MULTICLOUD APPLICATIONS TOWARDS
THE DIGITAL SINGLE MARKET

Deliverable D3.2

Intermediate architectural patterns for implementation, deployment and optimization

Editor(s):	Kyriakos Stefanidis
Responsible Partner:	Fraunhofer
Status-Version:	Final – v1.0
Date:	30/11/2018
Distribution level (CO, PU):	Public

BY-SA

Project Number:	GA 731533
Project Title:	DECIDE

Title of Deliverable:	Intermediate architectural patterns for implementation, deployment and optimization
Due Date of Delivery to the EC:	30/11/2018

Work Package responsible for the Deliverable:	WP3 - Continuous Architecting
Editor(s):	Kyriakos Stefanidis (Fraunhofer)
Contributor(s):	Simon Dutkowski (Fraunhofer) Anne Barsuhn (Fraunhofer) Kyriakos Stefanidis (Fraunhofer)
Reviewer(s):	Javier Gavilanes (EXPERIS)
Approved by:	All Partners
Recommended/mandatory readers:	WP2, WP4, and WP5

Abstract:	This report details the architectural patterns that comprise DECIDE ARCHITECT. It defines the description format and usage for those patterns together with an extended pattern set. It also describes the functional and architectural design of the ARCHITECT module.
Keyword List:	Multi-cloud application, Multi-cloud patterns, multi-cloud architecture, microservices, distributed applications, DevOps
Licensing information:	This work is licensed under Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) http://creativecommons.org/licenses/by-sa/3.0/
Disclaimer	This document reflects only the author's views and the Commission is not responsible for any use that may be made of the information contained therein

Document Description

Document Revision History

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
v0.1	28/09/2018	First draft version	FhG
v0.2	25/10/2018	Added/Updated sections on pattern models, new patterns, nfr structure, pattern language	FhG
v0.3	19/11/2018	Added section for pattern inferring. Updated rest of the document for intermediate deliverable	FhG
v0.4	21/11/2018	Updated abstract and executive summary	FhG
v0.5	26/11/2018	Updated based on internal review comments	FhG
V1.0	27/11/2018	Ready for submission	TECNALIA

(cc) BY-SA

Table of Contents

(cc) BY-SA

List of Figures

List of Tables

CC BY-SA

Terms and abbreviations

AGS	Authentication Gateway Service
API	Application Programming Interface
AWS	Amazon Web Services
CDP	Cloud Design Patterns
CI	Continuous Integration
CPIP	Cloud Provider Independent Pattern
CPSP	Cloud Provider Specific Pattern
CSP	Cloud Service Provider
DNS	Domain Name System
DoS	Denial of Service
EC	European Commission
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IP	Internet Protocol
JSON	Java Script Object Notation
KR	Key Result
LUN	Logical Unit Number
MTTR	Mean Time To Recovery
NFR	Non-functional Requirement
NIST	National Institute of Standards and Technology
PSP	Platform Specific Patterns
QoS	Quality of Service
REST	Representational state transfer
RPC	Request Procedure Call
SaaS	Software-as-a-Service
SKOS	Simple Knowledge Organization System
SOA	Service Oriented Architecture

CC BY-SA

SotA	State of the Art
SSO	Single Sign On
TPM	Trusted Platform Module
UI	User Interface
URI	Universal Resource Identifier
VM	Virtual Machine

(cc) BY-SA

Executive Summary

The deliverable at hand presents the intermediate architectural multi-cloud patterns for implementation, deployment and optimisation of multi-cloud native applications and is the output of T3.1 – "Multi-cloud native applications architectural patterns for implementation, deployment and optimization" of WP3 – "Continuous Architecting". The architectural patterns form the first building block for the DECIDE project and will be fed into the ARCHITECT tool (KR2) in order to aid developers in developing apps that are multi-cloud aware. It also presents the design of the ARCHITECT tool itself. There will be one more deliverable that presents the final results.

This deliverable is an update of the D3.1 – "Initial architectural patterns for implementation, deployment and optimization" and therefore reuses and updates the previous results and content. This is done so, for readability purposes.

The deliverable starts off by presenting the multi-cloud concepts and the benefits of adopting a multi-cloud strategy. This is strengthened by analysing at the state of the art (SotA) in which patterns have been developed for the implementation of multi-cloud apps.

Based on the results of the SotA, a categorisation of multi-cloud fundamental, optimisation, development and deployment patterns have been selected. The fundamental patterns are related to concepts that allow for an application to be multi-cloud aware and dynamically re-deployed and re-adapted to fulfil its non-functional requirements with no or minimal downtime. Furthermore, these patterns are in line with the approach considered by the DECIDE DevOps Framework. With this set of patterns as listed in this document, developers can ensure to address relevant NFRs at the design time of the application in question, this is exemplified using a microservices based application by applying the relevant multi-cloud patterns.

Following the patterns presentation, the description of the pattern inferring process that decides which patterns should be followed by the developer based on her chosen non-functional requirements is presented. This forms the core of the ARCHITECT tool. The output of this process is a set of pattern recommendations that, if adopted by the developer, can fulfil the developer's non-functional requirements.

Moreover the ARCHITECT tool and its design and implementation details are presented in the last chapters. The technical details of the tool, such as installation, configuration and usage, are presented in the appendices along with the first demonstration of the results.

(cc) BY-SA

1 Introduction

1.1 About this deliverable

This deliverable "Intermediate architectural patterns for implementation, deployment and optimization" is the second of the three deliverables that will be produced under task 3.1 in the context of WP3. This deliverable is an update of the D3.1 – "Initial architectural patterns for implementation, deployment and optimization" and therefore reuses and updates the previous results and content.

Nowadays, organisations and companies are changing the way they develop and deploy software. Agility is a major new paradigm adopted and correspondingly the use of Cloud technologies. In turn, the latter evolved and now these companies and organisations are working on getting the most out of the available Cloud landscape and leveraging resources based on their non-functional requirements (NFRs), such as scalability, availability, location, but also costs. A multi-cloud strategy is a product of this evolution and provides many benefits, but when developing apps for it there are certain aspects that need to be considered at the design time of an application.

This deliverable focuses on these aspects and discusses concepts and their related architectural patterns with regards to multi-cloud native applications. Architectural patterns provide a general and reusable solution to commonly occurring problems in the development and deployment of software artefacts. Why should one re-invent the wheel while others have already found solution to the problem and have documented it?

The intermediate set of patterns introduced in this deliverable are meant to aid developers to design their applications in a way that it is multi-cloud aware and that a set of non-functional requirements are always fulfilled when the app is running. Furthermore, by applying these architectural patterns the application will be able to be deployed, run and be monitored by the DECIDE DevOps Framework.

In the context of DECIDE, these patterns are fed into the ARCHITECT tool [KR2]. ARCHITECT then uses the information on the NFRs as input to recommend multi-cloud architectural patterns for the developer.

Lastly, it is important to note that this document does not describe each pattern fully, but gives directions on the need for applying specific patterns and refers to sources with their description.

1.2 Document structure

The deliverable at hand starts with a state of the art analysis in section 2. The analysis looks at different projects that have worked on cloud architectural patterns as well as multi-cloud architectural patterns. The type of patterns established in the projects have different natures, some of which are vendor specific and some vendor agnostic, some address cloud applications and their deployments some address the multi-cloud paradigm. These projects present initial ideas and work regarding these topics and bring about a number of architectural patterns that will be reused in the DECIDE project.

Section 3 "Multi-Cloud Native Application Architectures" discusses the benefits and aims of adopting a multi-cloud strategy and explains the differences between architectures and deployments in order to conclude on the type of applications we are targeting in the DECIDE project. The type of application is a multi-cloud application and in the context of DECIDE implies that the application is distributed over different CSPs and can be seamlessly re-deployed, i.e. ported across multiple heterogeneous CSPs.

Furthermore, Section 3 discusses considerations with regards to properties a DevOps team has to fulfil in order for apps to be able to run in a multi-cloud environment.

CC BY-SA

Section 4 is intentionally named “Cloud Computing Architectural Patterns for Multi-Cloud Apps” as it introduces and discusses architectural patterns from the Cloud Computing realm, as presented in Section 2, which when collectively applied, address the Multi-Cloud context.

The considerations made in Section 3 are translated into concepts and architectural patterns associated with them are listed. The concepts are regarded as fundamental ones that fulfil the requirements of a multi-cloud native application. In addition to this, these fundamental concepts and patterns allow for the correct functioning of the DECIDE DevOps Framework, in the sense that the application can be (re)-deployed, monitored and re-adapted. In the same section, further multi-cloud native application patterns are introduced, which when applied result in optimising the application to be multi-cloud aware as well as more patterns for deployment and development of multi-cloud native apps.

Lastly, the detailed description of the process that the ARCHITECT module follows in order to suggest patterns to the developer based on the given NFRs can be found in the last sub section.

Section 5 describes the design of the ARCHITECT tool [KR2] and details its functionality as well as its architecture.

Finally, in section 6, the conclusion is presented along with future work.

Appendix A contains the detailed property descriptions of the NFRs and the Pattern vocabulary

Appendix B contains the software documentation of the ARCHITECT tool

Appendix C describes the SockShop App, which is an exemplary application to showcase a microservices based application development. DECIDE Multi-cloud patterns are applied to the SockShop in order to render it multi-cloud aware based on hypothetical non-functional requirements.

2 State of the Art of Multi-Cloud Patterns

This section presents the state of the art analysis for multi-cloud patterns. In the context of the DECIDE project, multi-cloud patterns are those that enable the design, modelling and development of distributed applications over heterogeneous cloud resources in a faster and systematic way.

The existence of this notion of multi-cloud is relatively new and thus the work and research conducted regarding this matter is to date relatively immature.

We therefore included in our analysis research projects that handle patterns for cloud native apps and Cloud Computing but not just multi-cloud, with the goal to understand what we can learn and reuse from them. In general, it is important to note that by re-using well-known and established patterns we envision easing the development process for the developers by not adding an additional overhead for understanding and learning new patterns. Furthermore, we hope to borrow from the established pattern language as well as the content of each selected pattern. The research projects have been split into the two categories agnostic patterns and vendor specific patterns. The description and usefulness for the DECIDE project are briefly discussed below.

2.1 Vendor Agnostic Patterns

The following are research projects that have defined cloud patterns from an agnostic perspective. The content of this section has not been updated with respect to D3.1 [1] but it is kept in this document for readability purposes.

Table 1. Vendor agnostic patterns - Cloud Computing Patterns

Name	**Cloud Computing Patterns** [2]
Description	This book uses patterns to describe cloud service models and cloud deployment types in an abstract and provider agnostic form to categorize the offerings of cloud providers. Furthermore, it shows reoccurring cloud application architectural patterns on how to design, build, and manage applications that use these cloud offerings. The abstraction of these patterns makes them applicable to challenges faced by developers regardless of the actual technologies and cloud services that they are using. The authors of the book also created an icon language to describe and communicate these patterns efficiently.
Usefulness for DECIDE	The authors created a large library of different Cloud Computing patterns covering a number of use cases, which can be used by ARCHITECT; some of these patterns already incorporate the idea of a multi-cloud distributed application. Furthermore, the vendor agnostic description language provided can be used or adapted by DECIDE to effectively convey the function and idea of the patterns we are using to the user.

Table 2. Vendor agnostic patterns - ARTIST project

Name	**ARTIST –** Advanced software-based service provisioning and migration of legacy Software [3]
Description	The ARTIST project provides a method to move a non-cloud software application to the cloud. To help with this a catalogue of over thirty cloudification and optimization patterns has been developed and released. However, these patterns focus mostly on

(cc) BY-SA

	the cloud offerings from Microsoft, Azure and AWS and furthermore just on single cloud deployment instead of a multi-cloud deployment that DECIDE focuses on.
Usefulness for DECIDE	This means that these patterns most likely cannot directly be used but have to be adapted to a multi-cloud approach if possible. Nevertheless, such a collection of patterns for a multi cloud deployment can be very valuable, not just for DECIDE alone, but for the whole multi-cloud ecosystem, whose growth then in turn can be helpful for the success of DECIDE. Furthermore, the structured approach used to arrive at these patterns and to describe them can also be adapted to design the patterns needed for DECIDE.

Table 3. Vendor agnostic patterns - Toreador project

Name	***TOREADOR Project*** [4]
Description	The TOREADOR project envisioned to develop a Big Data Analytics-as-a-Service approach to support Big Data adoption in European companies and organizations. To accomplish this, research was conducted for new solutions for the problem of composing Cloud services to satisfy requirements. One of the outcomes was a paper presenting a semantic-based representation of Application Patterns and Cloud Services [4], with an example of its use in a typical distributed application, which shows how the proposed approach can be successfully employed for the discovery and composition of Cloud Services.
Usefulness for DECIDE	This Paper describes a method to construct vendor specific patterns from agnostic ones, which can be useful for the developer of an application to help them implement the patterns proposed by ARCHITECT.

Table 4. Vendor agnostic patterns - MODAClouds Project

Name	***MODAClouds Project*** [5]
Description	The MODAClouds project provides methods, a decision support system, an open source IDE and run-time environment for the high-level design, early prototyping, semi-automatic code generation, and automatic deployment of applications on multi-Clouds with guaranteed QoS. Model-driven development combined with novel model-driven risk analysis and quality prediction will enable developers to specify Cloud-provider independent models enriched with quality parameters, implement these, perform quality prediction, monitor applications at run-time and optimize them based on the feedback, thus filling the gap between design and run-time. Additionally, MODAClouds provides techniques for data mapping and synchronization among multiple Clouds.
Usefulness for DECIDE	This project approached the cloud provider agnostic vs. non-agnostic issue by developing a model language, which consists of Cloud provider-independent models and Cloud provider-specific models and can seamlessly translate between both. They also have created a number of multi-cloud patterns that are mostly relying on some heuristics like decomposing and encapsulating the features of an application into modular and reusable blocks. The approach taken in MODAClouds is reusable in this project, however, it needs to be investigated if simply designing an application as a

(cc) BY-SA

	whole using the MODAClouds patterns suffices a multi-cloud strategy or whether looking at the components (i.e. microservices) of the application individually needs to be taken into account.

Table 5. Vendor agnostic patterns – Cloud Migration Patterns

Name	***Cloud Migration Patterns – Multi-Cloud Architectural Description*** [6]
Description	The research of Jamshidi et al presents a catalogue of cloud architecture migration patterns that target multi-cloud strategies. The contribution aids application developers and architects in planning the migration and easily communicating the plan with non-technical stakeholders.
Usefulness for DECIDE	Although the title of the thesis addresses the multi-cloud topic, its results, i.e. the patterns, describe a systematic methodology for migrating on-premise applications into the cloud and multi-cloud. The patterns depict deployment strategies but do not consider the application as whole from an NFR perspective. This aspect led us to not use these patterns for our work as our approach considers more than just deployment strategies, namely the initial decomposition of the application in order to render it multi-cloud aware whilst respecting the NFRs assigned to it.

Table 6. Vendor agnostic patterns – Cloud Computing Design Patterns

Name	***CloudPatterns.org – Cloud computing Design Patterns*** [7]
Description	CloudPatterns.org is a community site dedicated to documenting a master patterns catalog comprised of design patterns that capture and modularize technology-centric solutions distinct or relevant to modern-day cloud computing platforms and business-centric cloud technology architectures.
Usefulness for DECIDE	CloudPatterns.org is a large collection of well-known design patterns that tackle the specific problems of contemporary delivery and deployment of cloud applications. The way that the patterns are decomposed and presented, follows the same approach as the DECIDE project and is a valuable resource of available patterns for the ARCHITECT module.

