system that will be aware of the live status of the application. The Deployer implements the mechanisms that will execute any required action coming from both Planner and/or Monitor.

In summary, in this first desing we have tried to concrete the strategies which are being studied at design level and analysed at implementation level in the SeaClouds platform.

References

1. U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. A cost-aware elasticity provisioning system for the cloud. In the 31st International Conference on Distributed Computing Systems (ICDCS), pp. 559 –570, 2011.

2. A. Verma, G. Kumar and R. Koller. The cost of reconfiguration in a cloud. In The 11th International Middleware Conference Industrial track, Middleware Industrial Track '10 (ACM), pp. 11-16, 2010.

3. V. Andrikopoulos, T. Binz, F. Leymann and S. Strauch. How to adapt applications for the Cloud environment. In Computing (Springer), vol. 95, no. 6, pp. 493-535, 2013.

4. D. Petcu. Portability and interoperability between clouds: challenges and case study. In Towards a Service-Based Internet. (Springer), pp. 62–74, 2011.

5. J. Miranda, J.M. Murillo, J. Guilľen and C. Canal. Identifying adaptation needs to avoid the vendor lock-in effect in the deployment of cloud SBAs. In Proceedings of the 2nd International Workshop on Adaptive Services for the Future Internet and 6th International Workshop on Web APIs and Service Mashups (ACM), pp, 12–19, 2011.

6. N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In Proceedings of the 4th Intl. Conference on Cloud Computing, ser. CLOUD 2011. (IEEE), pp. 500–507, 2011.

7. R. N. Calheiros, R. Ranjan, and R. Buyya. Virtual machine provisioning based on analytical performance and qos in cloud computing environments. In International Conference on Parallel Processing (ICPP), pp. 295 –304, 2011.

8. J. Schroeter, P. Mucha, M. Muth, K. Jugel, and M. Lochau. Dynamic Configuration Management of Cloud-based Applications. In Proceedings of the 16th International Software Product Line Conference - Volume 2, SPLC ’12 (ACM), pp. 171–178, 2012.

9. EMF: Eclipse Modeling Framework, http://www.eclipse.org/emf

10. D. Steinberg, F. Budinsky, M. Patenostro and E. Merks. EMF: Eclipse Modeling Framework, 2nd edn. Addison Wesley, Reading, 2008.

11. H. Mi, H. Wang, G. Yin, Y. Zhou, D. Shi, and L. Yuan. Online self- reconfiguration with performance guarantee for energy-efficient large- scale cloud computing data centers. In The 2010 IEEE International Conference on Services Computing (SCC '10) (IEEE), pp. 514-521, 2010.

12. N. Arshad, D. Heimbigner, and A. L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. In Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence (IEEE), pp. 39 - 46, 2003.

13. S. Gómez Sáez, V. Andrikopoulos, F. Leymann, and S. Strauch, Towards Dynamic Application Distribution Support for Performance Optimization in the Cloud. In Proceedings of CLOUD'14 (IEEE), pp. 248 - 255, 2014.

14. Cloud4SOA repository. Available at: https://github.com/Cloud4SOA/Cloud4SOA

15. Apache Brooklyn. Available at: https://brooklyn.incubator.apache.org/

16. MODAClouds EU Project. Available at: http://www.modaclouds.eu/

17. DEEP: DPWS Enabled dEvices Platform. Available at: http://com-gisum-deep.appspot.com/

18. Flyway. Available at: http://flywaydb.org/

19. CloudFoundry environment variables. Available at: http://docs.run.pivotal.io/devguide/deploy-apps/environment-variable.html#VCAP-SERVICES

20. Using environment variables in OpenShift. Available at: https://developers.openshift.com/en/managing-environment-variables.html#database-variables

21. Official definition of Brooklyn policies: https://brooklyn.incubator.apache.org/v/latest/concepts/policies.html/

22. SeaClouds Project. Deliverable D4.1 Definition of the multi-deployment and monitoring strategies (SeaClouds Consortium), http://seaclouds-project.eu/deliverables/SEACLOUDS-D4.1_Definition_of_the_multi-deployment_and_monitoring_strategies.pdf, 2014.

23. Oracle SQL Developer Migration. Available at: http://www.oracle.com/technetwork/database/migration/index- 084442.html

24. Flyway repository. Available at: https://github.com/flyway/flyway

25. LiquiBase. Available at: http://www.liquibase.org

26. Mysql workbench: Database migration. Available at: http://www.mysql.com/products/workbench/migrate/

27. Google Bulkloader. Available at: https://chromium.googlesource.com/external/googleappengine/python/+/200fcb767bdc358a3acb5cf7cad1376fe69f12c5/google/appengine/tools/ bulkloader.py

28. M. Scavuzzo, E. Di Nitto, and S. Ceri. Interoperable data migration between NoSQL columnar databases. In proceedings of the first International Workshop on Engineering Cloud applications and Services. In enCASE 2014. (IEEE), pp. 154 – 162. 2014.

29. B. Scoffield, Nosql – death to relational databases. In presentation at the CodeMash, 2010. Available at: http://www.slideshare.net/bscofield/ nosql- codemash-

30. A. Popescu. (2010, 02) Nosql at codemash – an interesting nosql categorization. Available at: http://nosql.mypopescu.com/post/ 396337069/presentation- nosql-codemash- an- interesting- nosql

31. M. Scavuzzo, Interoperable data migration between NoSQL columnar. Databases. Master's thesis, Politecnico di Milano, 2013. Available at: http://dinitto.faculty.polimi.it/wp-content/uploads/MarcoScavuzzo.pdf