Department of Electrical Engineering and Computer Science
Deanship - Board of Examiners

FACH
HOCHSCHULE
LÜBECK
University of Applied Sciences

Bachelor Thesis

for

Mr. Yuhang C h e n

Developing a Graphical User Interface for Describing Cloud-Native Applications for Elastic Container Platforms

[illegible]

Dipl.-Inf. Peter-Christian Quint

gefördert durch:

Date of handing out: 19th March 2018
Date of handing in: 19th June 2018

A. Hanemann

(Professor Dr. rer. nat. Andreas Hanemann)
Board of examinations

EMAS
GEPRÜFTES UMWELTMANAGEMENT
DE-150-00040

Task Description:

Cloud-native applications are intentionally designed for the cloud in order to leverage cloud platform features like horizontal scaling and elasticity – benefits coming along with cloud platforms. In addition to classical (and very often static) multi-tier deployment scenarios, cloud-native applications are typically operated on much more complex but elastic infrastructures. Furthermore, there is a trend to use elastic container platforms like Kubernetes, Docker Swarm or Apache Mesos. However, describing this service can be extensive and platform specific. Therefore, we developed a domain specific language for describing cloud-native applications, including features like model-to-model generation for specific elastic container platforms.

In this thesis, a graphical user interface for describing cloud-native application for multi-cloud deployments should be implemented.

For this purpose, the following sub-tasks have to be dealt with and documented (Bachelor's thesis):

Task

Note: Should individual aspects shown to be to difficult to achieve during the processing of this work, the task can be adapted - after plausible proof of the difficulty.

Preparation

- Familiarization with UCAML (Universal Cloud Application Modeling Language), cloud native applications (especially Docker), cloud computing and elastic container platforms
- Identification of the GUI requirements
- Design the software architecture of the GUI-service

Implementation and Evaluation of the GUI

- Implementation of the GUI as a web-service
- Automatic validation of the user input
- Practical evaluation of the GUI by using it for describing a cloud-native demo application (e.g., the Sock Shop)
- Writing a developer documentation

[illegible]

Dipl.-Inf. Peter-Christian Quint

Fachbereich Elektrotechnik und Informatik
Dekanat - Prüfungsausschuss

FACH
HOCHSCHULE
LÜBECK
University of Applied Sciences

Statement on the bachelor thesis

I assure that I have written the bachelor thesis independently, without outside help.

Only the given sources have been used for the writing of the bachelor thesis. Literally or the sense after taken parts are marked as those.

I agree that my work is published, in particular that the work is presented to third parties for inspection or copies of the work are made to pass on to third parties.

18/06/2018
Date

Yuhang Chen
Signature

I Abstract of Thesis

Elastic container platforms like Docker, Kubernetes and Swarm Mode have been become increasing popularity in the last years. These platforms can be used to deploy cloud-native applications (CNAs) into cloud infrastructure with microservices architecture. However, the deployments are varied among different platforms. Therefore, a domain specific language (DSL) which can achieve multi-cloud deployment was developed as part of the research project Cloud TRANSIT. In this project, the Unified Cloud Application Modelling language (UCAML) has been extended. This thesis has two major purposes: (1) to introduce the background of UCAML, including Cloud Computing, Microservices, container, CNAs and Container Orchestration. (2) to implement a graphical user interface (UCAML-GUI) to describe CNAs for multi-cloud deployments. This UCAML- GUI is implemented as a web-application by using the MVC design pattern. It can be used to create and edit CNA descriptions with UCAML format, then export it as platform specific description file with Model-to-Model (M2M) transformation.

II Table of Contents

1 Introduction

Generally, most traditional monolithic applications can run in cloud platform without any change only if this cloud platform supports corresponding computer architecture and operating system. However, this kind of working pattern cannot utilize virtual machine efficiently, which ignored what the cloud platforms are capable of. For the benefits of cloud platform and achieve distribution of computer resource on demand and elastic characteristic, cloud-native applications emerged.

A cloud-native application (CNA) is a distributed, elastic and horizontal scalable system composed of (micro)services which isolates state in a minimum of stateful components (Kratzke and Quint, 2017). Developers must break down applications into separate services that can run on several servers. Apparently, this kind of design pattern is designed specifically for cloud computing architecture and is also clear to achieve elasticity and reliability, now we refer to this as microservices. To encapsulate each microservice instance so that it can be easily started, stopped and migrated, a way was adopted to package a microservice into a “container” that would be easier to manage than a full virtual machine image (Gannon et al., 2017). It is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings. In addition, Containers isolate software from its surroundings such as differences between development and staging environments, which help reduce conflicts between teams running different software on the same infrastructure (Docker.com, 2018).

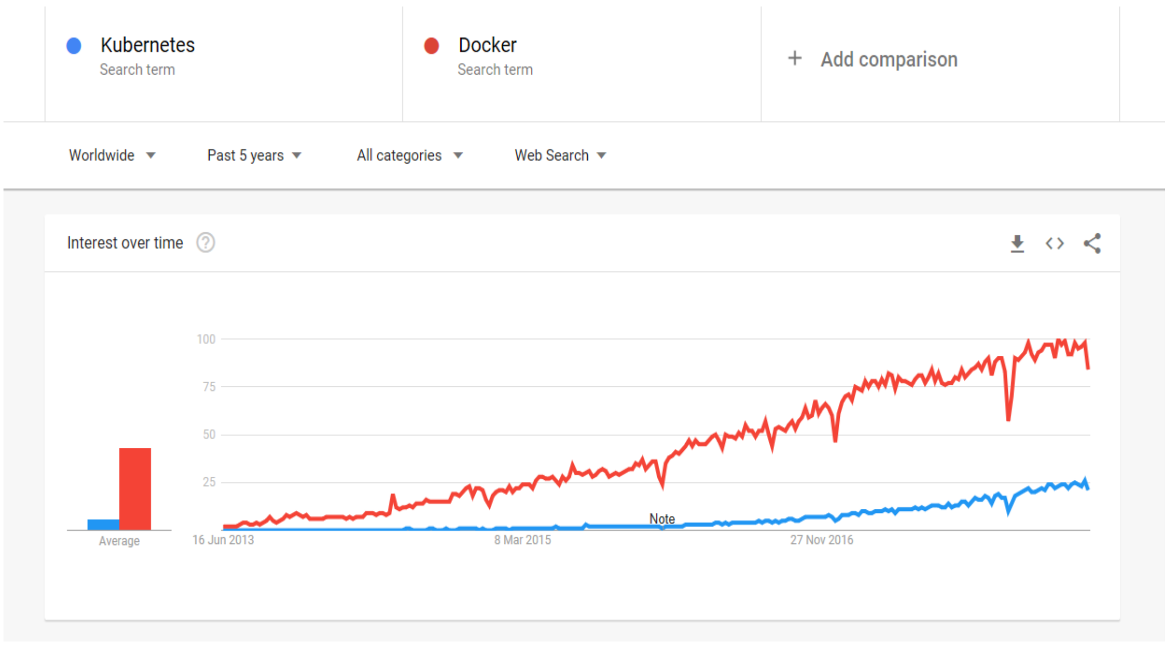


Figure 1: The Google Trends of Docker and K8s (Google Trend, 2018)

As shown on Figure 1, it shows the Google Trend of Docker and Kubernetes in recent 5 years. Obviously, elastic container platforms like K8s, especially Docker have become increasing popularity. However, it is not easy to describe this service due to its extensive and platform specific properties. Therefore, a domain specific language(DSL) for describing cloud-native applications was developed by team of Cloud TRANSIT project in University of Applied Science in Lübeck. This DSL can transform a special format (UCAML[1]) to platform specific format (e.g. Docker, Kubernetes). The UCAML framework determines how a CNA can be described in UCAML format and offers the possibility to test UCAML format CNA descriptions and transform them to a platform specific format. However, there is no possibility to describe a CNA in UCAML format without editing a text file only. Therefore, in this work a graphical user interface application, called UCAML-GUI, was developed to provide a more user-friendly way to describe CNAs.

This thesis intends to introduce the relevant background of UCAML and record the integral developing processes for UCAML-GUI. The UCAML-GUI was implemented as a web-service, which could describe cloud-native application and validate the user's input automatically. The thesis is structured as follows: the chapter 2 introduces related work and background, including Cloud Computing, container and microservices, Cloud-native applications, Container Orchestration and UCAML. In chapter 3 we provide a specification for UCAML-GUI. The implementation process is described in chapter 4. In chapter 5 we show the result of test and make an analysis. Finally, we conclude our finding in chapter 6.

1. https://bitbucket.org/cloudtransit/ucaml

2 Related work and background

2.1 Cloud Computing

Cloud computing is a kind of model that enable ubiquitous, on-demand network access to shared pools of configurable system resources and higher-level services that can be rapidly provisioned and released with minimal management effort (Mell and Grance, 2011). Sharing of resources is the key factor for cloud computing to achieve coherence and economies of scale. It is also one of essential characteristics of cloud computing.

According to the NIST definition (Mell and Grance, 2011), Cloud computing exhibits five essential characteristics:

On-demand Self-service. The end user can provision computing resources, such as server time and network storage on demand and without interaction of service providers.

Broad network access. The computing resources are available over the network and accessed through standard mechanisms. For example, the users just need to log in with an internet connection and then acquire the service from the service providers despite which devices are used.

Resource pooling. The provider's computing resources are pooled to serve multiple consumers according a specific multi-tenant model, which enables consumers to enter and use data in the cloud from any location, at any time.

Rapid elasticity. The providers can scale up or scale down according to the resources demand of consumers. For the providers, it eliminates the waste of resources (e.g., storage, processing, memory) in the inactive local infrastructure. For the consumer, the available capabilities usually appear to unlimited.

Measured service. The resource usage can be monitored, controlled automatically by cloud systems and providing transparency for providers and consumers. The consumers only pay what they use.

As shown in Figure 2, Cloud Computing can be broken down into three services models:

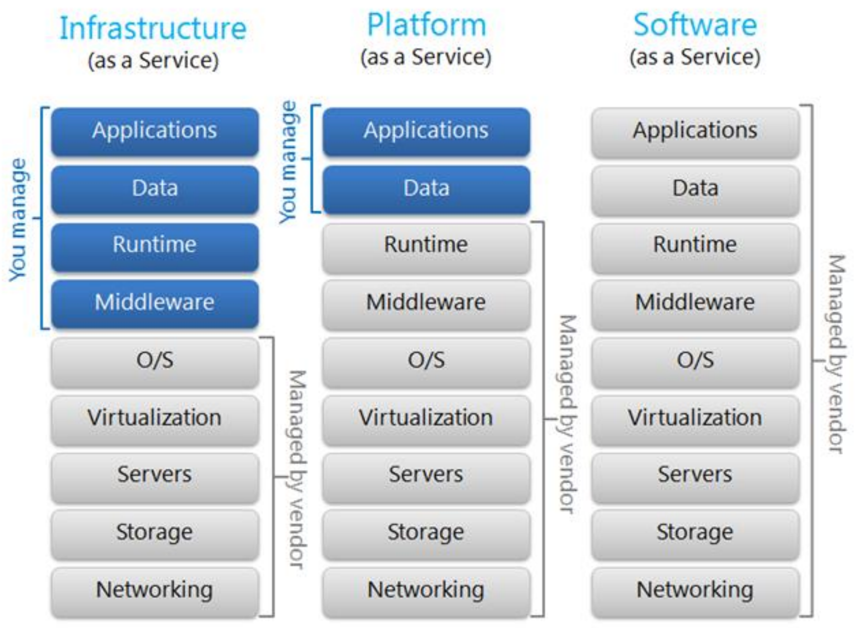


Figure 2: Service models of Cloud computing (Research Hub)

Infrastructure as a Service (IaaS). In the IaaS model, consumers are provided with elements of infrastructure such as Server, storage, virtualization and other fundamental computing resources. Therefore, they can deploy and run arbitrary software, including operating systems and applications.

Platform as a Service (PaaS). In the PaaS model, consumers are provided with capabilities to run their own applications or third-party applications. These capabilities refer to operating systems, programming language, service and tools supported by providers. Lower level elements of infrastructures don't need to be concerned by consumer.

Software as a Service (SaaS). In the SaaS model, consumers are provided with applications (e.g. needed for business) running on a cloud infrastructure. These applications are accessible from various devices like browser or mobile. The consumers need not be concerned about underlying cloud infrastructure including storage, operating systems and another running environment.

In addition to service models, (Mell and Grance, 2011) also defined cloud computing with four deployment models according to different usage for various agencies.

Private Cloud
The cloud infrastructure is configured to be dedicated to a single organization that includes multiple consumers (e.g., business units). It may be owned, managed, and operated by an organization, a third party, or some combination of these, and may be present on or off site.

Community Cloud
Community Cloud is exclusively used by a specific community of user who may come from different organizations but have common concerns (e.g., tasks, security requirements, policies, and compliance considerations). It may be owned, managed, and operated by one or more community organizations, third parties, or a combination of them.

Public Cloud
Public cloud provisions the cloud infrastructure for open use. It may be owned, managed and operated by a commercial, academic or governmental organization, or a combination thereof. It exists in the cloud provider's premises.

Hybrid Cloud
The Hybrid Cloud consists of two or more different cloud infrastructures (private, community or public). They are still unique entities but are bound together through standard or proprietary technologies to implement data and applications Portability (e.g., cloud bursting for load balancing between clouds).