2.2 Vendor Specific Patterns

The following are pattern projects belonging to vendors with a widespread use. The vendor specific patterns or platform specific patterns (PSP) are relevant as they give detailed instructions on how to implement these patterns on the infrastructure in question. The content of this section has been updated with respect to D3.1 [1].

Table 4. Vendor specific patterns - AWS Cloud Design Patterns

Name	***AWS Cloud Design Patterns*** [8]
Description	The AWS Cloud Design Patterns (CDP) are a collection of solutions and design ideas for using AWS cloud technology to solve common systems design problems. The CDPs are categorized by type of problem they are addressing and most are specific to the AWS infrastructure.

(cc) BY-SA

Usefulness for DECIDE	This collection will proof useful when the developer wants to deploy their application on the AWS cloud. Some patterns are also generic enough that their use with other providers seems feasible. However, since these patterns only focus on the deployment with a single cloud provider, they have to be adapted to a multi-cloud approach if possible.

Table 5. Vendor specific patterns - Microsoft Azure Cloud Design Patterns

Name	**Microsoft Azure Cloud Design Patterns** [9]
Description	The Microsoft Azure CDPs are categorized into "Challenges in cloud development"; some of these categories equal NFRs like Availability and Resiliency. Each pattern describes the problem that the pattern addresses, considerations for applying the pattern, and an example based on Microsoft Azure. Most of the patterns include code samples or snippets that show how to implement the pattern on Azure. However, most of the patterns are relevant to any distributed system, whether hosted on Azure or on other cloud platforms.
Usefulness for DECIDE	Like the AWS CDPs, these will be primarily useful for a deployment on Azure, but patterns descriptions are mostly general enough to be used on any cloud platform and since these descriptions are very in-depth they seem to be quite useful to decide which patterns ARCHITECT should suggest when. However, since these patterns only focus on the deployment with a single cloud provider, they have to be adapted to a multi-cloud approach if possible.

(cc) BY-SA

3 Multi-Cloud Native Application Architectures

The DECIDE project aims at tackling the issues that arise when dealing with multi-cloud native applications. Multi-cloud native applications are different from traditional cloud native applications as they are architected to be deployed on multiple, potentially heterogeneous clouds. The DECIDE project defines a multi-cloud native application as a distributed one whose components are deployed on different CSPs but still work in an integrated and transparent way for the end-user.

Companies nowadays are broadening their perspective on the use of different cloud services providers (CSPs) and adopting multi-cloud strategies in order to benefit from the best and most suitable cloud properties. That said, by adopting a multi-cloud approach and applying multi-cloud architectural characteristics to applications, leveraging non-functional requirements (e.g. cost, scalability, geo-presence, and data location) becomes feasible and provides a number of other benefits, such as:

- Utilising an on premise, hybrid, public and private clouds mix
- Utilising unique vendor-specific services
- Creating diversity while enabling redundancy and avoiding vendor lock-ins and latency
- Easy and faster disaster recovery

In general, adopting a multi-cloud strategy might seem basic but it is important to note that in the context of DECIDE it does not involve replicating existing applications (distributed or not) over several clouds (e.g. an application with high workloads is replicated and receives requests through a load balancer). Rather the focus in this project addresses issues that extend such a strategy, namely by deploying parts (i.e. microservices) of an application onto different cloud service providers with different capabilities by matching these capabilities to the developer's needs.

Figure 1 depicts the evolution in software development and deployment. There is a large difference between cloud native application and multi-cloud native architectures and deployments. It starts off with depicting a monolithic application (a)), b) depicts a distributed application in the traditional sense, c) depicts a microservices-based cloud native application, d) depicts a distributed application replicated or scaled across two CSPs and finally e) depicts a multi-cloud application's architecture and deployment as defined by the DECIDE project.

With this strategy[1], i.e. multi-cloud architecture and its deployment, considerations must be made regarding the architectural challenges and decisions that allow an application with its microservices to be seamlessly deployed and adapted across different CSPs.

[1] A pre-requisite for a multi-cloud strategy is a distributed application that is loosely coupled with stateless properties.

www.decide-h2020.eu
BY-SA

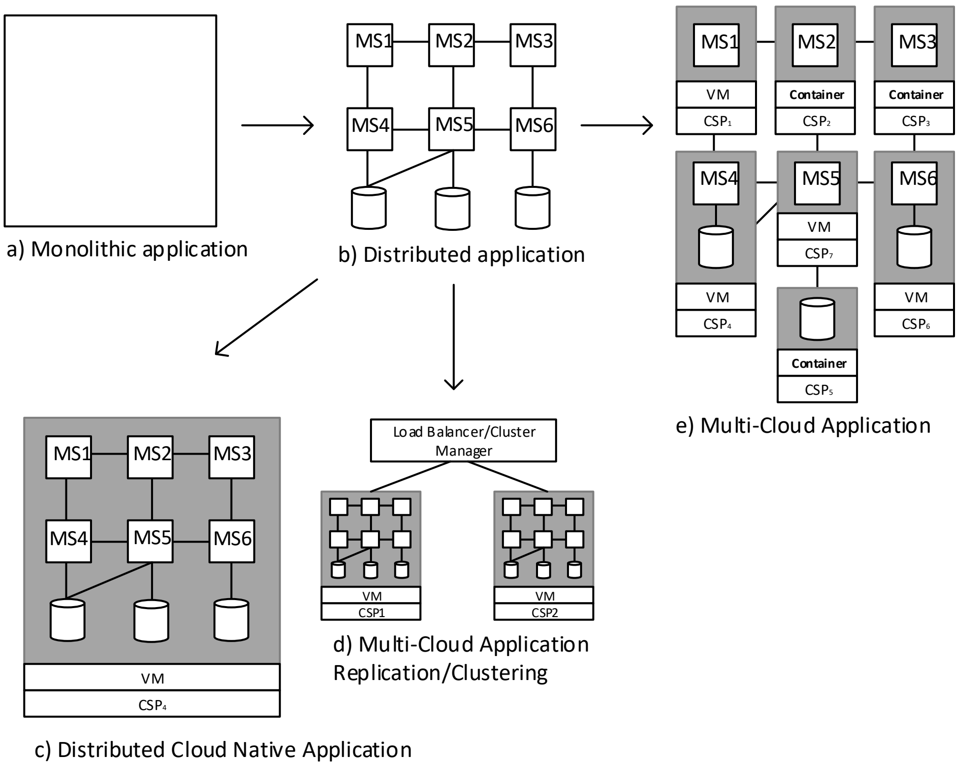

Figure 1. Evolution of Software Development and Deployment

The challenges that arise when designing a multi-cloud application are listed below and form the basis for all considerations when designing an application in the context of DECIDE:

- Resilience and portability of the components or microservices of an application; when porting processes across clouds MTTR must be decreased, disconnected scenarios and faults have to be avoided. In addition, cost effective deployment of the application, by abstracting from cloud vendor specifics and without having to manually adapt to new interfaces must be a given.
- Respecting the applications defined NFRs.
- The applications components (e.g. microservices) should work together in an integrated manner. Microservices' endpoints must be managed and discoverable in case of switching hosts (IP addresses).
- Just as the portability of microservices, data migration or replication should be easily handled and not pose a problem
- The use of provider specific SaaS and IaaS services, because of each service providers intricacies (e.g. different APIs, data storage), should be possible.
- Dynamic re-configuration of the application properties should be possible.

Further challenges address standardisation efforts for allowing portability among clouds and the management of multi-cloud strategies (these will be partly addressed in this work package, but also in a future deliverable).

In view of the fact that the outlined multi-cloud approach and its exploitation may be unfamiliar to most developers, the availability of Cloud offerings (features and service characteristics) are not comparable and providers or operators do not have a holistic view on things, it would be necessary to use patterns at the design stage of an application. Firstly, to improve applications from an architectural perspective and secondly, to enable new developers to easily understand the source code and deployment scenarios, as they would better understand the architectural approaches taken. Lastly, it

(cc) BY-SA

is worth noting that with a set of patterns common various pitfalls when designing a multi-cloud application can be mitigated.

In essence, the architectural patterns we will be proposing cover the aforementioned aspects in order to render an application multi-cloud. Architectural patterns should be regarded as solutions or best practices for commonly occurring problems [2]. With a pattern based approach we can additionally simplify and guide developers in the use of the DECIDE DevOps framework.

CC BY-SA

4 Cloud Computing Architectural Patterns for Multi-Cloud Apps

As explained in D3.1 [1], an architectural pattern provides a general, reusable solution to a commonly occurring problem in the development or deployment of software components.

The use of architectural patterns in the context of object-oriented programming and distributed applications has dramatically improved many aspects in software and systems engineering, such as their quality, speed maintainability and accessibility. For the same reason, a large number of Cloud Computing architectural patterns have been developed by [2] and others as presented in Section 2.

We can distinguish between two types of patterns, those that provide a description or template for solving a particular problem and are independent of the implementation details (Cloud Provider Independent Patterns (CPIP)) and those that target specific implementation techniques or rely on the use of specific software components (Cloud Provider Specific Patterns (CPSP)).

With the former (CPIP) the formulation of high-level solutions that cover a broad problem space is possible, whereas with the latter (CPSP) tailored solutions are provided, for instance to optimize an application in a very specific context.

In this deliverable, we will be looking at CPIP in the context of multi-cloud. For readability purposes, and in order to present a self-contained document as well as to get a complete idea of the pattern compendium that DECIDE has gathered, content from D3.1 [1] has been reused. New patterns in section 4.2 are clearly identified with (new). Section 4.3 is completely new for this version of the deliverable.

4.1 Definition of Multi-Cloud Architectural Patterns

In the context of Cloud Computing, patterns that address distributed applications are not in a broad sense addressing multi-cloud. These patterns when applied render the application a distributed one running on a single Cloud but not necessarily able to be distributed over multiple Cloud Service Providers.

For further elaboration, one could consider the following example: a distributed application, in a traditional sense, can just as well reside on one virtual machine (VM) where the components of the application listen on different ports on the same machine. This is something that is not effective, because the application would not benefit from the IDEAL (see [10]) properties and would not exploit the advantages of the essential cloud characteristics as defined by NIST [11].

One of the drawbacks resulting from a 1 VM instance deployment is the fact that the elasticity characteristic of Cloud Computing can only be partially supported. With elasticity, the resources (performance) provided for the application should be easily scaled up or out[2] in a flexible manner in order to cater to the currently experienced workload. With the lack of distribution in this deployment scheme (1 VM and 1 single cloud provider, application tailored for 1 single cloud provider), scaling out is not possible and therefore reacting to failures becomes difficult [10].

Therefore, in this project we define a collection of architectural patterns consisting of a number of cloud patterns for distributed applications (and presumably non-cloud patterns) that collectively address the multi-cloud issue as defined in the DECIDE project. These patterns have been selected from different sources ([2] [12] [13] [9] [8]) presented in the state of the art analysis in Section 2 and supplemented by additional ones that have been deemed necessary by the project's consortium.

[2] Scaling out means increasing the number of resources to adapt to a specific workload by creating additional instances. Scaling up means to increasing the capabilities of a single cloud resource.

(cc) BY-SA

In the next sub-sections, we define and describe the architectural patterns relevant in a multi-cloud environment and those that are in scope with our project goals. The comprehensive description of the patterns can be found in [2]. Those that have been produced in the DECIDE project are introduced below in their respective sections.

4.2 DECIDE Patterns

4.2.1 DECIDE Fundamental Patterns

The DECIDE fundamental patterns are mandatory for the use of the DECIDE DevOps Framework. They follow the DevOps principles adopted and reflected by the DECIDE tools and render the application compatible for the use of the DECIDE DevOps Framework, i.e. the application should be architected in a way that it can be seamlessly re-adapted and re-deployed in a multi-cloud environment. Moreover, the fundamental patterns allow to meet a number of different NFRs such as scalability, availability, cost, and location.

This section introduces fundamental concepts, explains their necessity and lists the patterns that need to be implemented in order to support these concepts.

Separation of Concern and Distribution

When one looks at the concept of multi-cloud from an architectural perspective it is primarily important that applications implement the concept of *separation of concern*. It is essential for maintaining and deploying systems on multiple cloud platforms with minimal effort [5].

The outdated concept of monolithic architectures and their deployments can lead to immense reduced performance and availability when any one service or solution component suffers an outage or a runtime exception [12].

With separation of concern, an application becomes easily modularised and thus distributed. This allows developers to leverage the specifics and intricacies of cloud resources based on the current needs and, of course, the non-functional requirements of each individual component or microservice. Currently the implementation of microservices to address this issue is regarded as the best choice to date. Designing software applications as several suites of independently developed and deployed services [14] fulfils our concern here and matches the DevOps approach followed in the DECIDE project.

The following patterns address this issue:

Table 6. DECIDE Fundamental Patterns for Separation of Concern and Distribution

Pattern Name	Short Description
Distributed Application	A cloud application divides the provided functionality among multiple application components that can be scaled out independently.
Two-Tier Cloud Application	Presentation and business logic is bundled to one stateless tier that is easy to scale. This tier is separated from the data tier that is harder to scale and often handled by a provider-supplied storage offering.
Three-Tier Cloud Application[3]	The presentation, business logic, and data handling are realized as separate tiers to scale stateless presentation and compute-

[3] The patterns Two-Tier and Three-Tier Cloud Application rule each other out.

www.decide-h2020.eu
CC BY-SA

Pattern Name	Short Description
	intensive processing independently of the data tier, which is harder to scale and often handled by the cloud provider.
Loose Coupling	A communication intermediary separates application functionality from concerns of communication partners regarding their location, implementation platform, the time of communication, and the used data format.

Containerized Services

In general, containerization technologies have significantly simplified and improved DevOps as they provide a solution to deployment problems caused by the lack of dependencies in production environments.

In addition to this, adopting a multi-cloud strategy emphasises the need for agility and leveraging the cloud resources based on the individual non-functional requirements of microservices. Containerization can aid in this context, because with containerization building and deploying a service becomes much easier and the services can be independently deployed and scaled [13].

Furthermore, high-performance recovery is given as containers are extremely fast to build and start. In addition to this, containers are much more cost-effective than VMs as the latter can impose a significant footprint by introducing a layer of intermediate processing [12] that ultimately can further increase costs.

Container-based orchestrators like the ones provided by some Cloud Services are indispensable for any production-ready microservice-based and for any multi-container application with significant complexity, scalability needs, and constant evolution [15].

Containers boost DevOps, by offering a significant advantage in the following key areas:

- Portability
- Service or Application Density
- Fault tolerance and Resilience through Fault Isolation and rapid replacement of faulty containers
- Suitability for Automation

For the use of the DECIDE Framework this is a fundamental pattern, because the ADAPT tool's mechanisms are built to deploy containerized services. As with the list above containers can simplify re-deployment.

The pattern *containerization* is a fundamental one in this case and describes the best-practices to package and deploy services. The following details said pattern and it is described using the pattern language as in [2]. The pattern is described in Table 7 and is derived from the following sources: [12] [13] [15] [16].

