



universität
wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„N2SkyC - Cloud Container-based Problem Solving Environment“

verfasst von / submitted by

Aliaksandr Adamenko

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2018 / Vienna, 2018

Studienkennzahl lt. Studienblatt /
degree programme code as it appears on
the student record sheet:

A 066 926

Studienrichtung lt. Studienblatt /
degree programme as it appears on
the student record sheet:

Informatik - Masterstudium Wirtschaftsinformatik

Betreut von / Supervisor:

Univ.-Prof. Dipl.-Ing. Dr. Erich Schikuta

Declaration of Authorship

I hereby declare that this master thesis is my unaided work. It is being submitted for the degree of Dipl.-Ing. in Business Informatics, at the University of Vienna. It has not been submitted before for any other degree or examination at any other university.

Vienna, March 2018

Signature:

ALIAKSANDR ADAMENKO

Acknowledgments

The project's architect is Aliaksandr Adamenko. My special thanks go to my scientific advisor Univ.-Prof. Dipl.-Ing. Dr. Erich Schikuta (Scientific Advisor) and to Andrii Fedorenko (Front-end Architect) for their assistance. I would also like to thank my wife for supporting me during the years of study.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Terms and Definitions	3
1.3	Related Work	4
2	The N2SkyC Architecture	8
2.1	Current Architecture Analysis	8
2.1.1	General Scope	8
2.2	Redesign Motivation	11
2.2.1	Scaling and Architecture	11
2.2.2	Microservices Architecture	12
2.2.3	Microservices Infrastructure	13
2.2.4	Component Decomposition Guidelines	16
3	Technological Stack	20
3.1	Cloud Infrastructure - OpenStack	21
3.2	Virtualization Technologies	25
3.2.1	Traditional Virtualization	26
3.2.2	Container-based Virtualization	27
3.2.3	Docker Platform	30
3.3	Orchestration Tool	36
3.3.1	Cloudify Platform	39
4	Components and Topology Description	41
4.1	Business Module	42
4.2	Monitoring Module	43
4.3	Infrastructure Module	44
4.4	Computational Module	46
4.5	UI Module	48

5	Development Guide	50
5.1	Cloud Infrastructure Guide	50
5.1.1	Environment Setup	50
5.1.2	DevStack Manual Installation	50
5.1.3	DevStack - Virtual Environment	54
5.2	Orchestration Management Guide	58
5.3	Neural Network Development Guide	61
6	User Guide	70
7	Conclusions and Outlook	75
7.1	Future Work	76
A	Abstract English	85
B	Abstract German	86

1 Introduction

1.1 Motivation

Neural networks and deep learning era approaches - these technologies became more and more popular in the last years. There is a clear trend that these domains will continue to grow. We observe that the theoretical and practical success and quality of the existing solutions are often highly connected with open-source community projects more than with commercial sphere despite the fact that many IT-related companies are fostering machine learning products. We believed that shared collaborative nature of the open-source communities is more than relevant for the future scientific and practical development of the machine learning and neural networks and that these concepts are reflected in the N2Sky platform. The N2sky system was initially developed by Erwin Mann and Erich Schikuta to provision neural network problem solving virtual organization where any member of the community can access or contribute neural network objects all over the Internet [25]. The N2Sky virtual organization administrates these neural network objects, provides access for users and fosters participating resource nodes. The number of neural network objects is supposed to be very large and continuously increasing. These neural network objects are expected to be shared on the Internet and provide easy access to the different neural network and machine learning topologies. Searching for specific neural network resources providing solutions to given problems can be a time consuming and difficult task. Therefore N2Sky should be designed to be fully responsive and easy to use and develop.

Problem Statement. Technically speaking, N2Sky is an artificial neural network simulator, which primary purpose is to provide different stakeholders with access to robust and efficient computing resource [25]. It was designed to provide natural support for Cloud deployment with distributed computational resources. However, the current N2Sky implementation is based on the Java programming language and deployed as a single monolithic application, which is not well aligned with the chosen paradigm. Monolithic architecture is a typical design pattern, and it's the excellent choice for the initial project architecture. It provides several guarantees on connectivity between modules, can speed up the development process due to the homogeneity of the system and allays the process of adding new features to the system. However, in our opinion, it is not the best choice in regards to the distributed artificial neural network simulator, as a monolith and previous implementation applied several

critical bottlenecks:

- Low level of development agility. N2Sky Java monolithic system should be reassembled and deployed each time any component is changed, which leads to the high development costs. As modules are tightly connected it becomes an issue to adjust the system,
- Low level of scalability. In distributed high-performance computing, as machine learning and neural network simulation, efficient usage of cloud computational resources is one of the crucial parts of the system. Although it is possible to scale monolithic application across the cloud, resource utilization in such case will be far from optimal as it will be necessary to deploy the whole application and some modules will be not used, which leads to the inefficiency,
- Language dependency - N2Sky was written fully in Java, and it applied several restrictions on the neural network development and deployment. As we've experienced, choice of Java for the main neural network development language can drastically affect the acceptance of the system in the scientific community and limit the ability to use the N2Sky platform. Usually, researchers should not be restricted to programming languages. They should be able to choose a language depending on the neural network or machine learning model requirements and to deploy models, which are written in any language,
- High dependency on one database - the previous implementation was fully dependant on one database, which was mapped to Java objects using an ORM. That led to the lower level of flexibility and was a limiting factor for the system, as any future developer needed to deal with the existing mapping, which became the part of the domain.

Taking into consideration all the mentioned problems, the novel microservice based N2SkyC architecture was designed and developed, which aims for increased extensibility, portability, dynamic orchestration and performance fostering cloud container technology.

Some parts and design decisions of the previous architecture definitely should be considered in the process of the analysis. Alongside the concepts, the new system implementation is aimed to provide full ViNNSL specification language support [5]. This project is highly influencing the implementation

details and requirements as it meant to be one of the new standards in the community. A several developments were made to ViNNSL during the years and system should be designed in a way, that it will be easy to implement them.

The layout of this Master's thesis is as follows: Chapter 2 consists of an analysis of the current architecture and design insights of the new N2SkyC architecture. Chapter 3 focuses on the technological stack, which is a base for implementation. Chapter 4 focuses on the components and topology description. Chapter 5 consists of the generalized development guides for the cloud infrastructure development as well as for neural network researchers. Chapter 6 presents user guide and is explaining underlying communication workflow between services. The thesis ends with Chapter 7, which is the outlook for the system and future development suggestions.

1.2 Terms and Definitions

Machine learning (ML) - is set of the artificial intelligent methods, which main characteristics are not the direct problem solvement, but the creation of the model based on the approximation of the results of the similar problems. The field is quite broad and includes a wide variety of heterogeneous approaches as decision trees, genetic algorithms, rule-based learning, classifier systems and many others.

Neural network (NN) - mathematical models and their implementations, which are trying to replicate the topology of the real neural activity. Its architecture is relatively simple and consists of two main components: simple parts which are called neurons and connection between neurons. Regarding problem-solving NN is a multiparametric non-linear programming optimization system. Parameters are coming through the network, and by resolving the coefficients between neurons, the network is trying to estimate and replicate dependencies between initial parameters and the output of the network.

Deep Learning (DL) - set of architectural models with a high level of abstraction, which is forming a multilayer system of non-linear transformations. Each new layer in such system is specifically designed to extract additional information from the initial data and to represent the initial data through different abstraction levels - from top to bottom. DL systems are characterized by a chain of transformations - credit assignment path(CAP). Credit assignment path is describing potential connections between the layers. Until now

there is no specification for the length of CAP to classify the model as a deep learning model, however, usually, the length of CAP should be more than 2.

Service-oriented architecture(SOA) is the architecture concept, which represents the application as a set of services. Services are small, detached software elements that solve one task and can be reused in many applications. SOA is based on the principle of weak connectivity, which means that each service is an isolated entity with limited dependencies on other shared resources.

Microservices architecture - is a subset of SOA, a modern representation of service-oriented architecture (SOA), used to create distributed software systems. As in SOA, the modules in the microservice architecture interact over the network with each other to communicate. Another similarity is that microservices use protocol-independent technology. This architecture is the first implementation of SOA that appeared after the introduction of DevOps, and it is gradually becoming the standard for continuously developing systems. In the architecture of microservices, the modules should be small, and the protocols should be lightweight.

DevOps - development and operations, a set of practices focused on the active interaction and integration of between developers and maintenance and deployment engineers.

LXC - operating system-level virtualization for starting several isolated instances of the Linux operating system on one node.

1.3 Related Work

distributedDataMining (dDM) is the open source project, which main goal is to leverage distributed computing by connecting interested parties into the research process [9]. Today, the project operates based on the Berkeley Open Infrastructure for Network Computing(BOINC), which provides both technical and software infrastructure to enable distributed computing[28]. BOINC is providing different software platforms to use the computation power of the nodes when they are at idle. Currently, dDM is running few scientific projects in different spheres: price prediction on the time series, evolutionary simulations, social network analysis and some others. The project is not restricted to the neural networks and trying to evaluate different machine learning techniques.

Neural Designer is the software product, which is aimed to provide data-mining functions alongside with neural network capabilities[4]. It is written in C++ and provides a desktop application to work with. Being a commercial product, it is developed based on the open-source library OpenNN. Although it is quite a robust and convenient software, it doesn't allow any distributed computing feature nor sharing or communication between users. The UI example is presented in Figure 1.

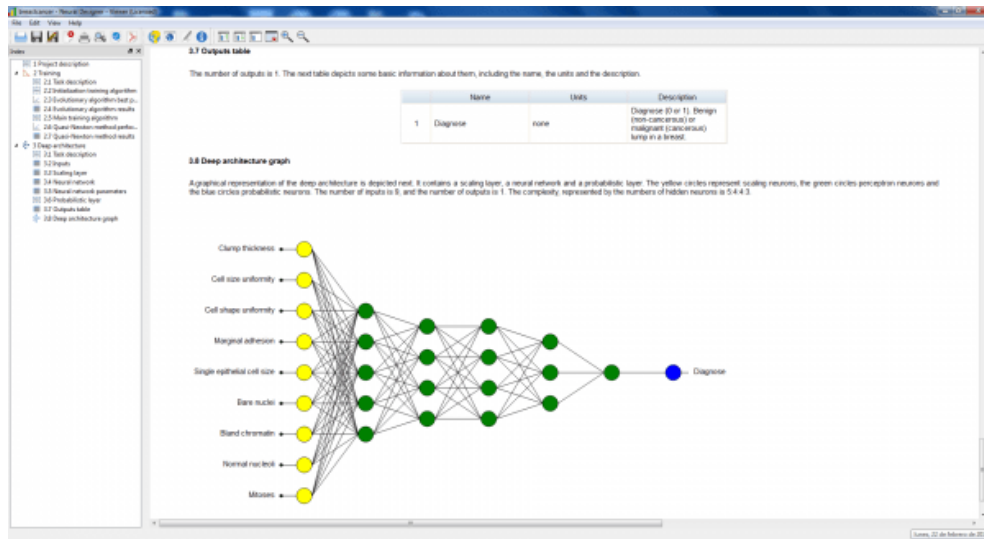


Figure 1: Neural Designer Screenshot
[27]

DNNGraph is a relatively new project, which aims at providing domain specific language for neural network description[2]. It is written in Haskell and allows the variety of optimizations which are automatically tuning the performance of the model. In some sense, it is similar to ViNNSL, but it doesn't have any execution environment or distributed nature. Nevertheless it gives a possibility to create complex deep neural network topologies and generate code from them for several popular frameworks. It is to consider, that N2Sky system can benefit from such add-on as new design will allow running language-independent code. The UI example is presented in Figure 2.

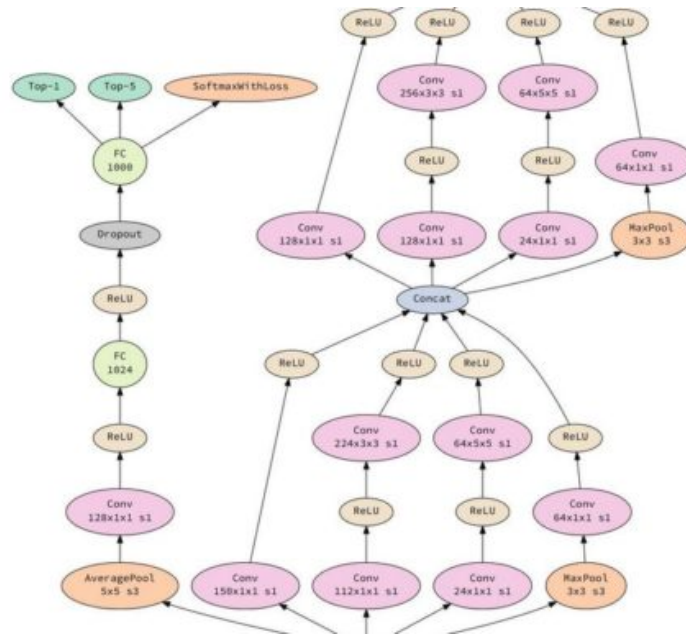


Figure 2: DNNGraph Screenshot

[2]

Neuroph is a platform developed by the University of Belgrad, and its purpose was to provide simple to use tool for modeling neural network activity[19]. It is written in Java language and provides both GUI and library to connect to own Java projects. In comparison to other platforms, it is not so sophisticated and mostly suitable for small development projects or university projects. It is still possible to develop a custom neural network but it becomes not a convenient task. *Neuroph* interface is presented in Figure 3.

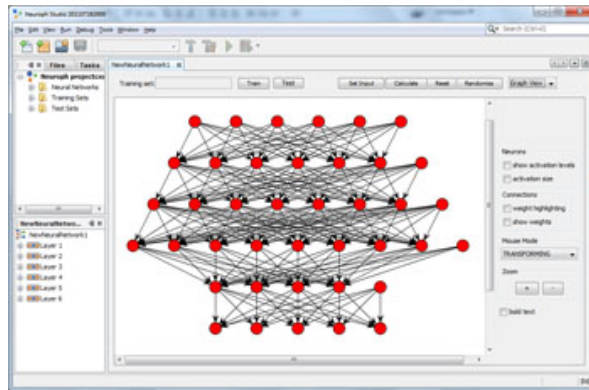


Figure 3: Neuroph Software Screenshot
[19]

The N2SkyC project had a long history and was prejudice by several projects, which were also developed at the University of Vienna:

1. N2Grid. The N2Grid system [24] which was fostering Grid technologies and aimed to provide a unified environment for storing and retrieving neural networks objects. It was designed in a way, that neural network objects should be stored as data blobs in the distributed storage.
2. N2Cloud. Next platform, which was developed in 2010[15]. Its main focus was to build up a simulation system which can be run in the cloud. It was the first platform, which tried to leverage SOA.
3. N2Sky. The last version of the N2Sky environment was an attempt to user RAVO description and guidelines and to design and implement virtual organization platform.
4. The current version of N2Sky is aimed to rethink the architecture approach and provide new architecture based on the microservices design. It aims to put together all conceptual design parts, which were developed before.

Previous version of the N2Sky UI is presented in Figure 4.



Figure 4: N2Sky UI Screenshot
[16]

2 The N2SkyC Architecture

2.1 Current Architecture Analysis

In this section we analyze the current N2Sky Architecture, identify possible bottlenecks and provide insights on limitations of the system.

2.1.1 General Scope

The N2Sky Architecture was designed to suit the distributed cloud infrastructure and provide several guarantees according to the system requirements. Requirements analysis was performed based on the questionnaires. However, mostly design of the system was influenced by requirements defined by Virtual Organization’s characteristics and not technical requirements. To a certain extent, it can be concluded that most architectural decisions in the first place were designed to be aligned with Virtual Organization guidelines. Implementation decisions were discussed very briefly, and there was no clear description of how the chosen technologies will enable described functional requirements. Nevertheless, functional requirements themselves are well aligned with the chosen domain and share common values with the new architecture. N2Sky was designed using a set of technologies which have several drawbacks both for cloud-based applications and machine learning development. The previous version of the architecture is presented in Figure 5.

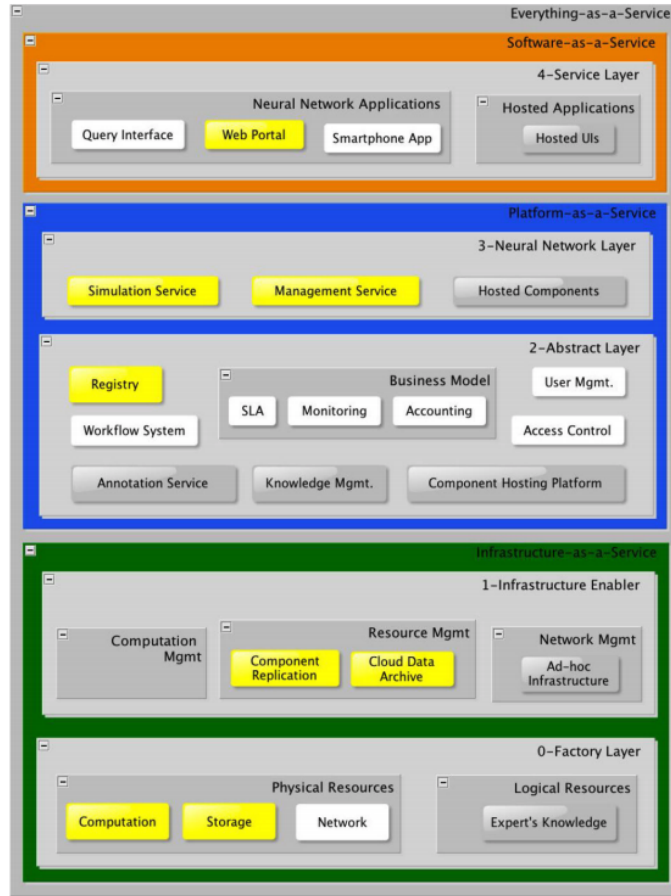


Figure 5: Previous N2Sky Architecture [16]

Java.