2.2 Microservice and Container

2.2.1 Microservices

Microservices is an architecture approach that can deploy applications and services in the cloud. Each microservice is an application with single a function and must be managed, scaled, upgraded and restarted independent of other services. The communication mechanisms used by microservice systems are varied: they include Representational State Transfer web service calls, remote procedure call (RPC) mechanisms such as Google's Swift, and the Advanced Message Queuing Protocol (Gannon et al., 2017). In addition, microservices use different programming languages, different data storage technologies, and keep a minimum centralized management.

Microservices vs Monolithic architectures
In Figure 3, there are two big differences between microservices and Monolith. Firstly, applying microservices means building applications from separate services running in different processes. Therefore, microservices can be deployed independently. In each service any technology or infrastructure can be used. Contrarily, in a monolith all functionalities are put into a single process (Buzachis Aris, 2014), isolated poorly and interacted through in-process method calls.

Secondly, in terms of scalability, microservices also show its advantages. If one functionality need be scaled, the monolith including multiple servers and other components must be scaled (Buzachis Aris, 2014), which waste resources and increase complexity. But extending microservices is easy. Since all services are independent applications, we can only scale specific services without disturbing others.

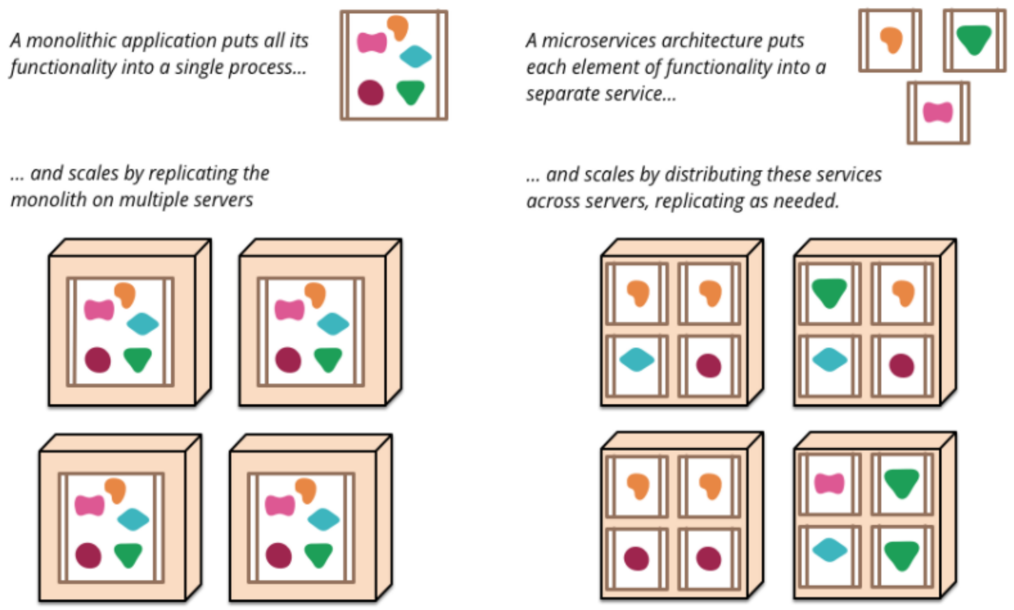

A monolithic application puts all its functionality into a single process...
A microservices architecture puts each element of functionality into a separate service...
... and scales by replicating the monolith on multiple servers
... and scales by distributing these services across servers, replicating as needed.

Figure 3: Microservices vs Monolith (Aris, 2014)

The usage of microservices architecture is increasing dramatically, because most traditional applications who are developed with monolithic architectures are difficult and expensive to maintain, hard to extended. Therefore, if developers want to deploy them into the cloud, they tend to split the applications into smaller microservices which can be maintained separately, scaled separately, or thrown away if needed (Thönes, 2015).

2.2.2 Containers

To encapsulate each microservice instance so that it can be easily started, stopped, and migrated, containers are used (Gannon et al., 2017). Container is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings. (docker.com, 2018). And containers can offer both efficiency and speed compared with standard virtual machine. They are lightweight and designed to run anywhere. Therefore, Container is optimum and ideal way for enabling microservices application development.

Containers vs VMs
Containers (left in the Figure 4) are isolated, they share the underlying host OS and infrastructures, and only package the necessary application and binary files. Each virtual machine (right in the Figure 4) runs its own guest operating system instance and provides its own libraries and binary files. Besides, Containers take up less space than VMs (container images are typically tens of MBs in size) and start almost instantly. Containerization is, in effect, OS-level virtualization, as opposed to VMs, which run on hypervisors with a full embedded OS (Bob Tarzey, 2016).

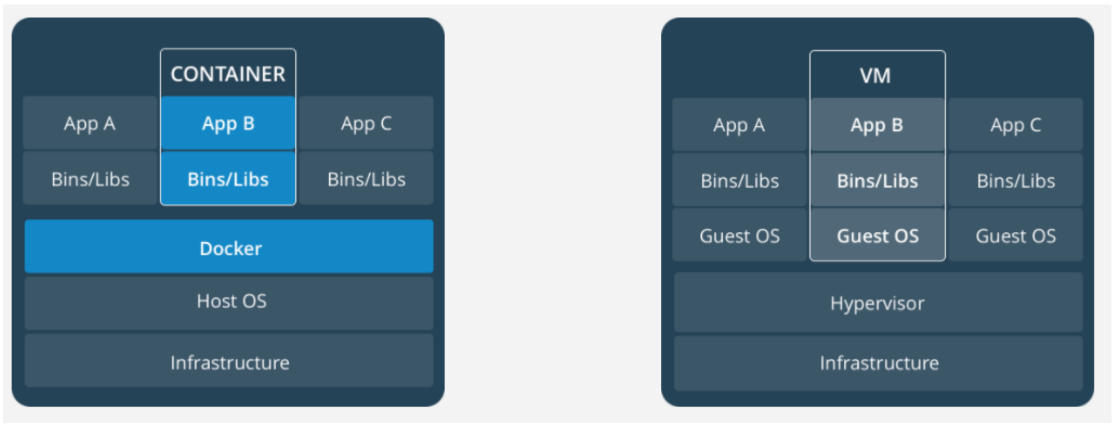


Figure 4: Containers vs VMs (docker.com, 2018)

Docker
Docker is an open source program that designed to make it easier to develop, deploy, and run applications by using containers. The Linux kernel provided an easy solution to the encapsulation problem by allowing processes to be managed with their own namespaces and with limits on the resources that they used. This led to standards for containerizing application components, such as Docker (Gannon et al., 2017).

A complete docker application usually is composed of following parts: docker client, docker Daemon, docker image and docker container. Since client-server model is used in docker, docker daemon in server would accept the request from client. A Docker container is an instance of a docker image, which is similar to the class and instance in object-oriented programming.

In this project, some simple commands in docker are required. For example:

Command:
```
$> docker run busybox echo 'Hello, World!'
```
Output:
```
   Hello, World!
```

This command is telling docker to run busybox image. If this image is not present, docker will attempt to fetch an image named "busybox" from public docker hub. Then

Docker sets up the layers of this image, all the cgroups and namespaces for this container environment, and executes "echo 'Hello, World!'" Also, we can use `$> docker pull fedora` to directly fetch an image named "fedora" from public docker hub. In addition, `$> docker images` will list all available images locally. There are many more docker commands like these. To see them all, see `$> docker help` (Vincent Batts, 2014).

2.3 Cloud Native Applications

According to (Kratzke and Quint, 2017): "A cloud-native application (CNA) is a distributed, elastic and horizontal scalable system composed of (micro)services which isolates state in a minimum of stateful components. The application and each self-contained deployment unit of that application is designed according to cloud-focused design patterns and operated on a self-service elastic platform".

Generally, the CNA natively utilizes the services and infrastructure provided by cloud computing providers such as Amazon EC2 or Force.com. In other words, the CNAs are completely based on the cloud computing model, including its essential characteristics and service models. Thus, its properties that are frequently cited and listed in the following are somewhat similar to Cloud Computing's characteristics.

1. CNAs typically **run on a real global scale**. Although ordinary websites can be accessed anywhere in the Internet, the real global scale means more. This means that the application's data and services are replicated to the local data center, minimizing interaction delays (Gannon et al., 2017).
2. CNAs must **scale well with thousands of concurrent users**. This is another parallel dimension that is orthogonal to the scale of the data needed for global scale distribution. It requires careful attention to synchronization and consistency in distributed systems (Gannon et al., 2017).
3. CNAs assume of **constant infrastructure mobility and failure**. This concept is the basis of the original design of the Internet Protocol, but applications built on a single PC, mainframe or supercomputer assume that the underlying operating system and hardware are rock solid. Therefore, when these applications are ported to the cloud, they can fail due to the first failure in the data center or network. Even if the hardware or network has a very low failure rate, the law of large numbers can guarantee that when you try out the scale of the world, something is always broken or about to burst. (Gannon et al., 2017).
4. CNAs are designed so that **upgrades and tests can be performed seamlessly without interrupting production**. Although not every cloud native application is intended for millions of concurrent users worldwide, most applications are designed for continuous operation. (Gannon et al., 2017). But all applications need to be upgraded without interrupting normal operations and then to test the upgrade.

5. **Security** is also a very important principle in CNAs. As we can see, many CNAs are built from many small components and these components must not have sensitive credentials. A firewall is not enough because access control needs to be managed at multiple levels of the application. Security must be part of the underlying application architecture (Gannon et al., 2017).

2.4 Container Orchestration

The appearance of the container has completely changed the way that cluster is released and operated. Because developers don't need to consider the consistency of environment. However, the usage of the containers brought some new problems. Firstly, although container is lighter than virtual machine, it still requires orchestration system to operate efficiently and reliably. The container resources need to be scheduled and the life cycle needs to be managed systematically. Secondly, it is still hard to scale for a cluster of containers. To solve these problems, elastic container platforms like Docker swarm, Kubernetes and Apache Mesos were born and there is a trend to use them by developers. Following paragraphs will mainly introduce Docker swarm mode and Kubernetes.

2.4.1 Docker Swarm Mode

Usually people are confused about Docker Swarm and Docker Swarm Mode. In fact, they both are container orchestration tools. Docker swarm is a standalone product of Docker while Swarm mode is integrated into the Docker engine in the Docker 1.12 release.

Compared with Docker Swarm, Swarm mode introduces the concept of services and provides many new features. Not only it can cluster and schedule containers, but also can allow user to control the entire lifecycle of application. In addition, Swarm Mode is easy to learn and convenient to use since it is part of Docker engine.

2.4.2 Kubernetes

Kubernetes (K8s), the third container-management system developed at Google, was conceived of and developed in a world where external developers were becoming interested in Linux containers, and Google had developed a growing business selling public-cloud infrastructure. Furthermore, K8s is open source program, which were developed as purely Google-internal systems (Burns et al., 2017). Now it is maintained by the Cloud Native Computing Foundation.

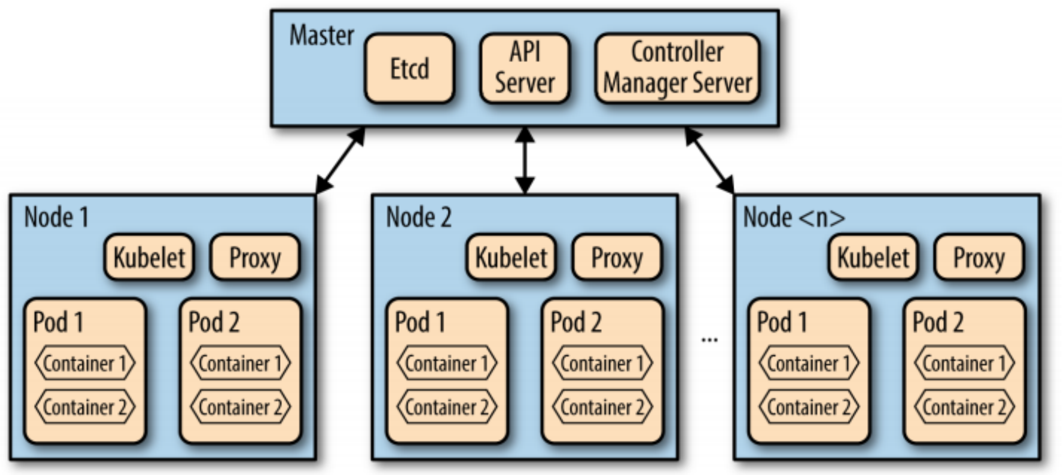


Figure 5: The Kubernetes Layout (Rensin, 2015)

According to (Rensin, 2015), K8s follows the master-slave architecture (slave also named “node”), which has been shown in Figure 5. As can be seen, master compose of three main items: API Server, Etcd and Controller Manager Server.

API Server. Almost all components on the master station and the node complete their tasks by making API calls. These are handled by the API server running on the master (Rensin, 2015).

Etcd. Etcd is a service, which aims to maintaining and replicating the current configuration and operational status of the cluster. It is implemented as a lightweight distributed key-value store and was developed in the CoreOS project (Rensin, 2015).

Controller Manager Server. The server’s task is to schedule containers on the target node. They also ensure that the correct amount of these things is always running (Rensin, 2015).

A master has charge of management of all nodes while each node must run the two important processes and some pods.

Kubelet. It is a special background process which can be used to process the tasks delivered by the master to parent node and manage the Pod and its container. The Kubelet will register the node information on the API Server, report the usage of the node resources to the Master periodically, and monitor the container and node resources through the cAdvisor (Rensin, 2015). Kubelet can be understood as an agent in the Server-Agent architecture, a pod manager on Node.

