FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Danylo Khalyeyev

Optimizing the deployment of cloud applications for multiple QoS parameters

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	doc. RNDr. Petr Hnětynka, Ph.D.
Study programme:	Computer Science
Study branch:	Software Systems

Prague 2021

I would like to thank my supervisor doc. RNDr. Petr Hnětynka, Ph.D. and my advisor prof. RNDr. Tomaš Bureš, Ph.D. for their guidance and invaluable advice. I would also like to thank my friends and family for their support throughout this work.

Title: Optimizing the deployment of cloud applications for multiple QoS parameters

Author: Bc. Danylo Khalyeyev

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Petr Hnětynka, Ph.D., Department of Distributed and Dependable Systems

Abstract: Guaranteeing Quality of Service (QoS) in an (edge-)cloud environment is one of the biggest open problems in the field of cloud computing. Currently, deployment of cloud applications is managed by cloud orchestration systems, such as Kubernetes. These systems make deployment of applications in cloud easier than ever, offering their users the benefits of availability, scalability and resilience. However, at the moment they are not capable of optimizing the deployment of cloud applications with respect to performance QoS metrics, such as response time and throughput.

The thesis proposes an approach that provides probabilistic guarantees on the performance QoS metrics in an (edge-)cloud environment. The approach is based on assessing the performance of cloud applications and subsequently controlling their deployment in a way that the applications are deployed only in the environments in which their performance does not violate their QoS requirements. The thesis also presents a proof-of-concept implementation of that approach. The implementation verifies the effectiveness of the approach and will serve for further research.

Keywords: Deployment Cloud Edge-cloud Quality of Service

Contents

1. Introduction

During the last decade, cloud computing grew from a novel and promising computing paradigm [1][2] to a widespread approach that had transformed many areas of information technology [3]. Apart from IT, cloud computing is extensively used in many other industries, such as manufacturing [4], healthcare [5], banking [6], etc. Large companies create their own on-premise private clouds [7]; while those who do not have means or a need to maintain a full-scale cloud infrastracture make use of public clouds, such as Amazon Web Services, Google Cloud Platform, Microsoft Azure, IBM Cloud, etc.

The benefits of cloud computing are numerous [8][9]. Among other things, cloud computing:

- Allows the maintainers of computing infrastructure to manage the computing resources in a more efficient and organized way.
- Reduces the overall computing infrastructure costs.
- Gives end users a location- and device-independent access to the services and applications.
- Simplifies the maintenance and support of server-side software.
- Allows companies to avoid the upfront costs of computing infrastructure by using public clouds for deployment of their applications.

All these benefits make cloud computing an attractive option to use even in such areas as Internet of Things, smart manufacturing, or cyber-physical systems [10][11]. However, there are several challenges that need to be addressed in order to make the adoption of cloud computing in these areas successful [12]. These challenges arise from the requirements specific for these areas.

As an example of such a challenge we can consider an interaction between a cloud service and an embedded device. Embedded devices are often real-time, i.e. the software they run is structured into *tasks* that have real-time deadlines. If a cloud function is called from inside of a real-time task, this would mean that the execution time of that task depends on (i) the latency between the device and the cloud, and (ii) the execution time of the cloud function. In order to make this use case feasible, we would need to (i) ensure that the latency between the device and the cloud is sufficiently low, and (ii) guarantee that the execution time of the cloud task is bounded by an appropriate value. Simply put, we need to bring real-time properties to the cloud, at least to some degree.

The existing approaches to handling this problem usually rely on changing the architecture of the cloud, bringing the processing closer to end users and embedded devices. This gave rise to such paradigms as edge computing [13], mobile edge computing [14], fog computing [15], cloudlets [16] and transparent computing [17]. These approaches, to a degree, are successful in reducing the latency [18], but they do not address the inherent inability of cloud software stack to support real-time task execution.

Quality of service in cloud computing

The challenge of supporting real-time devices outlined above is a special case of a broader problem: managing Quality of Service (QoS) in cloud environments. QoS management encompasses control over a large number of metrics that are important for delivering the services of sufficient quality to the end users (or, in some cases, they are important to the cloud service providers themselves, e.g. for energy efficiency). These include performance metrics, such as response time, service throughput, or data transfer rate; dependability metrics, such as availability, scalability, or resilience; and efficiency metrics, such as resource utilization or energy consumption [19][20]. Managing QoS in cloud is important for such applications as capacity allocation, load balancing, admission control, and energy management [19][21]. This problem is widely recognized as one of the biggest challenges that needs to be addressed in order to facilitate the adoption of cloud systems for many additional use cases [22][19].

QoS requirements are typically expressed in form of Service-Level Agreements (SLA) [23]. An SLA is a contract between the cloud service provider (CSP) and the maintainer of the application, which describes the requirements of the applications that CSP is going to guarantee. SLA usually come in form of the specific metrics and values, e.g. ”end-to-end latency under 20 ms” [23]. Ensuring that an SLA is always met is a task that has a lot to do with the environment of an application instance, its lifecycle and deployment.

Orchestration and deployment in cloud systems

Performance-wise, the behavior of an application instance in cloud depends very much on the hardware it is running on, on the amount of (physical or virtual) resources allocated to it, and on the other instances that are running alongside it on the same machine. Thus, in order to guarantee the performance properties of an instance, it is necessary to be able to control the *deployment* of this instance and of all the other instances in the system. That means deciding where exactly (on which machine in which data center) to put each instance in the cloud. These decisions have to take into account the performance properties of individual instances, and how those properties are influenced by the presence of other instances on the same machine.

An important part of cloud software stack are cloud orchestration systems. Their purpose is to manage the deployment of cloud applications in an automatic way. Among other things, orchestration includes:

- Deciding where exactly each application instance should be deployed.
- Allocating computing resources for the application instances.
- Connecting the instances with virtual networks.
- Load balancing incoming requests between the instances of the same application.
- Restarting failed instances.
- Scaling the number of instances up or down.

These functions allow cloud orchestrators to deal with such QoS metrics as availability, scalability and reliability. However, modern cloud orchestrators are not capable of managing performance QoS metrics, such as response time, latency, throughput, etc.

Management of performance-related QoS parameters is still an open problem in the area of cloud computing. Proposing a way to add such functionality to the cloud software stack is the main goal of this work.

1.1 Goals of the thesis

The goal of the thesis is to develop a framework for managing performance-related quality of service parameters in cloud applications. We are going to distinguish between two types of QoS management:

1. Providing QoS guarantees established in SLA. This means that SLA for an application, once accepted, must hold at all times.

2. Minimizing/maximizing certain QoS metrics. This means that the framework has to be aware about these metrics when making deployment decisions and, if possible, trying to make those decisions in a way that minimizes/maximizes those metrics.

The main focus of this work is going to be the former type. However, in cases where providing guarantees is not possible, it would be beneficial to provide at least some degree of the latter.

The three main challenges that we need to solve in order to achieve that goal are:

1. Expressing SLA properly.

2. Assessing if the SLA for a particular application can be met.

3. Ensuring that SLA for all applications hold at all times.

Thus, in this thesis we want both to propose an approach to solve the problem of performance QoS management in cloud and to provide a working implementation of that approach. We see that implementation as a proof-of-concept and do not expect it to fully satisfy all the properties that are usually put on production software systems (such as security or robustness). Since it is not feasible to create a framework for guaranteeing all types of QoS in the scope of this thesis, we are going to show the capabilities of the framework on two QoS metrics: response time and throughput. In addition to demonstrating the chosen approach, the framework is meant to serve for further research in the field of QoS management in cloud. Thus, we are going to implement a mechanism for extension of the framework in order to allow future developers to modify it according to possible future requirements.

1.2 Structure of the thesis

Chapter 2 provides an analysis of the problem of performance QoS management, explains the approach implemented in the framework, and provides the reasoning behind the design decisions taken in course of the development of the framework.

Chapter 3 provides an overview of the final implementation of the developed framework, including its integration with external projects.

Chapter 4 shows how the framework can be used from the point of view of different stakeholders (developer of the applications for the framework, maintainer of the framework, and developer of future extensions for the framework).

Chapter 5 discusses the efficiency of the final implementation and takes a look at its limitations and drawbacks.

Finally, chapter 6 summarizes the work that was done and takes a look into the possible future directions of the research in the field of QoS management in cloud.

The digital attachment that comes with the thesis contains the developed framework, the IVIS framework (includes both the code developed outside of the scope of this project and the integration written for this project), and a set of scripts for creating a virtual testing environment for the framework.

2. Problem analysis

In this chapter, we present our approach to the QoS management, outline the reasoning behind it, and describe the challenges that needed to be solved in order to implement it.

In the section 2.1 we introduce several key terms that are used throughout the rest of the thesis. Then, in section 2.2 we outline our approach in general terms. In the rest of the chapter, we explain the reasoning behind the individual elements of the approach and the design decisions taken during its implementation.

2.1 Key concepts and assumptions

2.1.1 Cloud and edge cloud

The term *edge computing* generally refers to any distributed computing model that brings data storage and computation closer to the end users. This contrasts the *edge cloud* from the *centralized cloud*, which does not have to take into account the geographic location of the deployed applications. There are several ways in which the term edge cloud is usually defined.

First, the cloud can be geographically distributed across several datacenters. In this case, the end users are supposed to be connected to the services in the closest datacenter available. This has such benefits of reduced network bandwidth consumtion, lower latency, etc. Ideally, the data of individual users should also be located as close as possible to them to ensure fast data access.

An alternative understanding of the term edge computing is a computational model in which a significant share of computation is moved to *edge devices*. An edge device is a device connected to the cloud from the outside, including mobile devices, smart sensors and actuators. These devices may rely on the cloud services for their correct functionality.