Infrastructure problems. Java programming language has quite a long history and was developed to help enterprises build websites and applications on top of application servers. Applications have since shifted from application-server-based model to a cloud-based model, a very different paradigm with distinct requirements for creating applications for reliability and scale. Cloud infrastructure allowed to shift to the complex distributed applications, which are no longer hosted on one single node. Together with cloud infrastructure development, new technologies as LXC had arrived and made the process of development, testing, building, and deployment more convenient. Despite the fact, that Java can be used for development of any applications, its object-oriented nature, and architectural patterns often lead to the monolithic design. Tight coupling of the modules leads to the inability to decompose the

system into smaller components, as there is lack of communication and a lot of shared properties and dependencies. Also, Java-based N2Sky needed several infrastructural requirements including Spring framework dependencies, Tomcat server to serve an application, Maven2[3] project management tool to build plugins, Jersey[20] framework to produce RESTful Webservices. All the data and data flow was designed to be managed by Spring Data JPA ORM[21], which is using many configuration files, which should be compliant with a chosen database. As a result, any developer and neural network researcher would need to familiarize himself with all these technologies. Before scaling the application to the cloud, any instance on the cloud should meet all these requirements.

Neural network development. Neural network development doesn't specifically imply any programming language. But the development of the virtual organization with a focus on the cloud infrastructure and shared resources should be designed in a way to provide services and open platform to the widest audience possible, including providing a possibility to extend the system. Currently, there exist two frameworks, which are enabling neural network development in Java: Deeplearning4j[18] and Neuroph. Both of them were not considered, and it is highly unclear on how they should be integrated into the existing N2Sky application.

Cloud infrastructure provider. Eucalyptus was chosen to be the main deployment platform. However, it is highly outdated. Starting as an open-source university project after some time Eucalyptus was bought by HP and shifted its main focus. Eucalyptus is mostly written in Java, uses SOAP as a communication protocol between services and this makes it highly unscalable and non-robust. Eucalyptus is a monolithic product, which is bundled as a package to fulfill client needs and whereas OpenStack is a set of loosely coupled components. Eucalyptus was focusing on providing an additional layer on top of the public clouds, but not as a separated cloud infrastructure provider. Being written in Java programming language, it is mostly suited for application deployment. On the other hand, there exist other cloud infrastructure providers, as OpenStack which is written in Python and provides a wide variety of services to configure for any task.

Web portal. The web portal was meant to be developed as a separate application using HTML5, jQuery, and The-M-Project but in reality, it was implemented as a part of monolithic Java application.

2.2 Redesign Motivation

2.2.1 Scaling and Architecture

During the development of the software engineering sphere, the performance, and accessibility requirements for information systems were continually growing both for new applications and for already existing ones. In order to meet new requirements, systems were usually scaled vertically - increasing the computational power of the node. However, vertical scaling efficiency was quite limited, as an increase in computational power of the node was not giving necessary results. That is why horizontal scaling was taken into consideration - approach, where computational power increase is not a result of adding more resources to the concrete node, but instead adding new nodes into the system. It should be noted that to ensure the efficiency of the application horizontal scaling, it is necessary to lay down its possibility in the architecture of the system at the stage of its design.

Scaling strategies can be presented in the form of a cube, where each side represents a specific scaling strategy[1]. Graphical representation of scaling cube concept is presented on the figure 6

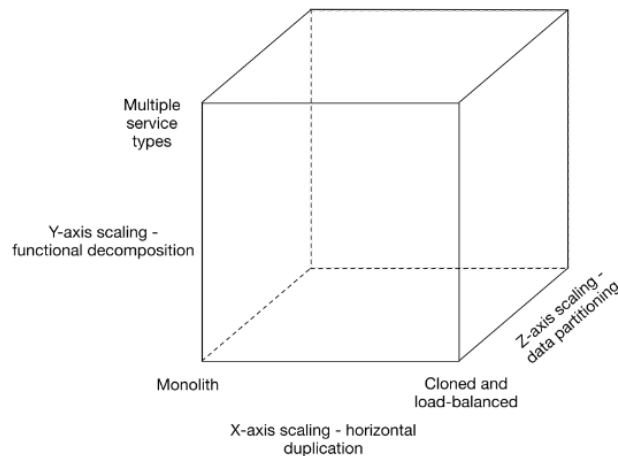


Figure 6: Scaling Strategy Cube

[1]

Scaling using X-axis is a common horizontal strategy - which uses load balancing. It is relatively easy and non-demanding scaling strategy for many types of application, including scientific computing. This approach works even in cases, where there was no initial scaling design. Monolithic applications are usually scaled in this way, as other design patterns were not considered

at the architecture design time - that scenario reflects the previous N2Sky implementation.

Scaling using Y-axis is a decomposition strategy, where an application is split into several services with an isolated functionality. System communication is designed to be managed by routing mechanisms. Although such approach can also be implemented in monolithic applications, it becomes a challenge to keep services or modules isolated from each other and to remain consistent.

Scaling using Z-axis is similar to X-axis scaling, but with a difference, that load is split based on the data requirements of the request. Therefore, it is usually achieved by sharding the database.

2.2.2 Microservices Architecture

Microservices architecture is an architecture, which enables horizontal scaling within "shared-nothing" cloud infrastructure on the design level. The main component of such design is application decomposition in a set of small services, which are run in a separate isolated environment and communicate with other parts of the system through lightweight messaging protocols as message queues and HTTP. A process of decomposition allows developing a system, which will be horizontally scalable by default, without additional effort. It exists minimal infrastructure and management requirement to implement such system, which often includes automated deployment strategies and monitoring. This allows using any frameworks and data storage technologies.

As with any decision, there are several drawbacks and compromises - as the N2SkyC is composed of a wide variety of services, there is an additional effort to maintain all of them and not only one monolithic application.

An additional requirement is that system should be designed in a robust and fault-tolerant way, as in cloud infrastructure some services can become not available, and the system should be able to identify this and try to provide a suitable way of resolving the issue. It implies additional implementation effort and usually is performed by a set of testing and monitoring services which primary purpose is to identify a problem and automatically create and deploy a working version of the service.

To ensure stability and maintainability of the system, it should be continuously monitored - both on the application and infrastructural levels. In case

of N2SkyC monitoring, the task also becomes a part of the load balancing, as new neural network parts can be spawned on request.

Monitoring is especially important because of system composition - asynchronous services communication can lead to unpredictable behavior.

Using each component as a separate service also allows to plan, develop and deploy new features more efficiently, as there is no need to rebuild and deploy the whole application, it's enough only to deploy new parts.

The drawback of such approach is that changes in services can be unsynchronized, which can lead to system fails. To ensure, that services are providing the same functionality different testing techniques should be considered.

To implement a stable system, it is needed to invest time and effort into designing automated testing and deployment pipeline - if done correctly it will allow having the same complexity level for all the components, as they will not differentiate in deployment.

Process automation should be split into several parts:

- Resource allocation automation - infrastructure layer should be able to deploy a new node on-demand. The most efficient way to achieve a high level of automation is to use cloud infrastructure,
- Build automation - to ensure, that new revision of the component is compliant with the previous revision. It can be achieved by versioning and testing techniques,
- Deployment automation - to ensure that services can be deployed in the production environment and met all the required dependencies.

It is very important to ensure that all the requirements of the microservices architecture should be met. It implies not only robust system design but also a choice of the right technological stack.

2.2.3 Microservices Infrastructure

Microservices deployment is a non-trivial process, and therefore it should be well designed as if done improperly it can lead to the exponential complexity growth. Continuous Integration gives control over the deployment process and bringing system components together.

It focuses on the checks and module testing each time new service or code is produced. One of the stages of this process is the artifact creation, which is used afterward in tests. We should also ensure that the tested artifact is the same while deploying the service across the cloud infrastructure.

To allow artifact consistency, there should exist a trusted repository, which holds the tested artifact versions. This technique gives additional advantages as it guarantees that new versions of the services will work with the previous versions. If there is no system which enables it, with each new code revision it will be much harder to ensure stability and consistency. There exist many test technologies to provide coverage, and some of them are more suited to the microservices architecture, as Contract Based Testing, but this discussion is out of the scope of this thesis.

If continuous integration is done without any compliance checks, it will only tell if the code syntax is correct, but provide no additional information on the system health.

The system should reject changes, which are not compliant with current services topology, as it can lead to the system failure.

Continuous integration usage should be designed in a way to allow the independent changes and deployment of each service. Several different strategies of an interrelationship between services and deployments are available:

- Monolithic deployment,
- Modular deployment, and
- Split deployment.

Monolithic deployment is characterized by single version control system, which is responsible for tracking all the changes of the source code. If the source code of any service changes it automatically triggers checks and deployment creation process. This approach is the simplest to implement, as there are a low number of repositories to track and it is efficient to use, but only in a case where there is no necessity to deploy services separately. The bigger problem also arises when there is no certainty on which artifact should be deployed. In this scenario all the services are deployed - if there is any failure in any service, the development process will be paused until the issue is resolved. Example of monolithic deployment workflow is presented in Figure 7.

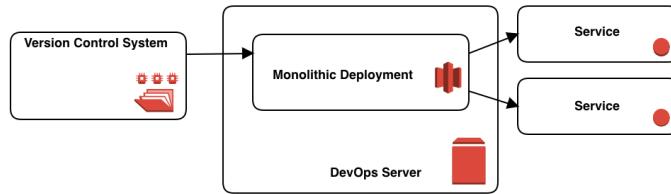


Figure 7: Monolithic Deployment Workflow

Next approach is modular deployment, which is characterized by single code repository, but continuous integration process is separate for each service. This approach leaves the code change reflection as simple as in monolithic approach, but there is a possibility that changes will be made to the several services, which in the same way will be reflected in tighter bounding between the components, which contradicts the idea of microservices architecture. Example of modular deployment workflow is presented in Figure 8.

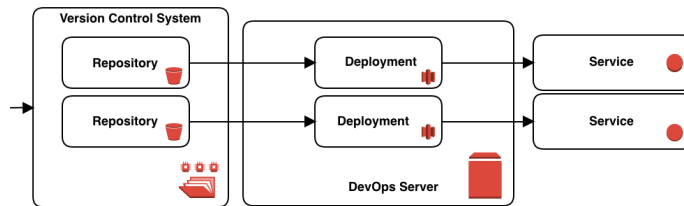


Figure 8: Modular Deployment Workflow

The third option is to create a separate repository for each service and invoke integration process only the service, which is affected. It becomes easier to have clear development process and to coordinate the development in general. However, it implies additional organization work to synchronize code bases of the several microservices. Split deployment example workflow is presented in Figure 9.

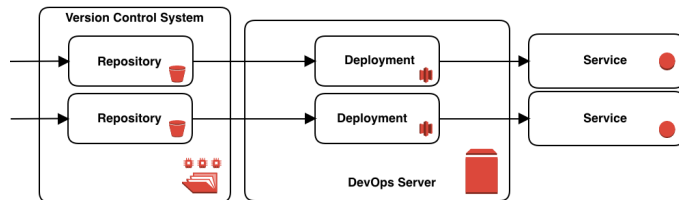


Figure 9: Split Deployment Workflow

During the work on the new N2Sky system architecture, we have evaluated and analyzed several methods of continuous integration and came to the conclusion that split deployment reflects the requirements of the microservices architecture the best.

2.2.4 Component Decomposition Guidelines

The main idea of microservices approach is to decompose the software system functionality into separate components, which are responsible for a certain task. Microservices do not offer any decomposition framework, so it is the responsibility of the software architect to distinguish areas of responsibility for each component. Despite the fact, that microservices definition is not very clear and can vary, common design decisions and features exist, which have to be considered:

- Services decomposition,
- Smart endpoints and dumb pipes, and
- Data management decentralization.

Taking into consideration challenges, which arises during the service decomposition process, we defined several guidelines that can be useful for the future implementation process.

These guidelines can be used to analyze which functionality should be split and how to define boundaries of the services. It is important to mention that current architectural version of N2SkyC was implemented to provide the conceptual framework and support the architectural change to the new design. It still can be decomposed further and changed according to the future needs - that was the main point of the refactoring of the system.

Taking into consideration previous N2Sky architecture design as a reference, an important step is to analyze the current system and deliver insights on how the system should be decomposed according to the different requirements. The previous revision of the N2Sky architecture is presented in Figure 10.

It is important to mention, that although components are described as "Web Services" they are not providing any endpoints to other components of the system and communication is encapsulated in the monolithic logic of the application.

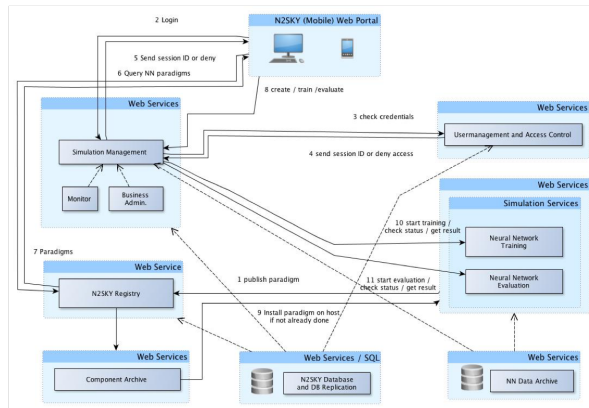


Figure 10: Previous N2Sky Architecture [16]

As a result of the decomposition and design process, following architecture was delivered - it is presented in Figure 11.

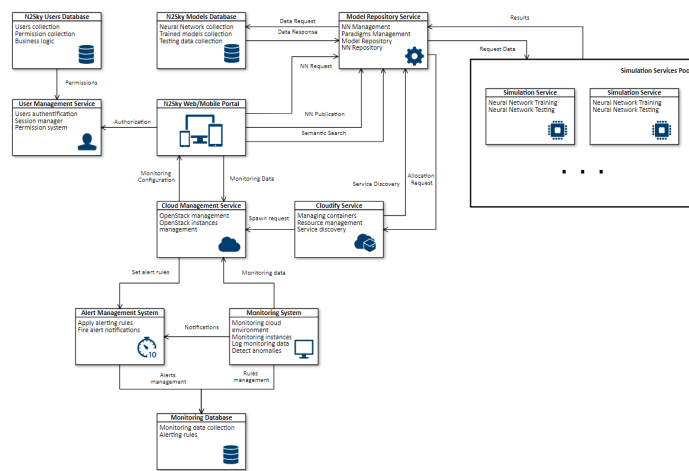


Figure 11: New N2Sky Architecture

To decompose the application into parts, we should keep attention on the set of different dimensions.

Domain capabilities and consumer needs. Analysis of the process of delivering value to the end-consumer of the product helps to understand, how components should be composed. N2Sky platform focused on providing cloud resources and neural network and machine learning development and execution environment for a wide variety of users. That is why we designed components, which are directly responsible for delivering this functionality. In the same

time, the second assumption is consumer related needs - it becomes clear, that services should be designed to fulfill the possible demand of the users. Splitting functionality in services allows bringing new features and controlling value-streams of the application. As a result, simulation services, which were separated into two different parts merged into one component, which allows to scale it across the cloud.

Service context. Partly adapting Domain Driven Design ideology[12], we should conclude which parts of the application and at the same time, which parts of the domain can be put together. The cloud infrastructure, for example, was not considered as a part of the system, so there were no any design decisions to support the communication between services and resources. However, the resource allocation and management are highly important part of the system, and there should be the technical possibility to scale the system across the cloud. Taking into consideration these assumptions, it leads to the conclusion, that all services and components of the system, being deployed on the cloud as separate applications should be managed all in the same manner. As a result, it was decided to design a component, which will be responsible for management of the services across the cloud.

Communication protocols. Having the system sharded and decentralized implies additional requirements on how components are communicating with each other. The communication patterns should reflect the data and operation flow - it is important that services have uniformed communication and that there is no service with too much responsibility. In N2Sky case, simulation management service wasn't composed well: it included not only domain-specific tasks as simulation but also business and administrative tasks. In the new architecture, simulation service functionality is encapsulated into one service and there, therefore, communication management is more clear. An additional point of interest is that the application should be designed in a way when there is no component which is highly dependant on others. In case such component exists, as cloud management service, it becomes important to ensure that there are systems which can monitor its health and scale it when needed.

Interconnectivity. If there will exist a necessity to change application functionality in the future, it is efficient to design the system in a way, that changes will affect only isolated services. Therefore, it becomes important to predict which functionality will be developed in the future, so when the new revision of services is developed, there is no need to change the API. Regarding mono-

lithic application, when components dependency is hidden and split across the application, it becomes challenging to change the application without breaking it. Interfaces description and specification can help to slow down the complexity growth, but it is a temporary mean. In the new N2SkyC architecture, each component can be changed independently and therefore there is less effort in developing the system.