Proxy. It is a simple web proxy that separates the IP address of the target container from the name of the service it provides (Rensin, 2015).

Pods. Pods are collection of containers and volumes that are bundled and scheduled together because they share a common resource—usually a filesystem or IP address (Rensin, 2015). In a conclusion, Kubernetes is managing in pod level instead of container level.

2.5 UCAML (Universal Cloud Application Modeling language)

2.5.1 Domain Specific Language

Currently there are some Domain Specific Languages (DSL) that can describe the Elastic container platform (ECP). The result of (Quint and Kratzke, 2018) has shown that Docker Compose and Kubernetes DSL can fulfill most of requirements for ECPs. However, they are both designed for specific elastic container platform (Docker Swarm or Kubernetes). Therefore, it is necessary for them to create a new DSL, which not only can support different elastic container platforms, but also provide the maximum flexibility in covering all the mentioned and derived requirements (Quint and Kratzke, 2018). And this new DSL is called UCAML (Universal Cloud Application Modeling language).

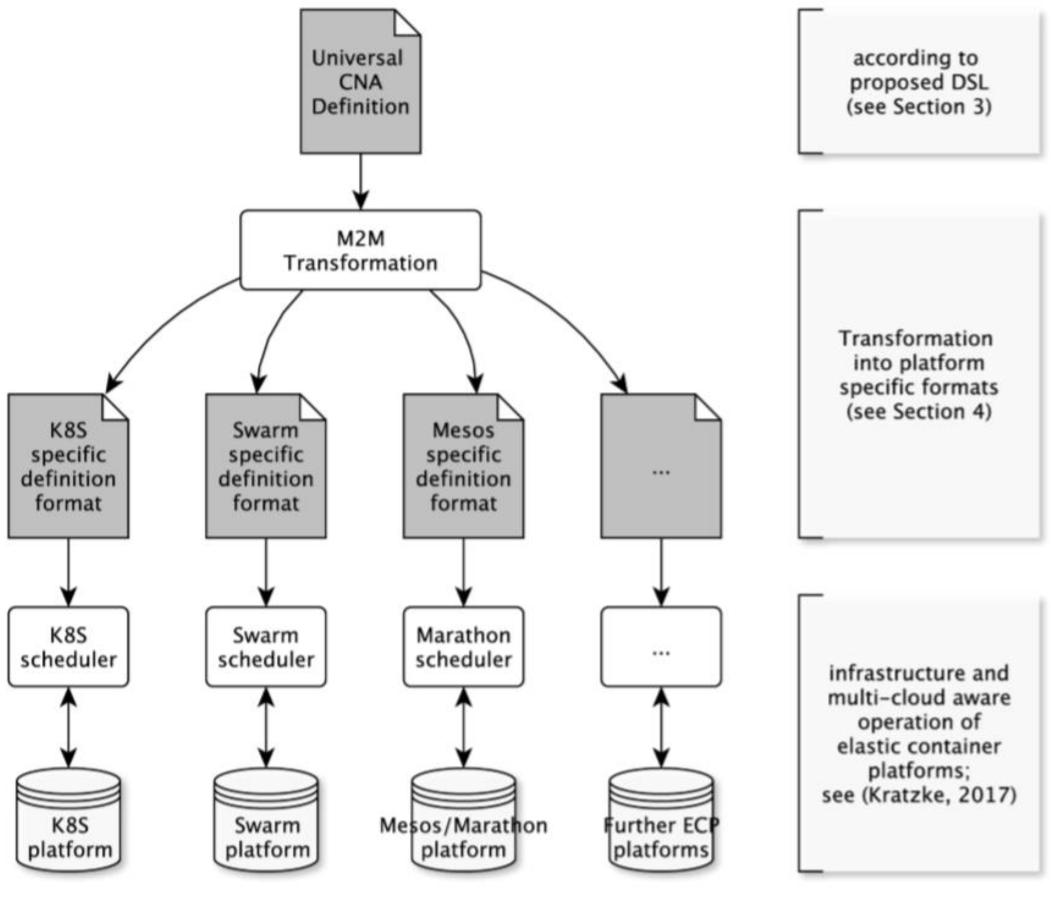

Universal CNA Definition
M2M Transformation
K8S specific definition format
Swarm specific definition format
Mesos specific definition format
...
K8S scheduler
Swarm scheduler
Marathon scheduler
...
K8S platform
Swarm platform
Mesos/Marathon platform
Further ECP platforms
according to proposed DSL (see Section 3)
Transformation into platform specific formats (see Section 4)
infrastructure and multi-cloud aware operation of elastic container platforms; see (Kratzke, 2017)

Figure 6: Deploying a CNA with M2M (Quint and Kratzke, 2018)

The UCAML is a model-to model (M2M) generator whose task is to deploy arbitrary Cloud native applications on specific elastic container platforms. The Figure 6 has shown that this universal DSL can be transformed into platform specific formats, such as Kubernetes, Swarm, Mesos, Nomad and more. Initially, this core language model is implemented as a declarative and internal DSL in Java. But to meet the demand for a representation of this core language model without the overhead of a full purpose language like java (Quint and Kratzke, 2018), the ruby is used to describe this language model and define the related features.

2.5.2 Features

Generally, the UCAML has following features:

Containerized deployments. The containers are the deployment units of a service and the UCAML is designed to describe and label a containerized deployment of discoverable services (Quint and Kratzke, 2018).

Application scaling. Elasticity and scalability both are important characteristics of cloud computing. Therefore, the UCAML can describe elastic services.

Compendiously. The UCAML intends to be much more lightweight than other DSL approaches. Its clear focus is to define an executable architecture of cloud applications without the need to consider complex technical details of platform or infrastructure specifics.

Multi-cloud-support. The UCAML can support multi-cloud operation, which meanwhile increases the using of hybrid cloud.

Independence. Using UCAML one can define an executable macro architecture of a cloud applications in a universal definition format and transform this format into container platform specific definition formats. In a word, it is independently from specific elastic container platform or cloud infrastructure.

Elastic Runtime Environment. The UCAML must define applications being able to be operated on an elastic runtime environment because the UCAML supports model-to-model transformation over different elastic containers platforms (Quint and Kratzke, 2018).

These features are described by defining some parameters in UCAML. For example, Figure 7 shows what return in a definition of application concept. Apparently, one application can have a namespace, one or more services and a set of optional parameters that can describe features such as Volumes, Scaling Rules and Scheduling Constraint. Besides that, each service also can define different parameters to describe features. But each service can only define one container.

```
return RSchema.define_hash {{
    name: pipeline(_String, predicate('name') { |n| n.match(Ucaml::NAMING_CONVENTION)}),
    services: array(Ucaml::service(:schema)),
    optional(:volumes) => array(Ucaml::volume(:schema)),
    optional(:labels) => Ucaml::labels(:schema),
    optional(:scheduling) => Ucaml::constraint(:schema),
    optional(:environment) => Ucaml::environment(:schema),
    optional(:request) => Ucaml::request(:schema),
    optional(:scale) => Ucaml::scalingrule(:schema),
    optional(:alive) => Ucaml::probe(:schema),
    optional(:ready) => Ucaml::probe(:schema),
}}
```

Figure 7: Application description

Figure 8 is an instance of UCAML file, which describe an application called "prime-service-app". Only one "prime-service" service is defined in this application. This service requests 100 Millicores CPU, 256 Megabytes memory and 2 Gigabytes ephemeral storage. Format similarly, it also defines some features like scaling rule, ports and one container. The namespace, image and CMD are included in this container.

```
Ucaml::application('prime-service-app',
  services: [
    Ucaml::service('prime-service',
      request: Ucaml::request(cpu: 100, memory: 256, ephemeral_storage: 2),
      scale: Ucaml::scalingrule(min: 1, max: 3, cpu: 66),
      ports: [80],
      expose: [80 => 80], #range depends on minikube/k8s config. Default 30001-32767
      container: Ucaml::container('prime-unit', 'transit/primesvc:latest', cmd: ["ruby", "hw-service.rb"])
    )
  ]
)
```

Figure 8: Prime-service-app instance

3 Specification

3.1 Project description

This thesis is based on the research project Cloud TRANSIT. Cloud TRANSIT is about to provide technological means manageable for Small and Medium sized enterprises (SME) who are facing special problems with cloud adoption. These solutions can avoid existential vendor lock-in situations which are still a major cloud adoption obstacle of cloud computing after 10 years (Kratzke et al., 2017). UCAML is one of these technological means provided by Cloud TRANSIT. It is a domain specific language which can determine how a CNA can be described in UCAML format and offers the possibility to test UCAML format CNA descriptions and transform them to a platform specific format.

Now we are developing a user-friendly graphical user interface for UCAML to avoid editing a text file. This graphical user interface will be implemented as a web-application, so more SME users can have easy access to it without downloading any software. We call this web-application “UCAML-GUI”. With UCAML-GUI, users can create and load UCAML file, finally transform it to target elastic container platform’s format (e.g. Docker, K8s).

3.2 Functional Requirements

3.2.1 Create and load UCAML

To deploy a CNA, specific parameters must be collected and described as UCAML file. Therefore, the UCAML-GUI must include a HTML form for UCAML transformation. In addition, the UCAML file must be also loadable and savable.

3.2.2 Avoid repetition

To avoid unnecessary effort on inputting same information, the UCAML-GUI must provide choice for users to save and reuse templates, including all parameters and services. This also results in better maintainability. For example, when the user finished the configuration of one service, he can save this service as a template and reuse it later.

3.2.3 Dynamic form

The form in the UCAML-GUI should be dynamic since the parameters in UCAML are diverse and inconstant. One CNA can define several optional features (e.g. Scheduling, request) in advance and have multiple services, each service can have fixed feature (e.g. Name) and define several optional parameters to override the same feature in CNA. Users can add parameter which is in the list to the form and delete it whenever they want. Therefore, this form should be dynamic and flexible.

3.2.4 Validation

As was stated above, since the parameters have different types and value range, the user's inputs must be limited to correct range and validated. Following (Table 1) is the list of parameters and requirements for the validation.

Parameters	**Data Type**	**mandatory**	**Other requirements**
Application name	String	×	
Volumes	array		
Volume name	String	×	
Volume type	array	×	
Volume nfs	Hash		
Volume git	Hash		
Volume secret	Hash		
Volume mountPath	String		
Labels	Hash		
Scheduling	Hash		
Environment	Hash		
Request	array		
Request cpu	Integer	×	>0
Request memory	Integer	×	>0
Request ephemeral_storage	Integer		>0
Scale	Array		
Scale min	Integer	×	>0
Scale max	Integer	×	>0
Scale targetCpu	Integer		>0 & <100
Service	Array	×	
Service name	String	×	
Service ports	Array(Integer)	×	>1 & <65535
Service capabilities	Hash		
Service expose	Array (Integer hash)		Exposed port must belong to ports
Container name	String	×	
Container image	String	×	Existing on docker-hub
Container cmd	Array(String)		

Table 1: Parameter list

As can be seen, the first column in Table 1 has shown all private and common parameters of application, service and container. The second column describe the data type of parameters and the third column shows whether they are mandatory. Some sub parameters are mandatory only if their optional parents are chosen (e.g. Request cpu is mandatory if Request is chosen). The last column lists other special requirements for corresponding parameter. In addition, every mandatory parameter cannot be empty. Therefore, there are many parameters need to be validated according different requirements.

3.2.5 Model-to-Model generation

After the data from client-side is transformed to UCAML file in server side, the UCAML file will be transformed into platform specific formats. Till now the UCAML supports Kubernetes and Docker Swarm. Thus, the CNA description must be saved as UCAML file and also exported as platform specific file.

3.3 Non-functional Requirements

3.3.1 Extendibility and reusability

Right now, UCAML supports Kubernetes and Docker Swarm. However, it is planned to support further container platform like Mesos, Nomad and so on. Accordingly, in such case only slight change of UCAML-GUI must be made. Therefore, this UCAML-GUI should be extendible and reusable to adapt new version that Mesos and Nomad are compatible with. For example, in client-side new parameters can be inserted into form and in server-side the UCAML generator is easy to accept new feature.

3.3.2 Fault tolerance

This UCAML-GUI should keep operating properly in the event of the failure of its components since the form in this website is dynamic and complex. To tolerate faults caused by users, not only validation but also exception handling mechanism can be applied to this UCAML-GUI. For instance, when the format of inputted parameter is incorrect, or server fail to generate UCAML file, helpful feedback and a ruby log-file should appear instantly.

3.3.3 Responsive design

Responsive design is currently a common requirement for website and even mobile application due to the variety of devices. Therefore, this website should also support different screen size. To ensure good operability on different screen, the contents and positions of graphical elements must be displayed dynamically.

3.4 Use case diagram

As shown in Figure 9, the actor and use cases including their relationship are visualized. In our case, the actor is UCAML-GUI user who plans to deploy cloud-native applications. The Table 2 shows the list of use cases and their descriptions.