Table 7. DECIDE Fundamental Pattern for Containerized Services

Pattern Name	Containerization
Short Description	Container-based solutions provide the important benefit of cost savings because containers are a solution to deployment problems caused by the

CC BY-SA

Pattern Name	Containerization
	lack of dependencies in production environments. Containers significantly improve DevOps and production operations.
Context	A container management system or container engine is used for the deployment and operation of containers.
Problem	How can an environment be provided with maximum support for services with high-performance recovery and scalability requirements? Services deployed on bare metal or virtual servers can impose a significant footprint. Virtualization improves portability but introduces a layer of intermediate processing that can further increase the footprint. Monolithic solution deployments can lead to widespread reduced performance and availability when any one service or solution component suffers an outage or a runtime exception.
Solution	Services are deployed independently, or together with composed services, as autonomous units that are packaged into independently manageable and autonomous container images, each of which includes the services' underlying system dependencies. Tooling is provided to manage the building, deploying and operating of the containers.

External Configuration Storage

Application and application instances (i.e. microservices) in a multi-cloud environment need to be scaled out as well as ported from one CSP to another in order to satisfy the given NFRs.

The configuration of a multi-cloud application does not only lie in the functional parts but also the deployment on and provisioning information for the underlying infrastructure [5] and services.

It is usually the case that the functional, deployment and provisioning configurations are stored in the application or its instances.

Examples of configuration elements include database connection strings, UI theme information, target CSPs, network locations, URIs or external APIs or the URIs of queues and storage used by a related set of applications [9].

It has been often demonstrated that it is challenging to manage changes to local configurations across multiple running instances of an application, especially in a cloud-hosted scenario. It can on one hand result in instances using different configurations settings while an update is still being deployed or on the other hand result in unnecessary redeployment resulting in drastic costly downtime.

Another point to be made is that with local configuration files the configuration is limited to a single instance, but it is sometimes beneficial to reuse these configurations settings across multiple applications or application instances [9] .

These problems can be solved by moving all relevant configuration information out of the application package and into a centralized location. Advantages with this approach, lie in giving easy access to deployment tools, enabling dynamic re-deployment and re-adaptation.

When outsourcing configuration information into a centralized location it is important to consider the following:

CC BY-SA

F1. Provide an interface that can be used to quickly and efficiently read and update configuration settings.
F2. The type of external store depends on the hosting and runtime environment of the application. In a cloud-hosted scenario it's typically a cloud-based storage service, but could be a hosted database or versioning system (e.g. Git – repositories for source control).
F3. The format of the configuration information (i.e. files) should be properly documented, validated and structured.
F4. Access control should be put in place in order to protect configuration data and enough flexibility provided to store versions of configuration (e.g. development, staging, production and releases).

With these points implemented, the DECIDE Framework can re-deploy and re-adapt the application cost effectively and with minimal downtime. The pattern to be implemented for the external configuration storage concept is *Managed Configuration*.

Table 8. DECIDE Fundamental Patterns for External Configuration Storage

Pattern Name	Short Description
Managed Configuration	Scaled-out application components should use a centrally stored configuration to provide a unified behaviour that can be adjusted simultaneously.

Service Registration and Discovery

In a traditional distributed system deployment, components communicate with one another and rely on functionality or data provided by other components. Components and services run at fixed, well known locations (hosts and ports) and can easily call one another using HTTP/REST or some form of RPC [13]. However, in a modern multi-cloud and microservices -based deployment, network locations of the components are dynamic and change frequently.

This is due to the fact that dynamic IP addresses are usually assigned to containers and virtual machines alike. Furthermore, re-deployments occur frequently throughout a single day due to specific NFRs that need to be met. For instance, workloads change throughout the day and thus suitable cloud providers are selected for a re-deployment to fulfil the currently needed resource capacities and to keep costs down.

This results into dynamic changes in the number of service instances along with the allocation of new network locations. In order to make clients easily and seamlessly able to determine the location of the service to which they send requests and service instances to register their new location, a mechanism is needed to make the services that have changed their network location, discoverable in an easy and simple manner.

This concept is not fundamentally necessary for the functioning of the DECIDE Framework but it allows for the seamless functioning of clients and services when a re-deployment takes place. Therefore, we consider it as a fundamental multi-cloud pattern.

The patterns below in Table 9 address this issue.

Table 9. DECIDE Fundamental Pattern for Service Registry and Discovery

Pattern Name	Short Description
Service Registry	Implement a service registry, which is a database of services, their instances and their locations. Service instances are registered with the service registry on

www.decide-h2020.eu
(cc) BY-SA

Pattern Name	Short Description
	start-up and deregistered on shutdown. Client of the service and/or routers query the service registry to find the available instances of a service. A service registry might invoke a service instance's health check API to verify that it is able to handle requests.
Client-side Discovery	When making a request to a service, the client obtains the location of a service instance by querying a Service Registry, which knows the locations of all service instances.
Server-side Discovery	Services typically need to call one another. In a monolithic application, services invoke one another through language-level method or procedure calls. In a traditional distributed system deployment, services run at fixed, well known locations (hosts and ports) and so can easily call one another using HTTP/REST or some RPC mechanism. However, a modern microservice-based application typically runs in a virtualized or containerized environments where the number of instances of a service and their locations changes dynamically.

4.2.2 DECIDE Optimization Patterns

Optimization Patterns are those that aid the developer in improving the applications NFRs by taking adequate measures in optimizing the application code to reflect on these requirements.

Optimization patterns can be of multi-cloud nature but their design could also optimize the use of cloud resources, such as elasticity.

An example is using the cloud persistence layer instead of implementing it as part of the application. Optimizations could therefore consist of the use of SaaS of parts of the application system.

Table 10. DECIDE Optimisation Patterns

Pattern Name	Short Description	NFR
Provider Adaptors	Provider interfaces are encapsulated and mapped to unified interfaces used in applications to separate concerns of interactions with the provider from application functionality.	Availability
Elasticity Manager	The utilization of IT resources on which an elastically scaled-out application is hosted, for example, virtual servers are used to determine the number of required application component instances. This is an optimization for the deployment.	Scalability
Resiliency Management Process	Application components are checked for failures and replaced automatically without human intervention.	Availability
Elastic Load Balancer	The number of synchronous accesses to an elastically scaled-out application is used to determine the number of required application component instances.	Availability, Scalability

CC BY-SA

Pattern Name	Short Description	NFR
Elastic Queue	The number of asynchronous accesses via messaging to an elastically scaled-out application is used to adjust the number of required application component instances.	Scalability
Cross-Storage Device Vertical Tiering (new)	A system is established whereby the vertical scaling of data processing can be carried out dynamically across multiple cloud storage devices.	Scalability
Dynamic Data Normalization (new)	Data received by cloud consumers is automatically normalized so that redundant data is avoided and cloud storage device capacity and performance is optimized.	Performance
Direct I/O Access (new)	The virtual server is allowed to circumvent the hypervisor and directly access the physical server's I/O card.	Performance
Direct LUN Access (new)	The virtual server is granted direct access to block-based storage LUNs via the physical host bust adapter card.	Performance
Micro Scatter-Gather (new)	A root container is utilized with special distributor and aggregator cloud services designed to compose and interact with multiple cloud services and cloud service instances, thereby carrying out the necessary high-performance composition logic.	Performance

4.2.3 DECIDE Development Patterns

Development Patterns are those that aid the developer with best practices for building a multi-cloud application. Example patterns are n-tier architectures (splitting the application into microservices), loose coupling, stateless.

There are at least some basic development patterns that obviously should be applied to all multi-cloud applications:

Table 11. DECIDE Development Patterns

Pattern Name	Short Description	NFR
Data Access Component	Functionality to store and access data elements is provided by special components that isolate complexity of data access, enable additional data consistency, and ensure adjustability of handled data elements to meet different customer requirements.	Scalability
Compliant Data Replication	Data is replicated among multiple environments that may handle different data subsets. During replication data is obfuscated and deleted depending on laws and security regulations. Data	Location

(cc) BY-SA

Pattern Name	Short Description	NFR
	updates are adjusted automatically to reflect the different data structures handled by environments.	
Stateless Component	State is handled outside of the application components to ease their scaling-out and to make the application more tolerant to component failures.	Scalability (Elasticity)
User Interface Component	Interactive synchronous access to applications is provided to humans, while application-internal interaction is realized asynchronously when possible to ensure Loose Coupling. Furthermore, the user interface should be customizable to be used by different customers.	Availability, Scalability
Processing Component	Possibly long running processing functionality is handled by separate components to enable elastic scaling. Processing functionality is further made configurable to support different customer requirements.	Scalability
Usage Monitoring (new)	Cloud usage monitors are utilized to track and measure the quantity and nature of runtime IT resource usage activity.	Availability Performance
Service State Management (new)	The cloud service is designed to integrate with a state management system allowing it to defer state data at runtime when necessary so as to minimize its IT resource consumption.	Performance
Dynamic Failure Detection and Recovery (new)	A watchdog system is established to monitor IT resource status and perform notifications and/or recovery attempts during failure conditions.	Availability
Multipath Resource Access (new)	Alternative paths to IT resources are provided to give cloud consumers a means of programmatically or manually overcoming path failures.	Availability
Resource Pooling (new)	An automated synchronization system is provided to group identical IT resources into pools and to maintain their synchronicity.	Availability
Synchronized Operating State (new)	A composite failover system is created to not rely on clustering or high availability features but instead use heartbeat messages to synchronize virtual servers.	Availability
Cloud Storage Data at Rest Encryption (new)	Secure data on the physical hard disks in order to prevent unauthorized access.	Availability
Cloud Storage Data Lifecycle Management (new)	A solution is introduced to automatically manage and migrate the data into a different type of cloud	Availability

(cc) BY-SA

Pattern Name	Short Description	NFR
	storage device, or delete the data based on its state in a defined lifecycle.	
Cloud Storage Device Masking (new)	A solution is implemented to isolate each cloud storage device from being presented to or accessed by unauthorized cloud consumers.	Availability
Cloud Storage Device Performance Enforcement (new)	A solution is implemented with the ability to match and compare the performance characteristics of datasets against performance capabilities of a destination cloud storage device.	Performance
Cloud Resource Access Control (new)	A cloud single sign-on (SSO) architecture is established, incorporating an authentication gateway service (AGS) and attribute authority for implementation of cloud resource access control.	Availability
Cloud VM Platform Encryption (new)	Encrypted containers are provided for use and storage of the various types of VM backups and replications.	Availability
Geotagging (new)	When trusted resource pools are generated, the geolocation is supplied as part of the compliance and regulatory assurance attributes.	Location
In-Transit Cloud Data Encryption (new)	A solution is implemented with capabilities that secure and protect data while it transfers between sender and receiver and also ensure that data will not be accepted by the receiver if the original data sent is modified.	Availability
Trusted Cloud Resource Pools (new)	Trusted resource pools made up of trusted geotagged computers are made available by the cloud provider, and can be verified by the consumer through direct monitoring or evidence through auditing.	Availability
Automatically Defined Perimeter (new)	A system is established that provides protected communications between consumers and providers whereby each provider either accepts or rejects communications based on privileges securely granted automatically by a perimeter controller.	Availability
Cloud Authentication Gateway (new)	An authentication service is implemented, allowing standard authentication, communication, and session establishment from a cloud consumer to the authentication service. The authentication service then authenticates to the cloud resource on behalf of the cloud consumer using the diverse protocols required by the cloud provider.	Availability

CC BY-SA

Pattern Name	Short Description	NFR
Cloud Key Management (new)	A cloud key management system is employed, available either as a physical or virtual network attached device.	Scalability
Leader Node Election (new)	One of the invoked cloud service instances is designated as the leader node, responsible for aggregating the other cloud service instances in a coordinated effort to complete the task.	Scalability

4.2.4 DECIDE Deployment Patterns

Deployment Patterns address how the deployment configuration for multi-cloud applications should be handled. For instance, managing the deployment scripts as well as storing them should be designed from a multi-cloud perspective. Here types of technological risks as well as geographical locations of the components (data or business logic) are accounted for.

Furthermore, the deployment patterns will take into account DECIDE principles of re-adaptability and re-deployment for multi-cloud environments.

Table 12. DECIDE Deployment Patterns

Pattern Name	Short Description	NFR
Content Distribution Network	Applications component instances and data handled by them are globally distributed to meet the access performance required by a global user group.	Scalability, Location
Hybrid User Interface	Varying workload from a user group interacting asynchronously with an application is handled in an elastic environment while the remainder of an application resides in a static environment.	Scalability
Hybrid Processing	Processing functionality that experiences varying workload is hosted in an elastic cloud while the remainder of an application resides in a static environment.	Scalability
Hybrid Data	Data of varying size is hosted in an elastic cloud while the remainder of an application resides in a static environment.	Scalability
Hybrid Backup	Data is periodically extracted from an application to be archived in an elastic cloud for disaster recovery purposes.	Scalability
Hybrid Backend	Backend functionality comprised of data intensive processing and data storage is experiencing varying workloads and is hosted in an elastic cloud while the rest of an application is hosted in a static data centre.	Scalability

CC BY-SA

Pattern Name	Short Description	NFR
Bare-Metal Provisioning (new)	Specialized discovery and deployment agents can be utilized within the remote bare-metal provisioning system to locate and provision available bare-metal servers with operating systems dynamically.	Scalability
Platform Provisioning (new)	A system can be established whereby ready-made platforms with packaged, pre-configured IT resources can be provided as turn-key environments for cloud consumers that do not wish to assume significant administrative responsibilities.	Scalability
Intra-Storage Device Vertical Data Tiering (new)	A cloud storage device capable of supporting multiple disk types is used to enable dynamic vertical scaling confined to the device.	Scalability
Storage Workload Management (new)	A storage capacity system is provided to distribute runtime workloads between different cloud storage devices, across the network, and to enable LUNs to be divided and managed.	Scalability, Performance
Redundant Physical Connection for Virtual Servers (new)	A redundant, physical backup network connection is established for virtual servers	Availability
Virtual Server Connectivity Isolation (new)	The virtual server is not allowed to connect to any part of the solution that has a communication path to the external network or internal network, outside of what is required	Availability
Virtual Server-to-Virtual Server Affinity (new)	Affinity rules are used to ensure that the virtual server group or bundled workload is always hosted by and moved to the same destination host.	Location
Virtual Server-to-Virtual Server Anti-Affinity (new)	Anti-affinity rules are used to ensure that the virtual servers or bundled workload are never simultaneously hosted together by the same destination host.	Location
Cloud Denial-of-Service Protection (new)	A cloud DoS protection service is incorporated into the security architecture to shield the cloud provider from DoS attacks.	Availability
Secure Connection for Scaled VMs (new)	A system can be established by controlling network traffic moving in and out of the VM using firewall agents or operating system firewalls. This will create a portable security solution that is location independent and scales as VMs are created.	Scalability

(cc) BY-SA

Pattern Name	Short Description	NFR
Trust Attestation Service (new)	An attestation service is implemented to maintain a trust policy for every attested host and to evaluate reports from the hardware roots of trust from trusted platform modules (TPMs) on each node to determine whether each node has undergone a trusted boot and is in compliance with the security policy.	Availability
Single Node Multi-Containers (new)	All composition participants are deployed in individual containers allowing each to scale independently and as required to fulfill high-performance requirements.	Performance

4.3 Inferring DECIDE patterns from NFRs

The collection of architectural patterns that we provide is only one part of the solution since not all of those patterns are applicable or even desirable for all kinds of applications and environments. The final decision maker, designer and implementer of the application is the person or team that we call, in the context of this deliverable, the "developer". It is the purpose of the DECIDE project, though, to guide the developer into making informed decisions on which patterns she should apply based on the application's nonfunctional requirements (NFRs) and at which part of the application development cycle should each pattern be applied.