These two paradigms are not mutually exclusive. A cloud system can consist of a number of geographically distributed datacenters, with each datacenter having many external edge devices connected to it. Since in this work we are concerned about guaranteeing the performance of services running in cloud environments, the term edge cloud for us primarily means a geographically distributed cloud. The external instances still can connect to it, but we do not consider them to be a part of edge cloud.

2.1.2 Architecture of cloud applications

Design time architecture. A typical application that we target with this framework is composed of several components. As we target primarily the applications with microservice architecture, we assume that each component is meant to carry out one specific function. Those components can be interconnected, i.e. they can call each other's functions (thus, several connected components can form a pipeline that carries out more complex functionality). When two components are connected, we say that one component has a *dependency* on the other.

The design time architecture of an application is described using *application desriptors*, which contain specification of the components and their relationships. An example of an application descriptor in the format used by our framework is shown in section 4.1.2.1.

Figure 2.1 shows the design time architecture of typical applications.

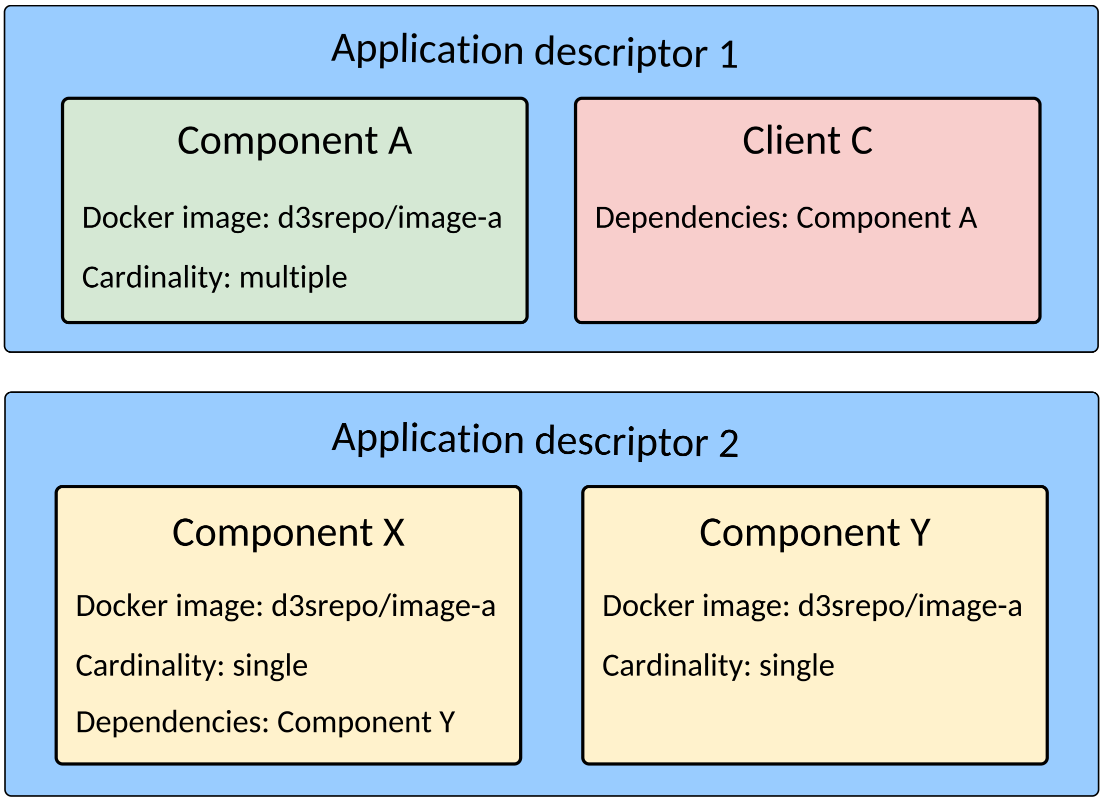


Figure 2.1: Design time architecture of two typical applications

Runtime architecture. At runtime, each component can have multiple instances running in the cloud (later in the text we refer to them as cloud instances or component instances). One instance corresponds to one Docker container managed by Kubernetes. We say that an instance A is deployed on a node N, if the corresponding Docker container runs on that node. Similarly, we say that an application is deployed within our framework if its application descriptor has been accepted and at least one instance that belongs to that application is deployed.

If component A has a dependency on component B, then at runtime each instance of component A needs to have a reference (e.g. an IP address) to an instance of component B. The number of instances of a component that run at any given time is determined based on its cardinality and demand.

Component cardinality can take two possible values:

1. **Single** cardinality means that there is only one instance of this component in the whole cloud. All the requests that are associated with this component are directed to that instance.

2. **Multiple** cardinality means that the number of instances running in cloud depends on the demand for the component's services at the current moment.

That demand can be expressed either as a number of connected clients or a number of incoming requests for component's services.

A *client* is an entity that connects to the cloud instances from the outside (i.e. it does not run in our cloud). This can be, for instance, an application running on a mobile phone, an embedded device, or a webpage opened in browser. When a client connects to our framework, it states which services (i.e. which components) it wants to connect to. Our framework then provides the client with addresses of the suitable instances.

However, an application may not contain any clients. A component may be designed to perform some computation on its own, without serving client requests.

Figure 2.2 shows a typical runtime architecture of an application.

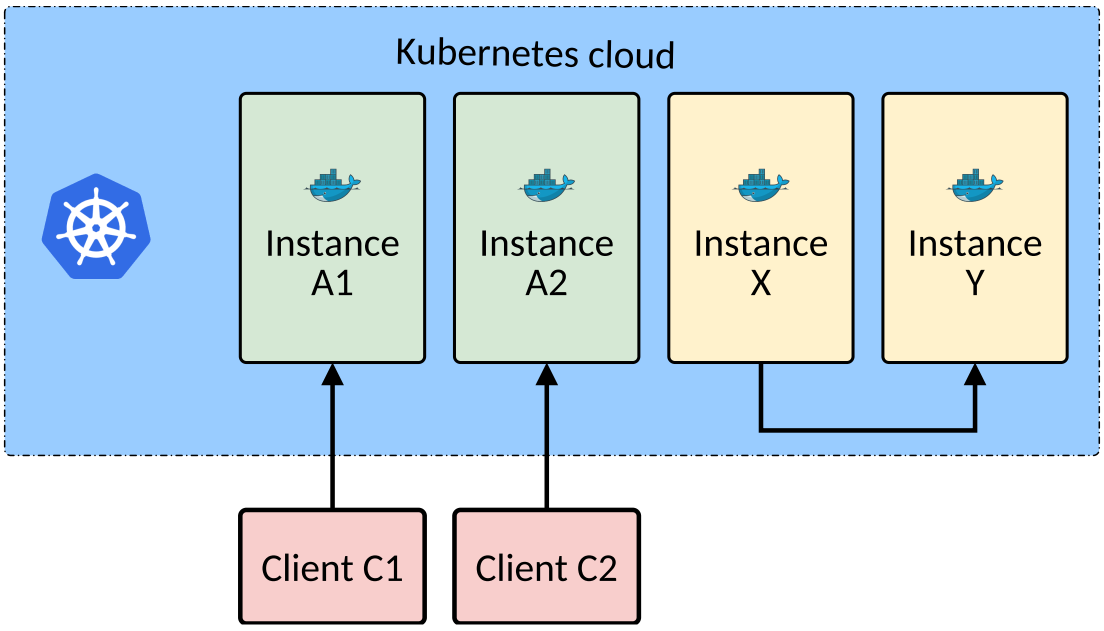


Figure 2.2: A possible runtime architecture of the applications shown on figure 2.1.

2.1.3 Structure of the cloud

Our framework assumes that on the physical level, the cloud under its management consists of *datacenters*, each containing one or more *nodes* (physical machines). As datacenters may be distributed across different geographic locations, it is also assumed that there is a non-zero network latency between them. The latency between nodes in a single datacenter is assumed to be small enough not to be taken into account.

All nodes in all datacenters are connected into a single Kubernetes cluster. The datacenter that contains the node which serves as a Kubernetes master is called the *central* datacenter. The central part of the framework is also meant to be executed in or near the central datacenter.

If there are no other datacenters besides the central one, than the cloud under the framework's management is a centralized cloud. Otherwise, we are talking about edge-cloud.

The nodes in the cluster may have different hardware configurations. However, we assume that the cluster is more or less homogeneous, i.e. the total number of hardware configurations is not very high.

2.2 Our approach to QoS management

Since it is not possible to provide strict performance guarantees in the cloud environment (see section 2.3.1 for explanation), our approach is based on providing *probabilistic* guarantees. This means that we still cannot ensure that the performance of a cloud operation will be within a specified limit all the time. However, as long as statistically, over the long term, its performance stays within limits, we consider an SLA to be satisfied.

An example of probabilistic SLA expressed in natural language may look in the following way: ”Response time of operation O provided by component C must be lower than 90 milliseconds in 95% of cases”. It is important to note that we define this contract over an individual operation.

In order to determine whether it is possible to fullfill an SLA, we deploy the instance in the controlled environment and measure its performance. We run the operations mentioned in SLA repeatedly, collecting detailed performance statistics from each run. The main goal of this process is to determine whether the SLA specified for these operations are realistic. In case that the performance analysis determines that the SLA cannot be fulfilled (e.g. someone specifies a limit of 100 ms for a long-running operation), we reject the application. Otherwise, we accept the application and proceed with further measurements.

In the second phase of the measurement process, we measure the performance of those operations under different background workloads. Again, we collect detailed statistics from each measurement in order to determine how the performance changes depending on the amount and type of background workload. This gives us an idea in which environments (on what hardware, under which background load) the SLA defined for operations will hold.