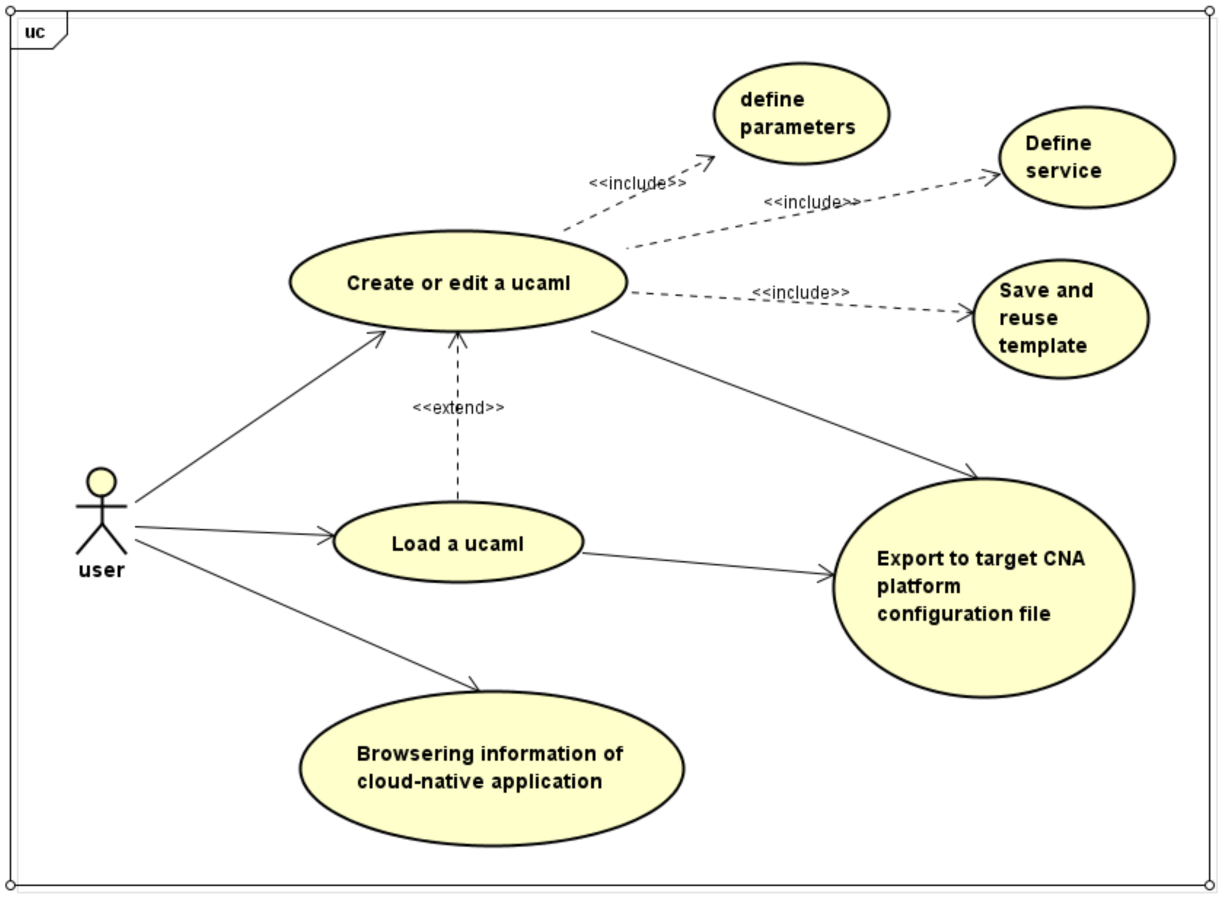


Figure 9: Use case diagram

Use Case	Description
Create or edit a CNA	A CNA can be created by filling the form. Also, user can load a UCAML file and edit it.
Load a CNA	Load a UCAML file which contains a description of a CNA from local disk. This use case can extend to “Create or edit a CNA” to update current CNA.
Define parameters	In application or service layer, users have choice to define or delete optional parameters.
Define service	The users can define one or more services in an application. They can enter to specific service page to specify every feature.
Save and reuse template	The users can save parameters and service that they want to reuse later.
Export to target CAN platform configuration file	The users can submit the form, and server will generate the UCAML file and export it to platform specific formats with M2M transformation.
Browsing information of cloud-native application	The users can learn some basic knowledge of cloud-native application by browsing this website.

Table 2: Use case specification

4 Implementation

4.1 Technology used

4.1.1 Technology

The UCAML-GUI is designed as a web-application. Accordingly, the following technologies were used:
-- HTML5, CSS and JavaScript for presenting UCAML-GUI.
-- PHP as server-side programming language.
-- ajax.js for the interaction of client-side and server-side.
-- Session for temporal storage of data.

Due to the diverse and dynamic features of CNAs, the way of storing and transferring data need to be considered carefully. The Figure 10 (parameters and datatype are not complete because the structure is too large) has shown the data structure of this project. As can be seen, it is very complicated, including the relationship and their datatype. Usually object-oriented programming and related data structure (e.g. list, map) will be used to transfer data between server-side and client-side. However, it would be more complex if they are used in this project. Because the database is not required, and the data that need to be stored is only the templates of parameters and services. In addition, the server only need to generate UCAML file after the reception of form from client-side. Hence, **JSON** is considered a better way to store and transfer data. In client-side, after the user submit the form, JSON will be generated automatically in server-side. And it can have the same

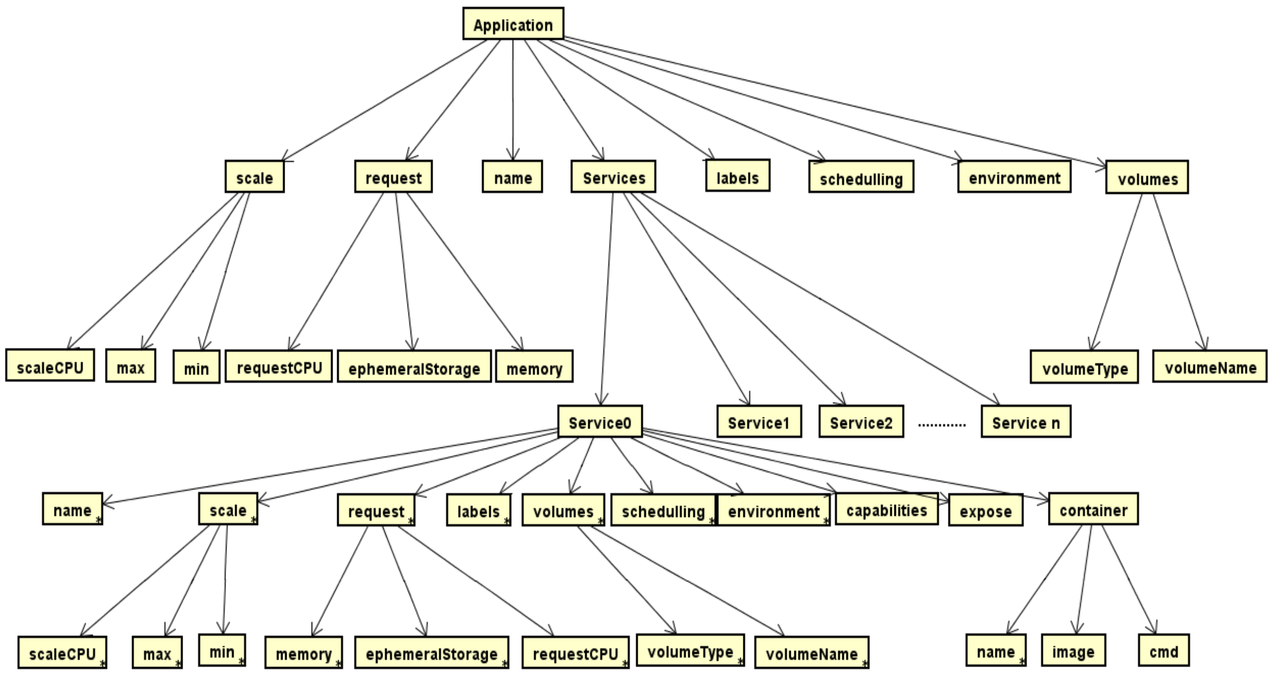


Figure 10: Data structure (Not complete)

structure as the Figure 10. But the name of input must be special format. For example, <input class="service0" name="services[0][name]" type="text">, the name means that this input is the name of first service, “services[1][request][CPU]” means that this input is the amount of CPU requested by user in first service. Therefore, JSON is an easier way to store and transfer structured data in this project.

The following programs and (web-) tools are also used to complete this project:
-- Balsamiq Cloud for designing prototype of UCAML-GUI.
-- Astah Professional for modelling with UML.
-- JetBrains PhpStorm as the IDE for programming.
-- Microsoft PowerPoint was used as form design.

4.1.2 Framework

Firstly, bootstrap[1], currently a very popular front-end framework, is used in the client-side. It contains HTML- and CSS- based design templates for forms, buttons, navigation and other interface components, which is richer and more abundant than other open-source front-end web frameworks. Furthermore, Bootstrap supports responsive web design. Its grid system and other utilities can help to design responsive web efficiently. In addition, Bootstrap is compatible with many mainstream browsers such as latest Google Chrome, Firefox, Safari, Internet Explorer and so on.

Secondly, jQuery[2], which is a fast and concise JavaScript framework. It is also an excellent JavaScript code library (or JavaScript framework) after Prototype. The purpose of jQuery design is "write Less, Do More", which advocates writing less code and doing more (Chaffer and Swedberg, 2010). It encapsulates JavaScript's common functional code, providing an easy JavaScript design pattern that optimizes following features:

Get document elements quickly
The jQuery selection mechanism is built on the CSS selector, which provides the ability to quickly query elements in the DOM document, and greatly enhances the way JavaScript gets the page elements.

1. https://getbootstrap.com/
2. https://jquery.com/

Provide dynamic effects
jQuery has a series of animation effects that can be used to create beautiful web pages. Many websites use jQuery's built-in effects, such as fade effects, element removal, and other dynamic effects.

Create AJAX no refresh page
AJAX can develop very sensitive and non-refreshing web pages (Chaffer and Swedberg, 2010). Especially when developing server-side web pages, such as PHP web sites, it is necessary to communicate with the server from and to, if AJAX is not used, each time the data update must Refresh the page. But if it is used, AJAX effect can perform a partial refresh on the page to provide dynamic effects.

Provides enhancements to the JavaScript language
jQuery provides enhancements to basic JavaScript structures such as element iteration and array handling. (Chaffer and Swedberg, 2010).

Enhanced event processing
jQuery provides a variety of page events, it can avoid the programmer to add too much event processing code in the HTML, and most importantly, its event handler eliminates a variety of browser compatibility issues. (Chaffer and Swedberg, 2010).

Change web content
jQuery can modify the content of a web page, such as changing the text of a web page, inserting or flipping a web page image. jQuery simplifies the way that JavaScript code needs to be processe

4.1.3 MVC design pattern

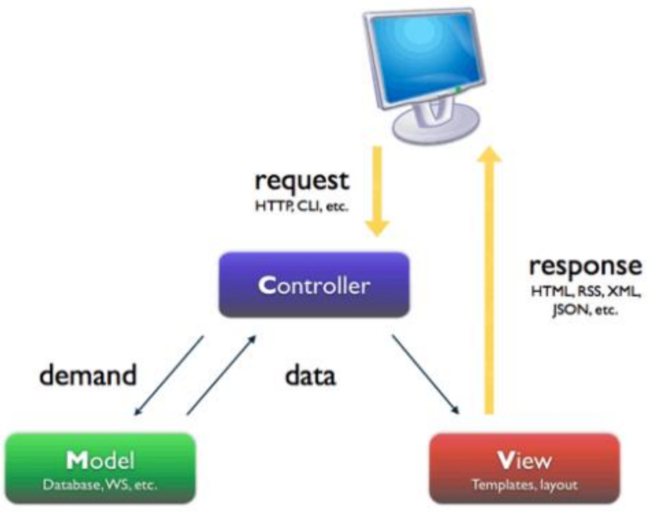


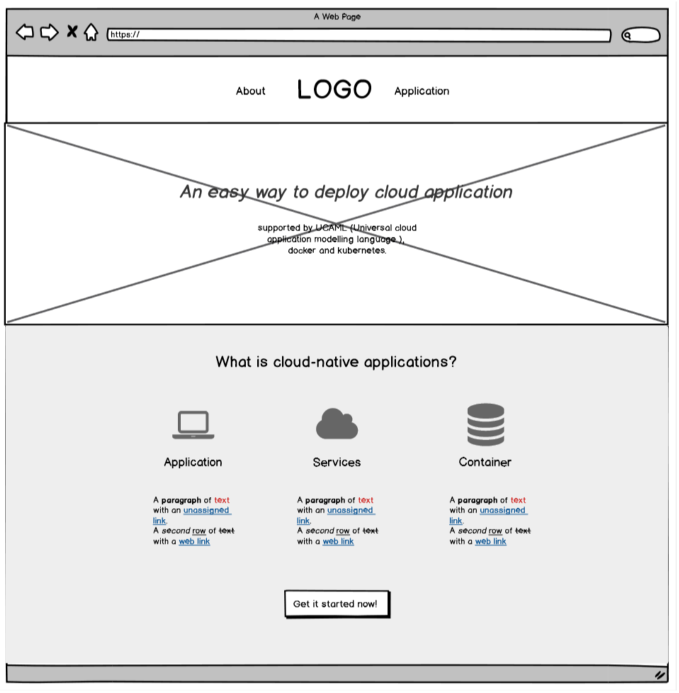
Figure 11: The MVC pattern in web application (Pop and Altar, 2014)

The Model-View-Controller (MVC) design pattern is to divide an application into three main categories: the model of the main application domain, the presentation of data in that model and user interaction (Pop and Altar, 2014). This designed pattern is used in this project because MVC is such a good fit for web application development which combine several technologies usually split into a set of layers.