In order to provide those pattern suggestions, we require from the developer to define at least a specific set of abstract properties for each NFR that will be used as input to the pattern inferring algorithm. We also classified each pattern based on which part of the development process each pattern should be applied. Lastly, we added a set of properties on each pattern that denote the impact of each pattern to the NFRs.

Based on the NFR properties as input and the pattern properties as weighting factors, the inferring algorithm can provide a prioritized set of mandatory and optional patterns that should/could be applied to the application for the successful fulfillment of the application's requirements.

In the rest of this new section, we will first define the goal of the inferring algorithm and, more broadly the ARCHITECT module as a whole, we will describe the whole process that starts at the definition of the NFRs and ends at the presentation of the ARCHITECT results, we will define in detail the NFR properties as well as the pattern properties that are relevant to this process and, lastly, we will describe the pattern inferring algorithm itself along with the presentation of the results.

4.3.1 Goals

The two main goals of this module as part of the DECIDE project can be defined as following:

1. Provide a repository of architectural patterns that can be applied in the context of multi-cloud applications
2. Suggest a set of architectural patterns to the developer based on the non-functional requirements as defined by the developer herself

Additionally, and in order to make those suggestions more relevant and help the developer to make informed decisions on the actual application of those patterns, we define three secondary goals

1. Define the minimum set of patterns that should be used in order to fulfil the NFRs

(cc) BY-SA

2. Give an additional set of optional patterns that could be used without lowering the NFRs
3. Categorize those suggestions based on the part of the development process that each pattern is relevant to.

Those goals can be achieved through the pattern inferring process that we describe in the next section and with the help of the ARCHITECT tool that acts as the user interface for the process.

4.3.2 Process

As described in section 4.2 there are a multitude of architectural patterns that can be applied in the context of a multi-cloud application. The set of patterns that we presented in that section is only the preexisting knowledge that should be distilled based on the requirements and transformed into a set of useful patterns for a given input. We will call this process the "pattern inferring process".

The pattern inferring process starts with the gathering and classification of the architectural patterns that can be applied in the context of a multi cloud application. In the context of this project, the results of this step is the collection of patterns that has been presented in section 4.2. That section contains only the short description of each pattern, the full description along with all the properties like "problem statement", "solution", "reference" or "model" is part of the actual knowledge repository that is being delivered as part of the software implementation of the ARCHITECT module. The patterns in the repository have already been classified based on the part of the development process that they are applicable (development, deployment, optimization, fundamental).

Moreover, each pattern has two properties that describe the impact of the pattern to the NFRs. Those properties are called "provides" and "requires" and their notion and meaning is described in the next sections. In the context of the process, those properties act as weighting factors to the decision process of the pattern inferring algorithm.

The second step of the process is the definition of the NFR properties. The NFRs, in the context of the DECIDE project, are classified into the following classes:

- Availability
- Scalability
- Performance
- Cost
- Location

Furthermore, each NFR has at least an abstract value and a real value that should be given by the developer as input. The notion of the usefulness of those values is described in the next section.

The third step of the process is the actual inferring of the set of patterns that are useful to the application based on the NFRs and the existing knowledge repository. This step is implemented by the pattern inferring algorithm. The main functionalities of this algorithm are the following three:

1. Select an initial superset of patterns that, when applied, can fulfill the given NFR properties
2. Prioritize the patterns in this set based on the impact that each pattern has to the given NFR properties
3. Define the minimum set of patterns that fulfil the given NFR properties and classify it as the mandatory set. Classify the remaining patterns as part of the optional set.

More details on the algorithm can be found in the next sections.

The last step of the process is the presentation of the results as a grid of classified patterns. The horizontal axis being the pattern classification "development"/"deployment"/"optimization". The vertical axis being the pattern classification "mandatory"/"optional". This presentation is done via the

CC BY-SA

ARCITECT tool based on the results of the algorithm and the preexisting classification of the patterns in the pattern repository.

4.3.3 NFR properties

The non-functional requirements, as given by the developer, serve as the input for the pattern inferring process. Apart from a specific value for each NFR, in the context of architectural patterns, we also define (and require) at least one more property that we call the "Abstract Value" which is measured in qualitative terms (e.g. low/medium/high availability).

The notion behind the existence of the abstract value is that, in the context of architectural patterns, the impact of each pattern to an NFR cannot be measured exactly but can only be described in abstract terms (e.g. redundant database storage provides high availability). This holds true for all vendor agnostic patterns. Only some vendor specific patterns could provide concrete numbers but even those numbers should be taken with a grain of salt and not be considered as part of the preexisting knowledge in the repository.

The basic NFR properties that we define in the context of the DECIDE project are the following:

- Abstract Value: The qualitative description of the NFR
- Value: The quantitative description of the NFR
- Unit: The relevant measurement unit of the "value" property

Each NFR has *different* possible values for each property since each NFR is defined by different metrics. For example, the abstract value of availability is low/medium/high while the abstract value of location is "single location"/"single country"/"cross border". The value of performance can be measured in ms (response time) while the value of cost can be measured in order of magnitude (0s)

Lastly, in the context of pattern inferring, the ordering can be either positive or negative depending on the NFR. Availability, Scalability, Performance have positive ordering which means that the higher abstract values are the better. Cost and Location have negative ordering which means that lower abstract values are better. The detailed table of possible values and their ordering is given in the appendix A. This distinction is taken into account during the pattern inferring algorithm during the threshold operations.

4.3.4 Pattern properties

Regarding the actual architectural patterns, for the purposes of pattern inferring, we added two properties that describe the impact of a specific pattern to a non-functional requirement. Apart from the regular properties for each pattern, the detailed properties can be found in appendix A, we added the properties "requires" and "provides". The possible values for those properties are a specific NFR and an abstract value as described in the previous section.

The property "*provides*" is a mandatory property and describes the impact that the application of this architectural pattern will have to at least one NFR. For example "leader node election" provides "high" "scalability". Each pattern can have one or mode "*provides*" properties. On the other hand, there are patterns that "*require*" a specific threshold in one or more NFRs (usually in terms of cost of location), therefore those patterns have one or more "*requires*" properties. This means that the application of that specific pattern is feasible only when a specific NFR threshold exists, therefore the inferring algorithm will suggest it only if the developer adds this NFR as input.

The full list of properties for each pattern is part of the software implementation of the ARCHITECT tool.

(cc) BY-SA

4.3.5 Inferring algorithm

Based on the above definitions, the pattern inferring algorithm can be described. The comparison operations of the NFR values take into account the positive or negative ordering depending on the NFR.

Input:

1. A set of patterns. Each pattern has at least one "*provide*s" property and zero or more "*requires*" properties. Each "*provides"/"requires*" is an "abstract value" of an NFR
2. A set of NFRs. Each NFR has an "abstract value"

Steps:

1. For each NFR
    a. Get the set of patterns from the repository that "*provide*" "at least" the "abstract value"
2. Merge the pattern sets and remove the duplicate patterns
3. For each NFR, given the merged set
    a. Remove the patterns that "*require*" "more" of the "abstract value"
4. Given the truncated set
    a. Find the smallest set of patterns that "*provide*" "all" the "abstract values" (multiple iterations)
    b. Assign the set as the "mandatory set".
    c. Assign the remaining set as the "optional set"
5. Split the patterns in each set into their deployment/development/optimization classification
6. Return a 3x3 matrix

Output:

1. A 3x3 matrix of patterns. The horizontal axis is the deployment/development/optimization and the vertical axis is the mandatory/optional

4.3.6 Presentation of results

The results of the pattern inferring process are presented to the developer via the user interface of the ARCHITECT module. The developer can then select one or more of the suggested patterns and the selection is written in the application description for further use from the rest of the DECIDE modules (mainly OPTIMUS) as well as the developer herself. Further details can be found in the user manual of the module in the appendix.

(cc) BY-SA

5 ARCHITECT Tool

The main purpose of the ARCHTECT tool is to help the application developer to prepare the application for a multi-cloud runtime environment. To make the best out of the cloud provider offerings is often very challenging. The ARCHITECT tool will try to propose patterns that are able to fulfil the non-functional requirements of the application as described in the previous section.

Multi-cloud computing patterns have been introduced in this deliverable as cloud provider-independent solutions to reoccurring problems in Cloud Computing. By using semantic technologies for modelling patterns, NFRs and their relationships, the ARCHITECT tool is able to infer related multi-cloud patterns to the developer. All patterns are semantically described in a pattern compendium together with a set of NFRs.

This compendium contains also semantical relationships which allow matching of related patterns. In further releases, the infer mechanism will be extended and optimized to incorporate more application information by identifying specific problems that can be addressed by specific multi-cloud architectural patterns. This step, however, requires detailed insight into the intricacies of the software architecture at hand, the multi-cloud computing paradigm, the offerings of specific cloud providers, and the utilization of the given application

Furthermore, based on the list of functional requirements, several use cases for the developer were identified. These are mainly the creation of a new project, a change of NFRs or a change of selected patterns of an already existing DECIDE project. Finally, the developer or the used CI tool should be able to enter the next DECIDE phase in triggering OPTIMUS for the most appropriate deployment configuration.

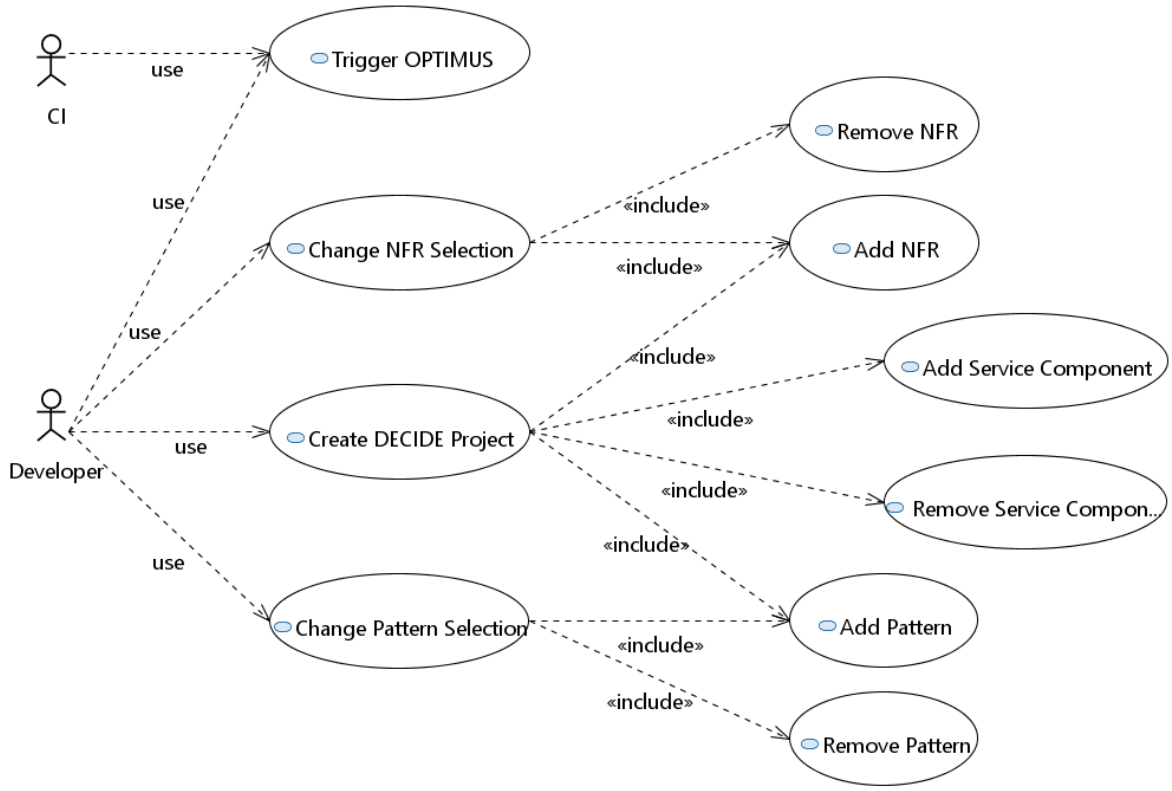

Figure 2. ARCHITECT Tool Use Cases

CC BY-SA

The following sequence diagram shows the “Create DECIDE Project” process. The other use cases are more or less an integrated part of this use case. This is an updated version of that presented in D3.1 [1].

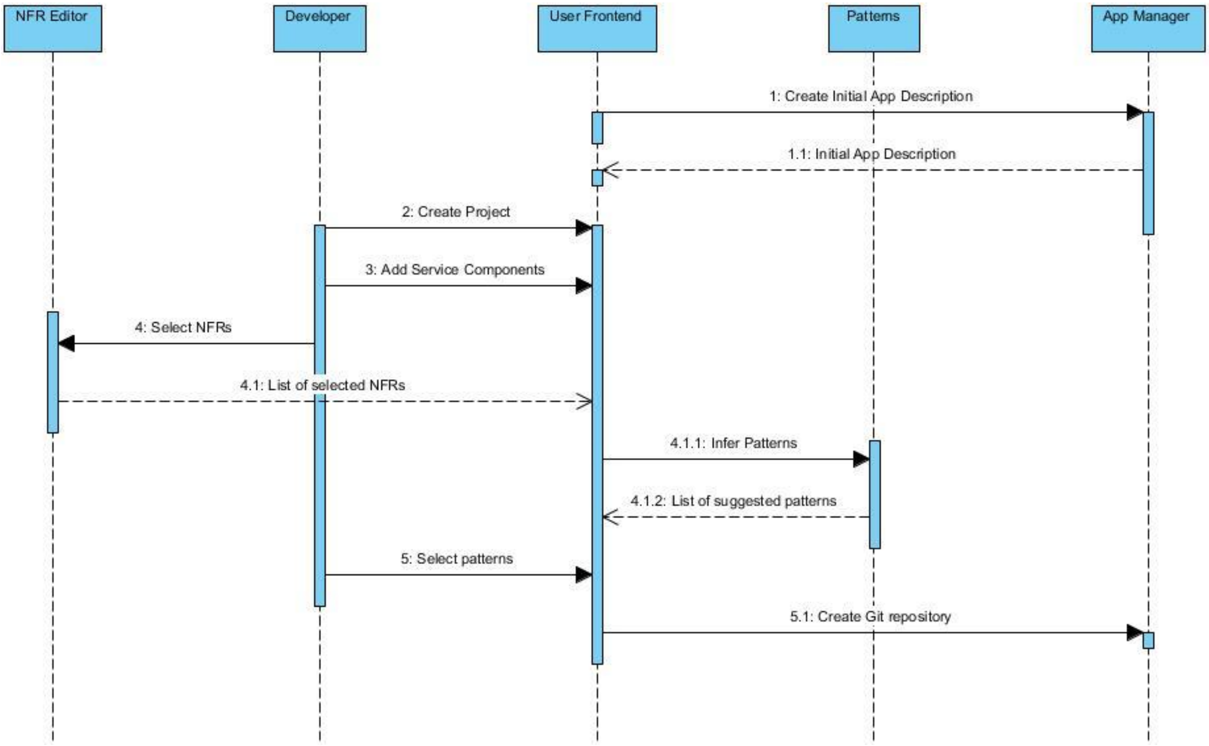

Figure 3. Use Case Create DECIDE Project

- The developer starts the creation of a new DECIDE project.
- The *User Frontend* part requests an initial *Application Description* from the *Application Manager*.
- The *User Frontend* shows to the user a form that requires the general information about the application, e.g. which micro-services are contained and how they are related to each other and to the application in general.
- The *User Frontend* shows to the user the NFR Editor where she can select a set of prioritized NFRs. The NFR Editor returns to ARCHITECT *User Frontend* with the selected list of NFRs.
- Based on the selected NFRs and the additional application information, a list of patterns is suggested to the developer. This list contains both fundamental and inferred patterns.
- The developer is asked to select any patterns from the catalogue that should or must be applied to the application design.