The detailed measurement statistics that we collect include various system counters related to CPU, memory and I/O usage. These data allow us to classify the operations with respect to the ”type of computation” they perform. This classification is then supplied to our performance model that allows us to predict which combinations of components can be collocated on a particular hardware configuration while keeping the SLA established for each component.

Then, the appplication is deployed in a production environment. Our framework makes sure that the instances that belong to the application are deployed only in those environments in which their SLA are expected to hold (according to the performance model). Throughout this process, we collect the data about performance of the deployed instances.

The following sections explain the reasoning behind the individual elements of this approach and explain them in more detail.

2.3 Establishing service-level agreements

2.3.1 Environment of a cloud instance

A cloud instance is a program running on a (physical or virtual) node in a datacenter. It is wrapped in a container, which provides a certain degree of isolation from other processes. A containerized program can see and use only those system resources that are available to its container. In Docker (and Kubernetes) it is possible to specify limits and requirements on the amount of CPU and memory that should be allocated for a particular container. Using this mechanism, it is possible to partition the CPU and memory resources perfectly, without sharing them between the different containers

However, in most cases such a strict partitioning of resources is not practical. As resource requirements of applications vary over time (depending, e.g. on the operation performed at the moment or on the current rate of requests), allocating a fixed chunk of memory or CPU for purposes of a single application would lead either to wasting a lot of these resources most of the time, or to the lack of resources for some of the containers. In other words, this would prevent us from making use of one of the biggest benefits of cloud computing – resource sharing. On top of this, it is very hard to predict how much memory or CPU a particular application will need for optimal functioning. Thus, in most cases the containers that run on a machine in the cloud share the memory and CPU between them.

Since the performance of a program may vary greatly depending on the amount of resources available, it will also depend on the other programs collocated with it on the same machine, and on their respective performance properties. The performance of an instance that runs in isolation on a given hardware may differ significantly from the performance of that same instance running alongside another instance. This is especially true if those instances compete with each other for the same resources (e.g. if both instances are CPU-intensive).

This is the main reason why it is not possible to provide any performance guarantees without having full control over the deployment of cloud instances. If we leave the job of choosing a node for deployment of each instance to Kubernetes scheduler, we will not be able to predict which instances will be collocated on the same node. Also, if the nodes in the cluster have different hardware configurations, we will not be able to predict on which hardware an instance will run. And since the collocated instances and the hardware have such a major impact on application performance, we will not be able to predict (and thus guarantee) that performance.

However, the performance still remains unpredictable even after we take the deployment under control. Two factors contribute to that.

First, if a cloud instance serves requests from external clients, its performance also depends on the rate of those requests and the type of those requests (some requests may take longer to process than the others). Since the requests can arrive aperiodically, this introduces an unpredictable factor into the performance.

Second, the operating system of the node on which a container is deployed is also a part of the instance's environment. In contrast to real-time operating systems, the operating systems on which cloud applications typically run do not support real-time scheduling and deadlines. The way in which operating system schedules threads looks rather random from the perspective of an application running under that OS. This means that every run of the same operation may differ in terms of duration of execution of that operation because of different OS thread scheduling. For this reason, it would not be possible to provide strict real-time guarantees even if we would take every other aspect of the environment under strict control.

2.3.2 Probabilistic guarantees

The considerations outlined in the previous section lead us to a conclusion that strict guarantees similar to those found in real-time systems cannot be provided in the cloud. Since this stems from the very nature of the software systems we use, we need to base our method on something different.

Once we assume that we have the deployment of applications under our control, two main things that prevent us from providing the strict guarantees are the rate of requests from clients and the scheduling decisions of the operating system.

To deal with the first issue, we can either (i) put an upper limit on the possible rate of requests, or (ii) assume that an instance is running under the maximum rate it can handle. A downside of the first approach is that it requires either cooperation on the side of the users or strict control over the rate of requests on the side of our framework. This was one of the main reasons that we have chosen the second approach. Another reason is that the measurement methodology [25] and prediction methods [26] that we use both assume the second approach.

From the point of view of a running program, the scheduling decisions of the operating system look random. If processes A and B run in parallel, there is no way to tell in advance which process will be executed by the CPU at a particular moment. To deal with this randomness, we can use statistical methods. Thus, instead of providing strict guarantees on performance we are going to work with *probabilistic* guarantees.

Even though these guarantees are not strict, they can still be very useful for many use cases. Since the primary focus of this work is to provide guarantees on response time and throughput, we will illustrate the usage of those metrics on two running examples.

Running example 1: Face recognition

Let us say that we have a client-side application running on a mobile phone. This application records a video stream and sends it to the server-side component running in the cloud. That component is tasked with recognizing the faces that may appear in the video stream. It returns the coordinates of the recognized faces to the client application. Based on this data, the client application augments the video stream, highlighting the recognized faces on the screen. Figure 2.3 illustrates the idea of the application.

An important property of this application is that it is real-time, but not safety-critical. A user of this application will want the video stream to be augmented timely, however if 1 frame out of 100 will not have the faces highlighted, they will likely not even notice. In order to improve user experience with this application, its developer might be interested in the following QoS metrics:

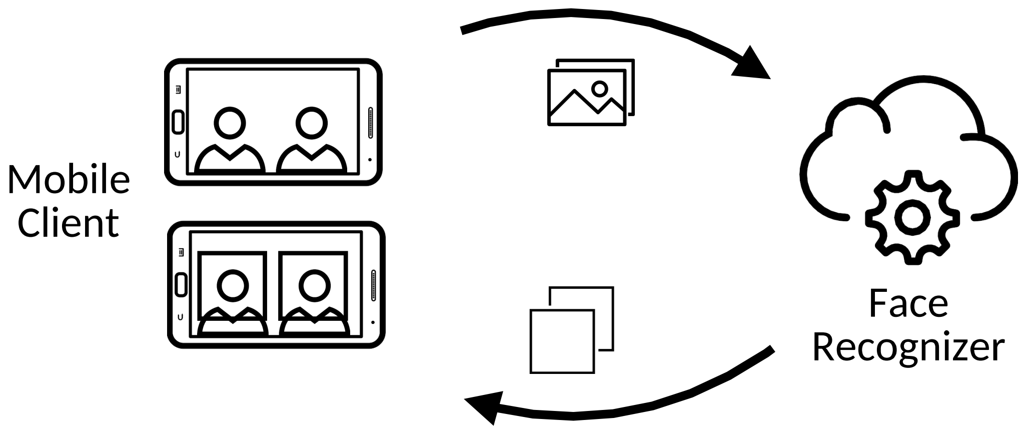


Figure 2.3: Running example 1: Face recognition

1. **End-to-end latency.** If the latency between the client and the server is high, the augmentation will lag behind the video. As users can connect to the cloud from different locations, it is generally may not be possible to fully guarantee that the end-to-end latency will be lower than a particular threshold. However, in the context of edge-cloud, we can minimize this latency as much as possible by deploying the server-side component into the datacenter closest to the user. Since the user in this example is mobile, we need to be able to redeploy the component if they change their location in order to preserve low end-to-end latency.

2. **Response time.** The duration of the face regonition operation is another factor that contributes to the timeliness of the video stream augmentation. As this operation is executed in a cloud instance, we can provide probabilistic guarantees for it. A guarantee like "The face recognition operation takes less than 50 milliseconds in 90% of cases and less than 100 milliseconds in 99% of cases" may be enough to ensure smooth user experience.

In our framework we aim to achieve both of those goals: minimizing end-to-end latency and probabilistically guaranteeing the response time.

Running example 2: Drone footage processing [1]

Let us say that we have a drone equipped with a multispectral imaging sensor. As this drone flies over the terrain, it shoots the images for a multispectral dataset that gets stored in a cloud database. A component running in the cloud takes those images from the database and detects feature points on them, as the first step in an image processing pipeline that aims to reconstruct 3D models from the drone footage (see OpenSfM [28] and OpenDroneMap [29] projects for the details of that process). Figure 2.4 shows the idea of the example.

This is a computationally intensive task that may take hours or days to compute for large datasets. If the footage from drones arrives continually, the stakeholders might be interested in guaranteeing some level of throughput, let us say

[1]This example is taken from the AFarCloud project in which our framework is being used [27]

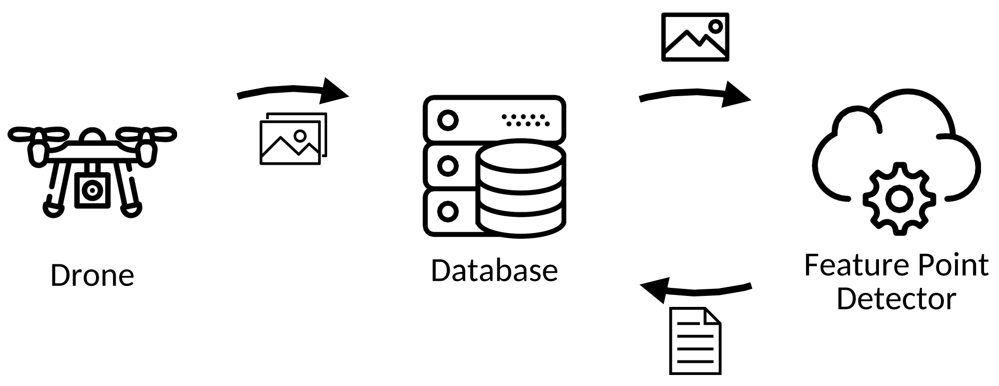


Figure 2.4: Running example 2: Drone footage processing

1000 images per hour. An SLA for this case may look as follows: ”The feature point recognition operation should be possible to call 1000 times in an hour on average”.

With large number of computationally intensive tasks running on the cloud nodes, the maintainers of the cloud may be interested in minimizing the energy consumption of their datacenters while keeping the guarantees on throughput and the other QoS parameters for the running applications. This introduces another QoS parameter to optimize for: energy consumption.