In Figure 11, the Model layer is responsible for the business logic of an application and manages all tasks related to data such as session state, data source structure and so on. The view layer is responsible for graphical user interface management, which include all forms, buttons, graphical elements and all other HTML elements. By separating the model from view, developers are clearer about what their task is and the developer for the model part cannot alter the graphical interface casually. It also reduces the risk of error (Pop and Altar, 2014). The controller is responsible for event handling. A controller accepts request and prepares the data for a response. In other words, it can transfer data from model to interface and vice versa.

4.2 Prototype implementation

The prototype implementation for this website experienced two versions. In first version (as shown in Figure 12), basic design and structure of this website are determined: first page (about page) is to describe the UCAML and CNAs briefly, giving user a rough idea about they are doing. The rest pages are for the usage of CNA definition and M2M generator according to the three layers of UCAML. Apparently, this website is divided into displaying page and functional pages. In functional page, the 3D model of application is applied to interaction so that users can feel which layer they are.


A Web Page
https://
About
LOGO
Application
An easy way to deploy cloud application
supported by UCAML (Universal cloud application modelling language), docker and kubernetes.
What is cloud-native applications?
Application
Services
Container
A paragraph of text with an unassigned link. A second row of text with a web link
A paragraph of text with an unassigned link. A second row of text with a web link
A paragraph of text with an unassigned link. A second row of text with a web link
Get it started now!

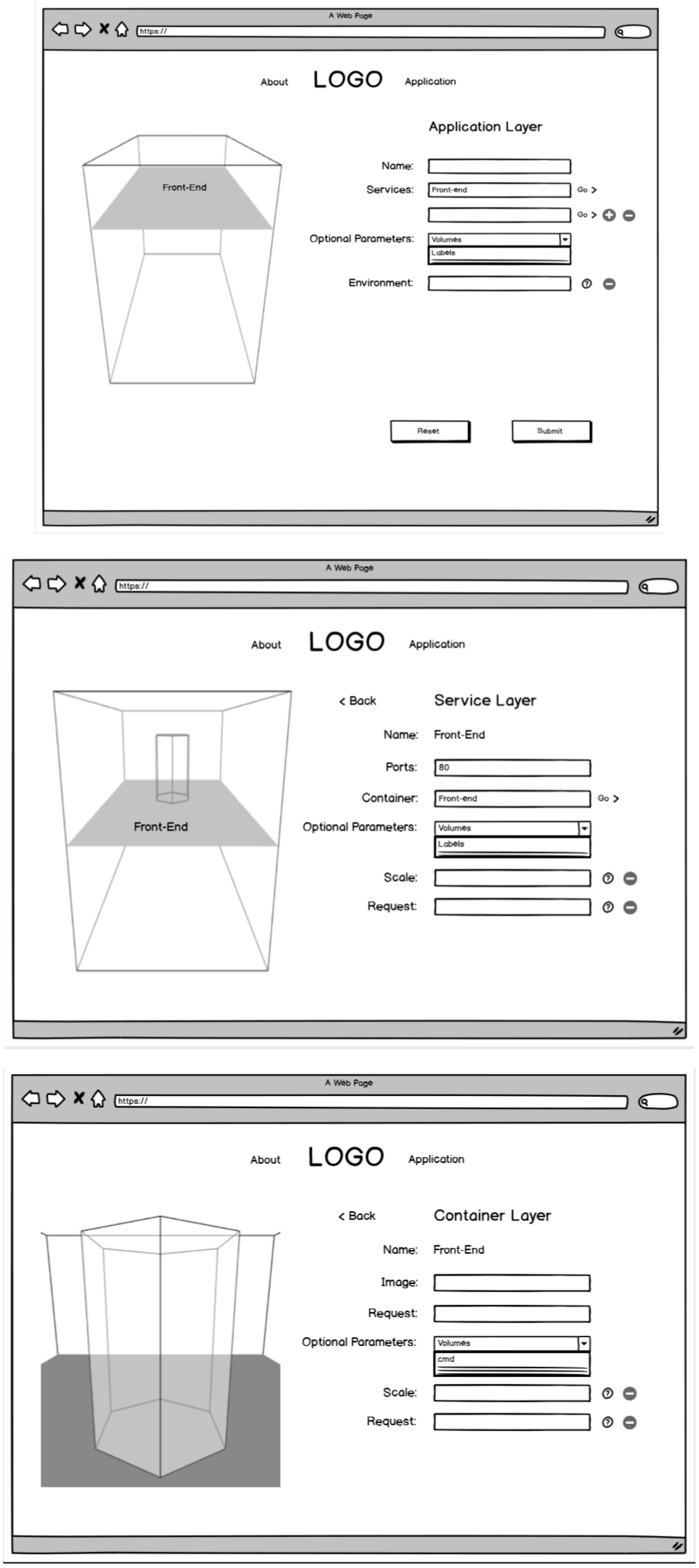
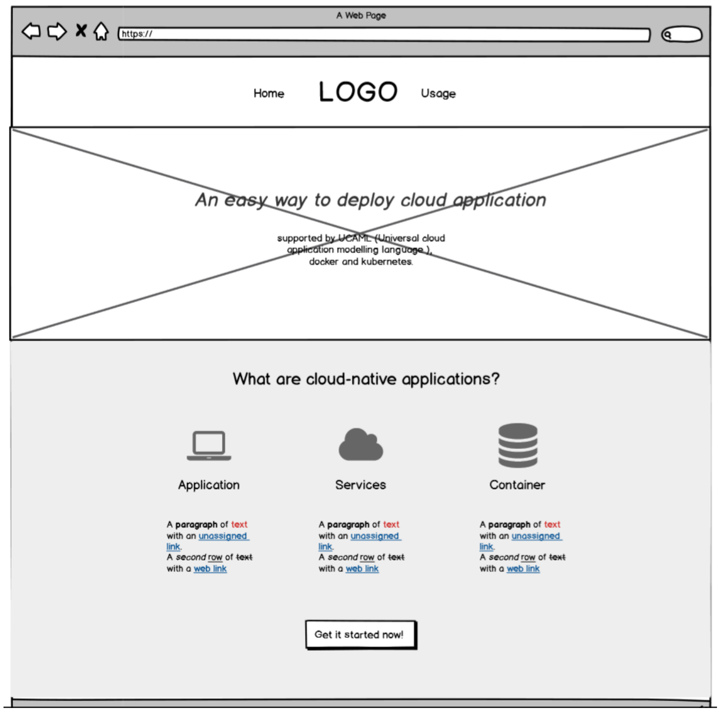
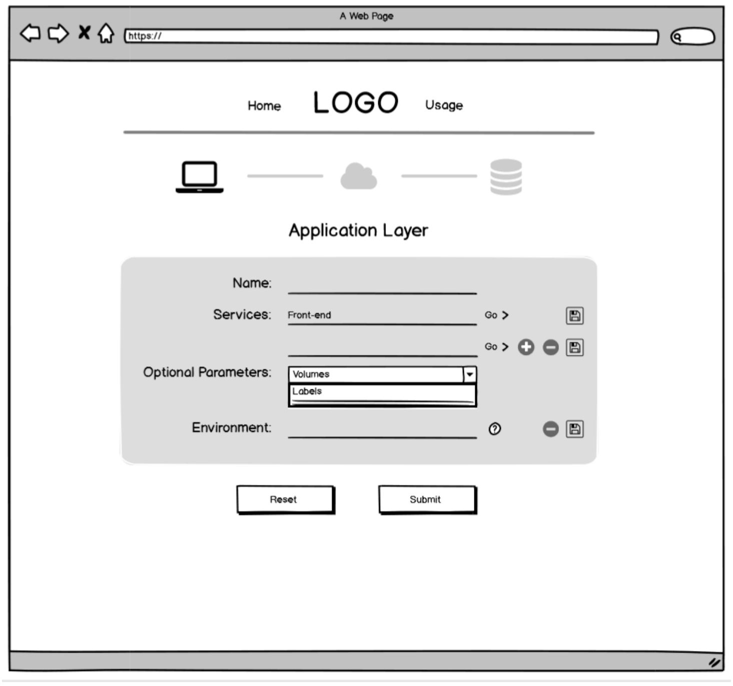
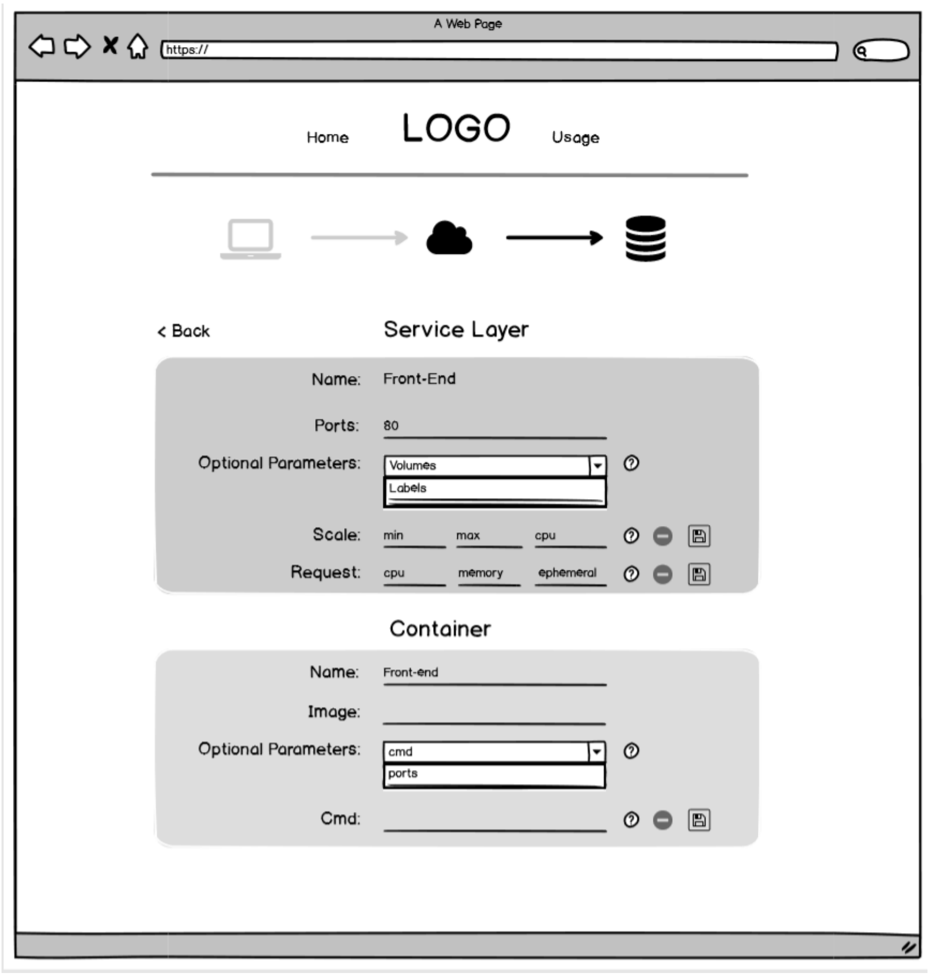


Figure 12: Prototype version 1

Following figures (Figure 13) are the second version of prototype, which removed the 3D model due to its impracticability. To make users focus more on the form, it is moved to the centre of page. Because each service has only one container, container layer is combined with service layer. In addition, three big icons for application, service and container are added to indicate users which layer the users are and show the complete process of this deployment.


A Web Page
https://
Home
LOGO
Usage
An easy way to deploy cloud application
supported by UCAML (Universal cloud application modelling language), docker and kubernetes.
What are cloud-native applications?
Application
Services
Container
A paragraph of text with an unassigned link. A second row of text with a web link
Get it started now!





Service Layer

Name: Front-End

Ports: 80

Optional Parameters: Volumes, Labels

Scale: min, max, cpu

Request: cpu, memory, ephemeral

Container

Name: Front-end

Image:

Optional Parameters: cmd, ports

Cmd:

Figure 13: Prototype version 2

4.3 Model

In Figure 14 we show the structure of model. As can be seen, in this project Session and JSON are the main technologies to store the data. When the Controller receive a request from View to store a template, an array of template will be transferred from controller to the model and store it in the sessions according to its data structure. The sessions can keep two types of templates, one is for single parameters and the another one is for service since the structure of service is more complicated than single parameters. Each parameter can have one or more templates, which contains the name and category of this template. When the controller receives the command to request a template from model, an array of corresponding template will be transferred to controller and the controller can load this template to the view. JSON is used to store and transfer the whole data of form. When user submit the form and the form is validated, then the JSON will be generated in the server-side and be transformed to UCAML file later.