After the developer has finalized the list of applied patterns for the application, the User Frontend finishes the creation process by persisting the final *Application Description* using the *Application Manager.*

5.1 DECIDE Context

The ARCHITECT tool is used during the design and development phase of the application. It is logically embedded into the DECIDE framework between the NFR editor and the OPTIMUS tool. Practically, the ARCHITECT tool utilizes the given NFR Editor for collecting the requirements and triggers OPTIMUS for entering the simulation phase:

CC BY-SA

Relationship to NFR Editor

ARCHITECT utilizes the NFR Editor for collecting the set of defined non-functional requirements from the application developer. ARCHITECT expects as return value from the editor the list of NFRs that the developer has selected.

Relationship to OPTIMUS

For a manual triggering of the simulation phase, ARCHITECT should be able to call OPTIMUS. The main artefact transferred is the *Application Description*. Depending on the provided interface of OPTIMUS it can either be referenced through the Git repository or be handed over as parameter in the API method. The result will be returned using the same mechanism. The *User Frontend* and the *Application Manager* (see Figure 4) may display the result in the current environment in an appropriate way.

5.2 Technical Description

ARCHITECT supports the developer with preparing the application for a multi-cloud deployment scenario by providing and suggesting a set of (multi-)cloud patterns, which must or should be applied to the application.

By means of the functional requirements, ARCHITECT is decomposed in several functional blocks and interfaces. The ARCHITECT component has a set of functional requirements that can be summed up in the following functionalities:

- Provide/recommend to the user (i.e. developer) architectural patterns based on his/her prioritized NFRs and additional information (supplied by the user), with guidelines on how to apply them, to which component these need be applied and in which order. This should be performed through a UI.
- Provide a repository of relevant multi-cloud patterns.

Beside these functional requirements, ARCHITECT will help to initiate the development of an application in the context of DECIDE. This includes the creation of the DECIDE project artefacts, mainly consisting of the Application Description contained in a Git repository.

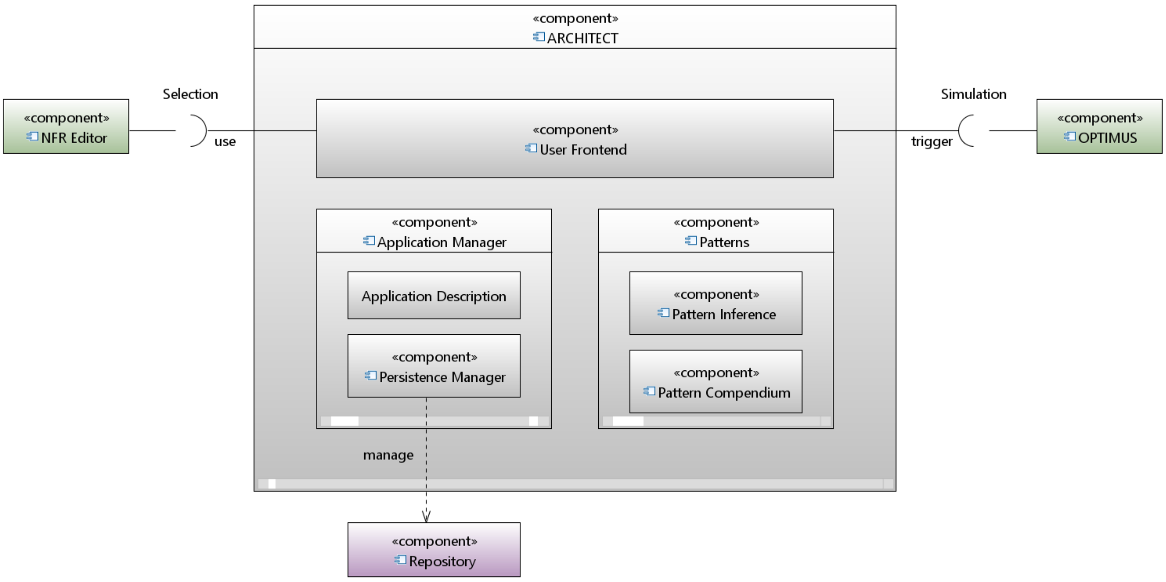

Figure 4. ARCHITECT Tool Architecture

(cc) BY-SA

ARCHITECT consists of three core elements. A frontend for user interaction, an application description manager for dealing with the DECIDE project model, and finally the patterns catalogue with the pattern inference engine.

ARCHITECT itself does not provide any external interfaces. Nevertheless, at least the *Patterns* component will be implemented as an autonomous library and its functionality could be offered as a micro-service in order to be accessible for other implementations. This allows an easy integration of ARCHITECT in a polyglot environment. Nevertheless, ARCHITECT does consume two interfaces, one from the NFR Editor and the other from the OPTIMUS component.

The Implementation is separated in three different projects (code repositories) representing the three main components of the architecture of the tool. Depending on their usage scenario, they are packaged with different purposes:

User Frontend

This element depends on the usage context. E.g. if ARCHITECT is integrated in an IDE, this part provides the mechanism of how the ARCHITECT component is plugged in. The main task is the interaction with the developer and provides necessary user interfaces to collect and maintain all general application information and to enable the use cases. The User Frontend is the workflow-controlling component of the ARCHITECT.

As a first step, the frontend is developed as an Eclipse Plugin. Therefore, it follows the common Eclipse development guidelines for plugins and is implemented in Java. In this release an alternative front end is also developed as a web application embedded in a "DECIDE Dashboard" component.

Application Manager

This element is responsible for a convenient abstraction level for the information model of the DECIDE application. It manages all application information in a persistent manner. That means, it encapsulates and hides the technical details, e.g. the fact that the application is coded and stored as a JSON structure inside a Git repository. The Application Manager encapsulates the Git repository functions and offers a convenient API for dealing with the Application Description JSON file. Most other components of DECIDE need similar functionality. Therefore, this module is provided as a small Java library which guarantees a high reusability.

Patterns

This element contains a catalogue of patterns, NFRs and their relationships. The contained information can be enriched to hold additional information experienced over time. The patterns catalogue provides functions that allow the inferring of patterns based on a given set of NFRs and optionally some fixed patterns.

5.3 Functionality and Requirements Coverage

The following is the list of the functionality implemented in Year 2 of the project for the ARCHITECT Tool:

- F1. Creating an initial DECIDE application project
- F2. Collecting and storing application meta information
- F3. Detecting changes in the application meta information and react with new proposed pattern recommendation when necessary
- F4. Providing a pattern catalogue
- F5. Recommend cloud patterns for the application

(cc) BY-SA

Table 13. Relationship between functionalities and requirements for the ARCHITECT tool

Functionality	Req. ID[4]	Coverage
F1	WP3-ARCH-REQ7	A DECIDE application project becomes manifested in a git repository containing all meta information required for the framework. The ARCHITECT tool prototype is implemented as an eclipse plugin, with an appropriate wizard driven UI to collect necessary initial information allowing the creation of the git repository and the contained initial application description.
F2	WP3-ARCH-REQ6, WP3-ARCH-REQ7	The eclipse plugin (F1) provides a wizard driven form based approach to allow the input of all required information by the developer. Validation and completeness check is included if applicable.
F3	WP3-ARCH-REQ7, WP3-ARCH-REQ8	The prototype ARCHITECT tool is implemented as an eclipse plugin. Changes are part of the application description stored in git. Almost all changes are coming either from the tool itself or the NFR editor which is triggered by the tool. For external changes, the eclipse plugin can apply any functions when the git repository is updated (e.g. by pulling the remote repo).
F4	WP3-ARCHI-REQ1, WP3-ARCH-REQ9, WP3-ARCH-REQ10	The prototype contains a separate library for managing a cloud patterns compendium. Pattern, NFRs, application meta-information and their relationships are described semantically in RDF format. The repository is a triple store. A wrapper to provide the compendium also as microservice is implemented.
F5	WP3-ARCHI-REQ3	The cloud patterns compendium allows the inferring of related patterns, based on a set of given NFRs. This is done by utilizing semantic technologies and infer engines. Simple dependencies between NFRs and multi-cloud patterns are identified and described in the compendium.

In addition to the implementation of these functionalities, ARCHITECT has defined and implemented the iframe for the integration with the DevOps framework [KR1], as well as the means that implement the interaction with the application description and OPTIMUS. Finally, bugs in the code have been corrected.

[4] The requirements for the ARCHITECT tool have been extracted from D2.2 – "Detailed Requirements Specification"

(cc) BY-SA

6 Conclusions

The deliverable at hand, as the second of three, provided an intermediate set of multi-cloud architectural patterns taken from a number of sources that have been deemed relevant in our context.

The deliverable also discussed the notion and definition of multi-cloud applications as defined in the DECIDE project. Furthermore, the benefits and challenges when adopting such a strategy have been laid out. These challenges give way to underpinning the need and usefulness of architectural patterns by which developers can be guided in the development, optimisation and deployment of an application. Hereinafter the deliverable provides set of relevant architectural patterns that have been selected for this matter.

Another aspect that has been discussed, is the need for specific architectural patterns for the use of the DECIDE DevOps Framework and design the application to be multi-cloud aware and be (re)deployed, monitored and (r-)adapted with no or minimal downtime. These are based on the fundamental concepts: *Separation of Concern and Distribution, Containerized Services, External Configuration Storage and Service Registration and Discovery.*

Furthermore, the deliverable gave a list of patterns that improve the development, deployment of the application and ultimately optimize it for a multi-cloud scenario. These have been selected from a number of previous projects and several resources as presented in the state of the art analysis.

The pattern inferring process that results in the suggestion of the architectural patterns that are relevant to an application based on given NFRs has been described in detail and shown how it operates as the basis of the ARHITECT tool.

The ARCHITECT Tool [KR2] that has been presented contains a wizard and a patterns compendium for guiding the developer in designing and architecting her multi-cloud application. The compendium describes patterns and their relationships using semantic technologies.

6.1 Future Work

In the next deliverable that follows we will be looking at how OPTIMUS can benefit from the pattern selection in an automated way by leveraging the modelling of the patterns and their properties. Lastly, the final version of the ARCHITECT tool integrated with OPTIMUS will be released.

(cc) BY-SA

7 References

[1] DECIDE Consortium;, “D3.1 Initial architectural patterns for implementation deployment and optimization,” 2017.

[2] C. Fehling, F. Leymann, R. Ralph, W. Schupeck and P. Arbitter, Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications, Springer, 2014.

[3] ARTIST Consortium, “The Artist Project,” 2014. [Online]. Available: https://cordis.europa.eu/project/rcn/105117_es.html. [Accessed November 2018].

[4] B. D. Martino, “Semantic Techniques for Multi-cloud Applications Protability and Interoperability”.

[5] T.-F. Fortiş and N. Ferry, “Cloud Patterns,” in *Model-Driven Development and Operation of Multi-Cloud Applications: The MODAClouds Approach*, Springer International Publishing, 2017, pp. 107--112.

[6] P. Jamshidi, C. Pahl, S. Chinenyeze and X. Liu, “Cloud Migration Patterns: A Multi-Cloud Architectural Perspective,” *10th International Workshop on Engineering Service-Oriented Applications,* 11 2014.

[7] Arcitura Education Inc., “CloudPatterns.org,” [Online]. Available: http://www.cloudpatterns.org/. [Accessed 2018].

[8] “AWS Cloud Design Patterns,” [Online]. Available: http://en.clouddesignpattern.org. [Accessed September 2017].

[9] “Microsoft Azure Cloud Design Patterns,” [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/.

[10] F. Leymann, C. Fehling, S. Wagner and J. Wettinger, “Native Cloud Applications: Why Virtual Machines, Images and Containers Miss The Point!,” in *Proceedings of the 6th International Conference on Cloud Computing and Service Science (CLOSER 2016)*, SciTePress, 2016, pp. 7-15.

[11] P. Mell and T. Grance, “The NIST Definition of Cloud Computing,” National Institute of Standards and Technology, Gaithersburg, MD, 2011.

[12] A. E. Inc, “SOA Patterns,” [Online]. Available: http://soapatterns.org/design_patterns/containerization. [Accessed 09 10 2017].

[13] C. Richardson, “Microservices.io,” [Online]. Available: http://microservices.io/patterns/deployment/service-per-container.html. [Accessed 10 10 2017].

[14] M. Fowler and J. Lewis, “Microservices, a definition of this new term.,” 25 3 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed 01 10 2017].

[15] C. de la Torre, “Microservices and Docker containers: Architecture, Patterns and Development guidance,” 2 08 2017. [Online]. Available:

(cc) BY-SA

https://blogs.msdn.microsoft.com/dotnet/2017/08/02/microservices-and-docker-containers-architecture-patterns-and-development-guidance/. [Accessed 10 10 2017].

[16] R. Stoffers, “Containers and Containerization - Applications and Services on Steroids?,” *Service Tech Mag,* pp. 3 --10, Q2 2016.

[17] “SockShop App,” [Online]. Available: https://microservices-demo.github.io/.

[18] U. Zdun y P. Avgeriou, «Modeling Architectural Patterns Using Architectural Primitives,» *SIGPLAN Not.,* vol. 40, pp. 133--146, 2005.

[19] U. Zdun, P. Avgeriou, C. Hentrich y S. Dustdar, «Architecting As Decision Making with Patterns and Primitives,» de *Proceedings of the 3rd International Workshop on Sharing and Reusing Architectural Knowledge*, Leipzig, 2008.

CC BY-SA

Appendix A. NFR Properties and Pattern Language

Table 14 describes the properties and structure for the NFRs. This structure is used to describe the qualitative and quantitative values for each NFR in the application description. It is also used as input for the pattern inferring algorithm as well as the OPTIMUS module.

Table 14. NFR Language

NFR			
Property	**Description**	**Type**	**Values**
Type	The type of the NFR	Enum	Availability Cost Location Performance Scalability
Availability			
Property	**Description**	**Type**	**Values**
AbstractValue	The qualitative property of the NFR	Enum	Low Medium High
Value	The availability as percentage	Percent	0%-100%
Unit	The availability unit	String	e.g. “Uptime”
Cost			
Property	**Description**	**Type**	**Values**
AbstractValue	The qualitative property of the NFR	Enum	Low Medium High
Value	The cost in currency unit and as order of magnitude	Number	10^n
Unit	The currency unit	String	e.g. “Euro”
Location			
Property	**Description**	**Type**	**Values**
AbstractValue	The qualitative property of the NFR	Enum	Single Location Single Country Cross Border
Value	The list of locations	ArrayOf(String)	e.g. [“France”,”Berlin”]
Performance			

CC BY-SA

Property	Description	Type	Values
AbstractValue	The qualitative property of the NFR	Enum	Low Medium High
Value	The response time in unit measurement	Number	e.g. 13.3
Unit	The unit of performance measurement	String	e.g. "ms"
Scalability			
Property	**Description**	**Type**	**Values**
AbstractValue	The qualitative property of the NFR	Enum	Low Medium High
Value	The scalability in unit of measurement and as order of magnitude	Number	10^n
Unit	The unit of scalability measurement	String	e.g. "requests/sec"

Table 15 lists the elements of the patterns language that will be used in the project to describe each pattern. The elements are consolidated from two projects namely [2] and [3]. The language is also reflected in the compendium component for inferring patterns based on NFRs.