It is not an easy task to ensure that the energy consumption is really minimized, let alone to provide strict guarantees for that parameter. However, we still can take energy consumption into account while making deployment decisions. If we try to deploy the instances in a way that minimizes the number of nodes used, the maintainers of the cloud can turn off the unused nodes, thus saving energy. This can be done only if we keep all the established SLA guaranteed, since energy efficiency is only a secondary goal for our framework.

In addition to managing response time and end-to-end latency as was described in the previous example, in our framework we also want to make it possible to provide throughput guarantees and take into account the energy consumption of datacenters.

2.3.3 SLA specification

The SLA contracts shown in the examples above are defined over the individual operations. In order for such a contract to be well-defined, we require the developer of an application to create a *probe* corresponding to each operation provided by a component.

Probe is a function defined on a component with the following properties:

1. It can be called without any inputs. Instead, it can either have hardcoded inputs or obtain the inputs on its own (e.g. by downloading them from a predefined location).

2. Once an instance is running, a probe can be called on it at any time.

3. Its performance strongly correlates with the function provided by the instance. Basically, a probe is meant to carry out the same task as the real operation.

These properties allow us to (i) define the SLA over the probes instead of the real operations, and (ii) measure the performance of the probe on our own, without needing any cooperation with the developer.

In the microservice architecture it is common for each microservice to have a single well-defined function. Thus, it is advised to define only one probe per component. However, we have decided to provide support for multiple probes per component in our implementation. This was done in order to provide support for possible future use cases that were not yet considered.

2.3.4 Submitting SLA and applications to the framework

We envision two possible scenarios how to submit the QoS-managed application into the framework. The first, straightforward way, looks as follows:

1. User submits the application, specifying the components, the probes, and their QoS requirements.

2. Our framework determines whether it is possible to guarantee the specified QoS requirements for this application.

3. In the case that it is possible, we accept the application and deploy it to the cloud.

Alternatively, we could split the application submission and QoS specification into two phases. Then, the process would look in the following way:

1. User submits the application, specifying the probes on which they would like to put QoS requirements.

2. Our framework determines the performance properties of these probes and suggests possible levels of QoS that it can guarantee for them (e.g. X ms at 90%, Y ms at 99%, 100 requests per hour, etc.).

3. User specifies their QoS requirements for the application and submits them to the framework.

4. The framework checks whether these requirements are possible to guarantee (based on the already measured data), and if so, accepts the application and deploys it to the cloud. Otherwise, the user can try to specify different QoS requirements.

We have decided that our framework will support both scenarios. There is a way to specify the SLA right away, and then there is a way to add them later. In both cases, however, we need to determine the performance properties of the application before we can decide whether QoS can be guaranteed.

2.4 Performance assessment of submitted applications

One of the central elements of our approach is assessing the performance properties of the components of an application before accepting that application for deployment. This process relies on the performance measurement methodology developed outside of the scope of this thesis [25]. The assessment phase consists of the following main steps:

1. Creating **measurement scenarios** out of the submitted components. A measurement scenario describes the combinations of components that are supposed to be run together on one or several nodes. It also describes which probes have to be executed on these components and how many times.

2. Deploying components in a special staging environment, also called the **assessment cluster**. This is a separate Kubernetes cluster, that normally consists of fewer nodes than the main (production) cluster, all located within a single datacenter. The components are deployed on suitable nodes according to the currently measured scenario. If after deploying a scenario there are free nodes in the assessment cluster, another scenario may be deployed on them (thus, with a sufficiently large cluster, several scenarios can be measured in parallel). No other workloads than the ones described in the scenarios can run on the nodes in the assessment cluster.

3. Running the measurements: executing the probes described in measurement scenarios.

4. Collecting the values of **performance counters** during before and after each probe execution. These values help us to characterize the probes from the resource utilization perspective. They include:

   - Hardware events: clock reference cycles, instruction count, cache misses, etc.
   - I/O (disk) events and counters: number of read/write/discard I/Os processed, wait time for requests, number of sectors read/written, etc.

   While choosing which performance parameters to measure we were guided by the approach outlined in [25]. According to that research, these counters are the most helpful in determining the resource utilization properties of operations.

5. Composing the **measurement data files** out of the measured data. These files contain the execution time of every iteration of the probe execution and the changes in the values of counters during that iteration.

After a scenario is finished, the measurement data files are submitted to the predictor module, that uses them to build the performance model for the application and predict its performance in the scenarios that have not been measured directly.

2.5 Performance prediction

The data collected in the assessment phase are supplied to the predictor module that builds a statistical model over these data and uses it to make predictions regarding the expected performance of the measured operations in different environments. The core of the Predictor module was developed outside the scope of this thesis. In the following section we briefly describe how this implementation works, a detailed description can be found in [26].

2.5.1 Predictor implementation

The purpose of the predictor module is to answer the questions about the response times of cloud operations running under a particular background load. An example of such a question may look as follows: "What will be the response time of operation O defined on component A at 90th percentile, while running on the same node with components B and C?". The ability of predictor to answer such questions is used by our framework while making deployment decisions.

Since the number of possible combinations of components may be far larger than is realistically possible to measure, predictor relies on data analysis in order to determine the likely response times for the combinations that were not measured directly. In order to explain how this data analysis works, we need to lay out several important concepts first:

- A **resource utilization property** is a metric that characterizes an operation with respect to the way it uses a particular system resource (CPU, memory, I/O). In the current implementation, these properties are represented with the values of performance counters collected during the assessment process.
- **Operation similarity.** The operations are similar to a degree that they use the system resources in a similar way. For instance, two CPU-intensive operations are likely to be more similar to each other than two a disk-intensive operation.
- **Cluster analysis.** The operations are grouped into clusters based on their similarities. Thus, operations in the same cluster tend to use the resources in similar ways.
- **Property weight.** Some of the resource utilization properties are more useful for determining similarity between the operations and clusters than the others. Thus, every property is assigned a specific weight according to its usefulness.

The currently available implementation of the predictor uses several different prediction methods to answer the questions about the response time. The accuracy of those methods varies. Predictor always chooses the most accurate method available. Those methods are:

- **Closest friend prediction:** substituting an operation from the combination with another, the most similar operation. If the new combination have already been measured, returning prediction for that combination.

- **Primary-cluster prediction:**
  1. Categorize the background operations based on the clusters they belong to;
  2. Calculate the expected response time for all operations in combinations with all the clusters;
  3. Predict the response time of the main operation as if it was running in combination with the clusters.
- **Cluster-cluster prediction:** the same idea as primary-cluster prediction, but the main operation also gets categorized into a cluster.
- **Polynomial regression prediction:** prediction based on a polynomial regression model.

Generally, the more data predictor has, the more accurate are its predictions. For this reason the assessment process runs additional measurement scenarios continually. With time, the database of predictor grows, and the framework can make better (and more efficient) deployment decisions.

2.5.2 Usage of the predictor in the framework

In our framework, predictor is a plug-in module with a well-defined interface. The currently available implementation has several limitations, that made it impossible to plug it in without implementing several extensions for it. These limitations include the following:

1. An instance of the predictor is parametrized by a percentile and a hardware configuration. Consequently, it cannot answer questions about other percentiles and hardware configurations.
2. Predictor cannot predict response time of a single process if there are several combinations measured. Thus, initially it is not capable of answering the questions about the response time of a single process after it has been measured in isolation.
3. Predictor cannot generate measurement scenarios. It would be useful to know which scenarios are the most important to measure based on the predictor usage. Predictor does not support this.
4. The implementation of predictor deals with individual operations while we need to make predictions about components. Operations do not always correspond to components one-to-one.
5. Predictor cannot specify different percentiles for different operations in a combination. Since every instance of predictor is parametrized by a percentile, it cannot make predictions for all operations in a combination if we need different a different percentile for each of those operations.
6. Predictor does not support questions about throughput, only about response time.

In order to use this implementation in our framework, we have developed the performance data aggregator module that includes the predictor and addresses those issues. Its implementation is described in section 3.3.

2.6 Ensuring the QoS guarantees

The main challenge that our framework faces is making sure that the SLA that it had accepted will be guaranteed throughout the time that the application is deployed in the cloud. This includes, among other things:

1. Ensuring that the instances get deployed only in those environments in which their performance will not violate the established SLA.
2. Creating enough instances to serve all the clients.
3. Reacting to the new incoming clients: providing them with addresses of the instances they need to use.
4. Managing the connections between the instances.
5. Collecting the runtime data from the instances.
6. Making sure that the instances are properly initialized, have access to all their data, etc.

The following sections provide some important details on our approach in addressing this challenge.

2.6.1 Implementing a self-adaptive system

As the conditions in the cloud that we are trying to control are always changing (the number of applications and their instances, the number of connected clients, etc.), we need to be able to adapt to these changing conditions if we want to continually uphold the SLA. For this reason, we have decided that the core component of our framework, the cloud controller, will be implemented as a self-adaptive system based on the MAPE-K architectural pattern [30].

MAPE-K stands for Monitoring, Analysis, Planning, Execution, and Knowledge. It is an architectural pattern widely used in the design of self-adaptive software systems [31]. The main concept of this pattern is the *adaptation loop* consisting of the following four phases:

1. **Monitoring:** collecting and systematizing the relevant information about the current state of the system.
2. **Analyzing** (sometimes is referred to as detecting): based on the information collected in monitiring, determining whether there is a need to make some changes into the system, and if so, which changes need to be made.
3. **Planning** (sometimes is referred to as deciding): finding a preferrable course of action in order to make the needed changes.

4. **Executing** (sometimes is referred to as acting): carrying out the actions planned in the previous phase, thus adapting the system to the current conditions.

These four phases run over a shared *Knowledge database*, which contains the data needed for performing the self-adaptation process.