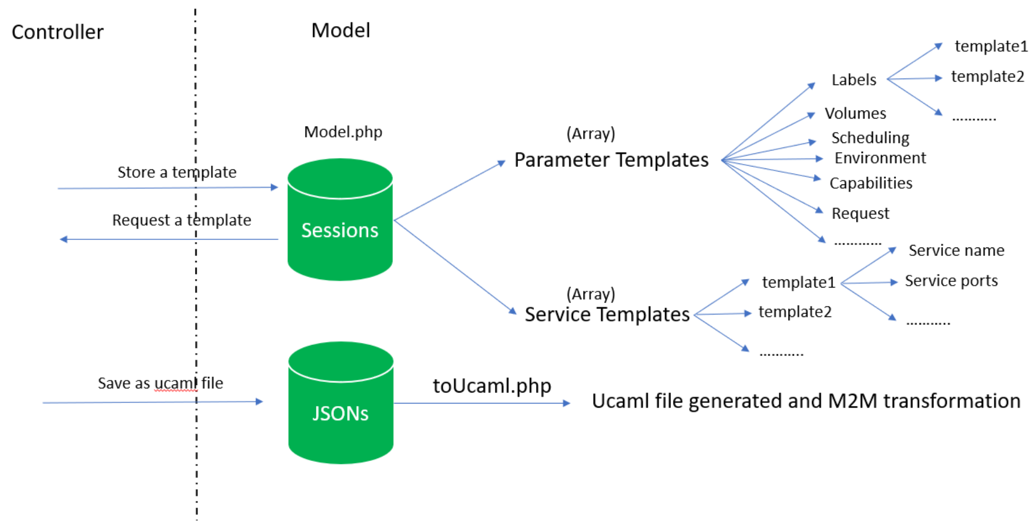


Figure 14: Structure of model

4.4 View

From the perspective of programming languages, the view is composed of three important files: ***index.html***, ***style.css*** and ***index.js***. From the perspective of design, the view is composed of navigation bar, main page and usage page. In addition, the implementation of responsive design and usability are also documented in this chapter.

4.4.1 Navigation bar

The navigation bar (Figure 15) consists of two buttons for moving to home and usage page conveniently. The picture in the centre is the logo of Cloud Transit team. This navigation bar is derived from the bootstrap's navigation bar, so it can change responsively according to the size of screen. In addition, animation is added. When user scroll down the page, the navigation bar will get short smoothly and a black line will appear to separate the navigation bar and pages (Figure 16).

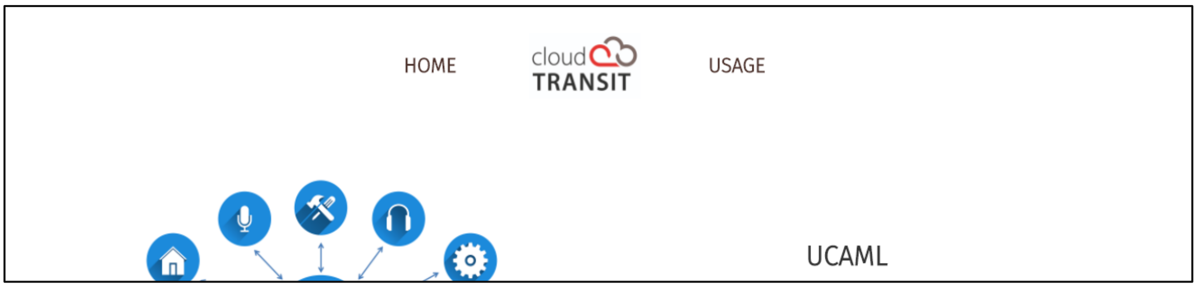
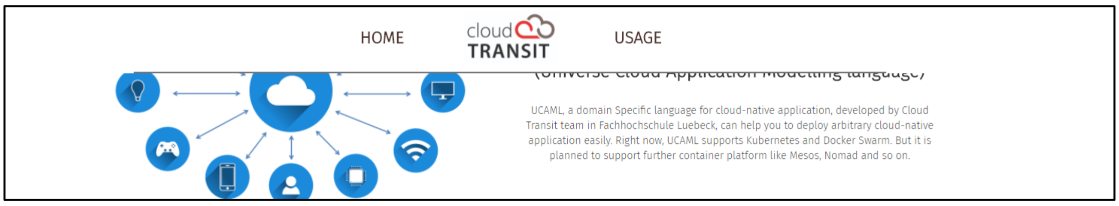


Figure 15: Original navigation bar



UCAML, a domain Specific language for cloud-native application, developed by Cloud Transit team in Fachhochschule Luebeck, can help you to deploy arbitrary cloud-native application easily. Right now, UCAML supports Kubernetes and Docker Swarm. But it is planned to support further container platform like Mesos, Nomad and so on.

Figure 16: Navigation bar after scroll down

4.4.2 Main page

The main page firstly provides users the introduction of UCAML. The introduction combines the text and pictures so that users can fully understand the description. Secondly, it presents some concepts of CNAs, including applications, services and containers. These concepts are related to the three layers of the deployment. Therefore, it would be easier for users to input the following form after browsing the main page. In addition, the main page used the grid system of Bootstrap to place the HTML elements. Thus, this page is complete responsive as the size of screen changes.

4.4.3 Usage page

The usage page provides a form for users to create a UCAML file so that server-side can deploy CNAs according to this UCAML. The form is divided to some sub tables to meet the requirement that one application can deploy one or more services. Therefore, as shown in Figure 17, there is a main table for application layer and one or more sub tables for each service. A “select” input element is added in this form so that users can choose optional parameters. Because most parameters’ data type is array and the lengths are inconsistent, the minus icon, plus icon and delete icon are used to add, delete new items for parameter. The tag icon can open the template window where uses can save or reuse a template for current parameter. In addition, the “go” icon can help user enter to the service layer.

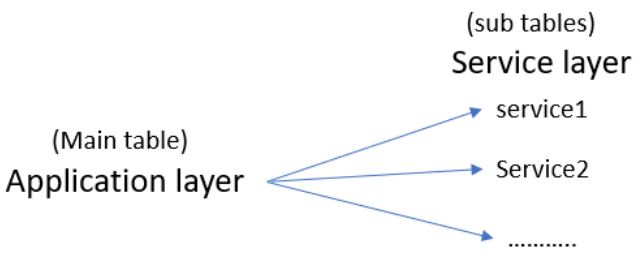
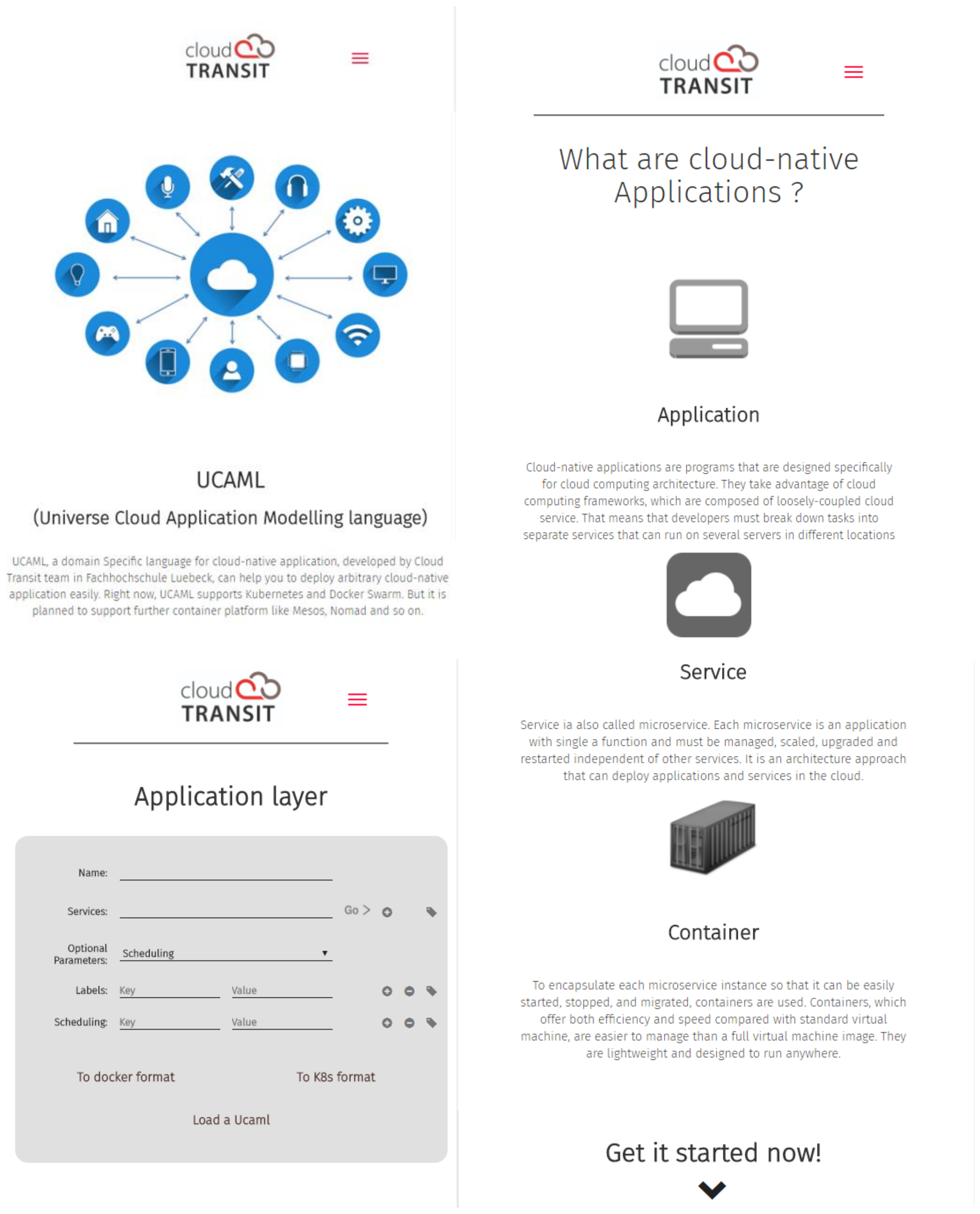


Figure 17: The structure of form

4.4.4 Responsive design

To make this website responsive to the screen's size, three main technologies are used. Firstly, the bootstrap's navigation component helps to change the style of navigation bar while the screen's size is changing. The items in navigation bar are hidden and user can click the toggle to uncover them. Secondly, the grid system in bootstrap helps to place specific elements properly according to the screen's size. Thirdly, the @media property also can make different style rules to adapt the screen's size. Following is the appearance of this website in small screen (e.g. mobile phone).



UCAML

(Universe Cloud Application Modelling language)

UCAML, a domain Specific language for cloud-native application, developed by Cloud Transit team in Fachhochschule Luebeck, can help you to deploy arbitrary cloud-native application easily. Right now, UCAML supports Kubernetes and Docker Swarm. But it is planned to support further container platform like Mesos, Nomad and so on.

Application layer

Name:

Services: Go >

Optional Parameters: Scheduling

Labels: Key Value

Scheduling: Key Value

To docker format

To K8s format

Load a Ucaml

What are cloud-native Applications ?

Application

Cloud-native applications are programs that are designed specifically for cloud computing architecture. They take advantage of cloud computing frameworks, which are composed of loosely-coupled cloud service. That means that developers must break down tasks into separate services that can run on several servers in different locations

Service

Service ia also called microservice. Each microservice is an application with single a function and must be managed, scaled, upgraded and restarted independent of other services. It is an architecture approach that can deploy applications and services in the cloud.

Container

To encapsulate each microservice instance so that it can be easily started, stopped, and migrated, containers are used. Containers, which offer both efficiency and speed compared with standard virtual machine, are easier to manage than a full virtual machine image. They are lightweight and designed to run anywhere.

Get it started now!

Figure 18: Responsive design in small screen

4.4.5 Usability in the view

Match between system and the real world
In this website, there are buttons with icons to show the functions more vividly. For example, as shown in Figure 19, the icons for application, service and container are from real stuff or something related to.

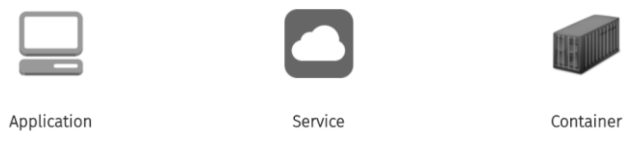
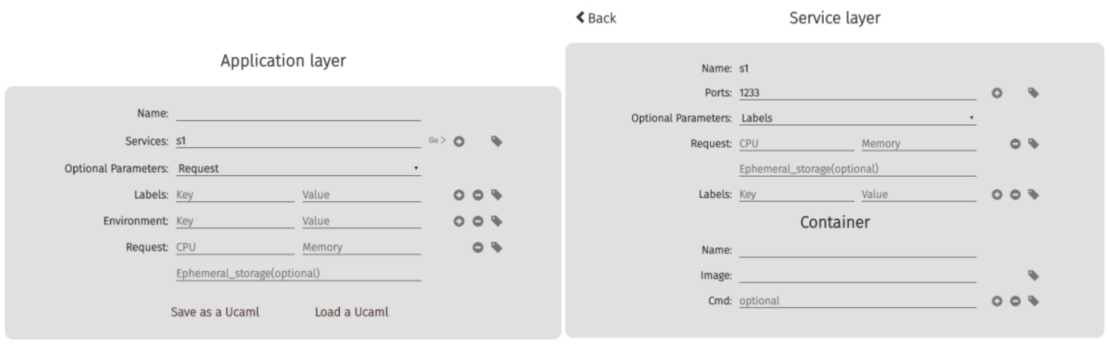


Figure 19: Match between system and the real world

Consistency and standards
In Figure 20, it shows that the layout of application layer and service layer are almost the same. Therefore, users can easily find the specific parameters and icons they want to operate no matter which layer they are.