Data architecture. Services have their data, and data flow and microservices architecture allow to use a separate database for each service. However, it brings an additional task to design data storage and data flow in a consistent way. If several services are dependant on the same data, it may worth to put the data in one place and scale it horizontally if needed. In case of N2Sky, one database was responsible for storing data of functionally different domain fields: user management, simulation management and registry for storing neural networks. Splitting this logic according to the data isolation principle led us to the decision to create three data storages - one for each specific microservice. One for user management, one for monitoring and alert system and one for neural network simulation services.

Merge and split. During the development, knowledge domain can become wider, and there will be a need for creating additional services and integrating them into the system. Architecture should enable efficient merge of the services or their split. If the system is too interconnected, that will decrease all the benefits from using the microservices approach, as regarding development it will on the same level of complexity as with monolith or even exceed it.

Service discovery. Parts of the challenge to use and deploy service across the cloud infrastructure is service discovery. By default, any microservice is unaware of other parts of the system and therefore as services communicate with each other using network, they should have a way to know that there are other parts. A monolithic application doesn't have such problem, but if we assume to scale the system horizontally across many nodes, it becomes an important task. In the new N2SkyC system design, the orchestration tool, which is part of the cloud management service and deployed separately is responsible for service discovery and identification.

Loose stability. As services are meant to be constantly developed and deployed, there can be a scenario in which several versions of the service exist simultaneously. N2SkyC is designed to be the platform which is used by different types of developers, and it is critical to enable the stability of the system while services are different. To resolve this problem, it is efficient to

design and develop services and architecture in such a way, that they should be redeployed only in case of critical changes.

Stateless design. Creating several instances of the service means, that each of the parts should have access to the data and resources. In the microservices architecture, that means, that services should not have own copy of the data, but rather fetch the data from a remote data storage. Additionally, the part of the same concept is that services should be deployed in one shot, and instead of upgrading old revision of the service it is more efficient to design the system in a way, that it can switch from usage of one service to another one. It meant to solve the problem with data and communication inconsistency.

Monitoring. High load systems, which are developed to be scaled across the cloud should be able to react accordingly to the additional resource demand. In the scientific field, it is especially important, because resources are not free and ideally they should be provided by demand. N2Sky system is not an exclude, and it was designed to fully leverage the cloud capabilities. However, the previous version of the architecture didn't answer the problem of workload detection and scaling of the system. There were no components designed, which are responsible for cloud management and therefore we decided to fill this gap. It is performed by two important services: monitoring and component orchestration. Whereas cloud monitoring controls cloud resource usage and orchestration management is tracking the performance of a specific service.

Combining these guidelines allowed us to decompose the previous application and develop a new N2SkyC architecture, which provides better guarantees and suits the cloud infrastructure.

3 Technological Stack

To enable described features of the system and reflect the new architecture design, it is highly important to choose the right technological stack. It implies no limitations on the development of the services themselves, but we believe that some underlying technologies can suit and reflect microservices architecture in a better way.

Although cloud platforms only provide infrastructure and environment for container deployment, and it is not related to the current software application design, cloud infrastructure products are different in terms of ideology

and communication means. Therefore we chose OpenStack[26] as a base for the cloud infrastructure as it is also built according to the microservices architecture approach. It provides great support for microservices paradigm implementation but implies no boundaries on the software developers. OpenStack consists of a variety of components, which are also designed in a modular way: computing, networking, storage, images, UI, authentication, and others. From the microservices perspective, it provides a high-level API to manipulate internal services.

Isolation of software functionality can affect the development and deployment process. However, benefits of the current implementation often will depend on a technology stack. Existing technology that fosters the microservices approach is Linux containers (LXC)[7]. Containers are a lightweight OS-level virtualization abstraction primarily based on namespace isolation and control groups.

As container quantity can grow very fast, it becomes clear that manual maintenance of hundreds or even millions of Docker containers can be a tough task, especially considering a cloud environment. For that reason, container orchestration software as Cloudify[14], Kubernetes [14] and others were developed. These tools are providing high-level interfaces to communicate with cloud-based platforms and control deployment and execution of containers. Putting all these technologies and design approaches together, we can a robust and efficient cloud-based environment, which can be easily scaled both horizontally and vertically. However, it is still important to be sure, that selected technologies will benefit the system.

3.1 Cloud Infrastructure - OpenStack

The Cloud computing paradigm provides access to large amounts of computing power by aggregating both hardware and software resources and offering them as a single system view. It hides the details of implementation and management of software and hardware from the end user. Cloud computing is the next step in the development of the distributed systems. Although it is not a new concept until now there were not so many technologies, which are suitable specifically for cloud infrastructures. Lately, with a rapid development of the DevOps, it became more convenient to design systems which are deployed across "share-nothing" infrastructure.

OpenStack is a system for organizing cloud infrastructure. In other words,

the tasks of this system include the organization and control of infrastructure and services that support the processes of creating virtual machines and containers and supporting their deployment. A user working with this platform can request system for resource allocation and run a virtual machine with specific functionality to perform any tasks. Figure 12 shows the architecture of the OpenStack system.

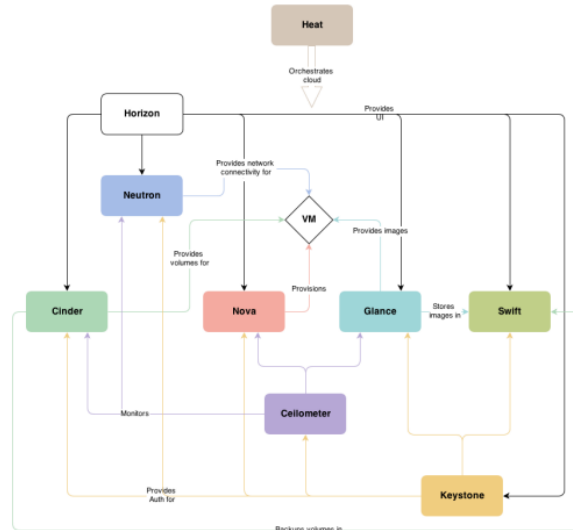


Figure 12: OpenStack Simplified Architecture [29]

OpenStack consists of a number of services, which provide all functionality. All modules support REST API. REST API - the interface of the application which is using HTTP requests for communication[22]. To implement REST, a client-server architecture is used. This method is used in OpenStack since all components are independent services and REST HTTP communication is a way to ensure consistency in communication means.

Nova. The core module of the OpenStack. The tasks of this module include launching virtual machines and their management. Nova components provide virtual machines resources necessary for their work. Management of virtual machines is performed by hypervisors. The Nova component consists of several weakly dependent components (the Nova architecture is shown in Figure 13).

Requests accepted by the interface are routed through HTTP depending on the request to a specific service. The message queue is used by components to interact with each other. A significant part of the functionality is provided

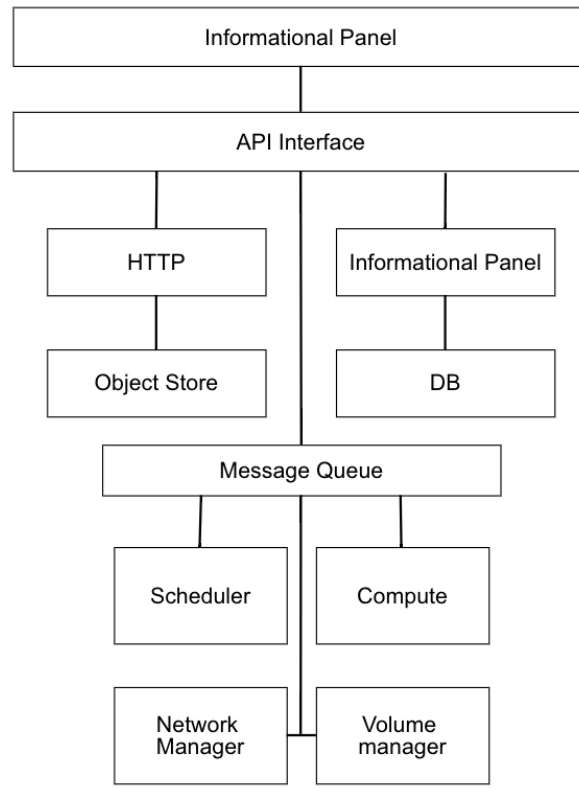


Figure 13: Nova Architecture

by the Scheduler and Compute daemons. In addition, the module stores information about virtual machines, such as time creation and modification of virtual machines, some network information, the state of the machines, information about the allocated resources and other. The Scheduler during the initialization phase of the virtual machine determines where it should be run depending on the resource requirement. The Compute daemon provides interaction with hypervisors and virtual machines. OpenStack provides support of the LXC containers, which are spawned directly using the docker driver, but this functionality is still under development.

Swift. It is a distributed, decentralized object storage with support mechanism of replication and other technologies that provide high fault tolerance. Swift uses the consistency model, which ensures that all queries will eventually return the last values if there are no new changes to the data. This approach provides system high availability. For data stored in the system, the following hierarchy is used: account, container, object. The account represents the top level of the hierarchy. Containers, which have the same names but located

in different accounts, represent different entities. Similarly for containers and objects. The objects are end-user data, such as documents, images, and more. The system supports data compression. It is important to note that Swift is not the only available object repository. For example Ceph[23] can be used as the object storage, which provides support for the Swift and Amazon S3 interfaces.

Heat. The orchestration system, designed specifically for the OpenStack infrastructure. Heat provides management of virtual machines and other objects of the OpenStack environment (such as users, security groups, etc.). The system manages entities through templates that describe use cases. Templates can be written in several different formats, including AWS format (Amazon Web Service) or HOT (Heat Orchestration Template).

Keystone. One of the main OpenStack services, which is responsible for users authentication and providing a catalog of services with their API. Any user before logging on to OpenStack must log in to Keystone. The user, in this case, can be a person or a service. After that, the user receives a token, which is valid for a certain time and in which holds the information on user permissions.

Glance. The store of images of virtual machines. Any storage can be used with Glance, as it is not responsible directly for storage and can be coupled with any file system. When prompted to create a new virtual machine, Glance loads the image from storage and sends it to Nova.

Cinder. Block storage service. This service provides virtual storage and assigns it to the virtual machine. The system architecture is presented on the. The volume manager controls volumes, devices for block storage. These volumes are connected to compute nodes using iSCSI technology, and they are used as long-term storage.

Horizon. The OpenStack graphical interface, one of the ways to control the system. Another option to manage OpenStack is CLI. It is assumed that the OpenStack administrators will modify it according to the domain needs. Therefore source code is provided in each OpenStack installation.

Neutron. The OpenStack module, which provides a network virtualization service for all other OpenStack subsystems, and for virtual machines in the OpenStack. Neutron is responsible for configuring virtual network interfaces and organizes a network which is used by all modules of the system. With Neutron, users can create complex network topologies if there is a necessity

to work with multi-level web applications. Neutron virtual switch connects virtual machines via ports. After creating the port, it receives fixed IP address. The address is issued either from the address pool or the Neutron service itself. Neutron allows creating complex network topologies, which are then connected through virtual interfaces.

Ceilometer. System monitoring for components of OpenStack. Ceilometer provides a collection of performance metrics for computational resources and network. In addition to collecting metrics, Ceilometer can notify the user of specific events. Information is collected through agents running on each node. There is an opportunity to use databases(PostgreSQL, MongoDB, etc.) to store collected information.

Sahara. The service which is responsible for the deployment of multiple distributed computing environments: Apache Hadoop, Spark, and Storm. It allows to run Map-Reduce tasks, aggregate the results and provide interfaces to cluster components.

As a result of such services composition OpenStack can provide a robust and scalable environment for the N2SkyC deployment and enables all the features of the microservices architecture on a design level. Also, it fosters the usage of LXC container technology, especially Docker and allows to deploy Docker containers across the cloud infrastructure with additional effort, which will present in the development guide.

3.2 Virtualization Technologies

One of the main ways to manage a large number of hosts is to divide existing physical machines into smaller parts. Traditional virtualization such as VMWare or AWS-based brought huge benefits in reduction of host management costs. However, a new development in this field provides new opportunities for microservice architecture. Running one instance of the microservice on the server without virtualization is not efficient. But in most cases, the deployment of a multiple microservices is performed on bare metal without virtualization. Running multiple services on bare metal without virtualization can lead to conflicts in the environmental settings between the different services, as they share the same OS and memory. Microservices are not isolated from each when they are running on the same machine. Also, the services deployed on the one server can occasionally consume the resources of other services, reducing their performance.

3.2.1 Traditional Virtualization

Running a large number of hosts is expensive. If using physical servers as hosts, this becomes more and more challenging. If nodes are used to horizontally scale the system, it will mean that resource usage will be highly inefficient. Virtualization allows splitting the physical servers into independent hosts, each of which will contain different application or application parts. Main software part, which enables virtualization is called hypervisor, and it has two main functions. First, it allocates the CPU and memory resources of a physical machine to the virtual host. Secondly, it serves as a control layer that allows manipulating virtual machines. A system which is run inside of the virtual machine is isolated from the host environment. The virtual machine is completely separated from the underlying physical host and other virtual machines through the hypervisor.

One of the problems with using virtual machines is the need to allocate resources for its operation. It is completely limiting the reuse of dedicated resources, such as CPU, I/O, and memory for other virtual machines, or the primary operating system. The more virtual machines are managed by the hypervisor, the more resources are required. Ultimately, the management overhead becomes a limiting factor for the further split of physical resources. In practice, this means a proportional increase in the resource management costs as the hypervisor is tightly connected to the increasing number of virtual machines.

Virtual machines are quite heavy. In this way, the launch of many small virtual machines on a physical host is not effective because it has large overheads, in fact, it is a waste of resources for the hypervisor management. If deployment of microservices is carried out in one virtual machine, the same problems that have been described above will occur. For example, deploying microservice written in Java programming language will lead to the shared JRE usage. Any change in the JRE on the host will affect all the services deployed on this machine. Also, if there is a need in specific OS settings for the service or certain library, it will be more difficult to manage the requirements, as they apply immediately for all services.

One of the principles of microservice architecture states that service should be isolated and autonomous, fully encapsulating the execution environment. To comply with this principle, all components, such as the operating system, runtime and executable code of the microservice must be autonomous and

isolated. The only way to achieve this - follow the approach of one microservice per virtual machine. However, this will result in inadequate resource usage of the virtual machine. Also in many cases, due to additional overhead costs can reduce all the advantages of the microservices.

3.2.2 Container-based Virtualization

The technology of containerization is far from new and not a disruptive, innovative technology. It has been used for quite a long time. Nevertheless, this technology began to gain considerable popularity with the arrival of cloud infrastructure. Disadvantages of traditional virtual machines have become a catalyst for increasing popularity of containers. Containerization technology was simplified by tool providers to reach the community faster. The popularity of DevOps and microservice architecture also accelerated the containerization technology development. Containerization technology provides an isolated environment in the operating system. This technology is also called virtualization of the operating system. In this approach, the kernel provides an isolated virtual space. Each of the virtual spaces is called a container. Containers allow other processes to create an isolated environment on the operating system, which is hosting these containers. Figure 14 shows different layers, involved in the process of containerization.

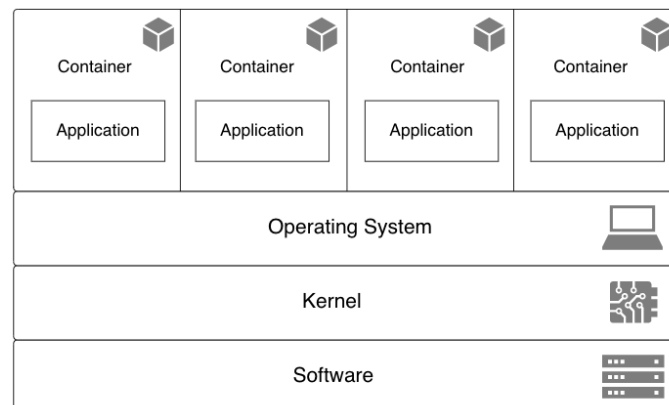


Figure 14: Container Architecture

On the one hand, containers are a simple mechanism, and on the another, they are powerful technology to deliver loosely coupled software components. In general, containers pack all executable files and libraries that are required to run the application and don't provide any additional interfaces to the ex-

ternal world without permission. Containers completely isolate the following elements:

- File system,
- IP address,
- Network interfaces,
- Internal processes,
- Namespace,
- OS libraries,
- Application data,
- Dependencies,
- Configuration files.

All containerization tools are based on the functionality of the Linux kernel. The main components of the Linux kernel for containerization are listed below:

- Namespaces,
- Control groups.

Namespaces are used to create an isolated environment, which is called a container. When the container is started, the system creates new namespaces for this container. The namespace provides an isolation layer. Each aspect of the container is performed in a separate context.

- PID,
- Networking,
- Inter-process communication,
- Mount points,
- Isolated kernel and version identifiers.