Table 15. Pattern Language

Element Name	Description
Pattern name	The name of the pattern
Logo	A logo to identify the pattern (if available)
Type of pattern	Optimization, Development or Deployment
Application context	When can the pattern be applied? Any constraints on the application?
Problem	What is the problem addressed?
Solution	How is the problem addressed?
Architectural mapping	Which component is addressed by the pattern? Application, storage, resource management
Architectural model	The pattern description in UML if applicable

(cc) BY-SA

Element Name	Description
Architectural model image	The UML model as an image
Impact on NFRs	Which NFRs are affected in this context?
Abstraction level	In which abstraction level is the pattern materialized? Among architecture level, application level and provider-specific type.
Related patterns	Which patterns are related to the described one? As it is compiled in the different catalogues we have studied

CC BY-SA

Appendix B. ARCHITECT Software Documentation

The following sections present the document related to the delivery and usage of the different elements comprising DECIDE ARCHITECT.

Appendix B.1 Delivery and Usage: The Eclipse Plugin

The Plugin consists of three Packages: the Update Site, the feature and the plugin Package. The first two have the standard Eclipse structure. The Plugin Package contains the actual implementation and consists of five sub packages:

- `eu.DECIDEh2020.architect.plugin.natures`
  contains the class `ProjectNature` which defines the projects Eclipse nature.
- `eu.DECIDEh2020.architect.plugin.perspectives`
  contains the class `Perspective`, defining the projects eclipse perspective.
- `eu.DECIDEh2020.architect.plugin.descriptorWizard`
  contains the Eclipse Wizard used to create a new DECIDE project.
- `eu.DECIDEh2020.architect.plugin.editor`
  contains the MultiPageEditor
- `eu.DECIDEh2020.architect.plugin.layoutComponents`
  consists of SWT Composites used by the wizard and the editor.

The Plugin uses two libraries developed for ARCHITECT: The Application Manager and the Cloud Patterns Library. The Application Manager manages the state of the Application Descriptor on disc and in the Git repository and its representation. It therefore consists of the model, a class that handles the serialization and deserialization to and from JSON and a class that controls the communication and handling of remote and local Git repositories. The Cloud Patterns Library is providing the pattern and NFR data.

Appendix B.1.1 Building from Source

The source code is provided via a zip file and a Git repository. For each source, a different preparation process is needed to import the project into Eclipse before building the plugin:

Via ZIP file

A special zip distribution is available in repository:

https://git.code.tecnalia.com/DECIDE_Public/DECIDE_Components/tree/master/ARCHITECT

When building the project from the zip file, the jar files from the Apache Jena project have to be extracted from http://archive.apache.org/dist/jena/binaries/apache-jena-3.4.0.zip and saved to the folder plugin/libs/jena/lib/. After doing this, the project can be imported to Eclipse with the import option General > "Projects from Folder or Archive".

From the Git repository

The source code is currently hosted in the following Gitlab repository:

https://git.code.tecnalia.com/DECIDE_Public/DECIDE_Components/tree/master/ARCHITECT

When building the project from the downloaded zip file, the jar files from the Apache Jena project have to be extracted from http://archive.apache.org/dist/jena/binaries/apache-jena-3.4.0.zip and

(cc) BY-SA

saved to the folder plugin/libs/jena/lib/. After doing this, the project can be imported to Eclipse with the import option General > “Projects from Folder or Archive”.

Building with Maven

The plugin can be built with Maven via the goal *clean install*. This will create the update Site in site/target/repository. The path to this folder can be used to install the plugin.

Appendix B.1.2 Installing the Plugin

Via an update file:

To register the update site with Eclipse, perform the following steps after unpacking the zip file:

- Select "Help -> Install New Software..." from the main menu to launch the "Install" wizard
- Click "Add..."
- Click "Local...", browse to the installation Folder in the dialog. Click "OK" to add the site
- Note that the "Install" wizard changes to display the contents of the added site

Via the Update URL

To register the update site with Eclipse, perform the following steps:

- Select "Help -> Install New Software..." from the main menu to launch the "Install" wizard
- Click "Add..."
- Enter the URL of the installation site. Click "OK" to add the site
- Note that the "Install" wizard changes to display the contents of the added site

Appendix B.1.3 User Manual

This Plug-in will create a new project type, a DECIDE project, with which the user will be able to create the DECIDE Application file via a GUI and synchronize it with a git repository. After creating this DECIDE project the user will have access to a GUI via which the DECIDE Application file can be modified. This GUI can also recommend relevant Design Patterns to the user.

The Wizard

There are three ways to create a DECIDE project in eclipse: The user can create a new DECIDE project from scratch, open an existing project or clone an existing project from a git repository. All these things can be accomplished via the new project dialog in Eclipse. It is also possible to convert an existing Eclipse project to a DECIDE project by adding the DECIDE nature to this project.

CC BY-SA

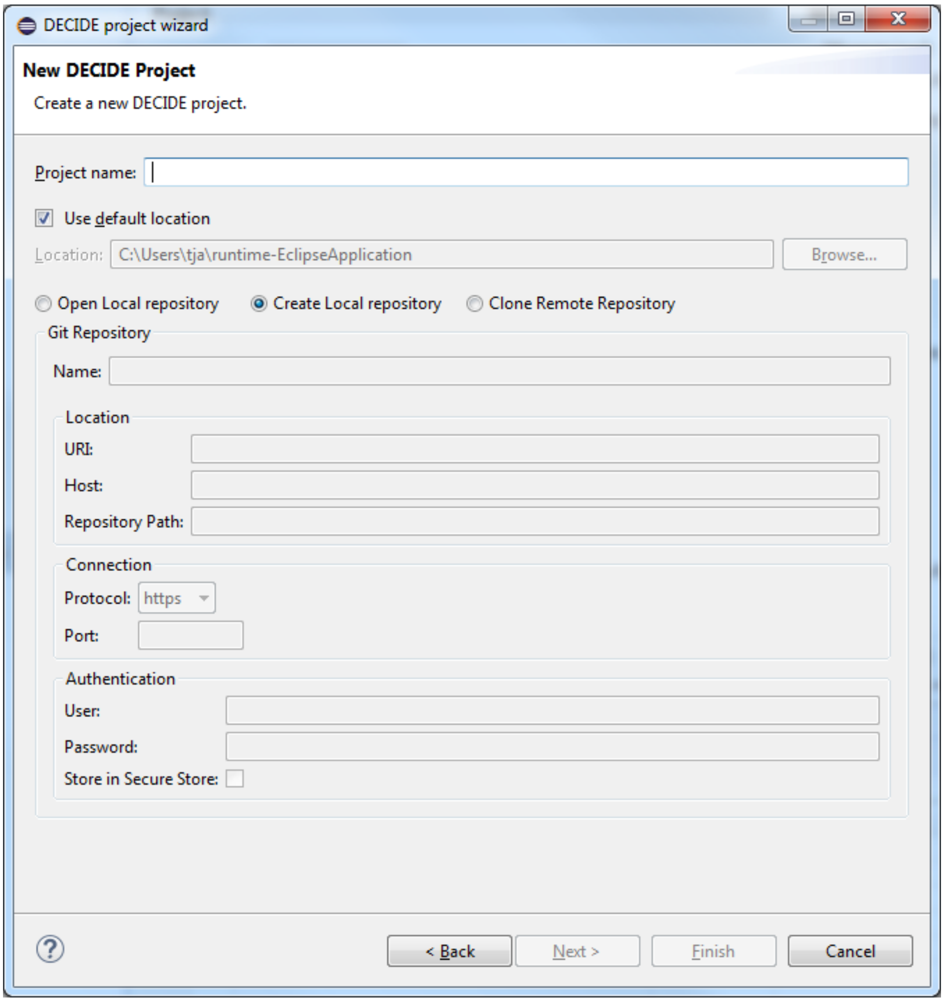

Figure 5. Create Project Wizard

When cloning from a Git repository, the user only has to input the Git URI and credentials in addition to the Project name and path and the repository will be downloaded and opened. In this case, this is also the last and only page of this wizard. If the repository is local, either new or existing, only the path to the repository is needed to open/create the project.

After deciding the location of the repository, the user can set a short text describing the application on the next page of the Wizard. On this page, the user can also set the number of Microservices that are known to be in the Application at the moment.

CC BY-SA

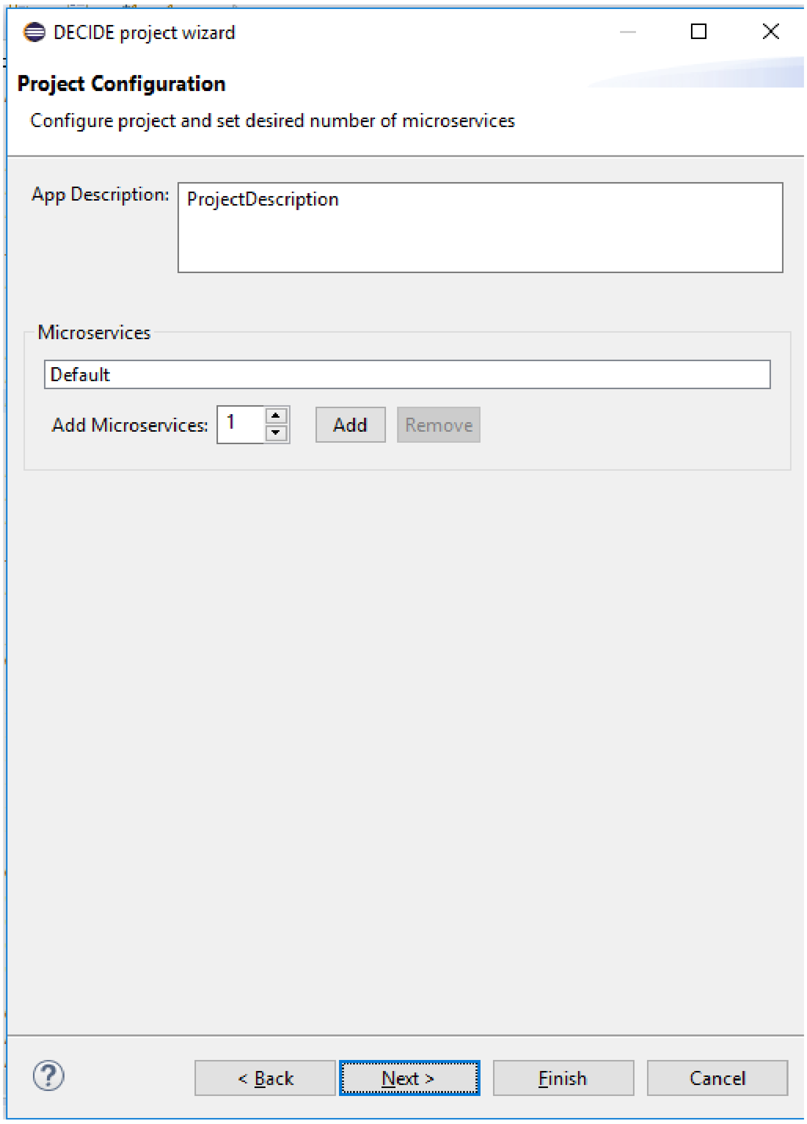

Figure 6. Wizard Add Microservices

The third and last page of the wizard is optional. On this page, the user can see a detailed view of each microservice and can set a name and other properties.

CC BY-SA

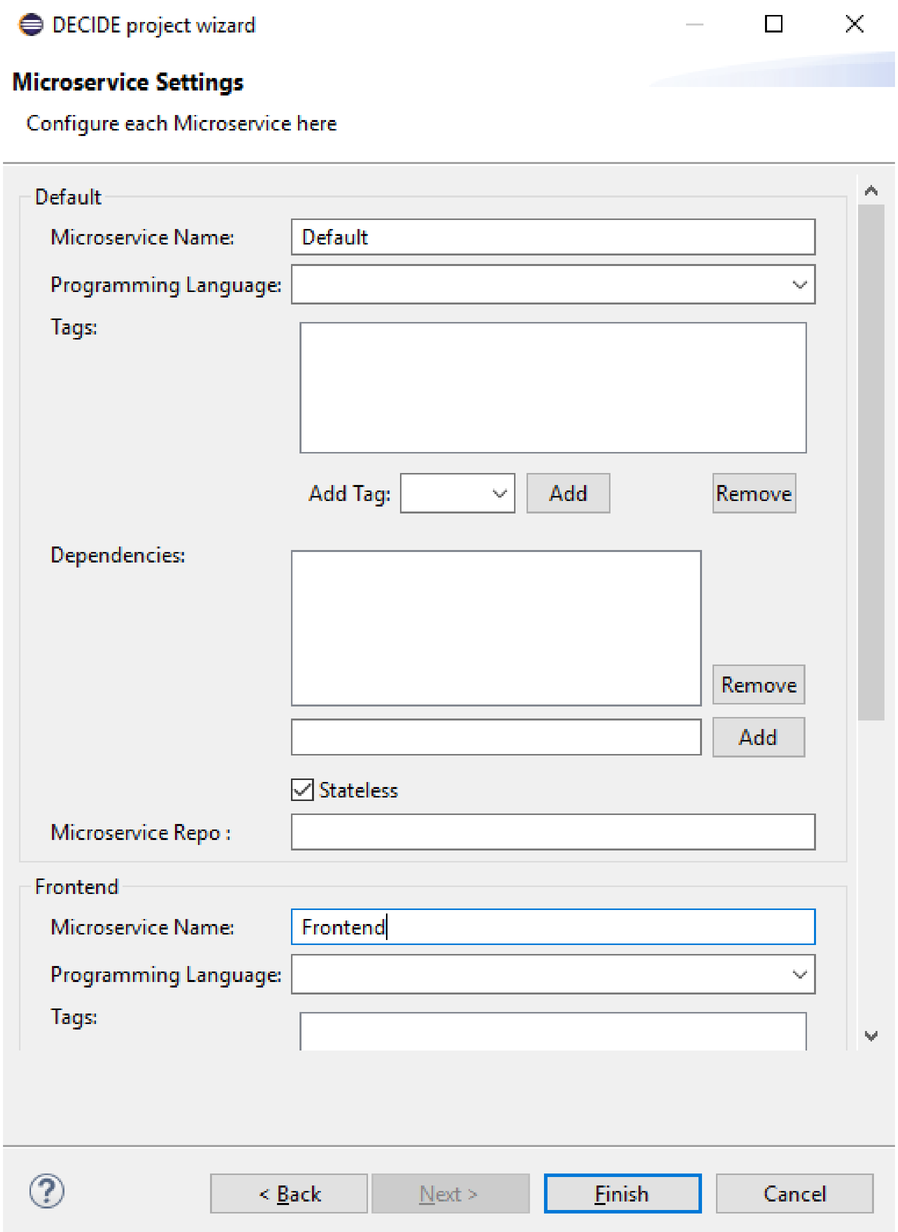

Figure 7. Wizard define Microservices

The Editor

After completing the wizard, the project containing the DECIDE.json will be created and the user will be prompted to open the DECIDE perspective, if it is not already open. In the editor view a Multi-Page editor with three pages will be opened: the first Page, titled "DECIDE.json" shows the raw JSON file; the second page contains a graphical project overview where most information contained in the Application description concerning Architect can be changed via this GUI, e.g. adding tags to microservices to link them to NFRs. The second page is labeled "NFR Editor" since here one can add, remove and edit all NFRs. NFRs can be linked to either one or more microservices or to the application as a whole via tags. Finally, the third page is called "Patterns" and presents the user with the required patterns inferred from the specified NFRs.