Application layer

Name:
Services: s1
Optional Parameters: Request
Labels: Key Value
Environment: Key Value
Request: CPU Memory
Ephemeral_storage(optional)
Save as a Ucaml Load a Ucaml

Back
Service layer

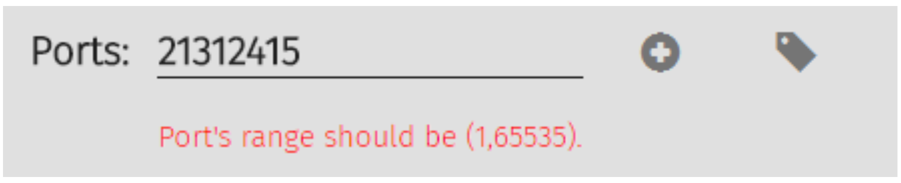
Name: s1
Ports: 1233
Optional Parameters: Labels
Request: CPU Memory
Ephemeral_storage(optional)
Labels: Key Value

Container

Name:
Image:
Cmd: optional

Figure 20: Consistency and standards

Help users recognize, diagnose, and recover from errors
If the user input some illegal characters, error message dialogue will be invoked and show the user where is wrong and how to solve it. For example, in Figure 21 the port that user inputted exceed the legal range, when users leave this text field the error dialogue will show that it should be an integer and the range should be from 1 to 65535. In addition, the colour of font will be red to make it more conspicuous.


Ports: 21312415

Port's range should be (1,65535).

Figure 21: Error detection and recover

4.5 Controller

4.5.1 Structure of Controller

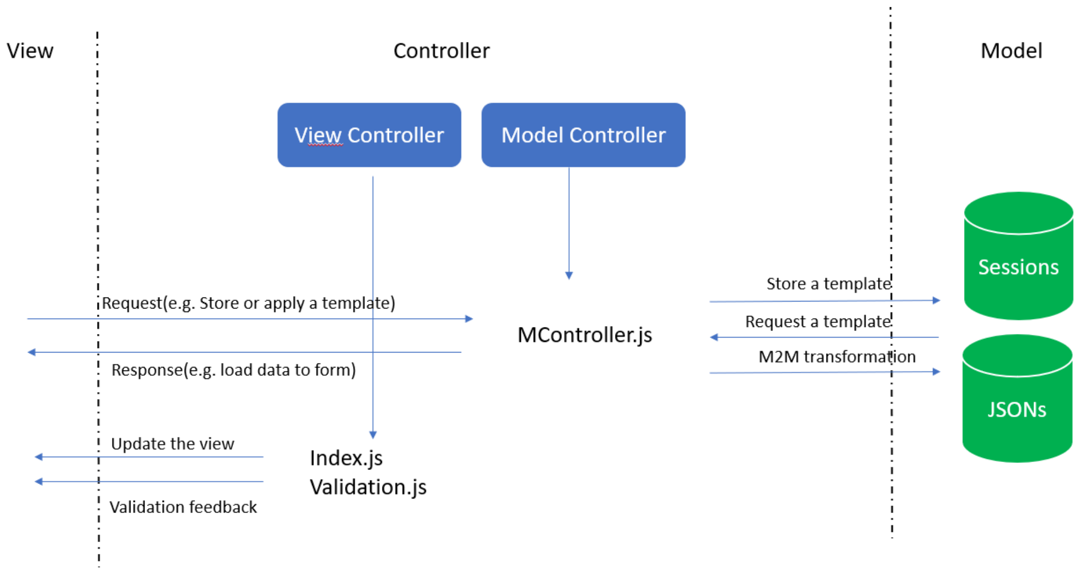


Figure 22: Structure of Controller

The figure 22 has shown the structure of controller, including its interaction with model and view. Controller is divided into two parts. The first part is view controller, which consists of index.js and validation.js. Index.js mainly control the update of view according the user's request. For example, the user needs to add an optional parameter or load a UCAML file to the form. Validation.js is to validate the format of parameters input by users and give feedback instantly. View controller has not permission to access to model. Therefore, the task of interaction with model layer is distributed to Model controller, which has only one MController.js file. The MController.js is to accept some requests that referred to model layer from view and response to view by updating the view. For example, one user wants to use a template, after controller accepts this request, it would get the template list from model and load it to the view.

4.5.2 Validation in the Controller

The validation of form is completed by validation.js. It has three important functions: ***checkEmpty ()***, ***checkInteger ()*** and ***validateCurrentTable ()***. ***checkEmpty ()*** is to check whether some necessary parameters are empty. ***checkInteger ()*** is to check whether parameter like ports, memory is integer. But there are some deeper validations for special parameter such as target CPU (>0 & <100) inside this function. The ***validateCurrentTable()*** function is to validate the whole service before the user leave service layer. Users are not allowed to leave service layer except this function return true.

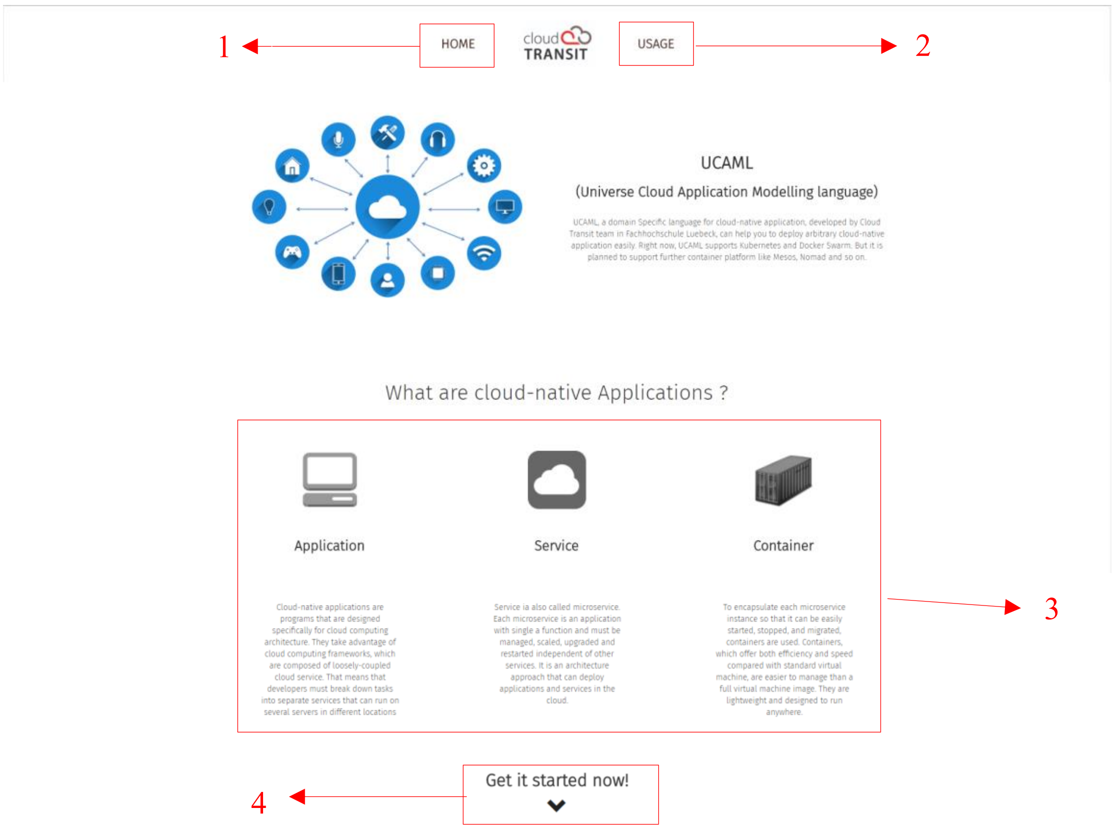
4.6 Model-to-Model transformation

The M2M transformation is mainly implemented by ***toUcaml.php***. ***toUcaml()*** function will generate the UCAML file from the form and then a DOS command which can transform UCAML to Kubernetes or Docker format is invoked by using PHP function ***exec()***. The final file would be downloaded in client-side automatically. Which format the UCAML will be transformed to is decided by users.

4.7 Instructions of this web application

4.7.1 Main page

The main page is the first page that users can see when open this website. It is mainly about introduction of UCAML and CNAs. See more in Chapter 4.4.1 and 4.4.2.



UCAML

(Universe Cloud Application Modelling language)

UCAML, a domain Specific language for cloud-native application, developed by Cloud Transit team in Fachhochschule Luebeck, can help you to deploy arbitrary cloud-native application easily. Right now, UCAML supports Kubernetes and Docker Swarm. But it is planned to support further container platform like Mesos, Nomad and so on.

What are cloud-native Applications ?

Application

Cloud-native applications are programs that are designed specifically for cloud computing architecture. They take advantage of cloud computing frameworks, which are composed of loosely-coupled cloud service. That means that developers must break down tasks into separate services that can run on several servers in different locations

Service

Service ia also called microservice. Each microservice is an application with single a function and must be managed, scaled, upgraded and restarted independent of other services. It is an architecture approach that can deploy applications and services in the cloud.

Container

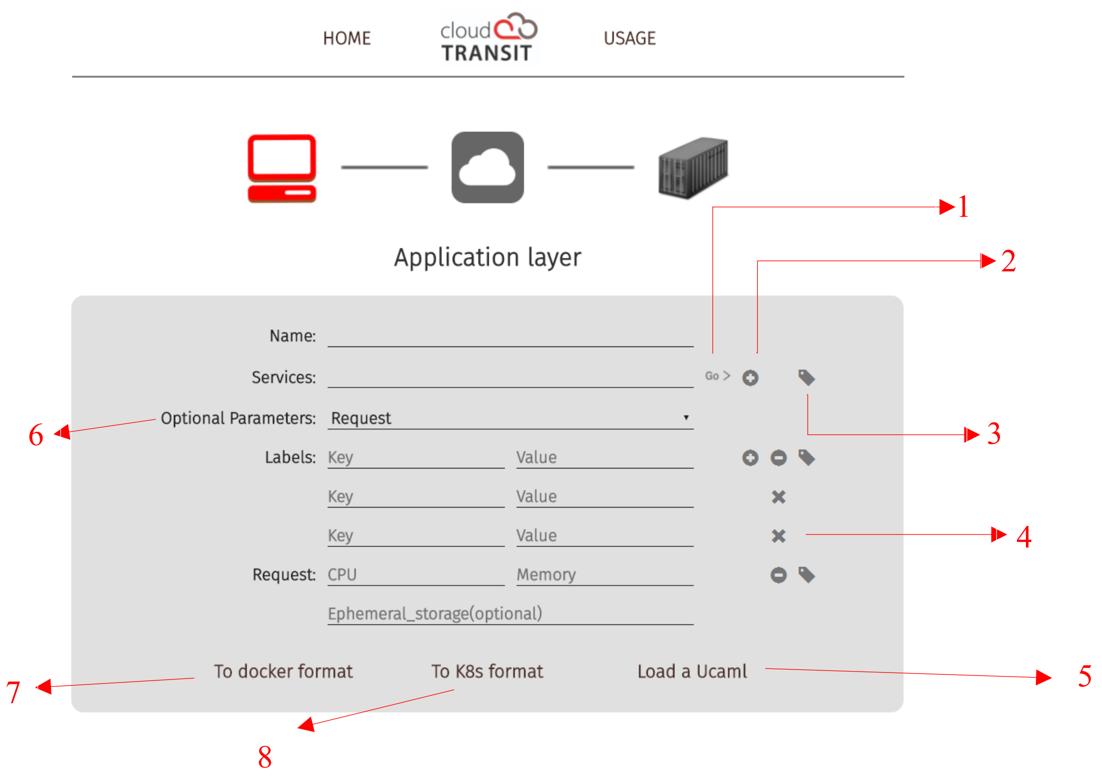
To encapsulate each microservice instance so that it can be easily started, stopped, and migrated, containers are used. Containers, which offer both efficiency and speed compared with standard virtual machine, are easier to manage than a full virtual machine image. They are lightweight and designed to run anywhere.

Figure 23: Main page

(1) Main page button, can move to main page quickly when clicked.
(2) Usage page button, can move to usage page quickly when clicked.
(3) Introduction of CAN.
(4) User can click it and move to usage page automatically.

4.7.2 Usage page

This page is intentionally designed for description of CNAs. Users can define their CNAs then a UCAML file will be generated in server-side and users can export it to a target platform format. More information can be found in chapter 4.4.3.



Name:

Services:

Optional Parameters: Request

Labels:	Key	Value
	Key	Value
	Key	Value
Request:	CPU	Memory
	Ephemeral_storage(optional)	

Figure 24: Usage page (application layer)

(1) Users can click "go" icon to move to service layer after the service name is defined.
(2) Users can click it to add corresponding parameters.
(3) Users can click it to open the template window where they can save or use a template.
(4) Users can click it to delete an item of corresponding parameter.
(5) Users can click it to load a UCAML from their local disk.
(6) Users can click it to select optional parameters that they want to define.
(7) Users can click it to save this form as UCAML and transform this UCAML file to Docker format.
(8) Users can click it to save this form as UCAML and transform this UCAML file to Kubernetes format.

4.7.3 Template window

Users can click the tag icon to open corresponding parameters' template window (Figure 25). They can save or reuse a template in this window.