Control groups, as well as namespaces, are contained in the Linux kernel. Control groups limit resources, consumed by the container. They also allow the Linux kernel to divide available hardware resources between containers and limit or expand these resources as necessary. For example, it is possible to limit memory usage for a particular container. At first glance, virtualization and containerization share the same characteristics. However, containers and virtual machines are quite different technologies, and they are also solving different virtualization problems. These differences are seen in Figure 15.

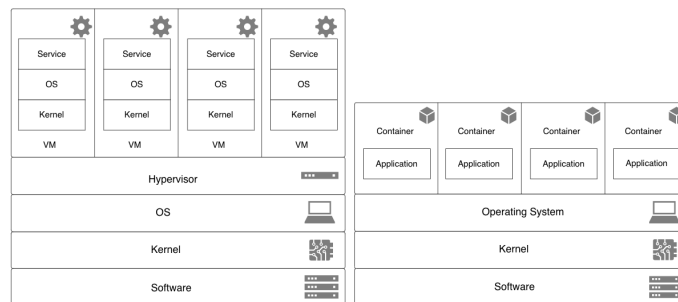


Figure 15: Differences between VM and LXC

Virtual machines work at a much lower level than containers. Virtual machines provide virtualization software, such as CPU, I/O, memory, etc. The virtual machine is an isolated component with the built-in operating system, which is called a guest operating system. The virtual machine contains the operating system completely and runs it without any dependence on the host environment. Because the virtual machine includes the operating system environment entirely, it is quite heavy. On the one hand, this aspect can be considered as an advantage, but it is also a disadvantage. The advantage is that all the processes, which are run in the guest OS are isolated. The disadvantage is a limited amount of virtual machines that can run on the host, as virtual machines reserve the resources necessary for their work. The size of the virtual machine directly affects the load time. Starting the virtual machine is a quite computationally heavy task, and usually, it takes some time.

On the other hand, containers do not emulate all hardware or whole OS. Unlike virtual machines, containers use certain parts of the kernel of a guest operating system and host OS. Containers do not use the concept of a guest operating system. A container provides an isolated execution environment directly on the top of host OS. This strategy is also both an advantage and a disadvantage. The advantage is that container is usually light and fast to

deploy. Because the containers share the host operating system, the use of resources by containers is rather small. As a result, a number of containers which can run on one host node are relatively high, which cannot be achieved with virtual machines. Because the containers are running on the same host operating system, there also exist limitations. For example, it is not possible to configure a firewall inside of the container. Processes which are run inside of the container are completely isolated and independent of processes from other containers, running on the same host system. Unlike virtual machines, the images of containers are publicly available on various portals. This greatly simplifies the development because software engineers don't need to spend time building images from scratch. They can use the basic images from certified sources and add additional layers of software components to the base image.

The lightweight nature of containers also provides opportunities for their automatic creation, publication, downloading and copying. Possibility to download, collect, deliver and run the container with only a few commands or with REST APIs is a more flexible way of software development. The deployment of a new container takes a few seconds. Thus, containers have a huge advantage over virtual machines, but virtual machines have their strong sides. For the N2SkyC implementation, we decided to choose containers as they suit for the chosen architectural pattern of microservices.

3.2.3 Docker Platform

Docker[17] is one of the most well-developed platforms fostering containerization technology. The Docker platform solves three major problems of services deployment:

- Delivery of the code to the server,
- Run the code,
- Uniform the environment.

Docker allows to isolate services from the infrastructure, so it is possible to deliver them much faster. It also allows managing the infrastructure using the same principles that used in managing applications. Using Docker tools for delivery, testing and deployment, the delay between new code deployment into the version control system and roll-out of the service on the production-based server is significantly decreased.

Docker provides the ability to pack and run services in the isolated environment, which is called a container. This isolation and security allow running multiple containers on the same host at the same time. Containers are lightweight because they do not require the work of the hypervisor. They run directly on the host machine's core or in case of the OpenStack infrastructure Nova service can spin them directly on the bare metal. Thus, it is possible to start more containers than virtual machines on the same physical equipment. Also, Docker containers can be run in the virtual machines. The container becomes an atomic unit for distribution and testing. After development, the application can be deployed on the production server manually or with the help of container orchestration tool. This deployment technique is uniform across all platforms, regardless of where the application will be deployed to the production server, in a local data center, cloud provider or hybrid environment.

Docker optimizes the development cycle by providing developers with a standardized environment. Containers suit well for continuous integration and continuous deploy. A typical software development scenario is that new application service or artifact is deployed after code edits are applied to the version control system. Using Docker, this artifact can be a container itself. Developers or QA engineers are using Docker containers in the testing environment to run automatized and manual tests. If a software bug is detected during the checks, the container can be re-deployed in the development environment, corrected and transferred back to the test environment, for re-testing. If all checks are successful, the corrected container can be deployed into the production environment. Docker provides high portability. Portability of the docker containers, as well as his lightness, allows to dynamically change the load, increasing or reducing the number of applications and services, almost in real time.

Docker uses client-server architecture, which is presented in Figure 16.

Docker client interacts with the background process, which is called Docker server and it starts containers. Client and background process can be performed in one OS, and it is possible to connect a client to a remote background process. Docker client and background process interact via REST API over sockets or via a network interface. Background process Docker (dockerd) receives requests and manages Docker objects.

The background process can also interact with other background processes. The background process manages the following objects:

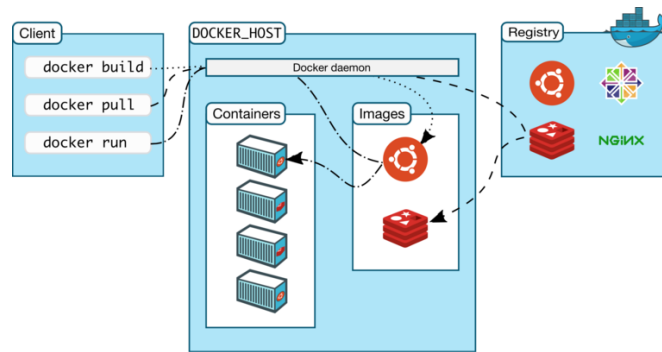


Figure 16: Docker Architecture
[10]

- Images,
- Containers,
- Network interaction,
- Volumes.

Container repository The container repository stores images. This repository can be public, and private. Docker Hub[11] is the official open repository, which stores official images from various technology providers, for example, Ubuntu or nginx. Official Docker image repository is free for use. By default, Docker is configured to search for images in this public repository. Also, there is a possibility to create private image repositories. When using the docker pull or docker run commands, the necessary images will be downloaded from the configured registry. Using docker push, the specified image will be placed in the registry.

Docker image An image is an immutable template containing instructions for creating a container. In most cases, the image is based on another image, providing it with new functionality. For example, it is possible to take an image, which is based on the image of the OS Linux Ubuntu, infuse it by installing the Apache web server with the developed application, and configuring the server for the application to function correctly. It is also possible to create new images or re-use already created ones from the repository. To create a new image, it is needed to create a configuration file called Dockerfile. This file contains instructions for creating and running an image. Each instruction in the configuration file creates a layer in the image. When the configuration

file is changed, and the image is assembled only those layers that have changes will be rebuilt. It is this technology will allows containers to be lightweight and fast compared to other virtualization technologies.

Docker container A container is a deployed instance of an image. Containers can be created, started, stopped and deleted using the Docker API or CLI. The container can be connected to one or several networks. Part of the file system can be mounted to its host machine. It is also possible to create a new image from the current state of the container. By default, the containers are relatively well isolated from other containers and the host machine. It is possible to manage isolation level of network interaction between containers and connected volumes between other containers and the host system. The container is created from the image, and also contains all configuration parameters that are provided when it is created and started. When the container is stopped, all changes in its state, which are not fixed in the permanent store, will disappear.

Image Layers Each image is linked to a list of unchanged layers, which represent differences in the file system. Layers are stacked on top of each other to form the base for the root file system of the container, which is presented in Figure 17.

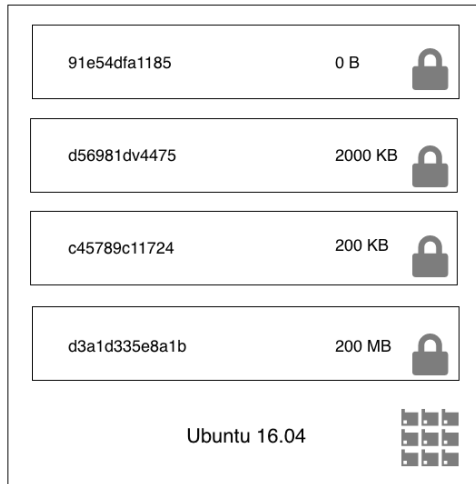


Figure 17: Docker image layers

When a container is created, an additional layer is added to the image, which is called the container layer. All changes made in this running container, such as creating, modifying and deleting files are written to this variable layer. Figure 18 shows the layers of the launched container.

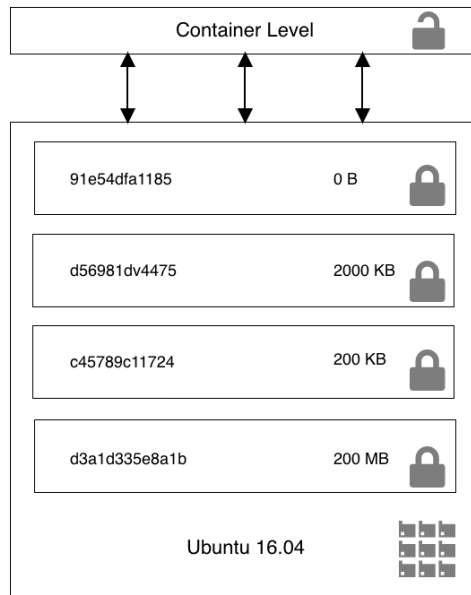


Figure 18: Running container layers

The main difference between the container and the image is a changeable top layer. All container changes that add or change the data are stored in this variable layer. When the container is deleted, the changeable layer is deleted alongside with it, and the underlying image remains untouched. Because each container has its mutable layer and all changes are stored in it many containers can share the underlying image and have a different state. The visual representation of the concept is presented in Figure 19.

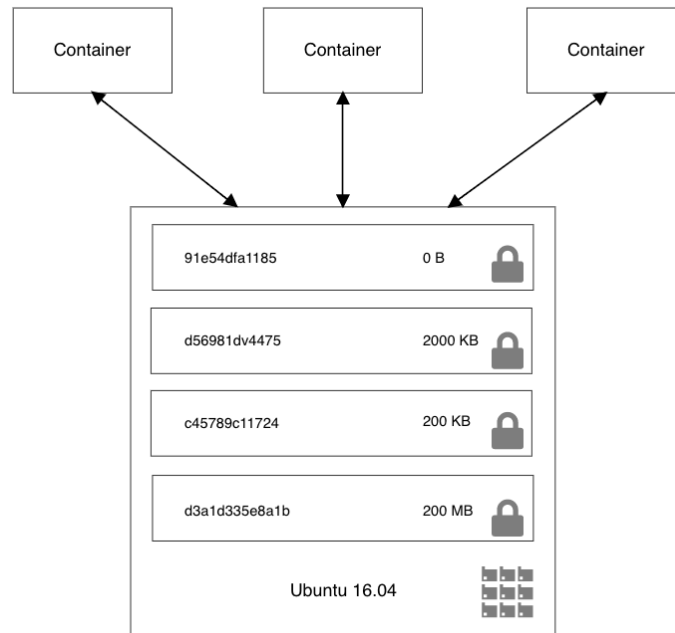


Figure 19: State sharing schema

Volumes As was mentioned above, when the container is stopped, all data from the mutable layer will be deleted. To persist the data used by the container, it should be taken away from the internal container environment and put into docker volume. A volume is a directory or file on a host machine that is mounted directly into the container. Any number of volumes can be mounted in a container. Also, many containers can share one or more volumes. Thus, volumes are created for the persistent storage of data, which is independent of the life cycle of the container. Therefore, the Docker will not automatically delete the volume when the container is stopped.

3.3 Orchestration Tool

Docker is an excellent tool that provides a full set of functionality to ensure building, deployment, starting and stopping containers. It is convenient to use for isolating the environment and it solves one of the problems of the microservice architecture, which is the encapsulation of various technologies in a unified format. Using containerization while developing microservices, software engineers are no longer required to monitor the configuration of runtime environment, the configuration of environment variables, and the various libraries that they are used by the service. All the dependencies are encapsu-

lated and packed into a container. But when using a microservice architecture, there are hundreds and even thousands of different services, which also need to be scaled. The Docker client is great for managing a small number of containers on a single host. To automatically deploy multiple services, it is needed to describe their relationships, as well as the number of instances that someone needs to run. Also, it is required to limit the resources, which are available to microservices. Thus, if the service is scaled horizontally, it is required to automatically distribute the load between the started services. To solve this problem, there are tools for containers orchestration. They are the primary container management interface for administrators of distributed systems.

The orchestration is a wide term which describes the management of containers distribution, cluster management, and the ability to add additional hosts. The Container scheduler accepts files settings that define the launch options for a particular container, and also algorithms for distributing containers to different hosts. Orchestration schema is presented in Figure 20.

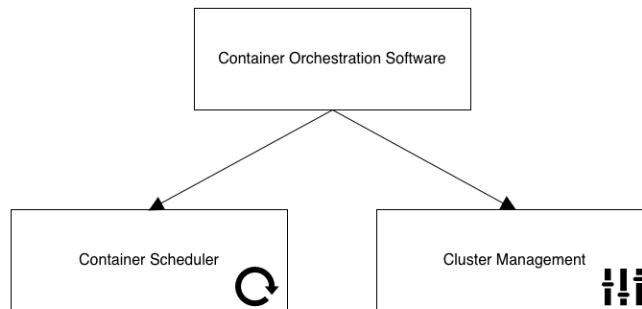


Figure 20: Orchestration Responsibility Split

Cluster management is the management of a group of hosts. This can include adding and removing hosts from the cluster, obtaining information on the current status of hosts and containers, as well as process management. Cluster management is closely related to the distribution of load between hosts since the scheduler must have access to each cluster for task distribution. For this reason, it is often one application which is responsible both for cluster management and for distribution of the tasks between hosts. Taking into consideration, that some cloud infrastructure providers, like OpenStack, allows deploying containers on the bare metal the task of load distribution between virtual machines becomes depreciated. However, in the scenario, where there exists middleware between cloud orchestration service and container orches-

tration service, load balancing between nodes is highly relevant.

Cluster management. The distribution of load between containers is often closely related to cluster management, as both of these functions require to work directly with both containers and the cluster as a single entity. Cluster management software can be used to retrieve information about the containers in the cluster and to provide fine-tuning for individual containers. All these functions can be scheduled and automated. Often, cluster management is also associated with service discovery service, as it holds all the information about started containers and their availability. Such services suit well for storing container configuration information about the cluster topology due to the distributed nature of the cluster itself. Containers can be marked with special labels that can be used for distribution of tasks between individual containers and groups of containers.

When the application is separated into small isolated services, these services should be perceived as a whole. In the microservice architecture deployment of one service without deploying its dependency services makes little sense, since they must work together. Therefore one additional task, which should be performed by cluster management tool should be a grouping of the services. Group management allows an administrator to work with a set of containers, as with single entity. Running group of containers which are tightly connected forming one entity, simplifies the management of the application, without sacrificing the benefits of functional decomposition. Ultimately, this allows developers to take advantage of containerization and microservice architecture, minimizing the effort for system management. Grouping services allow to distribute them in such a way, so that they are run together, allowing to start and stop the group at the same time. Also, the grouping of services supports more complex tasks, for example, setting up individual subnets for each group of applications or scaling or group based scaling.

To meet all the requirements of the cloud orchestration, it was decided to take advantage of OASIS's Topology and Orchestration Specification for Cloud Applications (TOSCA)[6], which provides an efficient and clear framework for orchestration of complex applications. It can be adapted to develop a cloud-based application based on the microservices approach. It uses YAML language to describe templates, which represent component topology with all the information about other components of the system. There are two main syntax elements in TOSCA specification, which specify concrete topology entity and relationship between them: node and relationship. Node is a wide

concept, which can include not only the infrastructure system components, such as networks, sub-networks, servers or cluster of servers, but also software components. A relationship describes how nodes are related to each other. Only a few of the orchestration tools on the market have already implemented OASIS TOSCA support. As an example, we refer to Cloudify[14] orchestration tool, which implements a domain-specific language based on TOSCA YAML specification. It has a built-in domain specific language parser, which is responsible for validation of the provided template and mapping to the specified topology entities.