In our initial design, we have implemented the four phases of the adaptation loop as four separate processes. We have encountered two problems with that design. First, since the duration of individual phases varies, synchronization between the phases is a major concern under that design. Second, as all the phases work with the shared Knowledge database, preserving the integrity of that database becomes quite challenging and makes the interaction beween the processes inefficient due to the fact that the four phases work with the same data.

Due to these concerns, we have reimplemented the adaptation loop as a single sequential process consisting of four phases. This, however, led to a different challenge. The adaptation loop has to be fast enough to be able to timely respond to the changes in the managed system. At the same time, some phases (in particular the analysis and the execution) need to run long-running calculations in order to carry out their functions effectively. We discuss the ways in which we have approached this challenge and the efficiency of the final implementation in section 5.2. The final implementation itself is described in section 3.5.

In the following sections we discuss the challenges that had to be addressed in the implementation of individual phases.

2.6.2 Analysis phase: finding a deployment plan

The goal of the analysis phase in our framework is to find a way to guarantee the QoS requirements of all cloud instances under the current circumstances. It means to determine (i) how many instances of each component should run, (ii) where to put each of those instances, and (iii) how the instances should be connected to the clients and to each other. We call this determining the *deployment plan* for the instances, or the *desired state* of the cloud.

While doing this, we need to take into account not only the number of instances that need to be created and their requirements, but also the current state of the cloud. It is important to base our decisions on the current state in order to avoid unnecessary redeployments of already running instances.

Initially, we have considered several possible strategies for finding a deployment plan, including linear programming, constraint programming, and various heuristics. Running several strategies in parallel and choosing the best or the first available result was also considered. One heuristics-based strategy was implemented, but proved to be inefficient and inflexible. Eventually, we have implemented a strategy based on constraint satisfaction.

A constraint satisfaction problem consists of a collection of *variables*, and a collection of *constraints* defined over them. Each variable has a defined set of values that can be assigned to it (e.g. a range of integers). A constraint is an expression over a subset of the variables that establishes a relationship between their values that must hold. A *solution* to a constraint satisfaction problem is any assignment of values to the variables under which all the defined constraints are satisfied.

The process of searching for a solution consists of sequentially assigning the values to variables, and then evaluating the constraints defined over them after each assignment. An application of a constraint limits the range of possible values for some of the variables. If the range of possible values for a variable drops to zero, then the current assignment of variables is inconsistent. In this case, the solver cancels the last assignment and tries a different value or variable. This process continues until all variables have a value or until the search space is exhausted. Various strategies may be used to determine the order of variable value assignment and constraint propagation.

Additionally, an *objective function* can be specified for a problem. An objective function is an integer expression over a subset of the declared variables. Its value represents the relative ”quality” of a solution. A constraint satisfaction solver may be tasked with an objective to find a solution that maximizes or minimizes that value.

Under this approach to the analysis phase, we model the state of the cloud as a constraint satisfaction problem. The constraints defined in that problem ensure that in a valid solution to the problem all probabilistic QoS requirements of the deployed applications must be satisfied. Futhermore, we add an objective function that includes the weak QoS requirements (those that are meant to be optimized, not strictly guaranteed, i.e. latency and energy consumption).

The implementation of this phase is described in section 3.5.2. A discussion of its efficiency can be found in section 5.3.1.

2.6.3 Changing the state of the managed system

Another major challenge that needs to be solved in order to ensure that the QoS guarantees hold is making sure that the desired state corresponds to the current state of the cloud. This may also include changing the state of inividual deployed instances, of the nodes, of the connected clients, etc.

To do this we compare the desired state found in the analysis phase with the current state determined in the monitoring phase. Then, we determine the changes that must happen in order to bring the cloud to the desired state. These changes are structured into *tasks* that have to be executed. Creating the tasks is done in the planning phase, while carrying them out is the job of the execution phase.

We have considered multiple ways to implement task creation and execution. Our initial implementation relied on the concept of *execution plans*. An execution plan is a data structure that contains the tasks and the dependencies between them. If task B is said to depend on task A, that means that the execution of task B cannot start until task A completes successfully. An execution plan thus captures synchronization between the tasks.

However, as we have changed the architecture of our MAPE-K loop to be single-process, it became apparent that the execution plan pattern is not an optimal solution. There are two main problems with it:

1. In the single-process MAPE-K loop each phase is supposed to take no more

than several seconds. However, quite often execution of a single task may take much longer. For instance, if a task includes calling Kubernetes API to create a Kubernetes deployment, sometimes it may take minutes until Kubernetes spins up a container for that deployment. Thus, all the subsequent tasks in the execution plan will have to wait until that happens.

2. Execution plan pattern is hard to extend or alter in case if some additional functionality needs to be added to planning and execution phases. Adding a task into a plan may require changing the structure of the dependencies between the tasks that are already present in the plan. Thus, in order to insert a new task into an execution plan, it is necessary to know the whole structure of that plan.

Making the execution plans asynchronous helped us to partially solve the first issue, but did not address the second. Eventually we have decided to move from the concept of execution plans to the concept of *task registry*. In this implementation, a task can define a set of *preconditions* that must hold before its execution can start. If the preconditions of the tasks are defined correctly, there is no need to specify the dependencies between them. Consequently, support for additional tasks can be added into the framework without modifying the existing code. The final implementation of the planning and execution phases is described in section 3.5.3

2.7 Statefulness support

In the cloud, managing stateful services is very different from managing the stateless ones. Statefulness in cloud environment introduces a number of problems that do not exist for stateless applications:

1. **Persistent data storage.** The data cannot be stored inside the containers of the instances, since the instances are being created and destroyed all the time. Sometimes, e.g. when a client leaves the system, the containers that served that client may get destroyed. However, their data need to be stored somewhere until the client connects to the system again.

2. **Data migration.** As an instance may get redeployed into a different datacenter, it is important to ensure that it still has an access to its persistent data. Thus, there needs to be a mechanism that would ensure that the data get migrated following the migration of cloud instances.

3. **Access control.** Any application and any cloud instance must have an access only to the data that belong to it. Sometimes that only means restricting data access between the different applications, while in other cases it means isolating the data of different users (clients).

Since statefulness is an important property for a large class of applications, we have decided to implement statefulness support in our framework. We distinguish three types of statefulness for the components running in our cloud:

1. **No statefulness.** The option for stateless components that do not need persistent data storage.

2. **Per-client statefulness.** The data that belong to different clients are isolated. Since it is always assumed that all the instances that serve a single client are located in the same datacenter, the data of that client are always moved into that datacenter following redeployment the instances.

3. **Per-component statefulness.** The data are shared by all instances of a component. Instances of other components cannot access those data. Since different instances can be located in different datacenters, data migration is not performed for this statefulness type.

Our implementation of the statefulness support and the way in which it solves the challenges outlined above is described in section 3.5.5.

2.8 Integration with the IVIS platform

IVIS is a platform for IoT (Internet of Things) and CPS (Cyber-Physical Systems) data processing and visualization that has been deployed in a number of international research projects [32]. It stores the data received from the external sources (primarily sensors) in an SQL database, and allows users to create and run data processing tasks working with those data. The data stored in the database can then be visualized in many different ways by defining custom visualization templates. The platform comes with a web-based user interface, in which users can (i) define data processing jobs, (ii) schedule the defined jobs, (iii) define visualization templates, and (iv) produce data visualizations.

The IVIS platform has two limitations that are relevant in the context of our work:

1. The data processing jobs in the IVIS platform can be executed only locally (on the same machine where the IVIS backend runs). This limits the number and the scale of the jobs that can run concurrently.

2. The jobs can be scheduled with a periodic trigger, i.e. a user can define that a job has to be executed repeatedly with a particular interval. However, the IVIS platform cannot provide a guarantee that a job execution will be completed by the time when it has to be executed again.

   This is especially important when the number of scheduled jobs becomes large enough that the jobs start to interfere with each other's performance. In that case, a job that previously was running within its intervals may suddenly start to miss the deadlines.

   Since the IVIS platform targets primarily the IoT and CPS use cases, the data usually come in real time from various sensors. Because of this, missing the deadlines may potentially lead to the problems in the whole data processing pipeline.

In order to address these limitations, we have developed an extension to the IVIS platform that integrates it with our framework and allows IVIS users to deploy the data processing jobs in the cloud (in addition to regular local execution). Additionally, it allows the IVIS users to specify the QoS requirements on the response time and throughput of the data processing jobs. Those requirements are then handled by the framework in the regular way. With this extension, we have added real-time properties to the job execution in the IVIS platform.

Furthermore, we have employed the IVIS platform data visualization capabilities in order to automatically produce visualizations of the performance data of running cloud instances. Users can use these visualizations to monitor whether the QoS requirements are kept.

Developing this integration allowed us to employ our framework in the AFarCloud [27] and FitOptiVis [33] international research projects. The way in which this integration is implemented is described in section 3.6.

3. Framework architecture and implementation

This chapter describes the architecture of the developed framework in terms of modules it is composed of, their roles, and interactions between them. It also serves as a high-level overview of the framework implementation, explaining the main concepts that it uses and the processes involved in delivering the implemented functionality. Section 3.1 provides a high-level overview of the main modules of which the framework is composed, while the subsequent sections describe the functionality of these modules and explain how this functionality is implemented.

3.1 Overview of the architecture

A high-level overview of the framework architecture is shown on figure 3.1. The arrows, if present, represent the general directions of the data flow between the modules.