The developer has to commit the changes they made either via the Eclipse Git tool or an external Git tool of their choice.

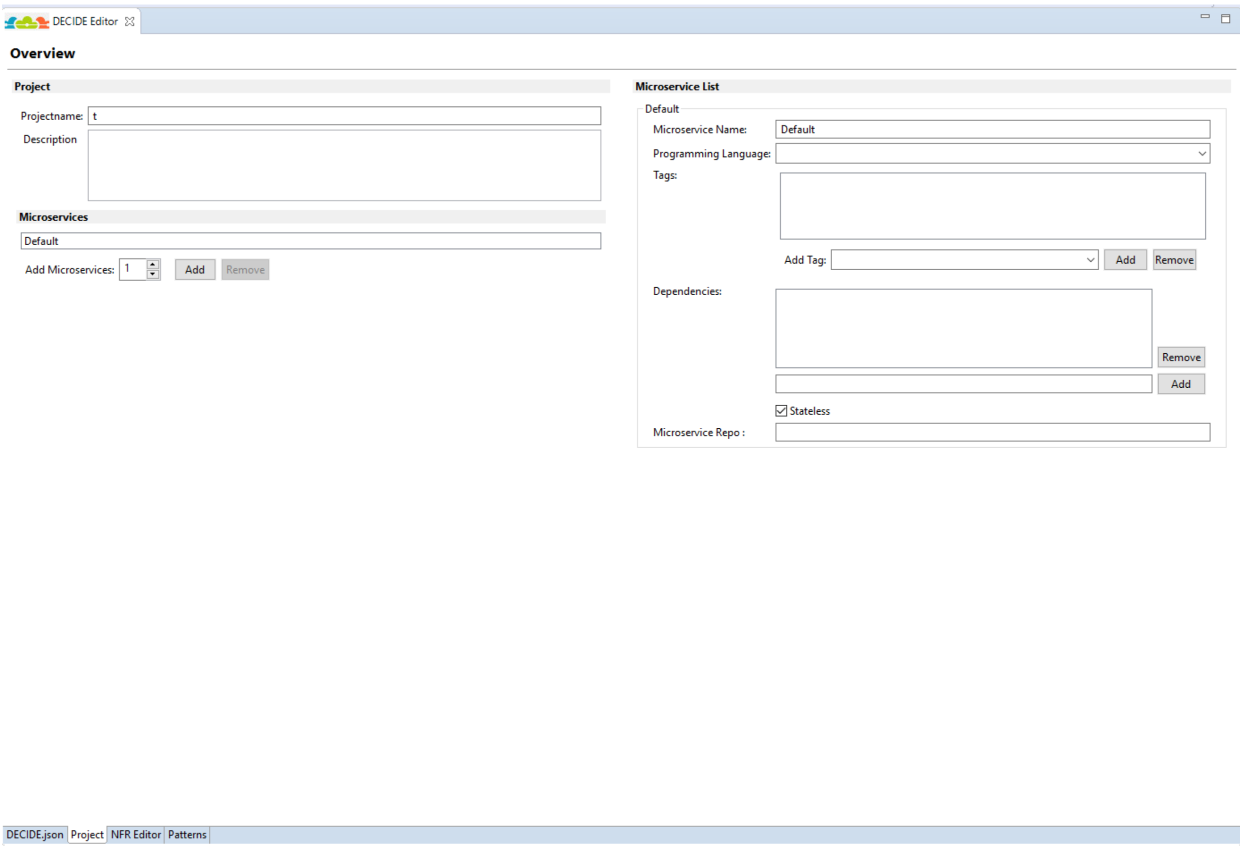

Figure 8. Application Description Editor

CC BY-SA

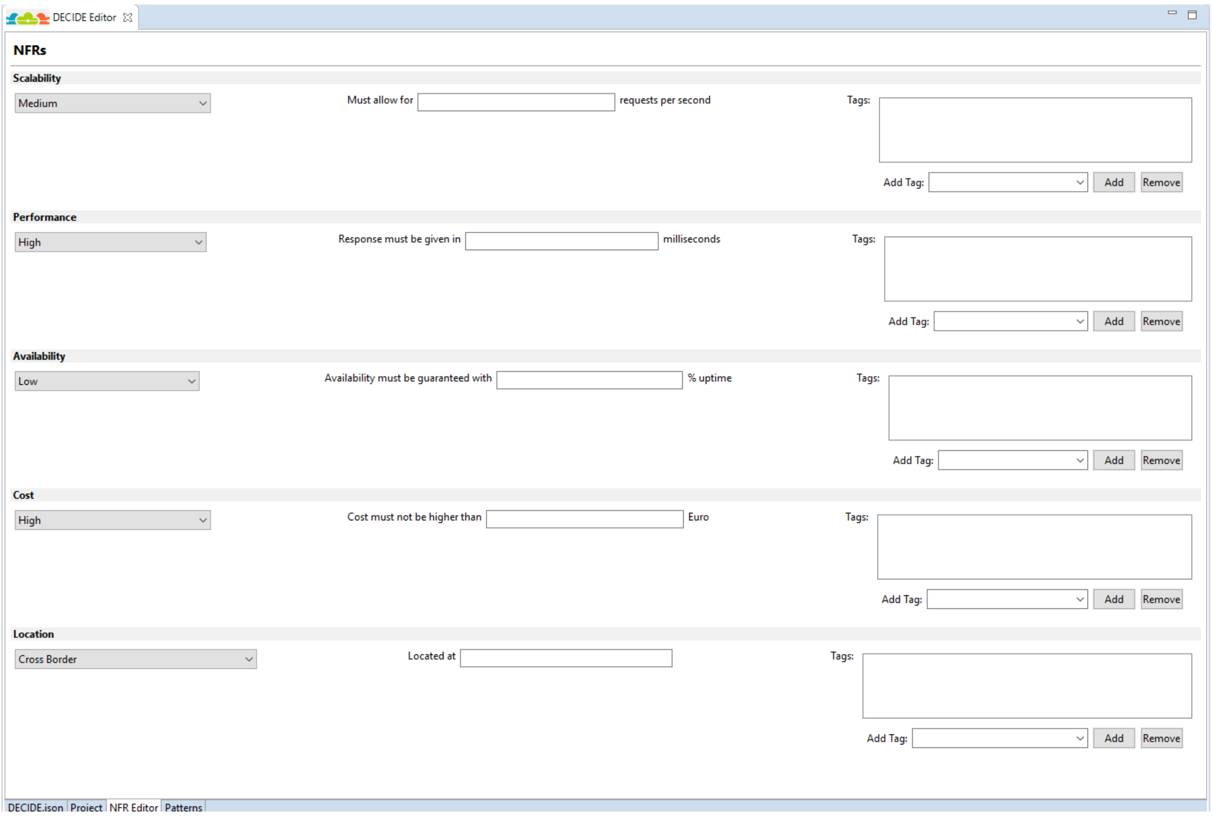

Figure 9. NFR Editor

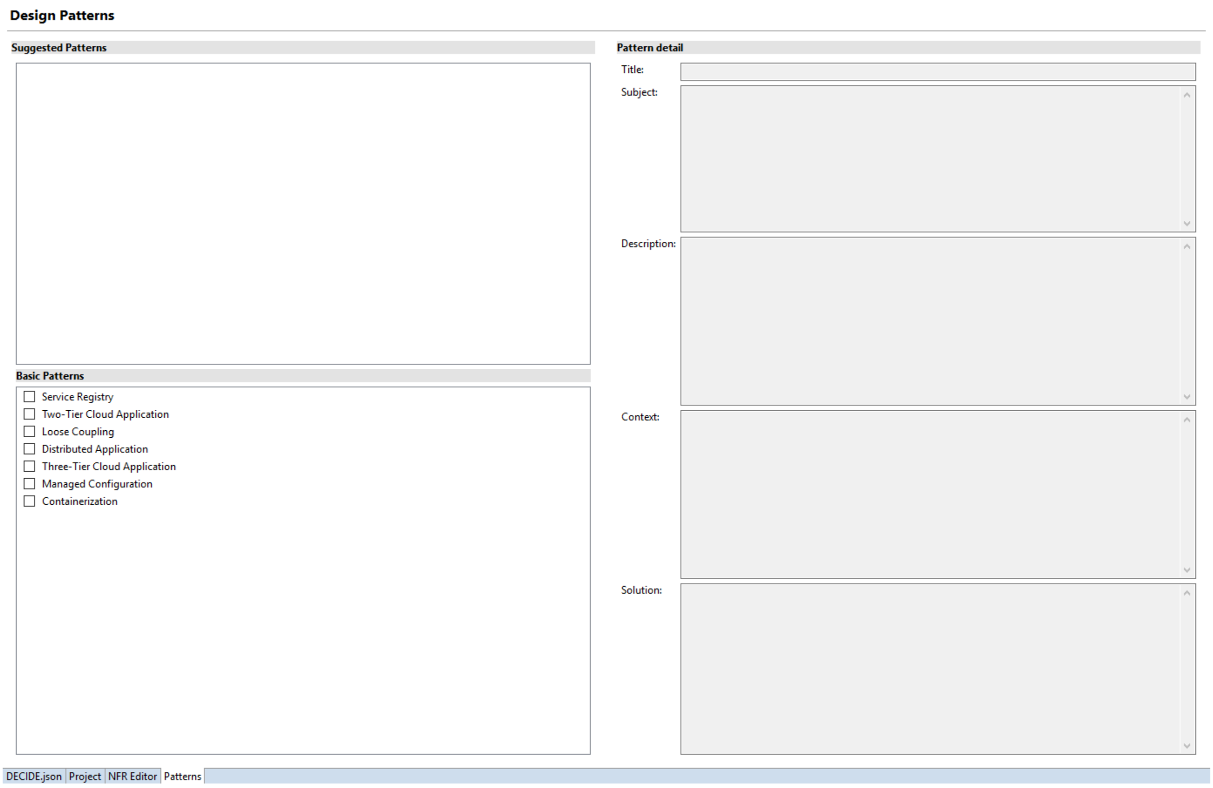

Figure 10. Inferred Patterns

CC BY-SA

Appendix B.2 Delivery and Usage: The Cloud Patterns

The Patterns Compendium is implemented in Java and packaged and provided as .jar library. The library contains a model of patterns defined as ontology. The patterns, NFRs and their dependencies are described in RDF format.

The Ontology "DECIDE Patterns Vocabulary"

DECIDE defines its own pattern ontology. The ontology uses the following namespace:

```
<http://decideh2020.eu/ns/patterns/>
```

In the first release, it defines two main classes and a few properties (we use 'dp:' for the namespace) in order to capture each aspect from the viewpoint of DECIDE:

Class *NFR*

```
dp:NFR
   a rdfs:Class ;
   rdfs:label "Non Functional Requirement"@en ;
   rdfs:comment "A non-functional requirement."@en ;
   rdfs:subClassOf rdfs:Resource ;
   rdfs:isDefinedBy dp: .
```

Class *Pattern*

```
dp:Pattern
       a rdfs:Class ;
       rdfs:label "Pattern"@en ;
       rdfs:comment "A pattern is a reusable solution for a common problem."@en ;
       rdfs:subClassOf rdfs:Resource ;
       rdfs:isDefinedBy dp: .
```

Property *context*

```
dp:context
   a rdf:Property ;
   rdfs:label "context"@en ;
   rdfs:comment "Describes the context of the pattern's problem."@en ;
   rdfs:domain dp:Pattern ;
   rdfs:range rdf:langString ;
   rdfs:isDefinedBy dp: .
```

Property *solution*

```
dp:solution
   a rdf:Property ;
   rdfs:label "solution"@en ;
   rdfs:comment "Describes the solution of the pattern's problem."@en ;
   rdfs:domain dp:Pattern ;
   rdfs:range rdf:langString ;
   rdfs:isDefinedBy dp: .
```

Property *hasImpactOn*

```
dp:hasImpactOn
```


CC BY-SA

```
    a rdf:Property ;
    rdfs:label "Has impact"@en ;
       rdfs:comment "The subject has an impact on the object."@en ;
       rdfs:domain rdfs:Class ;
       rdfs:range rdfs:Resource ;
       rdfs:isDefinedBy dp: .
```

Property `provides`

```
dp:provides
    a rdf:Property ;
    rdfs:subPropertyOf dp:hasImpactOn ;
    rdfs:label "Provides"@en ;
       rdfs:comment "The subject provides to the object."@en ;
       rdfs:domain rdfs:Class ;
       rdfs:range rdfs:Resource ;
       rdfs:isDefinedBy dp: .
```

Property `requires`

```
dp:requires
    a rdf:Property ;
    rdfs:subPropertyOf dp:hasImpactOn ;
    rdfs:label "Requires"@en ;
       rdfs:comment "The subject requires from the object."@en ;
       rdfs:domain rdfs:Class ;
       rdfs:range rdfs:Resource ;
       rdfs:isDefinedBy dp: .
```

In addition, also a SKOS concept theme for the NFR categories are defined:

```
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .

<http://decideh2020.eu/resources/patterncategories>
    a skos:ConceptScheme ;
    rdfs:label "Cloud Pattern Categories"@en .

<http://decideh2020.eu/resources/patterncategories/fundamental>
    a skos:Concept ;
    skos:inScheme <http://decideh2020.eu/resources/patterncategories> ;
    skos:prefLabel "Fundamental Pattern"@en .

<http://decideh2020.eu/resources/patterncategories/development>
    a skos:Concept ;
    skos:inScheme <http://decideh2020.eu/resources/patterncategories> ;
    skos:prefLabel "Development Pattern"@en .

<http://decideh2020.eu/resources/patterncategories/optimization>
    a skos:Concept ;
    skos:inScheme <http://decideh2020.eu/resources/patterncategories> ;
    skos:prefLabel "Optimization Pattern"@en .

<http://decideh2020.eu/resources/patterncategories/deployment>
    a skos:Concept ;
    skos:inScheme <http://decideh2020.eu/resources/patterncategories> ;
    skos:prefLabel "Deployment Pattern"@en .
```

Describe Patterns Semantically

CC BY-SA

Using the concepts from 5.4.1, together with a few other common vocabularies, we are able to describe our patterns semantically in relation to non-functional requirements. Currently we are limited to only define the abstract pre-defined relation “has impact on”, maybe slightly more concrete as “has negative impact on” or “has positive impact on”. The following shows a complete pattern definition in RDF:

```
<stateless-component>
    a dp:Pattern ;
    dct:title "Stateless Component"@en ;
    dct:type <http://decideh2020.eu/resources/patterncategories/development> ;
    dp:icon <urn:stateless_component.png> ;
    dct:subject "How can elasticity and robustness of an application component be
increased?"@en ;
    dct:description "State is handled external of application components to ease
their scaling-out and to make the application more tolerant to component
failures."@en ;
    dp:context "The components of a Distributed Application are deployed among
multiple cloud resources to benefit from this distributed runtime environment
through scaling out.The most significant factor complicating addition and removal
of component instances in this scope is the internal state maintained by them. In
case of failure, this information may even be lost."@en ;
    dp:solution "Application components are implemented in a fashion that they do
not have an internal state. Instead, their state and configuration is stored
externally in Storage Offerings or provided to the component with each request."@en
;
    dct:license <https://creativecommons.org/licenses/by/4.0/> ;
    foaf:page [
        a foaf:Document ;
        foaf:topic "Stateless Component Pattern" ;
        foaf:primaryTopic
<http://www.cloudcomputingpatterns.org/stateless_component/> ;
    ] ;
    dct:relation <relational-database> ;
    dct:relation <key-value-storage> ;
    dct:relation <blob-storage> ;
    dct:relation <message-oriented-middleware> ;
    dp:provides <scalability> .
```

This allows us to simply ask for all patterns that have a positive impact on scalability. In future releases, the full power of the semantic description approach will enhance the query and analytical capabilities for a much better selection of applicable patterns.