3.3.1 Cloudify Platform

Cloudify[14] is an orchestration tool which allows achieving a smooth transition to the cloud and easy automation of the most complex applications throughout their life cycle. It is possible to organize the creation of the entire cloud infrastructure needed for the application, starting from computing resources to networks and storage devices. Cloudify can deploy applications to the cloud (Amazon EC2, OpenStack, VMWare Vsphere or even bare metal), display progress and perform scaling when necessary. It supports various platforms. Therefore it is possible to choose a provider of cloud services, having the same front end - monitor, manage and scale applications within Cloudify, regardless of the stack, topology, and platform where they are deployed. The sample configuration file is provided below.

```
1 floating_ip :
2   type: cloudify.openstack.nodes.FloatingIP
3   interfaces :
4     cloudify.interfaces.lifecycle :
5       create :
6         inputs :
7           args :
8             floating_network_name: external_network
9
10
11 network :
12   type: cloudify.openstack.nodes.Network
13   properties :
14     resource_id: private_network
15
16
17 subnet :
18   type: cloudify.openstack.nodes.Subnet
```

```

19 properties:
20   resource_id: subnetwork
21 interfaces:
22   cloudify.interfaces.lifecycle:
23     create:
24       inputs:
25         args:
26           cidr: 1.2.3.0/24
27           ip_version: 4
28   cloudify.interfaces.validation:
29     creation:
30       inputs:
31         args:
32           cidr: 1.2.3.0/24
33           ip_version: 4
34 relationships:
35   - target: network
36     type: cloudify.relationships.contained_in
37
38
39 security_group:
40   type: cloudify.openstack.nodes.SecurityGroup
41 properties:
42   resource_id: security_group_name
43   rules:
44     - remote_ip_prefix: 0.0.0.0/0
45       port: 8080
46
47
48 server:
49   type: cloudify.openstack.nodes.Server
50 properties:
51   resource_id: server_name
52 interfaces:
53   cloudify.interfaces.lifecycle:
54     create:
55       inputs:
56         args:
57           image: 8672f4c6-e33d-46f5-b6d8-ebbba12fa02
58           flavor: 101
59   cloudify.interfaces.validation:
60     creation:
61       inputs:
62         args:
63           image: 8672f4c6-e33d-46f5-b6d8-ebbba12fa02
64           flavor: 101

```

```

65 relationships :
66   - target: network
67     type: cloudify.relationships.connected_to
68   - target: subnet
69     type: cloudify.relationships.depends_on
70   - target: floating_ip
71     type: cloudify.openstack.server_connected_to_floating_ip
72   - target: security_group
73     type: cloudify.openstack.server_connected_to_security_group

```

Listing 1: Cloudify TOSCA-based sample configuration

The configuration file defines the services and their topology in the infrastructure.

Cloudify includes support for a variety of additional technologies, including Bash, Chef, OpenStack, Puppet, Windows, and Linux. Cloudify includes a management node, a gateway, which allows access to applications. The management of Cloduify is can be done through GUI, CLI and a REST API.

4 Components and Topology Description

Using the previously defined approaches and technologies, the new architecture of the system was delivered. It was composed by applying decomposition guidelines from Chapter 2 and implemented using the chosen technologies which allow ensuring the required properties of the system. This Chapter provides an overview of the services together with their functionality description and provided interfaces.

N2Sky architecture is presented on the Figure 21.

Functionality can be aggregated into five major upper-level modules :

- User interface module,
- Infrastructure module,
- Monitoring module,
- Computational module,
- Business module.

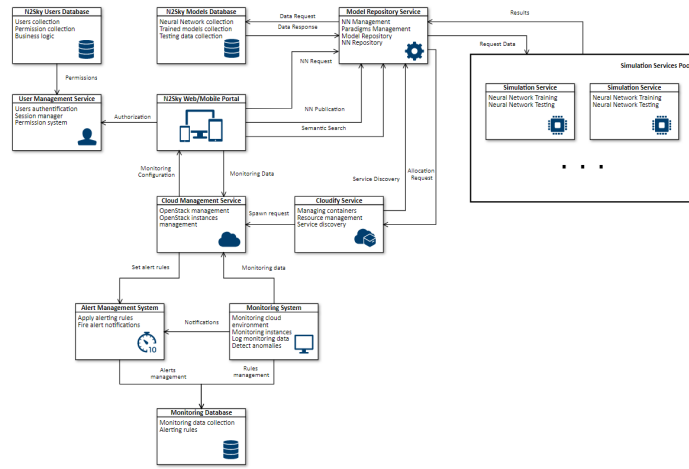


Figure 21: N2SkyC Architecture

4.1 Business Module

The previous revision of the N2Sky system put a high emphasis on the business cases of the application[25]. This domain is mostly out of the scope of the research questions of this thesis. Therefore we limit the business domain by services which are responsible for user management. The services composition of the business module is presented in Figure 22.

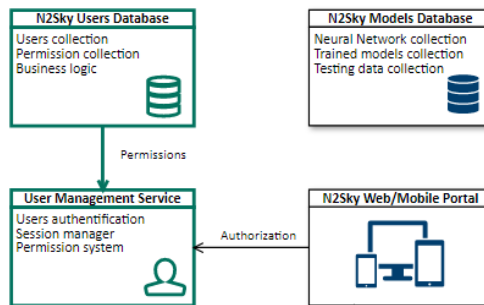


Figure 22: Business Module

The functionality of the module is insured by persistent DB storage and user management service which provides CRUD operations, it's endpoints with description are listed in Table 1.

Table 1: Business module description

Service	Interface	Methods	Description
User management	/users	GET	Getting the list of users
	/user/:username	GET	Getting data of the specific user
	/user/login	POST	Login method
	/user/signup	POST	Registration method
	/user/delete/:username	DELETE	Deleting specified user

4.2 Monitoring Module

This module is responsible for tracking the cloud infrastructure, setting up the alerts for undefined behavior of the system and providing access to log information. The services, which are included into this module, are presented in the Figure 23. Communication between OpenStack internal services and monitoring is performed through the OpenStack REST API. The module consists of several services, which are presented in Table 2.

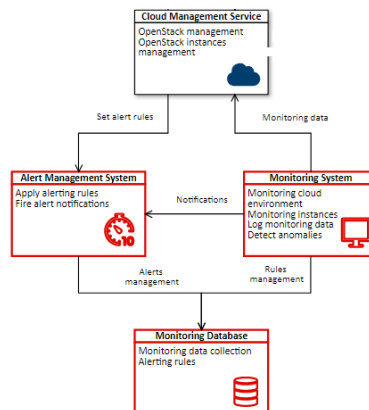


Figure 23: Monitoring Module

Table 2: Monitoring module description

Service	Description
Alert Management	Alert management system is responsible for creating custom alerts on specific conditions, defined by users.
Monitoring System	Monitoring system is responsible for collection data from OpenStack services and running custom defined checks against it.
Monitoring Database	Monitoring Database is used to store monitoring information received from the infrastructure.

4.3 Infrastructure Module

This part of the system consists of two crucial systems, each of which are split further into microservices of smaller size. On the one hand, there are set of OpenStack services, which can be used directly through API.

The second part of this module is Cloudfify management system, which is orchestrating microservices, which are deployed across the cloud. By using orchestration tool, there should be no need for the manual OpenStack deployment, as Cloudfify can automatically deploy necessary services using stored configuration blueprints. Services, which belong to this module are illustrated in the Figure 24.

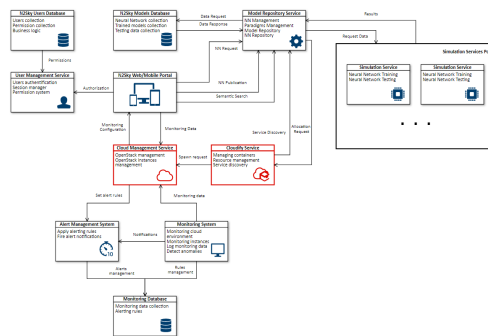


Figure 24: Infrastructure Module

List of the most important interfaces available on the infrastructure services is presented in Table 3.

Table 3: Infrastructure module description

Service	Interface	Method	Description
Cloudify	/api/v3.1/blueprints/:blueprint-id?application_file_name=:blueprint-id.yaml	PUT	Deploys Cloudify configuration
	/api/v3.1/deployments/deployment-id	PUT	Creates a deployment
	/api/v3.1/executions	GET	List all running executions
	/api/v3.1/executions	POST	Creates an execution
	/api/v3.1/node-instances	GET	List all spawned instances
	/api/v3.1/node-instances/{node-instance-id}	GET	Get monitoring information
OpenStack Management	Provides set of API for each OpenStack service		

4.4 Computational Module

The computational module is responsible for supporting the main workflow of the system and can be called the main logic module. It provides interfaces for the main functionality of the platform and provides end-points for neural network management and usage(Figure 25).

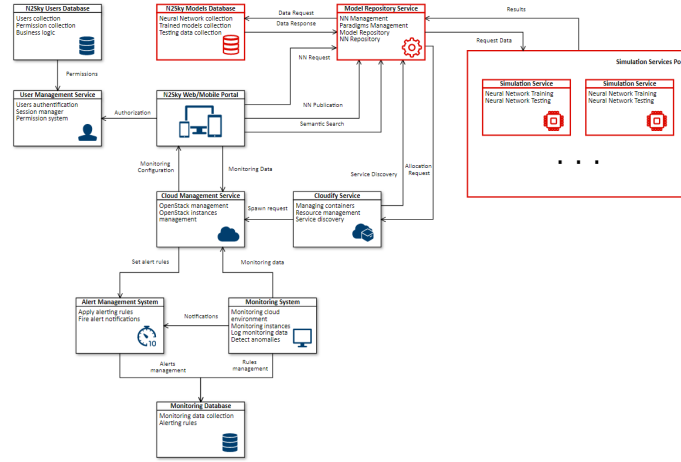


Figure 25: Computational Module

There exist two type of entities:

- Neural network descriptions - which is an object that is containing a description of the topology of a neural network including different parameters as learning rates, activation functions, number of neurons, etc. N2SkyC is designed to use and adopt ViNNsL language,
- Neural network objects - which are representing a state of the instances of neural networks, defined by neural network descriptions. In our application design, each instance of simulation service is deployed by orchestration manager based on the neural network description. After training the neural network, the object is delivered from simulation service and stored in the database.

Each simulation service should provide unified interfaces, which are specified by development guide and is a responsibility of neural network developer. The functionality and end-points of the services are presented in the Table 4.

Details on the requirements for neural network developers are presented in Chapter 5.

Table 4: Computational model description

Service	Interface	Method	Description
Model Repo	/vinns1/description/create	POST	Creating a Neural Network description object
	/vinns1/description/:id	GET	Returning a Neural Network description
	/vinns1/description/run/instance/:id	POST	Run a new instance
	/vinns1/description/:id/instances	GET	List of running Simulation Services
	/dockerhub/:user	GET	Getting Docker images of the user
Simulation	/train	POST	Perform training
	/test	POST	Perform testing
	/logs	GET	Produce logs for a specific Neural Network
Models DB	Stores all relevant Neural Networks data		

Thus, this module of microservices is providing all necessary functionality to allow robust and scalable neural network simulation environment.

4.5 UI Module

The UI module is the main component which enables user-friendly access to the platform services.

N2Sky was developed with a focus on the user-centered design. Front-end architecture was designed and developed by my colleague Andrii Fedorenko[13]. Front-end design fully reflects not only functional and business requirements of the different types of users, but also it is composed in the same manner as other parts of the application. That means that all the front-end services are composed of Docker containers and put into groups, so front-end module can fully benefit from the overall system design. Moreover, some Front-end design decisions were inspired by microservices architecture. Place of the frontend service is right in the center, as it provides all the users with endpoints, which is presented in Figure 26.

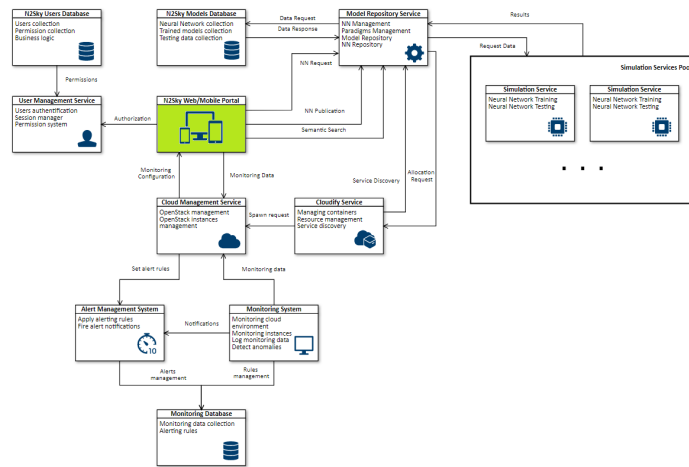


Figure 26: User Interface Module

The Frontend service provides the main way of communicating with an application. Its responsibility can be split into three main categories, according to the business needs of the users:

- Arbitrary user requests - requests to use a certain neural network and experiment with existing solutions,

- Developer requests - requests from neural network developers, who are aiming to upload their solutions,
- Administration requests - requests from the administrator of the N2Sky platform, who is responsible for management of the system.

It is important to mention that services and databases, which were implemented during the N2Sky architecture project are heterogeneous and using different technologies. This was possible by adapting microservices architecture and designing communication in a unified way.

As a result, by implementing architectural changes to both infrastructure and software levels of the N2Sky platform, we achieve higher scalability and make the platform more suitable for usage in a cloud environment. The shift to new technologies, as container base execution environment and orchestration tools for load and deployment management, drastically increased efficiency and scalability of N2Sky neural network platform for future development.

5 Development Guide

This Chapter provides guidelines on environmental setup and neural network and machine learning development frameworks. We use Ubuntu 16.04 Server as a main OS for these guidelines.

5.1 Cloud Infrastructure Guide

As we choose OpenStack as a first cloud infrastructure provider, therefore these guidelines are relevant to the OpenStack distribution.

OpenStack is a quite complex software product, and its full configuration requires some effort, which is not always suitable if we aim for fast development. There exists OpenStack distribution, which is called DevStack, which is specifically created to deploy OpenStack within a shorter time frame. It tracks the same number of repositories as main distribution but requires fewer resources to bring OpenStack up.

We differentiate between two use cases of the DevStack installation, which are a bit different in details. The first option would be to install DevStack on the bare metal and the second option is to install it inside a virtual machine.

5.1.1 Environment Setup

DevStack installation script should be executed with an account, which has sudo privileges. Therefore, it is needed to create a user with right permissions and assign a directory to him.

```
1 sudo useradd -s /bin/bash -d /opt/stack -m stack
2 echo "stack ALL=(ALL) NOPASSWD: ALL" | sudo tee
   /etc/sudoers.d/stack
```

Listing 2: DevStack user creation

5.1.2 DevStack Manual Installation

After cloning a DevStack repository, it is essential to edit configuration file according to the settings of the OS. DevStack configuration is managed by the local.conf file. It consists of several sections, each of which starts with the header of the following format.


```
1 '[[ <phase> '|' <config-file-name> ']]'
```

Listing 3: DevStack configuration header specification

There are several phases, which will be executed during the OpenStack installation:

- local - loads the environmental variables specified in local header
- pre-install - will be executed after system packages installation, but before any service,
- install - will be executed, after all, services are pulled from repositories and installed,
- post-config - will load services configuration right before the start of the services,
- extra - will be executed after all services are installed and run.

The first section which is defining global settings of the DevStack is called local, and it holds the network configuration along with security settings.

```
1 [[ local | localrc ]]  
2  
3 ADMIN_PASSWORD=password  
4 DATABASE_PASSWORD=password  
5 RABBIT_PASSWORD=password  
6 SERVICE_PASSWORD=password  
7  
8 HOST_IP=172.16.116.5  
9 IP_VERSION=4  
10 Q_USE_SECGROUP=TRUE  
11 FLOATING_RANGE="172.16.116.0/24"  
12 FIXED_RANGE="10.0.0.0/24"  
13 Q_FLOATING_ALLOCATION_POOL=start=172.16.116.65,end=172.16.116.126  
14 PUBLIC_NETWORK_GATEWAY="172.16.116.2"  
15 PUBLIC_INTERFACE=enp0s3
```

Listing 4: DevStack main configuration file

It is important to configure the networking component of the DevStack properly so that the instances will be available from a host machine and other machines on the network. To ensure this, it is important to setup host IP address the same as the machine, where DevStack is going to be installed. The

floating range is used to determine, which pool of IP addresses will be used to allocate IP addresses to the spawned instances. It is possible to manually set up the pool of the addresses. Public interface and public network are used to specify which network adapter should be used to connect to the external network and which gateway to use to access the Internet.

Next useful setting is repositories and branches for OpenStack services. They can be configured separately, to use a specific version of the service.

```
1 NOVA_REPO=\$GIT_BASE/openstack/nova.git
2 NOVA_BRANCH=master
```

Listing 5: DevStack components repository specification

To enable monitoring capabilities, it is possible to activate a Vitrage plugin, which is responsible for pooling the monitoring information from the OpenStack services. To do that, we need to add following lines to the configuration file.

```
1 [[ local | localrc ]]
2 enable_plugin vitrage https://git.openstack.org/openstack/vitrage
3
4 [[ post-config | \$NOVA.CONF ]]
5 [DEFAULT]
6 notification_topics = notifications , vitrage_notifications
7 notification_driver=messagingv2
8
9 [[ post-config | \$NEUTRON.CONF ]]
10 [DEFAULT]
11 notification_topics = notifications , vitrage_notifications
12 notification_driver=messagingv2
13
14 [[ post-config | \$CINDER.CONF ]]
15 [DEFAULT]
16 notification_topics = notifications , vitrage_notifications
17 notification_driver=messagingv2
18
19 [[ post-config | \$HEAT.CONF ]]
20 [DEFAULT]
21 notification_topics = notifications , vitrage_notifications
22 notification_driver=messagingv2
23
24 [[ post-config | \$AODH.CONF ]]
25 [oslo_messaging_notifications]
26 driver = messagingv2
```

```
27 topics = notifications , vitrage_notifications
```

Listing 6: DevStack enabling monitoring of services

This will enable notifications push from services to Vitrage monitoring service.