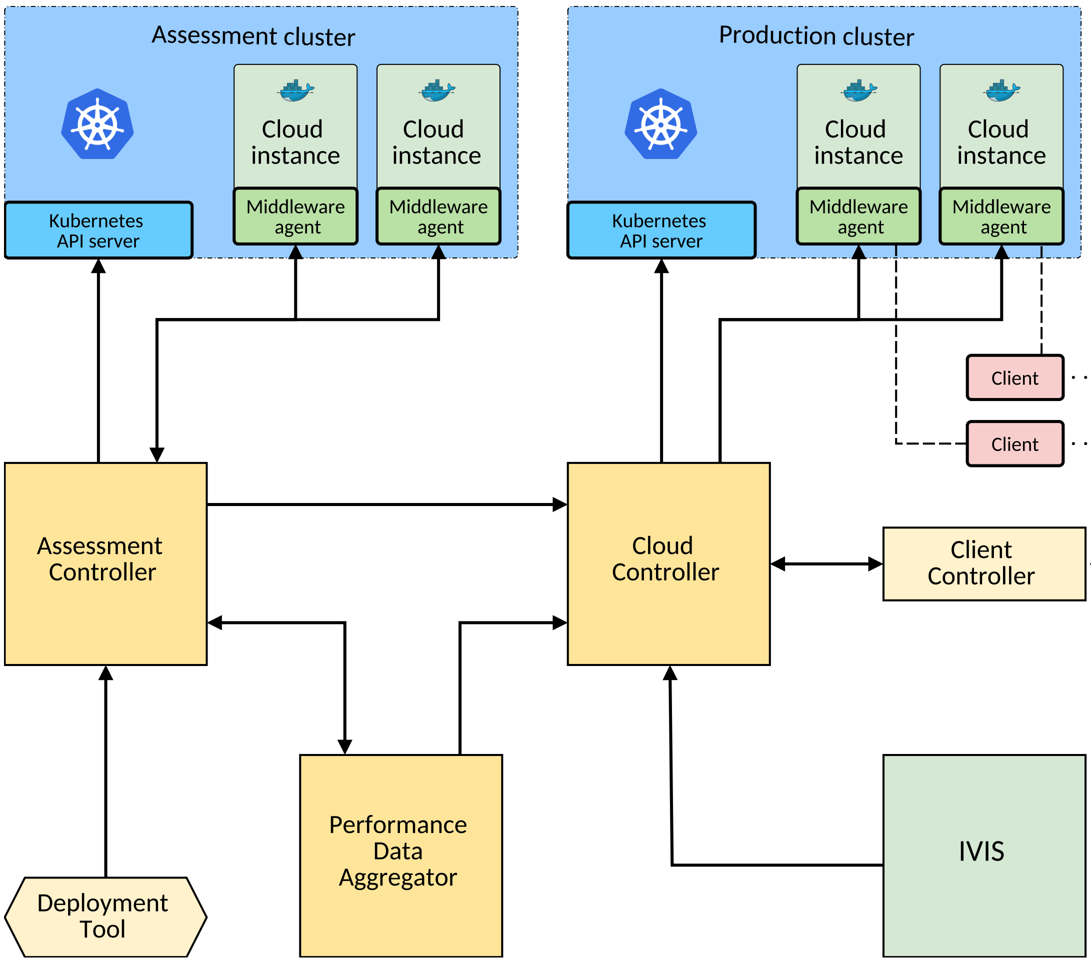


Figure 3.1: High-level overview of the framework architecture.

The core of the framework is composed of 3 modules running in separate processes: assessment controller, cloud controller, and performance data aggregator.

Assessment controller is the module responsible for the process of performance assessment of the submitted applications (which was introduced in section 2.4). It has a separate Kubernetes cluster under its control in which it deploys the measurement scenarios. This module is covered in section 3.4.

Cloud controller is the module responsible for controlling the main Kubernetes cluster, in which the applications are deployed after passing the assessment phase. Thus, its main purpose is to ensure that the QoS requirements of the deployed are actually being held. The implementation of this module is explained in section 3.5.

Both assessment controller and cloud controller manage the cloud instances through the **middleware agents** that must be integrated into every container that is meant to be deployed within the framework. The structure of cloud instances and the functions provided by the middleware agent are described in section 3.2.

Performance data aggregator is composed of several submodues that together are responsible for the following three functions:

1. Generating measurement scenarios.
2. Deciding whether an application can be accepted for deployment into the cloud (i.e. whether its QoS requirements are realistic). We call this process *application review.*
3. Predicting whether a combination of instances collocated on a particular hardware configuration will have all their QoS requirements satisfied.

The first two functions are used by the assessment controller. The third function is used by the cloud controller while making decisions about the deployment of instances. The implementation of the performance data aggregator is described in section 3.3.

Deployment tool is a simple command-line utility for communication with the framework. It allows users to submit applications for deployment into the framework, specify QoS requirements for the applications, and retrieve the data about the measurement status of the submitted applications. Its usage is covered in section 4.1.2.

Client controller is technically a submodule of the cloud controller even though it runs in a separate process. The purpose of this submodule is to serve as an entry point for the clients that want to use the applications deployed through our framework. Since in production-grade version of this framework, the number of connecting clients may potentially be very large, it is possible that this module will have to be implemented as a distributed system. For this reason, it was decided to separate this module from the rest of the cloud controller into a separate process. Its functionality and implementation are described in section 3.5.4.

IVIS is an independent Web application that can run alongside our framework. We have developed an extension that connects IVIS to the framework. With that extension, IVIS can be used as a GUI for deployment of applications into our framework and for visualization of their performance data. The main

IVIS concepts used in connection to our framework were explained in section 2.8. The implementation of our extension is described in section 3.6.

The communication between the processes is implemented mostly with gRPC remote procedure calls framework [34]. gRPC has a number of benefits, including high performance, cross-language compatibility, and being designed to support the microservice architecture. These qualities are especially important for communications with the middleware agents running on the cloud instances.

3.2 Cloud instances

One active cloud instance corresponds to one running Docker container. The Docker image for the container must be specified in the descriptor of the application the instance belongs to. For every instance, our framework creates a Kubernetes deployment (so that Kubernetes would handle possible failures and restarts of the container), and a Kubernetes service (to provide the container with a stable IP address). In order to make sure that the instance will be deployed on the correct node, the framework adds a node selector to its Kubernetes deployment that includes only that specific node.

In order to be deployable within our framework, an instance must integrate our middleware library. In particular, that means running the **middleware agent**. Middleware agent exposes a gRPC interface through which our framework communicates with cloud instances. It includes methods for instance initialization, finalization and probe calling.

Initialization of an instance means providing it with necessary data, such as initial addresses of its dependencies, access to the services provided by the framework, information about the probes that the instance should provide, etc.

Middleware agent provides three modes in which the probes can be executed:

1. **Regular execution.** Running the probe once. The response time of this run gets measured and reported.
2. **Measured execution.** Executing a requested number of probe runs. In this case, not only the response time, but also the values of system performance counters are measured and saved for each run. The data resulting from these measurements can be subsequently fetched from the instance through the middleware agent.
3. **Background execution.** Running the probe repeatedly, until the middleware agent receives a request to stop probe execution. The response time is not measured in this case, and the performance counters are not collected.

There are two ways to specify executable code for the probes:

1. Registering a callable Python procedure (a function, a method or a lambda) on the middleware agent under the probe's name. This can be done only if the middleware agent runs as a part of another Python process running on the container.

2. Providing a file with Python code and referencing it from the application descriptor. This file will be sent to the instance during its initialization. A probe execution in this case is execution of that file with the Python interpreter. Additionally, parameters that can be passed to that process or written to its standard input can also be specified in the application descriptor.

Likewise, there are two ways to integrate the middleware agent: (i) to run it as the main process of the container, and (ii) to create and start it within another Python process (which in turn runs as the main process of the container). In the first case, all the probes have to be specified in Python files. In the second case, both ways may be used. The process of middleware agent integration is shown in section 4.1.1.

3.3 Performance data aggregator

This module exposes a single gRPC interface that provides access to the three functions that the module is responsible for. The way in which these functions are carried out is described below.

3.3.1 Measurement scenario generation

The purpose of running measurement scenarios is to collect enough data on the performance of an operation so that the predictor would be able to profile it and make accurate predictions for it. The most important attributes specified in a measurement scenario are the following:

- A probe to measure (also called the main probe of the scenario).
- The background load: probes that have to run on the same machine as the main probe while it is being measured.
- Hardware configuration of the node on which the instances containing the described probes have to be deployed.
- How many runs of the measured probe to execute.

The number of runs to execute in each scenario can be configured in the framework settings. The rest of the attributes are unique to each scenario. Creation of measurement scenarios is a responsibility of the **scenario generator** submodule. This submodule distinguishes between the three different kinds of measurement scenarios:

1. Isolation scenarios. These scenarios measure performance of the components running in isolation (without any background load). They are needed in order to perform the application performance review: if an operation does not perform according to its QoS requirements while running in isolation, then the application that this operation belongs to has to be rejected.

2. Initial combination scenarios. Since the predictor cannot profile an operation based only on one scenario, it is necessary to measure several more scenarios before registering an operation with the predictor.

3. Additional combination scenarios. These are generated in order to provide the predictor with additional data, and thus improve the accuracy of prediction.

When a new application is submitted, the scenario generator creates one isolation scenario per every operation described in the application descriptor. After the application passes the review and gets accepted, it generates several initial combination scenarios for every operation (the exact number is configurable, by default it is 3). When created, both of these scenario types are put into a priority queue. The isolation scenarios have a higher priority than the initial combination scenarios.

The assessment controller asks for new measurement scenarios one by one. The scenario generator returns the first scenario from the priority queue. If the queue is empty, it generates an additional combination scenario. The operations that are measured in that scenario are determined randomly, however the scenario generator takes into account the frequency of requests about the different operations. Thus, if the predictor gets asked questions about operation A more frequently than about operation B, a scenario for the operation A is more likely to be generated.