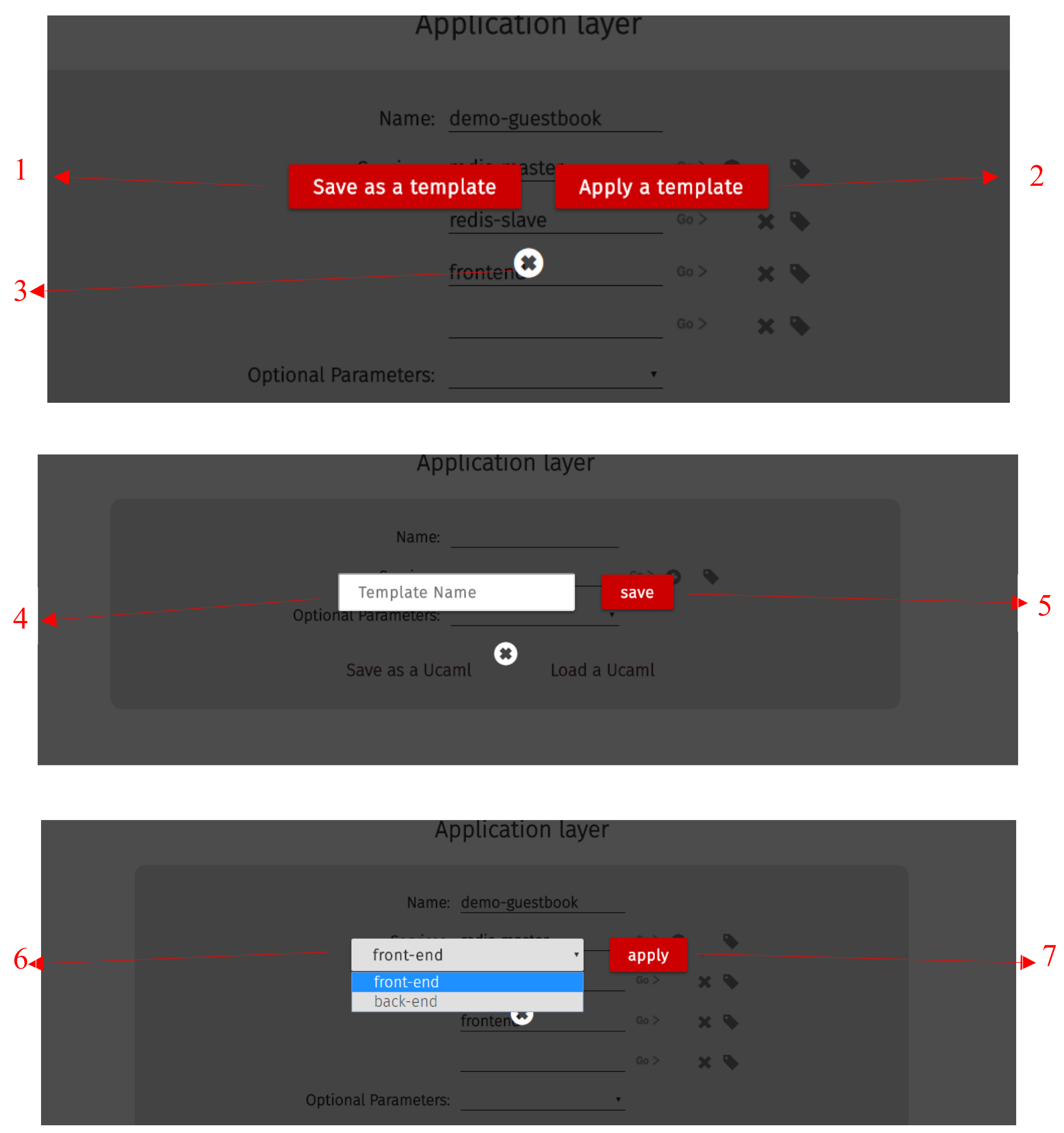


Figure 25: Template window

(1) Users can click it to save current parameter as a template.
(2) Users can click it to use a template.
(3) Users can click it to close the template window.
(4) Users can input the template name for the parameter which they want reuse.
(5) Users can click it to save this template.
(6) Users can select a template which they want to use.
(7) Users can click it to use this template and the data of this template will be loaded in the form.

5 Test and Evaluation

5.1 Boundary Testing

Boundary testing is the process of testing between extreme ends or boundaries between partitions of the input values. As shown in Figure 26, the basic idea in boundary value testing is to select input variable values at their Minimum, just above the minimum, a nominal value, just below the maximum and maximum. In Boundary Testing, Equivalence Class Partitioning plays a good role. Equivalent Class Partitioning is a black box technique (code is not visible to tester) which can be applied to all levels of testing like unit, integration, system, etc. In this technique, you divide the set of test condition into a partition that can be considered the same (Guru99, 2018).

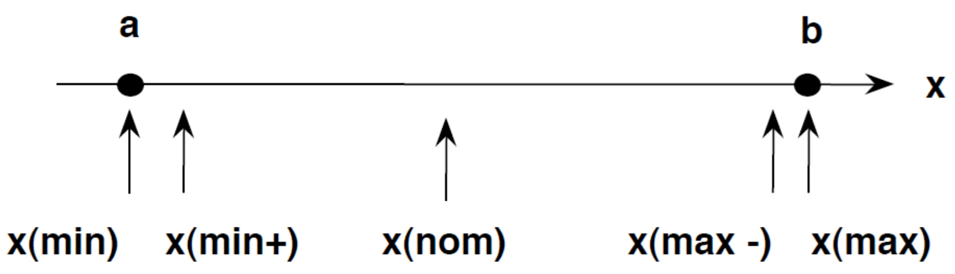


Figure 26: Test cases in boundary testing

Many errors usually occur at the input or output boundary. Therefore, designing test cases for various boundary conditions is necessary to detect more errors. A small boundary test has been done in this project to check some special parameters' boundary. As shown in Figure 27, the tested input is Target CPU, the valid Equivalence Class (EC) is from 0 to 100 and Invalid EC has also been shown. Many test cases have been tested and the result showed that the practical input boundary is completely correct.

Target CPU: integer

Valid EC:
Target CPU1 = {value | 0 < value < 100}

Invalid EC:
Target CPU2 = {value | value >=100}
Target CPU3 = {value | value <= 0}

Test cases for: -2147483649, -1, 0, 45, 100, 101, 2147483649, one, z, 2.2

Figure 27: Boundary testing example

5.2 Usability test

To test the usability of this project, a usability test has been done. Four testers who are majoring in Information Technology participated this test. They need to finish 3 given tasks by using UCAML-GUI without any instruction. These tasks are sequentially creating a UCAML and M2M transformation, load a UCAML, save and use a template. The result of this test has been shown in Table 3.

Item to test	Tester information	Task Completed Yes/No	Comments
Create a UCAML and M2M transformation	ZAHNG Xiaoyue FHL students majoring in Information Technology	Yes	He was confusing about the scaling rule and request due to the missing of units.
	DAI Wei FHL students majoring in Information Technology	Yes	Well done.
	LI Fangtian FHL students majoring in Information Technology	Yes	Tester suggest that in application layer the error warning for validation should be placed on the bottom of form so that it is more obvious since it's closer to submit button.
	HU Bocheng FHL students majoring in Information Technology	Yes	Well done.
Load a UCAML	ZAHNG Xiaoyue FHL students majoring in Information Technology	Yes	The last scale in application layer is not cleared in current loading.

	DAI Wei FHL students majoring in Information Technology	Yes	The optional parameters in application layer were not loaded into form.
	LI Fangtian FHL students majoring in Information Technology	Yes	Well done.
	HU Bocheng FHL students majoring in Information Technology	Yes	Some labels are missed after loading.
Save and use a template	ZAHNG Xiaoyue FHL students majoring in Information Technology	Yes	Well done.
	DAI Wei FHL students majoring in Information Technology	Yes	Well done.
	LI Fangtian FHL students majoring in Information Technology	Yes	Well done.
	HU Bocheng FHL students majoring in Information Technology	Yes	Well done.

Table 3: Usability test

As shown in table 3, every tester can finish task “Save and reuse template” without any obstacle or confusion. But in other tasks, they found some problems and gave a feedback. After this usability test, some improvements have been done:

1. Units and hints are added in the placeholders of “request” and “scale” parameters.
2. In application layer, the error warning for validation will appear on the bottom of form.
3. Fixed the bug that form is not completely cleared when loading a UCAML.
4. Fixed the bug that optional parameters are not loaded into form when loading a UCAML.
5. Fixed the bug that some labels are missing when loading a UCAML.

Even though most testers are not familiar with cloud computing and CNAs, this usability test is still very helpful to find some bugs and several unusable places.

5.3 Evaluation

In table 4, the evaluations for each requirement are listed. As can be seen, some requirements are fulfilled well while others are not satisfied. In general, the UCAML-GUI can complete the basic functions and tasks successfully.

Requirements	Result (Not finished/Moderate/Good)	Evaluation
Create and load UCAML (see chapter 3.2.1)	Good	The users can create their own UCAML file easily but can only load the JSON of UCAML file from local disk to form.
Avoid repetition (see chapter 3.2.2)	Moderate	The users can save parameters as a template and select one template from session to reuse. They can create but can’t update a template in the template list.
Dynamic form (see chapter 3.2.3)	Good	This form meets the requirement of dynamic form. The optional parameters can be added, edited and deleted in this form whenever necessary.
Validation (see chapter 3.2.4)	Moderate	Most requirements for validation are fulfilled except “expose” and “image”.
M2M generation (see chapter 3.2.5)	Good	The UCAML file can be transformed to platform specific formats successfully.

Extendibility and reusability (see chapter 3.3.1)	Moderate	This project doesn't strictly follow the MVC pattern, so it is not very extendible and reusable.
Fault tolerance (see chapter 3.3.2)	Moderate	This UCAML-GUI can tolerate user's fault in some aspect like validation. But some exception handling mechanisms are still missing.
Responsive design (see chapter 3.3.3)	Moderate	All pages of UCAML-GUI are responsively designed except template page and load UCAML window.

Table 4: Evaluation

6 Conclusion and future work

This thesis has shown firstly the relevant background that refers from general Cloud Computing to specific Elastic Container platforms and the UCAML which developed in Cloud TRANSIT project, secondly the complete developing process for the graphical user interface that describe CNAs for multi-cloud deployments. In general, it was complex task to create such an inputting system for defining UCAML due to the data structural complexity and different properties (e.g. some parameters are mandatory while some are optional). Furthermore, the data must experience multiple transformation procedures to achieve final M2M transformation: from JSON to html form, to JSON again and to UCAML, finally finish the M2M transformation.

The UCAML-GUI is implemented as a web-application. Therefore, users have easier access to it and there are adequate frameworks and libraries to utilize in developing phase. The MVC design pattern also play a significant role for providing a clear structure so that the code is easier to maintain and reuse. But if we take much more time to developing phase, some defects or imperfection can be corrected or improved. For example, the users cannot update a template in list, the template window and load UCAML window are not designed responsively and more exception handling mechanisms are required to tolerate faults caused by users or systems. Furthermore, the validations for "expose" and "image" are still missing. In addition, we still see the demand for more strictly following MVC design pattern to enhance the reusability and extendibility. However, this UCAML-GUI can fulfil most of requirements for describing multi-deployments for CNAs, including creating UCAML files and loading UCAML json, template system for avoiding repetition, validation and exporting platform specific configuration files. Since currently the UCAML only support Docker and K8s, it remains to be seen whether this UCAML-GUI can be extended to adapt further UCAML after more Cloud platforms are compatible with.

7 Acknowledgments

At the point of finishing this paper, I'd like to express my sincere thanks to all those who have helped me in the process of my writing this paper. This thesis is based on the Cloud TRANSIT project and guided by my supervisor Peter-Christian Quint. Mr. Quint is a very responsible supervisor who provide great help for me, such as instructions on implementing UCAML-GUI, very thorough feedback for my thesis and so forth. Furthermore, the regular tutorials on writing thesis which offered by Lenka Hanesova are also very helpful. Without their help, it would be harder for me to finish my project and this paper. Thanks to all of you!

8 Bibliography

[1] Kratzke, N. and Quint, P. C. (2017). Understanding cloud-native applications after 10 years of cloud computing -A systematic mapping study. *Journal of Systems and Software*, 126(January):1–16.

[2] Gannon, D., Barga, R., & Sundaresan, N. (2017). Cloud-Native Applications. *IEEE Cloud Computing*, *4*(5), 16-21. [Online].

[3] https://www.docker.com/what-container, 02/04/2018.

[4] Mell, P., & Grance, T. (2011). The NIST definition of cloud computing. 50-58. [online]

[5] Thönes, J. (2015). Microservices. IEEE Software, 32(1), 116-116. [online]

[6] Bob Tarzey, (2016). https://www.computaerweekly.com/feature/What-are-containers-and-microservices, 01/05/2018

[7] Vincent Batts, (2014). https://opensource.com/business/14/7/guide-docker 01/05/2018

[8] Rensin, D. K. (2015). Kubernetes-Scheduling the Future at Cloud Scale.

[9] Quint, P. C., & Kratzke, N. (2018). Towards a Lightweight Multi-Cloud DSL for Elastic and Transferable Cloud-native Applications. arXiv preprint arXiv:1802.03562.

[10] Buzachis Aris, https://blog.buzachis-aris.com/2014/12 /microservices-vs-monolithic-architectures/ 01/06/2018

[11] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, omega, and kubernetes. Queue, 14(1), 10.

[12] Chaffer, J., & Swedberg, K. (2010). Jquery reference guide: a comprehensive exploration of the popular javascript library. Packt Publishing Ltd.

[13] Pop, D. P., & Altar, A. (2014). Designing an MVC model for rapid web application development. Procedia Engineering, 69, 1172-1179.

[14] Kratzke, N., Quint, P. C., Palme, D., & Reimers, D. (2017). Project Cloud TRANSIT-Or to Simplify Cloud-native Application Provisioning for SMEs by Integrating Already Available Container Technologies. European Project Space on Smart Systems, Big Data, Future Internet-Towards Serving the Grand Societal Challenges. SCITEPRESS.

[15] Guru99, https://www.guru99.com/equivalence-partitioning-boundary-va lue-analysis.html 10/06/2018

9. Appendix A - List of figures

10. Appendix B - List of tables