To enable support for Docker hypervisor, to be able to deploy containers without a VM following configuration should be applied. First it is needed to enable docker plugin.

```
1 rm -rf /opt/stack/nova-docker
2 sudo mkdir -p /opt/stack
3 sudo git clone https://git.openstack.org/openstack/nova-docker
  /opt/stack/nova-docker
4 cd /opt/stack/nova-docker
5 sudo pip install .
```

Listing 7: DevStack download Docker hypervisor

After that, we should configure OpenStack to include docker hypervisor in the distribution.

```
1 echo "$GLANCE_APLCONF" >> local.conf
2 echo "[DEFAULT]" >> local.conf
3 echo container_formats=ami, ari, aki, bare, ovf, ova, docker >>
  local.conf
4 echo >> local.conf
5 echo "$NOVA_CONF" >> local.conf
6 echo "[DEFAULT]" >> local.conf
7 echo compute_driver=novadocker.virt.docker.DockerDriver >>
  local.conf
8 echo >> local.conf
```

Listing 8: DevStack enable Docker hypervisor support

To bootstrap DevStack after configuration, only one command should be executed, keep attention that it is executed by a user with right permissions.

```
1 ./stack.sh
```

Listing 9: DevStack bootstrap script

It is important to mention that DevStack installation is not persistent upon restart - that means that all data and settings including stack volumes, bridge configuration, and routing settings will be lost. Currently, there is no way to prevent this behavior. To bring cloud infrastructure online after reboot it is enough to run stack script again.

5.1.3 DevStack - Virtual Environment

The second option on how to create a development cloud environment is to use a virtual machine. The challenges in this method are that there will be nested virtualization involved and network settings should be properly configured to allow traffic and connectivity between all the nodes. Two technologies, which are used in such type of deployment are:

- VirtualBox - it is easier to deal with dependencies in case if something goes wrong,
- Vagrant - which provides a robust way to configure virtual environments.

Both of these distributions can be installed in a straightforward way.

```
1 sudo apt-get install virtualbox
2 sudo apt-get install vagrant
```

Listing 10: Cloudify environment setup

As Vagrant is providing a configuration file to unify the deployment of virtual machines, we should create one for DevStack deployment.

```
1 # -*- mode: ruby -*-
2 # vi: set ft=ruby :
3
4 config.vm.box = "bento/ubuntu-16.04"
5 config.vm.hostname = 'devstack'
6
7 config.vm.network "forwarded_port", guest: 80, host: 80
8 config.vm.network "forwarded_port", guest: 5000, host: 5000
9 config.vm.network "forwarded_port", guest: 9696, host: 9696
10 config.vm.network "forwarded_port", guest: 8774, host: 8774
11 config.vm.network "forwarded_port", guest: 35357, host: 35357
12 config.vm.network "private_network", ip: "171.15.19.31"
13
14 config.vm.provider :virtualbox do |v|
15   v.customize ["modifyvm", :id, "--memory", 8096]
16   v.customize ["modifyvm", :id, "--cpus", 4]
17 end
18
19 config.vm.provision "shell", path: "devstack_install.sh"
20
21 end
```

Listing 11: Vagrant VM settings

In this configuration file, we specify, which image should be installed in the virtual machine, which network routing rules should apply to it and which virtualization provider should be used. In this case, it is VirtualBox provider with memory limitation of 8096Mb and four cores.

Also, it is to specify how DevStack will be installed. To provide Vagrant with DevStack configuration, we specify that certain script should be run after virtual machine deployment.

```
1#!/bin/bash
2
3echo "Spinning virtual machine..."
4echo "Installing Git..."
5sudo apt-get install git
6
7echo "DevStack installation setup..."
8sudo useradd -s /bin/bash -d /opt/stack -m stack
9echo "stack ALL=(ALL) NOPASSWD: ALL" | sudo tee
   /etc/sudoers.d/stack
10su - stack -c "cd /opt/stack && git clone
   https://git.openstack.org/openstack-dev/devstack"
11su - stack -c "cat > /opt/stack/devstack/local.conf << END
12[[ local|localrc ]]
13enable_plugin vitrage https://git.openstack.org/openstack/vitrage
14
15HOST_IP=$(ip addr | grep 'state UP' -A2 | tail -n1 | awk '{print
   \$2}' | cut -f1 -d'/')
16HOST_IP_IFACE=eth1
17FLAT_INTERFACE=br100
18PUBLIC_INTERFACE=eth1
19FLOATING_RANGE=192.168.56.224/27
20
21DATABASE_PASSWORD=password
22RABBIT_PASSWORD=password
23SERVICE_TOKEN=password
24SERVICE_PASSWORD=password
25ADMIN_PASSWORD=password
26
27[[ post-config|\$NOVA_CONF ]]
28[DEFAULT]
29notification_topics = notifications , vitrage_notifications
30notification_driver=messagingv2
31
32[[ post-config|\$NEUTRON_CONF ]]
33[DEFAULT]
34notification_topics = notifications , vitrage_notifications
```

```

35 notification_driver=messagingv2
36
37 [[ post-config |\$CINDER_CONF ]]
38 [DEFAULT]
39 notification_topics = notifications , vitrage_notifications
40 notification_driver=messagingv2
41
42 [[ post-config |\$HEAT_CONF ]]
43 [DEFAULT]
44 notification_topics = notifications , vitrage_notifications
45 notification_driver=messagingv2
46
47 [[ post-config |\$AODH_CONF ]]
48 [oslo_messaging_notifications]
49 driver = messagingv2
50 topics = notifications , vitrage_notifications
51 END"
52
53 echo "Installing DevStack..."
54 su - stack -c "cd /opt/stack/devstack && ./stack.sh"
55
56 end

```

Listing 12: DevStack VM configuration

This configuration file is slightly different from the installation on the bare metal, as a virtual machine will have different network topology. Therefore we need to dynamically specify the IP address of the host and to provide the script with information on which network interfaces are used in the virtual machine. After configuration OpenStack can be bootstrapped in the virtual environment with the following command.

```
1 vagrant up
```

Listing 13: Spinning Vagrant VM

If everything was finished successfully, following message will appear, and OpenStack will be available at the specified address.

```

1 =====
2 DevStack Component Timing
3 (times are in seconds)
4 =====
5 run_process           12
6 test_with_retry       2
7 apt-get-update       9
8 pip_install           271

```

```

9 osc 103
10 wait_for_service 12
11 git_timed 255
12 dbsync 16
13 apt-get 126
14 -----
15 Unaccounted time 271
16 =====
17 Total runtime 1077
18
19 This is your host IP address: 171.15.19.31
20 This is your host IPv6 address: fe80::a00:27ff:fe85:f57f
21 Horizon is now available at http://171.15.19.31/dashboard
22 Keystone is serving at http://171.15.19.31/identity/
23 The default users are: admin and demo

```

Listing 14: Sample DevStack installation output

Last preparations before actual usage are enabling OpenStack CLI and changing the default security policy to be able to remotely control instances, deployed in the cloud. To get necessary privileges on the OpenStack CLI, it is needed to identify a user in Keystone service. To do that, credentials should be provided as environmental variables. For convenience, the following file can be used.

```

1#!/usr/bin/env bash
2export OS_AUTH_URL=http://171.15.19.31/identity/v3
3export OS_PROJECT_ID=1d9a7fa2144a4df7a76334db1786ea12
4export OS_PROJECT_NAME="admin"
5export OS_USER_DOMAIN_NAME="Default"
6if [ -z "\$OS_USER_DOMAIN_NAME" ]; then unset
    OS_USER_DOMAIN_NAME; fi
7export OS_PROJECT_DOMAIN_ID="default"
8if [ -z "\$OS_PROJECT_DOMAIN_ID" ]; then unset
    OS_PROJECT_DOMAIN_ID; fi
9unset OS_TENANT_ID
10unset OS_TENANT_NAME
11export OS_USERNAME="admin"
12echo "Please enter your OpenStack Password for project
    \$OS_PROJECT_NAME as user \$OS_USERNAME: "
13read -sr OS_PASSWORD_INPUT
14export OS_PASSWORD=\$OS_PASSWORD_INPUT
15export OS_REGION_NAME="RegionOne"
16if [ -z "\$OS_REGION_NAME" ]; then unset OS_REGION_NAME; fi
17export OS_INTERFACE=public

```

```
18 export OS_IDENTITY_API_VERSION=3
```

Listing 15: Keystone configuration

The settings will depend on the information provided to the DevStack configuration file before deployment.

To open necessary ports and to allow traffic, following rules should be added to the default security group.

```
1 nova secgroup-add-rule SECURITY_GROUP_NAME tcp 22 22 0.0.0.0/0
2 nova secgroup-add-rule SECURITY_GROUP_NAME icmp -1 -1 0.0.0.0/0
```

Listing 16: Configuring cloud security group

OpenStack can also be configured through the GUI, which is available on the specified address. The GUI presented in Figure 27.

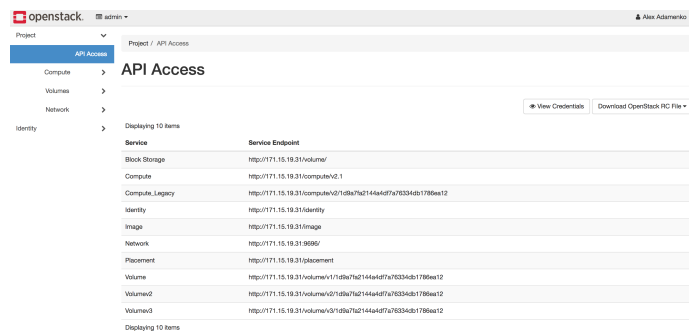


Figure 27: OpenStack GUI

After successful installation, OpenStack can be used as a cloud infrastructure provider for the N2Sky platform.

5.2 Orchestration Management Guide

In the current implementation, Cloudify 3.3.1 is used. It is quite outdated, although Cloudify decided to stop providing support for the older version and switch the business model to the commercial one. Cloudify Manager will be deployed on the node within the OpenStack, and it will be accessible through the GUI and CLI. It is important that OpenStack and host machine networks are configured properly as Cloudify manager will require the Internet access while bootstrapping.

The first requirement is to install Python 2.7. It is recommended to use a virtual environment to isolate the Python distribution from the OS in case of troubles. Cloudify installation scripts also require virtualenv, in another case, it will be needed to use sudo. Python and its package manager tool can be installed simply by following commands.

```
1 install python2.7 python-pip
```

Listing 17: Installing Python distribution

The second requirement is Cloudify CLI, which will communicate with OpenStack and enable deployment of the virtual machine with Cloudify Manager. Following code is used to install Cloudify CLI and configuration scripts. It is needed to enable virtual environment before executing following commands.

```
1 virtualenv cloudify
2 pip install 'cloudify==3.3.1'
```

Listing 18: Installing Cloudify CLI in virtual environment

After Cloudify CLI is installed, it is needed to pull out the configuration files, which are required by Cloudify bootstrap process.

```
1 git clone -b 3.3-build \
2 https://github.com/cloudify-cosmo/cloudify-manager-blueprints.git
```

Listing 19: Fetching Cloudify configuration files

This repository consists of several configuration files for different cloud infrastructure providers, including OpenStack. The bootstrap script needs two files to be provided. First one is topology description file, which explains how Cloudify Manager should be instantiated in the OpenStack. There is no need to change it, except cloud infrastructure is not configured in a default manner. The second file is providing inputs for the first one, consisting of specific setting of the cloud. The following code shows a possible configuration of the Cloudify inputs file.

```
1 keystone_username: 'demo'
2 keystone_password: 'password'
3 keystone_tenant_name: 'demo'
4 keystone_url: 'http://171.15.19.31/identity/v2.0/'
5 region: ''
6
7 use_existing_manager_keypair: false
8 use_existing_agent_keypair: false
```

```

9
10 ssh_key_filename: ~/.ssh/cloudify-manager.pem
11 agent_private_key_path: ~/.ssh/cloudify-agent.pem
12
13 manager_public_key_name: 'cloudify-manager'
14 agent_public_key_name: 'cloudify-agent'
15
16 image_id: '4cb4e4ee-c53f-49f2-9e9d-baa800bc9bf6'
17 flavor_id: 'cloud1'
18
19 external_network_name: 'public'

```

Listing 20: Cloudify Management input specification

Although Cloudify is providing OpenStack support most of the times, it is not compliant with the last changes. Cloudify is using an outdated version of the Keystone OpenStack component, and it is needed to specify it explicitly. To do that, some changes need to be applied to the credentials OpenStack authentication file - specifying that user uses the second version of the Keystone.

```

1 #!/usr/bin/env bash
2 export OS_AUTH_URL=http://171.15.19.31/identity/v2
3 export OS_PROJECT_ID=1d9a7fa2144a4df7a76334db1786ea12
4 export OS_PROJECT_NAME="admin"
5 export OS_USER_DOMAIN_NAME="Default"
6 if [ -z "\$OS_USER_DOMAIN_NAME" ]; then unset
   OS_USER_DOMAIN_NAME; fi
7 export OS_PROJECT_DOMAIN_ID="default"
8 if [ -z "\$OS_PROJECT_DOMAIN_ID" ]; then unset
   OS_PROJECT_DOMAIN_ID; fi
9 unset OS_TENANT_ID
10 unset OS_TENANT_NAME
11 export OS_USERNAME="admin"
12 echo "Please enter your OpenStack Password for project
   \$OS_PROJECT_NAME as user \$OS_USERNAME: "
13 read -sr OS_PASSWORD_INPUT
14 export OS_PASSWORD=\$OS_PASSWORD_INPUT
15 export OS_REGION_NAME="RegionOne"
16 if [ -z "\$OS_REGION_NAME" ]; then unset OS_REGION_NAME; fi
17 export OS_INTERFACE=public
18 export OS_IDENTITY_API_VERSION=3

```

Listing 21: OpenStack Authentication for Cloudify

After the process of spawning the virtual machine and installing Cloudify Manager, the following message will appear.

```
1 installing cloudify-ui...
2 cloudify-ui installation successful.
3 deploying cloudify agents
4 cloudify agents installation successful.
5 bootstrapping complete
6 management server is up at 10.10.10.227 (is now set as the
   default management server)
```

Listing 22: Cloudify Bootstrap

After Cloudify manager is installed, we can use it as an orchestration tool for instances, deployed in the cloud. It will be configured to be used in a specific cloud environment and provide functionality, described in the documentation[8].

5.3 Neural Network Development Guide

One of the main goals of the new architecture was a shift from Java language to provide a more flexible environment for scientific developers. It was achieved by introducing containers technology, where each neural network application can be developed in any language independently, and therefore there will be no language limitations. As a consequence, after simulation service was completely separated from other logic of the application, each neural network application will still need to provide certain guarantees on its interfaces to be able to communicate with other management components of the system.

Thus, neural network creation guidelines were developed. Technically, each neural network model should be an application, which is packed into Docker container with all the necessary dependencies. In this way, we will be able to provide all necessary guarantees, specifically when this container will be spawned, it will be able to be used by end-users.

To make the development process as simple and as flexible as possible, we decided to bring only several requirements:

1. Each machine learning or neural network application should be a web-service,
2. Each application should provide train and test endpoints,
3. Each application should be packed into a Docker container with all the libraries and dependencies,

4. Each application should serialize trained model and respond it after training phase,
5. Developers should provide data format specification - description on which data formats are accepted by their service,
6. It is preferred, that each application is designed and described according to the ViNNSL Specification language.

It becomes possible to develop in any programming language and use any libraries with this approach. These requirements are only ones, and if fulfilled the applications can fully utilize benefits provided by architecture and infrastructure of the platform.

This guide refers to the backpropagation neural network robust implementation in Python using Keras framework, Tensorflow computational library, HDF5/JSON serialization, and Flask web framework.