3.3.2 Application review

In order to accept an application we need to make sure that all of its requirements on response time and throughput can be satisfied. In the current implementation we consider that if an instance behaves according to its QoS requirements (i.e. has a required response time and throughput) while having all of the system resources to itself (i.e. running without any background load), then it is possible (in principle) to guarantee those requirements. Of course, at this point it does not mean that these requirements can be guaranteed under any background load. If they cannot, that would mean that each instance will need to have a whole node to itself, and thus the deployment of the corresponding application might be very costly. Yet, in our implementation we do not deal with the issue of cost of the deployment of individual instances, only with the question whether it is possible to guarantee those requirements at all.

Thus, the process of application review happens after all of the operations that belong to the application have been measured in isolation. The results of those measurements are supplied to the **measurement aggregator** module. With these data, the module can answer the questions about response times of the operations at different percentiles and about the mean response time.

If the response time of an operation at a given percentile is lower then the one specified in a QoS requirement, then we consider that requirement to be satisfied. The throughput requirements are reformulated as requirements on the mean response time (e.g. if the mean response time of an operation is 100 milliseconds that is equivalent to the average throughput of 10 operation executions per second).

In case that the application has been submitted without QoS requirements, the application review is postponed until the QoS requirements are specified. At this point the users can get the information about the response times of operations at different percentiles through the deployment tool and use it to specify their QoS requirements.

3.3.3 Prediction

As we have mentioned before, the core of the predictor module was developed outside of the scope of this thesis.

Our framework interacts with that module through its main class `Predictor`. This class has a method `predict` that takes a combination of operations and a time limit for the main operation, and returns a boolean value that tells whether the response time of that operation at a given percentile is going to be lower than the time limit.

An instance of the `Predictor` class deals only with predictions for a single hardware configuration and a single percentile. For this reason, we have implemented the **multipredictor** module that creates an instance of the `Predictor` class whenever a new hardware configuration or a new percentile is registered. It subsequently manages those instances, providing new measurement data files to them, destroying the instances when they are no longer needed, etc.

Another limitation of the `Predictor` class is that it deals with the operations, and not component instances (as our framework does). Additionally, one question to the predictor deals with the response time of a single operation, while we need to determine whether the QoS requirements of each individual instance in a combination of collocated instances will be satisfied.

Therefore, our implementation converts a single question about a combination of instances coming from the cloud controller into a series of questions about the response time of operations. These questions are forwarded to the corresponding predictor instances. A positive answer is returned only if all of those questions resolve positively.

3.3.3.1 Throughput prediction

Throughput can be expressed as the reverse of the mean response time, e.g. a mean response time of 100 milliseconds corresponds to the average throughput of 10 executions per second. As it was said in section 2.5.2, the currently existing implementation of the predictor can only answer the questions about percentiles of the response time, not about the mean. Predicting the mean response time is very similar to predicting a percentile and can be done with the same statistical methods as those used in the existing implementation of the predictor. However, since the predictor does not support that function at the moment, we have implemented a workaroud that allows us to determine a tight upper bound on the mean response time using the current predictor implementation. We have used the following observation:

Let us say that $p_0, p_1, \ldots, p_n$ is a sequence of percentiles (expressed as real numbers from 0 to 1) where $p_0 = 0, p_n = 1$ and $p_i < p_{i+1}$. Then, let us say that t_{p_i} is the response time of a particular process at percentile p_i, and m is the mean response time of the same process. Then, $\sum_{i=1}^{n}(p_i - p_{i-1}) * t_{p_i} \geq m$.

This observation allows us to determine an upper bound on the mean response time in a constant number of requests to the predictor. The more requests are done (i.e. the more percentiles we have in the sequence), the tighter is the bound.

We have tested this approach for mean response time estimation on the default dataset of the predictor module, which contains 100 different operations. With 3 percentile requests, the upper bound was within 5% of the real value of mean response time for 82 out of 100 operations, while with 4 requests it was within 5% for 99 out of 100 operations. In both cases, it was within 10% for all 100 operations.

We consider 10% to be a sufficiently tight upper bound and use 3 requests per one throughput question by default. However, the number of those requests, as well as the values of the corresponding percentiles can be easily changed in the framework configuration.

Upper bound on the mean response time gives us a lower bound on throughput, and thus allows us to provide throughput guarantees.

3.4 Assessment controller

The assessment controller module has two gRPC interfaces. The first interface is used mainly for submission of application descriptors and QoS requirements, while the second is an internal interface through which the cloud controller fetches the application descriptors of accepted applications.

Since scenario generation and application review are carried out by the performance data aggregator, the main functions of the assessment controller are (i) scenario deployment, (ii) execution of measurements, and (iii) collection of the results of executed measurements.

Assessment controller fetches the scenarios generated by the performance data aggregator one by one. Even though a scenario specifies the load of only one node, more than one node may be needed to deploy a scenario. The reason for this is that the components described in a scenario can have dependencies on other components, which also need to be deployed before the scenario measurement can start. The assessment controller determines how many nodes it needs for the fetched scenario, and then, if those nodes are available, it starts to deploy the scenario into the assessment cloud. At the same time, it fetches the next scenario, and decides whether it can be deployed as well. If no nodes are available, it waits until one of the running scenarios finishes and frees the nodes.

The process of scenario deployment includes deployment and initialization of individual instances, and is very similar to the way in which the instances are deployed in the main (production) cluster. For this reason, a modified version of the MAPE-K loop is used for scenario deployment (implementation of the MAPE-K loop is described in section 3.5).

After deployment of a scenario finishes, the assessment controller connects to the middleware agents of each of the instances deployed on the main node of the scenario, and runs the specified probes on them. The probe is executed in the measured mode on the main instance of the scenario, and in the background mode on the other instances.

The middleware agent ensures that before and after each operation execution, the values of the performance counters are collected and processed. Additionally, it measures execution time of every run. After the measurements of the main probe are finished, the scenario orchestrator fetches these measurements and composes them into a measurement data file.

Then, the assessment controller reports scenario completion to the performance data aggregator, sending the address of the data file with the report. The performance data aggregator then moves that file into its own database.

At this point, the scenario is finished, and all the instances that belong to it are removed from the cloud. The nodes are then used for the purposes of the next measurement scenario.

Since the scenario generator can generate a virtually unlimited number of scenarios, the process of assessment never stops until the assessment controller is running. If the assessment controller is stopped, the rest of the framework will continue to work, managing the already deployed applications. However, new applications cannot be submitted in that case.

3.5 Cloud controller

The cloud controller module is built around the concept of MAPE-K loop that was described in section 2.6.1. Each phase has a module responsible for carrying it out (**monitor**, **analyzer**, **planner**, and **executor** respectively). The four phases run sequentially, one by one, in an infinite loop. The functions carried out by the individual phases include the following:

1. **Monitoring:** retrieving the newly submitted applications that need to be deployed into the system; processing connecting/disconnecting clients; collecting the data about probe execution times from the cloud instances; monitoring the state of the Kubernetes cluster (the nodes present, entities deployed, etc.).

2. **Analyzing:** finding the preferred *deployment plan*: the state of the cloud in which all the established SLA are expected to hold. This includes determining the number of instances of each component that should run, on which nodes those instances should run, how they should be interconnected, and which clients they should serve.

3. **Planning:** comparing the desired state of the cloud determined in the previous phase to its current state, determining which actions need to be taken in order to get the cloud to the desired state. Structuring these actions into *tasks* and specifying in which order these tasks need to be executed.

4. **Executing:** carrying out the planned tasks (creating and initializing new instances, providing instances with addresses of their dependencies, managing Kubernetes entities, etc). Checking preconditions for the task execution, updating the model of the system depending on the results of the execution.

The **knowledge** module is a set of data structures used by all phases. It contains the model of the entire system managed by the framework, including both internal (cloud-based) entities and external clients.

In the following sections, we explain the implementation of the individual phases and submodules.

3.5.1 Monitoring

`Monitor` is an abstract class that serves as a base class for all classes that implement monitoring functionality. The default implementation of this class (which carries out the whole monitoring phase) is the `TopLevelMonitor` class. It serves as a wrapper around a collection of other, domain-specific monitors.

By default, there are 4 monitors registered in the top level monitor. Those are:

1. **Kubernetes monitor.** Collects information about Kubernetes entities (nodes, pods, namespaces, etc.) from the Kubernetes API server. Extracts the relevant information from the descriptors of those entities, and updates the knowledge model accordingly.

2. **Application monitor.** Fetches the descriptors of newly accepted applications and requests for application deletion from the assessment controller.

3. **Client monitor.** Processes new client events received from the client controller, such as client connections, disconnections, and changes of client locations.

4. **External equipment monitor.** Interacts with the client controller. Gathers information about the external equipment from which the clients are connected to the cloud. This is needed primarily to control whether the clients connect from the allowed IP addresses (some applications may specify whitelists or blacklists of IP addresses from which the clients can/cannot connect).

Figure 3.2 shows the UML component diagram of the monitor module and the external modules it communicates with.

The functionality of the monitoring phase can be easily extended by adding new monitors to the top level monitor, which can be done without modifying any of the existing classes. Also, some of the default monitors can be removed, if needed. For instance, the version of the monitoring phase used in the assessment controller includes only Kubernetes monitor, since the assessment controller does not need to retrieve new applications or work with clients.