Appendix B.2.1 Building from Source

The project is available via Git repository. Download the source code from https://git.code.tecnalia.com/DECIDE_Public/DECIDE_Components/tree/master/ARCHITECT. The component relevant for this is named CloudPatterns.

The project uses Maven as build tool. So, the only thing to do is to call

```
$> mvn clean package
```

in order to build the jar. You will find the jar in the `target` directory.

Appendix B.2.2 Installation and Usage

For non-Maven based projects you can take the build jar file located in the target directory after the build command and put it in the classpath of your application.

CC BY-SA

For Maven based projects you need to install it in a Maven repository which your application can access. E.g. to put it in your local maven repository, you can simply call

```
$> mvn install
```

Finally, your application pom.xml requires the following dependency:

```
<dependency>
    <groupId>de.decideh2020</groupId>
    <artifactId>cloudpatterns</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

`src/main/test`
contains examples in the class `PatternsTest` for how to use the library.

`src/main/resources/patterns`
contains the DECIDE pattern ontology and the turtle-based RDF pattern descriptions.

Appendix B.3 Delivery and Usage: The Cloud Patterns Microservice

The project CloudPatternsCompendium wraps the CloudPatterns library and allowing the library to be deployed as Microservice, offering a convenient REST interface. Patterns and NFRs are exported in JSON format for now. Later releases will support RDF based formats and a SPARQL endpoint as well if necessary.

Appendix B.3.1 Building from Source

The project is available via Git repository. Download the source code from https://git.code.tecnalia.com/DECIDE_Public/DECIDE_Components/tree/master/ARCHITECT.

The component name relevant for this trial is called CloudPatternsCompendium.

The project uses Maven as build tool. So, the ony thing to do is to call

```
$> mvn clean package
```

The packaged jar file can be found in the `target` directory. To start the Microservice type

```
$> java -jar target/cloudpatternscompendium-<version>-fat.jar
```

Appendix B.3.2 Building and Using a Docker Image

You can build and run a Docker image of the microservice:

```
$ docker build -t decide/cloudpatternscompendium
$ docker run -t -i -p 8080:8080 decide/cloudpatternscompendium
```

Appendix B.3.3 Usage

The microservice can be accessed under the following URL:

http://localhost:8080/

CC BY-SA

The OpenAPI 3 specification of the API. The root URL shows a documentation of this specification. There you will find any details about the API.

http://localhost:8080/health

Shows health status of the microservice

http://localhost:8080/metrics
Gives metric information about the API usage. This requires to start the Microservice with `-Dvertx.metrics.options.enabled=true`

Appendix B.4 Delivery and Usage: The AppManager

The AppManager manages the state of the Application Description on disc (local repository) and in the git repository and its representation. It presents an interface to Applications that want to work with the Application Description that is it helps with converting the JSON file into a Java class and back and also helps with the git repository of the app descriptor.

Appendix B.4.1 Building from Source

The project is available via Git repository. Download the source code from https://git.code.tecnalia.com/DECIDE_Public/DECIDE_Components/tree/master/ARCHITECT.

The Patterns Compendium is used as a dependency which means that the usage instructions of the relevant module need to be followed before building the AppManager.

The project uses Maven as build tool. So, the only thing to do is to call:

```
$> mvn clean package
```

in order to build the jar. You will find the jar in the `target` directory.

Appendix B.4.2 Installation and Usage

For non-Maven based projects you can take the build jar file located in the target directory after the build command and put it in the `classpath` of your application.

For Maven based projects you need to install it in a Maven repository which your application can access. E.g. to put it in your local maven repository, you can simply call

```
$> mvn install
```

Finally, your application pom.xml requires the following dependency:

```
<dependency>
    <groupId>de.decideh2020</groupId>
    <artifactId> appManager</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

`src\eu\DECIDEh2020\architect\appManager\models`
contains the App Descriptor models.

CC BY-SA

`src\eu\DECIDEh2020\architect\appManager\persistence` contains the helper classes for handling the json file and git repository.

Appendix C. Sock Shop example app

This section investigates an exemplary application and showcases which multi-cloud patterns, as presented in this deliverable, can be applied in order to render it multi-cloud aware.

Again, multi-cloud aware in the context of DECIDE implies that the application is distributed over different CSPs and can be seamlessly re-deployed, i.e. ported across multiple heterogeneous CSPs.

The selected application, the Sock Shop App [17], is a loosely coupled microservices demo application. It simulates the user-facing part of an e-commerce website that sells socks. It is available as open source software and has been developed with the intention to aid in demonstrating and testing microservices and cloud native technologies.

Appendix C.1 Architecture

The Sock Shop app is designed to provide as many microservices as possible. The microservices are defined by functionality required in an e-commerce site and are loosely coupled. Sock Shop microservices are designed to have minimal expectations, using DNS to find other services. The Application uses a message broker for sending messages using queues.

All services communicate using REST over HTTP. Furthermore, the Sock Shop app is polyglot being built using Spring Boot[5], Go kit[6] and Node.js[7] and is packaged in Docker[8] containers.

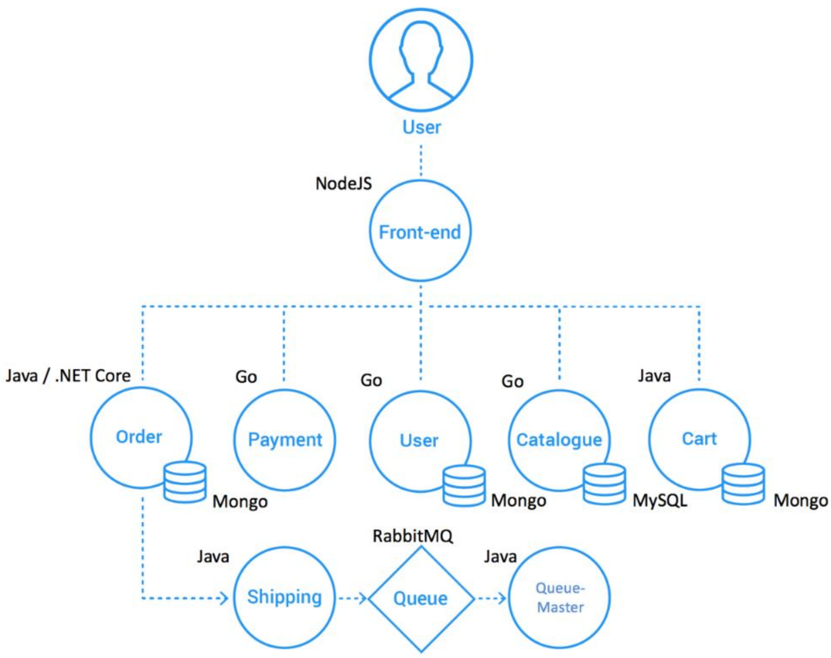

Figure 11. Architecture of SockShop App *[17]*

[5] Spring Boot: http://projects.spring.io/spring-boot/
[6] Go kit: http://gokit.io/
[7] Node.js: https://nodejs.org/
[8] Docker: https://docker.io

CC BY-SA

Appendix C.2 Non-functional Requirements

For the Sock Shop App, we define the following NFRs based on hypothetical assumptions in the context of an e-commerce application and prioritize them as follows:

- **Scalability** – Hypothetically, we may assume that market research and data analytics show that the user base is active during morning hours and after 8 PM otherwise we have unpredictable workloads.
- **Performance** – the performance of the application is important as the users expect rendering of the website and the transactions speed to be at most 2 seconds response time.
- **Availability** – To maintain a good reputation the service has to be available at 99% or at all times
- **Cost** – As the Sock Shop is a start-up, keeping costs to a minimum is vital.

Appendix C.3 Candidate DECIDE Patterns

As noted previously in section 0 patterns provide solutions or best practices for commonly occurring problems [2] with a pattern based approach we can additionally simplify and guide developers in the use of the DECIDE DevOps framework.

This section introduces a selection of patterns that address the NFRs as well as the use of the DECIDE DevOps Framework.

Appendix C.3.1 DECIDE Fundamental Patterns

The Sock Shop app fulfils evidently a number of patterns that are fundamental for the use of the DECIDE Framework. These are:

- **Distributed Application -** The Sock Shop app consists of microservices. This allows the application to be deployed in a distributed manner.
- **Loose Coupling -** The microservices communicate via REST over HTTP.
- **Three-Tier Cloud Application -** Front-end, Business Logic (Order, Shipping, Payment), Persistence (Order, User, Catalogue, Cart)
- **Containerization -** The Sock Shop app is developed as container based architecture.

With these patterns the ground work for the properties: **Scalability, Performance, Cost and Availability** can start to be addressed. Furthermore, the patterns facilitate the use of the DECIDE DevOps Framework.

Other fundamental patterns that still need to be applied are:

- **Managed Configuration –** Deployment and configuration scripts have to be stored in a central area external to the built files.
- **Service Registry -** The Sock Shop app uses DNS to discover services. This is an outdated way of doing things, as DNS propagation is slow. Using DNS tables is probably problematic in a multi-cloud scenario, because of deployment and access issues. Furthermore, given the fact that many instances will be spawned or scaled out and there is probably a number of communications taking place between the different microservices, this needs to be handled by a service. Therefore, we propose the service registry pattern, with which a type of database or table holds the current location of the services, their instances and locations. Registration

CC BY-SA

and de-registration of the service instances takes place during start-up and shutdown, respectively.

Appendix C.3.2 DECIDE Optimization Patterns

The Sock Shop app fulfils a number of optimization patterns that are part of the DECIDE Multi-Cloud pattern catalogue. These are:

- **Elastic Load Balancer –** As workloads are unpredictable at certain times (during the day) it is vital to scale out automatically depending on the current experienced workload. The components resulting in being scaled out by an elastic load balancer are Front-End, Order, Payment, User, Catalogue and Cart.
- **Elastic Queue –** since the Sock Shop app uses message queues in its architecture and scalability is an important NFR, an Elastic Queue should be employed to manage the number of instances (Shipping and QueueMaster) depending on the number requests to be queued.

Appendix C.3.3 DECIDE Development Patterns

The Sock Shop app fulfils a number of development patterns that are part of the DECIDE Multi-Cloud pattern catalogue. These are:

- **Data Access Component –** The microservices, which access a data base are themselves implemented in a way that isolates complexity of data, enable additional data consistency, and ensure adjustability of handled data elements to meet different customer requirements.
- **User Interface Component –** The front-end microservice is decoupled from the rest of the application (i.e. microservices) and loosely coupled. The front-end is therefore, exchangeable and customisable. Furthermore, it can be scaled out independently if need be.
- **Processing Component –** The Sock Shop app's microservices can be scaled out independently, as separation of concerns has been considered here at the design time of the application.

Other development patterns that still need to be applied are:

- **Compliant Data Replication –** This pattern becomes useful if the SockShop App becomes available internationally and certain countries do not allow storing specific data, e.g. a certain subset of user's personal data cannot be stored in country x. If location is important, this pattern should be regarded.

Appendix C.3.4 DECIDE Deployment Patterns

Hybrid * - The patterns involving hybrid cloud, such as hybrid user interface, hybrid processing, hybrid data, hybrid backup, hybrid backend, and hybrid application functions all involve using multiple hosting environments that best suit the requirements and needs of the application. This is relevant in a multi-cloud strategy and can drastically reduce cost if, for instance, certain microservices do not require elasticity they can be hosted on a private cloud that does not feature these capabilities. Also sharing IT-resources between different tenants can drastically reduce costs.

Appendix C.4 Resulting Architecture

Figure 12 depicts the architecture for the Sock Shop app after the recommended patterns have been applied. As one can see, an independent **Configuration Manager** component has been introduced to

CC BY-SA

allow for dynamic configuration of the microservices as well as to allow other automated deployment and provisioning tools to access the configuration information needed for their tasks.

Furthermore, a **Service Registry** has been introduced in order to facilitate the discovery of the location of the microservices that have been newly instantiated by the **Elastic Load Balancer**. And lastly an **Elastic Queue** manages the number of needed Queue Masters depending on the number of the messages received by the Rabbit MQ.

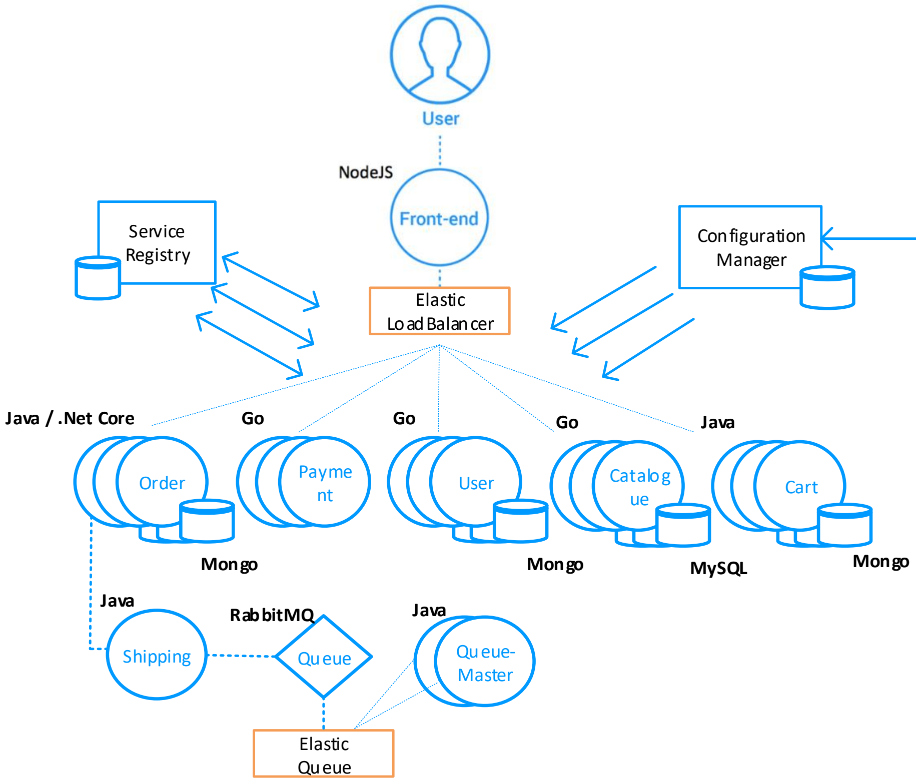

Figure 12. SockShop App Updated Architecture

CC BY-SA