First, a developer should be familiar with ViNNSL specification language, as data flow between the application and other components is performed based on the ViNNSL template.

```
1 let parameters = new Schema({
2     parameter: String ,
3     defaultValue: String ,
4     possibleValues: [String]
5 });
6
7 let connectionsShortcuts = {
8     from: String ,
9     to: String ,
10    isFullConnected: Boolean
11 };
12
13 let inputOutputLayer = new Schema({
14     nodesId: [String] ,
15     amount: Number
16 });
17
18 let hiddenLayer = new Schema({
19     id: String ,
20     nodesId: [String] ,
21     amount: Number
22 });
23
24
```

```

25 let endpoints = new Schema({
26   name: String ,
27   endpoint: String
28 });
29
30 let image = new Schema({
31   imageType: String ,
32   endpoint: String
33 });
34
35 let VINNSL_Description_NN = new Schema({
36   metadata: {
37     name: {type: String},
38     description: {type: String},
39     paradigm: {type: String},
40     createdOn: Date ,
41     version: {
42       major: {type: String},
43       minor: {type: String}
44     }
45   },
46   creator: {
47     name: {type: String},
48     contact: {type: String}
49   },
50   problemDomain: {
51     propagationType: {
52       propType: String ,
53       learningType: String
54     },
55     applicationField: [String],
56     problemType: {type: String},
57     networkType: {type: String}
58   },
59   endpoints: [endpoints],
60   structure: {
61     inputLayer: inputOutputLayer ,
62     hiddenLayer: [hiddenLayer],
63     outputLayer: inputOutputLayer
64   },
65   connections: {
66     fullyConnected: {
67       isConnected: Boolean
68     },
69     shortcuts: {
70       isConnected: Boolean ,

```

```

71         connections: [connectionsShortcuts]
72     }
73 },
74 parameters: {
75     input: [parameters],
76     output: String
77 },
78 data: {
79     description: {type: String},
80     tableDescription: {type: String},
81     fileDescription: {type: String, Default: "no file needed"}
82 },
83 executionEnvironment: {
84     lastRun: Date,
85     isRunning: {type: Boolean, Default: false},
86     hardware: String,
87     isPublic: Boolean,
88     image: image
89 }
90 });

```

Listing 23: ViNNSL Specification Adaptation

Currently, this specification language is defined to be able to describe neural network topologies, but in the future, it should be able to reflect any machine learning algorithm. Mostly, there are three crucial components: training data, labels of the training data and parameters of the model. Neural networks are more complex in this case, as they imply complex topologies with several heterogeneous layers. Using such schema as a reference, the developer should be able to produce a mapping between ViNNSL specification and his neural network implementation. The example of such mapping can look like this.

```

1 def parse_vinnsl(vinnsl):
2
3     nn_structure = {}
4
5     parsed_json = json.loads(vinnsl)
6     parameters = parsed_json['parameters']['input']
7     structure = parsed_json['structure']
8
9     learning_rate = parameters[0]['defaultValue']
10    biasInput = parameters[1]['defaultValue']
11    biasHidden = parameters[2]['defaultValue']
12    momentum = parameters[3]['defaultValue']
13    activationFunctionOutput = parameters[4]['defaultValue']

```

```

14 activationFunctionHidden = parameters[5]['defaultValue']
15 threshold = parameters[6]['defaultValue']
16 target_data = parameters[7]['defaultValue']
17 number_epochs = parameters[8]['defaultValue']
18
19 connections = parsed_json['connections']
20
21 fully_connected = connections['fullyConnected']['isConnected']
22 shortcuts = connections['shortcuts']
23 shortcuts_connections = shortcuts['connections']
24
25 print(fully_connected)
26
27 input_layer = structure['inputLayer']
28 input_neurons = input_layer['amount']
29
30 outputLayer = structure['outputLayer']
31 output_neurons = outputLayer['amount']
32
33 hidden_layers = structure['hiddenLayer']
34 hidden_layers_neurons = []
35
36 for layer in hidden_layers:
37     hidden_layers_neurons.append(layer['amount'])
38
39 nn_structure['input_neurons'] = input_neurons
40 nn_structure['output_neurons'] = output_neurons
41 nn_structure['hidden_layers'] = hidden_layers_neurons
42
43 nn_structure['learning_rate'] = learning_rate
44 nn_structure['biasInput'] = biasInput
45 nn_structure['biasHidden'] = biasHidden
46 nn_structure['momentum'] = momentum
47 nn_structure['activationFunctionOutput'] =
activationFunctionOutput
48 nn_structure['activationFunctionHidden'] =
activationFunctionHidden
49 nn_structure['threshold'] = threshold
50 nn_structure['target_data'] = target_data
51 nn_structure['number_epochs'] = number_epochs
52
53 return nn_structure

```

Listing 24: ViNNSL Parser

This information which is contained in ViNNSL should be enough to reproduce any single neural network topology. Therefore, the network can be

created, and training can be performed.

```
1 def train_model(training_data , description , model_id):
2
3     model = Sequential()
4
5     dimensions = training_data[0].size
6
7     input_layer = description[ 'input_neurons' ]
8     output_layer = description[ 'output_neurons' ]
9     hidden_layers = description[ 'hidden_layers' ]
10    target_data = description[ 'target_data' ]
11    number_epochs = int( description[ 'number_epochs' ] )
12
13    target_data = np.array( eval( target_data ), "float32" )
14
15    learning_rate = float( description[ 'learning_rate' ] )
16    momentum = float( description[ 'momentum' ] )
17    activationFunctionOutput =
18    description[ 'activationFunctionOutput' ]
19    activationFunctionHidden =
20    description[ 'activationFunctionHidden' ]
21
22    model.add( Dense( input_layer , input_dim=dimensions ,
23                    activation=activationFunctionHidden ) )
24    for layer in hidden_layers:
25        model.add( Dense( layer , input_dim=dimensions ,
26                        activation=activationFunctionHidden ) )
27    model.add( Dense( output_layer ,
28                    activation=activationFunctionOutput ) )
29
30    sgd = optimizers.SGD( lr=learning_rate , decay=1e-6,
31                          momentum=momentum, nesterov=False )
32    model.compile( loss='mean_squared_error' , optimizer=sgd ,
33                 metrics=[ 'binary_accuracy' ] )
34
35    model.fit( training_data , testing_data , epochs=epoche ,
36             verbose=2, callbacks=[remote] )
37
38    model.save( 'models/my_model.h5' )
39
40    return model
```

Listing 25: Generic backpropogation Neural Network

To be integrated into the system, a developer should provide two interfaces, which are expecting to receive ViNNNSL compliant file both for training and

for testing. Therefore, an application should be able to communicate over the network and provide these two endpoints.

```
1 def parse_vinnsl(vinnsl):
2
3 @app.route('/train', methods=['GET', 'POST'])
4 def train():
5     vinnsl_description = request.form['vinnsl']
6     training_data = request.form['training_data']
7     model_id = request.form['model_id']
8
9     description = vinnsl_decoder.parse_vinnsl(vinnsl_description)
10
11     model = nn.train_model(description, model_id)
12     model.save('models/my_model.h5')
13     proc = subprocess.Popen(['python',
14                             'serialization/encoder.py', 'models/my_model.h5'],
15                             stdout=subprocess.PIPE,
16                             stderr=subprocess.STDOUT)
17     return proc.communicate()[0]
18
19 @app.route('/test', methods=['POST'])
20 def test():
21     model = request.form['model']
22
23     with io.open('models/model.json', 'w', encoding='utf-8') as f:
24         f.write(model)
25
26     if os.path.exists('models/model.h5'):
27         os.remove('models/model.h5')
28
29     p = subprocess.Popen(['python', 'serialization/decoder.py',
30                         'models/model.json', 'models/model.h5'],
31                         stdout=subprocess.PIPE,
32                         stderr=subprocess.STDOUT)
33
34     p_status = p.wait()
35
36     model = load_model('models/model.h5')
37     predictions = model.predict(testing_data).round()
38
39     return str(predictions)
```

Listing 26: Server endpoints

Train and Test endpoints are responsible for performing training and testing of the neural network. Although there are no requirements on the implementation language or paradigm itself, there is a specification of these inter-

faces which should be met.

1. Train - accepts training data along with a ViNNsL description of the network and returns a serialized model in JSON format,
2. Test - accepts testing data alongside serialized model and responds with predictions data.

As developers will need to serialize and de-serialize their models, they need to choose proper method or library for that. That choice will fully depend on the developer himself, as different languages and neural network libraries are producing different output. Taking the responsibility of format mapping out of the system, it becomes possible to work with any service which is compliant with requirements of the system, which are platform independent and easy to meet.

As an example of Keras model serialization, an N2Sky backpropagation network is serialized using HDF5 format and transformed to JSON.

After the training phase, a compiled model is returned to our platform and stored for future testing by other users. The process of creating or adapting an existing application to such requirements is quite straightforward and doesn't require much re-factoring.

As the last step, any neural network developer should be able to produce a Docker artifact to upload his application to the N2SkyC system. Docker file is a configuration file, which consists of all the dependencies of the application. The Dockerfile for the neural network, which was taken as an example will look like this.

```
1FROM debian:8
2
3ENV LANG=C.UTF-8 LC_ALL=C.UTF-8
4
5RUN apt-get update --fix-missing && apt-get install -y wget bzip2
   ca-certificates \
6   libglib2.0-0 libxext6 libsm6 libxrender1 \
7   git mercurial subversion
8
9RUN echo 'export PATH=/opt/conda/bin:\$PATH' >
   /etc/profile.d/conda.sh && \
10  wget --quiet \
11  https://repo.continuum.io/archive/Anaconda2-5.0.1-Linux-x86_64.sh \
12  -O ~/anaconda.sh && \
```

```

13 /bin/bash ~/anaconda.sh -b -p /opt/conda && \
14 rm ~/anaconda.sh
15
16 RUN apt-get install -y curl grep sed dpkg && \
17     TINI_VERSION='curl
18     https://github.com/krallin/tini/releases/latest | grep -o
19     "/v.*\'" | sed 's:^\..\.(\.*\).$:\1:' && \
20     curl -L
21     "https://github.com/krallin/tini/\
22     releases/download/v\${TINI_VERSION}/ \
23     tini-\${TINI_VERSION}.deb" > tini.deb && \
24     dpkg -i tini.deb && \
25     rm tini.deb && \
26     apt-get clean
27 ENV PATH /opt/conda/bin:\PATH
28 COPY backprop /root/
29 EXPOSE 6006 8888 5000
30
31 WORKDIR "/root"
32
33 RUN conda install libgcc
34 RUN pip install keras
35 RUN pip install -Iv tensorflow==1.3
36 RUN pip install h5json
37
38 CMD ["python", "server.py"]

```

Listing 27: Dockerfile example configuration

Dockerfile specifies the sequence of commands which will be executed when the container will be deployed. This Dockerfile can be deployed to the open community repositories, stored locally or stored in any remote repositories. When neural network developer wants to submit his model to the N2Sky platform, he will be prompted to provide a path to the Dockerfile of his application. Based on the information provided, the N2Sky system will generate necessary configuration files to deploy the application to the cloud infrastructure and provide access to it.

This guide covers all the requirements for the end developer who wants to participate in the N2Sky community. Details of availability and GUI use-cases are designed and developed by my colleague Andrii Fedorenko. We believe that such requirements will not be limiting factors to the developers as they are receiving full flexibility and language independence in comparison to other

systems or the previous revision of N2Sky.

6 User Guide

As it was mentioned before, N2SkyC UI is the main way to interact with the platform. User Guide presents description on how the execution flow is organized together with typical use-cases.

The first component a user can interact with is user management component. In the UI it is presented as login and register form respectively. It is presented in Figure 28.

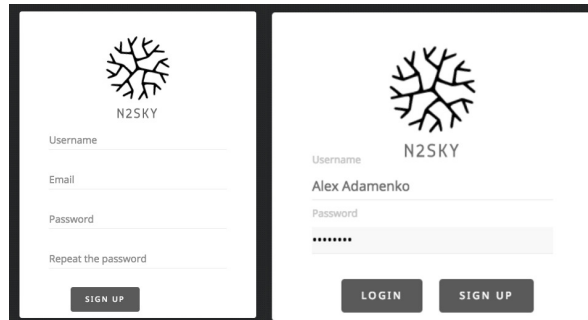


Figure 28: User login/register forms

These forms are submitted to the User Management Service, processed or stored in the respective database in JSON format according to the schema.

Upon deployment, the N2SkyC platform doesn't include any neural network paradigms. Therefore, no functionality will be available. On this step, it is up to developers or administrators of the system to provide neural network descriptions. The interface, provided by the Model Repository Service allows uploading a new neural network description with ViNNSL specification language support. This interface is reflected in the UI Form presented in Figure 29.

UPLOAD NETWORK IN VINNSL FORMAT

Fields which will be overridden:

Creator: Alex Adamenko
 Not running by default
 Image to Docker image

alexadamenko

alexadamenko/app

Choose file XOR Network Internal.json

```

"image": {
  "imageType": null,
  "_id": "5ab4f596b48a3500bf45f1b0"
},
"data": {
  "description": "[X] for each output",
  "tableDescription": "[X],[X]",
  "fileDescription": "TXT"
}

```

CREATE | +

Figure 29: Paradigm upload

This form contains important information for deployment of the new neural network paradigm. The developer should provide information about a DockerHub account, where Docker image of the neural network is stored and a ViNNsL network description, specifying data format for training and testing data and network specific settings.

After the ViNNsL description is validated, Model Repository Service produces and stores a YAML topology description for Cloudify manager, which is used to deploy a Simulation Service on the OpenStack. This description can be used later, to spawn an instance of the neural network.

As soon as neural network description is provided and stored, new neural network objects can be created from it, using the provided specification. All the provided and validated descriptions are stored in the Models database.

Next step would be to spawn a new neural network deployment blueprint using one of the previously specified neural network paradigm descriptions. It is performed by requesting a new instance from Model Repository Service. The UI form which is reflecting this step is in Figure 30.

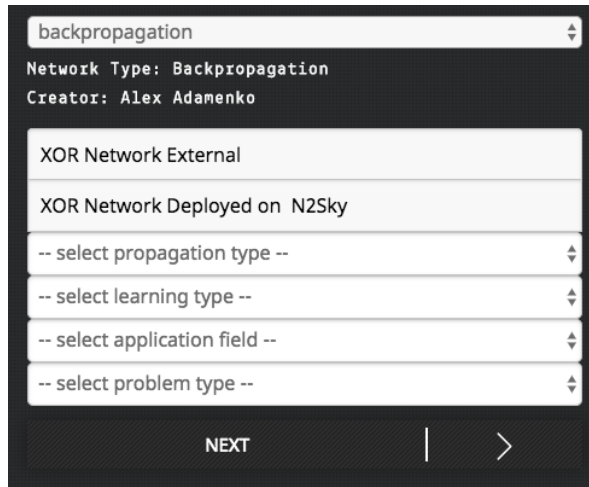


Figure 30: Defining neural network deployment

This form represents POST request data payload, which is required to create a new neural network deployment blueprint. The list of available neural network paradigms is pulled out of the Models database and presented to a user. According to the ViNNNSL specification submitted by neural network developer, a user should provide a system with meta-information on parameters, relevant to the chosen neural network paradigm. It is important to mention, that information provided on these steps is stored in ViNNNSL itself as a part of it.

As paradigm description only specifies neural network specific properties, next step would be to generate a topology description, which is a concrete number of neurons and connections between them. These properties are specified in ViNNNSL and mapped to the responsive UI, which is presented in Figure 31.

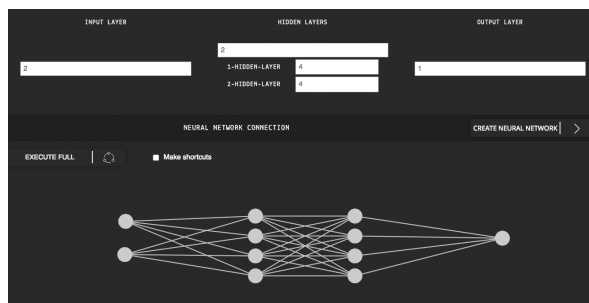


Figure 31: Neural network editor

Currently, the platform supports a multilayer perceptron topology specifi-

cation, but it is only temporary UI limitation, as other N2SkyC components are designed in a way to support any neural networks and machine learning algorithms. To provide users with GUI interface for all the type of other neural network specifications as long short-term memory blocks or recurrent neural network additional interfaces modules should be developed. As a result of this step, system updates the ViNNsL description of this neural object, produced from paradigm and stores it in the Model database.

After creation of neural network blueprint with all necessary information, next step is to request a new running instance of the network. There are two possibilities to run a new instance: either it is run on the N2SkyC cloud infrastructure, or it is already deployed externally and available on the Internet. The form, which is responsible for creating a request to Model Repository Service is presented in the Figure 32.

RUN INSTANCE	
<input type="radio"/> Deploy on N2sky environment <input checked="" type="radio"/> Instance already deployed on external environment	
<input type="text" value="http://35.158.150.84"/>	
Endpoint for training	http://35.158.150.84/train
Endpoint for testing	http://35.158.150.84/test
<input type="button" value="RUN INSTANCE"/> <input type="button" value="+"/>	

Figure 32: New instance deployment

In case service is deployed on the running N2SkyC platform, no additional actions are required. Model Repository Service forms a PUT HTTP request, which contains TOSCA topology description of the service and sends it to Cloudify Service to spawn an instance. Cloudify Manager creates a new deployment and executes it on the OpenStack infrastructure. As soon as the new instance is spawned, Cloudify responds with it's IP address which is then stored in the Models Database by Repository Service. In the second case, a user needs to provide an address, where compliant with the N2SkyC platform service is running.

At the end of these steps, we received a running neural network instance, which can be used by other users of the system for training or testing tasks. Endpoints of these objects are stored in the Models Database, and they are available for requests. Training and testing input requirements are specified by concrete neural network paradigm and can vary.

At this step, such neural network is not trained, and it can't be used. First it should be trained to rebalance the weights between neurons. It is performed by sending a POST request with the required ViNNNSL payload on the /train endpoint of the corresponding neural network service. The UI for forming such request is presented in Figure 33.

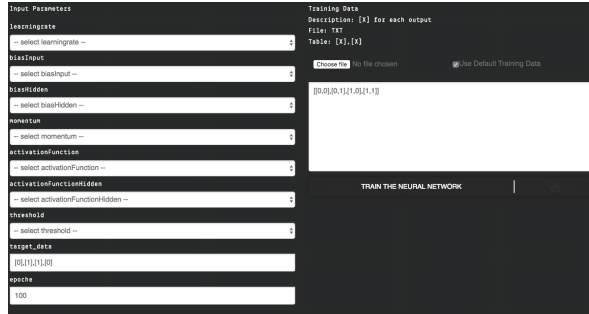


Figure 33: Training request

POST HTTP request is sent by Model Repository Service using the endpoint stored in the ViNNNSL description of the neural network instance. Neural Network performs training and replies with training information alongside serialized model, which is stored in the Models Database. The sample training result is presented in Figure 34.

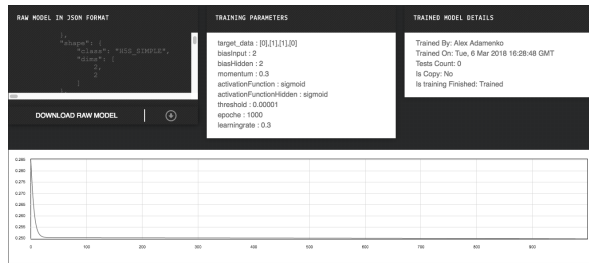


Figure 34: Training result

After this step, we have trained serialized model stored in the Models Database, which can be used by other users for testing. If needed, the neural network deployment blueprint can be copied by a user and deployed as another instance to retrain the network. A user can also perform testing by sending a POST request with testing data payload on any of the running neural network services, which are already trained. Sample UI form for testing is presented in Figure 35.

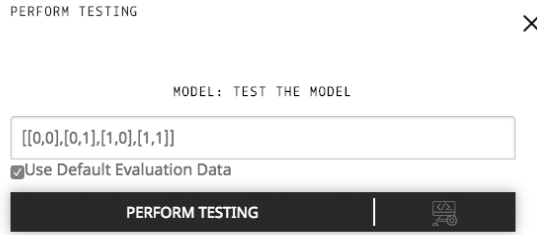


Figure 35: Testing request

This user guide presented sample user interaction with the N2SkyC platform, explaining underlying services communication and workflow. As workflow shows, there is no complex interaction involved and each service is responsible for its own domain functionality. By design, all services communicate through HTTP messages using a unified data format. Microservices architectures allowed to structure complex functionality into a clear and robust system, which is also reflected in the UI of the system.

7 Conclusions and Outlook

The N2Sky platform, its architecture, and implementation, which were the main focuses of this master thesis, allowed to deeply analyze the difference in the approaches to the software architecture. During the research, several crucial points were discovered, which can drastically influence the robustness of the development process. Such topics include optimization of the process of automated delivery and deployment and differences in the approaches of the system monitoring. Nevertheless, right design decisions and well-argued technology stack can help to overcome the difficulties.

The research started with the analysis of the architectural design decisions of the N2Sky followed by the source code analysis. After analysis of the current state of the system were delivered limitations and possible bottlenecks both in the design and chosen technological stack. Followed by analysis of the existing software development approaches with a focus on the microservices architecture which led to the formulation of the insights on how such systems should be composed and which features are relevant for the robustness of the software. Taking into consideration delivered analysis results application functionality decomposition guidelines were produced. They describe which topics

are relevant and which suggestions should be considered, while decomposing application into the microservices. To enable benefits of microservices architecture, it was necessary to choose a right technological task. After research of the existing solutions, several of them were picked to be part of the new N2Sky implementation.

As a result, previous N2Sky was architecture decomposed into microservices, and new architecture was delivered and implemented. To provide detailed insights on the process of both infrastructure deployment and development, several guidelines were produced. First, they cover the process of N2Sky cloud infrastructure deployment starting from installation and configuration of the OpenStack platform both on bare metal and virtual machines. Second, they provide information on the bootstrapping and configuration of the Cloudify orchestration management tool. Last, the developer guidelines for neural network researchers and developers are providing conceptual descriptions on how to implement new neural networks in a way that they will be compliant with the N2SkyC platform. As any live system N2Sky is developing further and new possibilities are available now, after the re-design project.

7.1 Future Work

N2Sky can be developed in several directions to extend the platform and provide more functionality:

Cloudify and OpenStack.

Although these two products are working will together, there are few drawbacks in the current versions. OpenStack is shifting to the full container orchestration support, but the service which will be responsible for that is still in development. Cloudify, on the other hand, is depreciating its support for native container virtualization for OpenStack. To overcome these difficulties, it is suggested to use middleware between OpenStack and Cloudify, as Kubernetes, which will give more control and flexibility over the virtual resources of the cloud.

OpenStack in Microservices.

Currently, OpenStack is deployed as an application on the bare-metal without the usage of any virtualization technologies. Possible development direction would be to re-design OpenStack services in a such a way so they can be run as containers on the host machine.

Smart Load balancing.

One of the possible extensions of the system can be a unified load balancing system, which will be responsible for routing user requests depending on the current load of the services. Heuristics can be developed to estimate the average load and computational complexity to provide necessary resources. Using the monitoring system, it will be possible to collect usage and load data and use the N2SkyC system itself to build predictive models.

DevOps and Repositories.

Cloudify provides a certain level of versioning of the services, but in regards internal services development the management of the source code is handled by developers themselves. One of the possible extension could be an internal DevOps server, which will be responsible for the whole project deployment and testing.

ViNNSL extension.

Currently, ViNNSL specification language provides support only for neural network description. One of the possible extension could be the new specification of the ViNNSL, which supports other machine learning algorithms. New architecture enables deployment of any machine learning and neural network service. Therefore such extension will be easy to implement.

Deep Learning Support.

Deep learning is highly popular topic nowadays, and it will be beneficial to add support for stacking several neural networks together. Combined with the support of other machine learning algorithms, such extension will be highly appreciated by the scientific community.

These topics are highly interesting and can be considered as future development perspective of the N2Sky platform.

References

- [1] ABBOT, M. L., AND FISCHER, M. T. *The Art of Scalability*. Addison-Wesley Professional, Boston, 2009. ISBN: 978-0134032801.
- [2] ANDREW TULLOCH. DNNGraph. [online]. <https://github.com/ajtulloch>, last viewed January 2018.
- [3] APACHE. Maven. [online]. <https://maven.apache.org/>, last viewed January 2018.
- [4] ARTELNICS. Neural Designer. [online]. <https://www.neuraldesigner.com/>, last viewed January 2018.
- [5] BERAN, P. P., VINEK, E., SCHIKUTA, E., AND WEISHAUPL, T. Vinnsl - the Vienna neural network specification language. In *Neural Networks, IJCNN(IEEE World Congress on Computational Intelligence)*. *IEEE International Joint Conference on* (2008), IEEE, pp. 1872–1879.
- [6] BINZ, T., BREITER, G., LEYMAN, F., AND SPATZIER, T. Portable cloud services using toasca. *IEEE Internet Computing* 16, 3 (2012), 80–85.
- [7] CANONICAL LTD. Infrastructure for container projects. [online]. <https://linuxcontainers.org/>, last viewed January 2016.
- [8] CLOUDIFY, INC. Dockerhub. [online]. <https://hub.docker.com/>, last viewed January 2017.
- [9] DISTRIBUTEDDATAMINING PROJECT. DistributedDataMining Project. [online]. <https://www.distributeddatamining.org/>, last viewed January 2018.
- [10] DOCKER, INC. Docker architecture. [online]. <https://docs.docker.com/engine/docker-overview/#what-can-i-use-docker-for/>, last viewed January 2017.
- [11] DOCKER, INC. Openstack plugin. [online]. <http://docs.getcloudify.org/3.3.1/plugins/openstack/>, last viewed January 2017.
- [12] EVANS, E. *Domain-driven design: Tackling Complexity in the Heart of Software*. Addison-Wesley, Boston, 2003. ISBN: 978-0321125217.

- [13] FEDORENKO, A., ADAMENKO, A., AND SCHIKUTA, E. N2Sky - A Neural Network Problem Solving Environment Fostering Virtual Resources. In *International Joint Conference on Neural Networks, IEEE World Congress on Computational Intelligence. In Press on* (2018), IEEE.
- [14] GIGASPACE TECHNOLOGIES. Clodify. [online]. <http://docs.getcloudify.org/3.4.1/intro/what-is-cloudify/>, last viewed January 2016.
- [15] HUQQANI, A. A., LI, X., BERAN, P. P., AND SCHIKUTA, E. N2Cloud: Cloud based neural network simulation application. In *Neural Networks (IJCNN), The 2010 International Joint Conference on* (2010), IEEE.
- [16] MANN, E. *N2Sky - a cloud-based neural network simulation environment*. Master's thesis, University of Vienna, 2013.
- [17] MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 239 (2014), 2.
- [18] OPEN-SOURCE. Deeplearning4j. [online]. <https://deeplearning4j.org/>, last viewed January 2018.
- [19] OPEN-SOURCE. Neuroph. [online]. <http://neuroph.sourceforge.net/>, last viewed January 2018.
- [20] ORACLE CORPORATION. Jersey. [online]. <https://jersey.github.io/>, last viewed January 2018.
- [21] ORACLE CORPORATION. Spring Data. [online]. <http://projects.spring.io/spring-data/>, last viewed January 2018.
- [22] R.T.FIELDING. Representational state transfer. [online]. <https://www.openstack.org/assets/science/OpenStack-CloudandHPC6x9Booklet-v4-online.pdf>, last viewed January 2016.
- [23] SAGE A. WEIL AND SCOTT A. AND BRANDT ETHAN AND L. MILLER AND DARRELL D. E. LONG. Ceph: A Scalable, High-Performance Distributed File System. [online]. <https://www3.nd.edu/~dthain/courses/cse40771/spring2007/papers/ceph.pdf>, last viewed January 2018.

- [24] SCHIKUTA, E., AND BERAN, P. P. A gridified artificial neural network resource. In *IEEE International Conference on Tools with Artificial Intelligence on* (2007), IEEE.
- [25] SCHIKUTA, E., AND MANN, E. N2sky - neural networks as services in the clouds. In *Neural Networks (IJCNN), The International Joint Conference on* (2013), IEEE, pp. 1–8.
- [26] SEFRAOUI, O., AISSAOUI, M., AND ELEULDJ, M. OpenStack: toward an open-source solution for cloud computing. *International Journal of Computer Applications* 55, 3 (2012).
- [27] SOFTWARE ADVICE. Neural designer. [online]. <https://www.softwareadvice.com/bi/neural-designer-profile/>, last viewed February 2017.
- [28] UNIVERSITY OF CALIFORNIA. Berkeley Open Infrastructure for Network Computing. [online]. <https://boinc.berkeley.edu/>, last viewed January 2018.
- [29] UNIXARENA. Openstack architecture and components overview. [online]. <https://www.unixarena.com/2015/08/openstack-architecture-and-components-overview.html>, last viewed January 2016.

List of Figures

1	Neural Designer Screenshot	5
2	DNNGraph Screenshot	6
3	Neuroph Software Screenshot	7
4	N2Sky UI Screenshot	8
5	Previous N2Sky Architecture	9
6	Scaling Strategy Cube	11
7	Monolithic Deployment Workflow	15
8	Modular Deployment Workflow	15
9	Split Deployment Workflow	15
10	Previous N2Sky Architecture	17
11	New N2Sky Architecture	17
12	OpenStack Simplified Architecture	22
13	Nova Architecture	23
14	Container Architecture	27
15	Differences between VM and LXC	29
16	Docker Architecture	32
17	Docker image layers	34
18	Running container layers	35
19	State sharing schema	36
20	Orchestration Responsibility Split	37
21	N2SkyC Architecture	42
22	Business Module	42
23	Monitoring Module	43
24	Infrastructure Module	44
25	Computational Module	46
26	User Interface Module	48
27	OpenStack GUI	58
28	User login/register forms	70

29	Paradigm upload	71
30	Defining neural network deployment	72
31	Neural network editor	72
32	New instance deployment	73
33	Training request	74
34	Training result	74
35	Testing request	75

List of Tables

1	Business module description	43
2	Monitoring module description	44
3	Infrastructure module description	45
4	Computational model description	47

Listings

1	Cloudify TOSCA-based sample configuration	39
2	DevStack user creation	50
3	DevStack configuration header specification	50
4	DevStack main configuration file	51
5	DevStack components repository specification	52
6	DevStack enabling monitoring of services	52
7	DevStack download Docker hypervisor	53
8	DevStack enable Docker hypervisor support	53
9	DevStack bootstrap script	53
10	Cloudify environment setup	54
11	Vagrant VM settings	54
12	DevStack VM configuration	55
13	Spinning Vagrant VM	56
14	Sample DevStack installation output	56
15	Keystone configuration	57
16	Configuring cloud security group	58
17	Installing Python distribution	59
18	Installing Cloudify CLI in virtual environment	59
19	Fetching Cloudify configuration files	59
20	Cloudify Management input specification	59
21	OpenStack Authentication for Cloudify	60
22	Cloudify Bootstrap	61
23	ViNNSL Specification Adaptation	62
24	ViNNSL Parser	64
25	Generic backpropogation Neural Network	66
26	Server endpoints	67
27	Dockerfile example configuration	68

A Abstract English

N2Sky was developed as a neural network simulation environment, which main purpose was to provide different stakeholders with access to a robust and efficient computing resource. However, the current N2Sky implementation is based on the Java programming language and deployed as a single monolithic application, which was not well aligned with the distributed cloud-based paradigm. That led to a decision of redesign of the N2Sky platform using microservices approach and the new technological stack for the cloud infrastructure, which will allow to fully utilize the benefits of cloud computing.

Master thesis is focusing on two main parts of the architectural redesign process: infrastructure redesign and architecture redesign. An important infrastructural change is a switch from Eucalyptus cloud infrastructure provider to OpenStack. Internal Eucalyptus endpoints were previously designed to be accessed using SOAP technology, and it added overhead to the infrastructure. OpenStack internal services are based on REST interfaces and support orchestration tools, which suits better for the project infrastructure.

Application redesign is performed by adapting microservices approach - whole application functionality is decomposed into separate modules, each of them can be accessed through the provided API. Containerization technology perfectly suits the microservices architectural approach. It allows not to be restricted to a specific programming language or database storage technology: all the components which are designed in a way that they are interacting with each other through the API, so they are not aware of any internal implementation details. As container quantity, can grow very fast, it becomes clear that manual maintenance of numbers of containers can be a tough task, especially considering a cloud environment. For that reason, is considered container orchestration middleware. As a result, new architecture design of the N2Sky system was delivered, alongside with decomposition guidelines and development guides. New system revision is highly scalable and provides necessary features to fully support agile development and fulfill needs of all types of stakeholders.

B Abstract German

N2Sky wurde als Neuronennetz Simulationsumgebung entwickelt. Die Idee war, den verschiedenen Stakeholder Zugang zu den robusten und effizienten Computerressourcen gewährt werden. Es wurde konzipiert, um natürliche Unterstützung für die Cloud-Bereitstellung mit verteilten Computerressourcen zur Verfügung zu stellen. Das aktuell N2Sky basiert jedoch auf der Java-Programmiersprache und als eine einzige monolithische Applikation ist deployed, die auf das verteilte cloudbasierte Paradigma nicht gut ausgerichtet ist. Das führte zur Entscheidung für ein Redesign der N2Sky-Plattform unter Verwendung des Microservices-Ansatzes und des neuen technologischen Stacks für die Cloud-Infrastruktur, der es ermöglichen wird, die Vorteile des Cloud-Computing voll auszuschöpfen.

Die Masterarbeit konzentriert sich auf zwei Hauptteile des architektonischen Redesignprozesses: Infrastruktur-Redesign und Architektur-Redesign. Ein wichtiger Infrastrukturwandel ist der Wechsel vom Eucalyptus Cloud zu OpenStack. Interne endpoints des Eucalyptus wurden früher so konzipiert, dass Zugang nur durch SOAP Technologie möglich war. Das hat ein zusätzliches Schwirigkeitsneveu zum System hinzugefügt. Die internen OpenStack Services basieren auf REST-API und unterstützen Orchestrierungs-Tools, die besser für die Projektinfrastruktur passen.

Die Neugestaltung des Systems erfolgt durch Anpassung des Microservices - die gesamte Applikationenfunktionalität wird sich in separate Module zerlegt. Jeder von Ihnen kann über die zur Verfügung gestellte API zugegriffen werden. Die Containerisierungstechnologie passt perfekt zum Microservices-Architekturansatz. Es kann auf eine bestimmte Programmiersprache oder Datenbanktechnologie nicht beschränkt werden. Alle Komponenten, die so erstellt sind, dass sie über die API miteinander interagieren, sodass sie sich keine internen Implementierungsdetails bewusst sind. Da die Containermenge sehr schnell wachsen kann, wird es klar, dass die manuelle Wartung von Containern eine schwierige Aufgabe sein kann, insbesondere unter Berücksichtigung einer Cloud-Umgebung. Aus diesem Grund wird Container-Orchestrierungs-Middleware betrachtet. Als Ergebnis wurde eine neue Architektur des N2Sky-Systems geliefert, daneben mit Dekompositionsleitlinien und Entwicklungsanleitungen. Die neue Systemrevision ist hoch skalierbar und stellt zur Verfügung die notwendigen Eigenheiten, um die agile Entwicklung voll zu unterstützen und die Bedürfnisse aller Stakeholder zu erfüllen.