3.5.2 Analysis

This phase deals with determining the desired state of the cloud by solving a constraint satisfaction problem (CSP). Internally, it uses the Google OR-Tools constraint satisfaction solver [35]. The phase consists of the following steps:

1. Creating an instance of a CSP, and a corresponding solver.

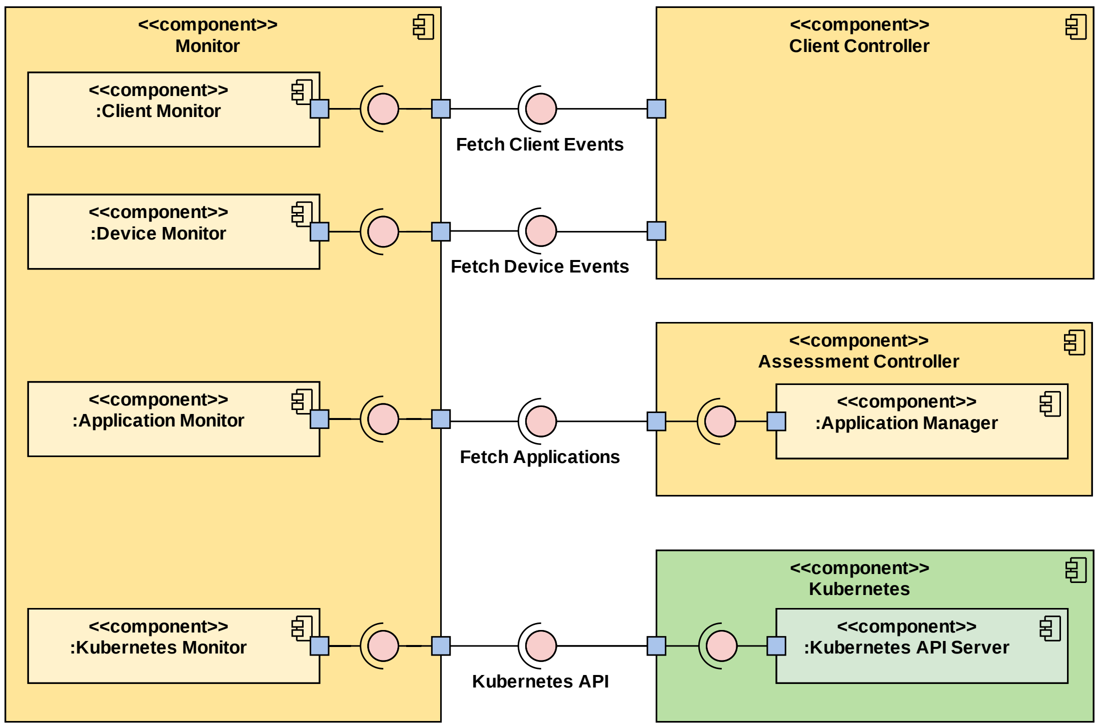


Figure 3.2: UML component diagram of the components participating in the monitoring phase.

2. Adding variables to the problem. Together, the variables represent all possible states of the cloud. E.g., there is a boolean variable for every instance-node and every instance-datacenter pair; if this variable is set to true than the instance should be deployed on that node or in that datacenter, respectively.

3. Adding constraints to the problem. Constraints are needed to filter out the impossible or inconsistent states of the cloud (such as an instance being deployed on two nodes at once). An example of a constraint, expressed in natural language, is ”If the instance A is deployed on the node N, then it is deployed in the datacenter in which the node N is located”. One of the most important constraints is the QoS satisfaction constraint. There is one instance of this constraint for each node in the cluster. Basically, it says: ”QoS requirements of each instance that is deployed on this node must be satisfied”. When this constraint is evaluated, it calls the predictor module to get a prediction for the combination of instances for which the variables are currently set to true. The constraint evaluates as true if and only if the predictor returns a positive prediction.

4. Adding the objective function. An objective function is a numerical expression over the variables that the solver should try to minimize. Basically, it represents the ”cost of the deployment plan” as an abstract number. By default, it consists of three terms, summed together (relative weight of these terms can be configured):

   (a) The number of running nodes. This number is taken as a proxy for energy consumption. Optimizing for this parameter ensures that we use as few nodes as possible while keeping the QoS guarantees.

(b) Distance between the clients and the datacenters in which their dependencies are located. This ensures that the latency between the (latency-sensitive) clients and their dependencies is kept as low as possible.

(c) The number of redeployed instances. Generally, we try to keep the already deployed instances in place to avoid the overhead associated with redeployments.

5. Running the OR-Tools solver on the problem. We let solver to search for the solutions to the problem for 60 seconds (by default). A solution is an assignment of the values to variables under which all constraints are satisfied. A value of the solution is the value of the objective function evaluated over it.

6. Retrieving the solution with the highest value and converting it into a model of the desired state.

If no solution was found in 60 seconds, this can mean one of two possibilities: (i) either there is not enough computational resources in the cluster to support the current number of clients/applications, or (ii) the CSP became so large that it cannot be solved within 60 seconds. In this case, the analysis phase produces the same desired state that was found in the previous iteration of the phase. At the same time, it continues the search for a solution asynchronously, while the rest of the MAPE-K loop continues to run. Then, the analysis phase is effectively skipped until one of the following two events happens: (i) the asynchronous search yields a result, or (ii) an application gets deleted from the framework.

The definition of the CSP problem can be easily altered/extended by adding more variables/constraints, or changing the objective function. All of this can be done without modifying the existing code (see section 4.3.2).

Alternatively, the default implementation of the analyzer module can be changed for a different one. For instance, the implementation of analyzer used in the assessment constroller does not use constraint satisfaction. Instead, it determines the desired state according to the measurement scenarios.

3.5.3 Planning and execution

Planning and execution phases are related to each other by the fact that they both work with *tasks*. Conceptually, a task is a sequence of actions (expressed in a piece of code) that influences the state of the system that we control (state of the cloud, of an instance, of a connected client, etc.). There are multiple *task types*, each type is represented by a class that inherits from the `Task` abstract base class. A particular task instance can be uniquely identified by its task type and parameters, i.e. there cannot exist two tasks of the same type with the same parameters.

All the created tasks are put into the **task registry**. Both planning and execution phases have an access to the registry. Its main purpose is ensuring the integrity of the task system. It prevents task duplication, keeps track of the lifecycle of the tasks, and ensures that tasks are handled correctly (e.g. that the same task does not get executed twice).

The goal of the planning phase is to determine which actions, i.e. which tasks need to be carried out in order to make the current state correspond to the desired state, and to react to the changes that were observed in the monitoring phase. This function is done by a collection of the `Planner` objects. Each planner specializes on some particular type of changes in the system, and is responsible for creation of tasks that deal with those changes. Below is the list of planners present in the framework by default, and their areas of responsibility:

1. **Application creation planner:** tasks that need to be executed once a new application gets deployed into the framework: creation of namespaces, registering applications with the client controller, etc.

2. **Application removal planner:** tasks that remove an application and all of the entities associated with it from the framework after an application removal request is received.

3. **Instance deployment planner:** tasks concerned with the deployment of the cloud instances: creation and deletion of Kubernetes services and deployments, initialization/finalization of cloud instances.

4. **Dependency planner:** tasks related to managing the dependencies between cloud instances.

5. **Client dependency planner:** tasks that send dependency addresses to clients.

6. **Statefulness planner:** tasks related to statefulness management and data migration.

Each of those planners has an access to the model of the system contained in the knowledge module. A planner looks into the relevant parts of the model and decides which tasks need to be created based on the current state of the system, or on the comparison between the current and the desired states. It is also important for a planner to verify that the tasks that were created earlier, but were not yet executed are still relevant. For example, if there is a pending task for creation of an instance that is no longer present in the desired state, instance deployment planner will cancel it.

Execution phase is carried out by the **task executor** module. It fetches the pending tasks from the task registry, and submits them to the thread pool. Each task is executed with an **execution context** specific for that task type. An execution context contains the references to the objects, and/or helper methods that may be needed to carry out that task. For instance, `CreateInstanceTask` is executed with `KubernetesExecutionContext` that provides access to the Kubernetes API. Other contexts can provide tasks with access to such things as the client controller, the statefulness controller, middleware agents of the running instances, etc.

Execution of a tasks is composed of the following steps:

1. Checking the preconditions of the task. A precondition is a function evaluated over the task parameters and the current state that returns a boolean value. For instance, the precondition of the `InitializeInstanceTask` checks whether the instance already exists and is accessible by trying to connect to its middleware agent. If one of preconditions fails, execution of the task cannot proceed.

2. Running the `execute` method of the task. This method contains the main logic of the task that actually changes the state of the system. For some task types the execution can fail even if all preconditions were met. For this reason, this method returns a boolean value. If it returns false, the task is treated in the same way as if it did not meet a precondition.

3. Running `update_model` method of the task. This method ensures that the changes made by the task are reflected in the model.

4. Changing the status of the task depending on the outcome. While the task is being executed, it is in the *running* state. If its preconditions were not met, it gets reverted back to the *pending* state. Successfully executed tasks get to the *completed* state, while the tasks that exit with exceptions get to the *failed* state.

A UML diagram of the main classes participating in the process of planning and execution is shown on figure 3.3.

The planning and the execution phases can be extended by creating new task types, new planners, new execution contexts, and by adding preconditions to already existing task types. All these ways are shown in section 4.3.2.

3.5.4 Client controller

The purpose of the client controller is to manage communication between the framework and the external clients by reacting to client requests and other client-related events. The client controller module has two gRPC interfaces: external (for communications with the clients), and internal (for communication with the framework). It keeps its own internal model that aggregates the information about the clients. Below we describe the functionality of this module by showing how it reacts to different events.

- **The cloud controller fetches the client events.** This event happens in every iteration of the monitoring phase. The client controller responds with a stream of all client events that happened since the last monitoring phase: client connections, disconnections, and location changes.

- **A new application gets deployed.** An application descriptor can contain description of the client types, which includes specifications of their dependencies (i.e. the component types that this client type depends on). If there is at least one client type in the descriptor, the application is added to the client controller's model. Immediately after that, the client controller adds several clients of each of the described types into its model (the exact number can be configured). These clients are called *virtual clients*. They

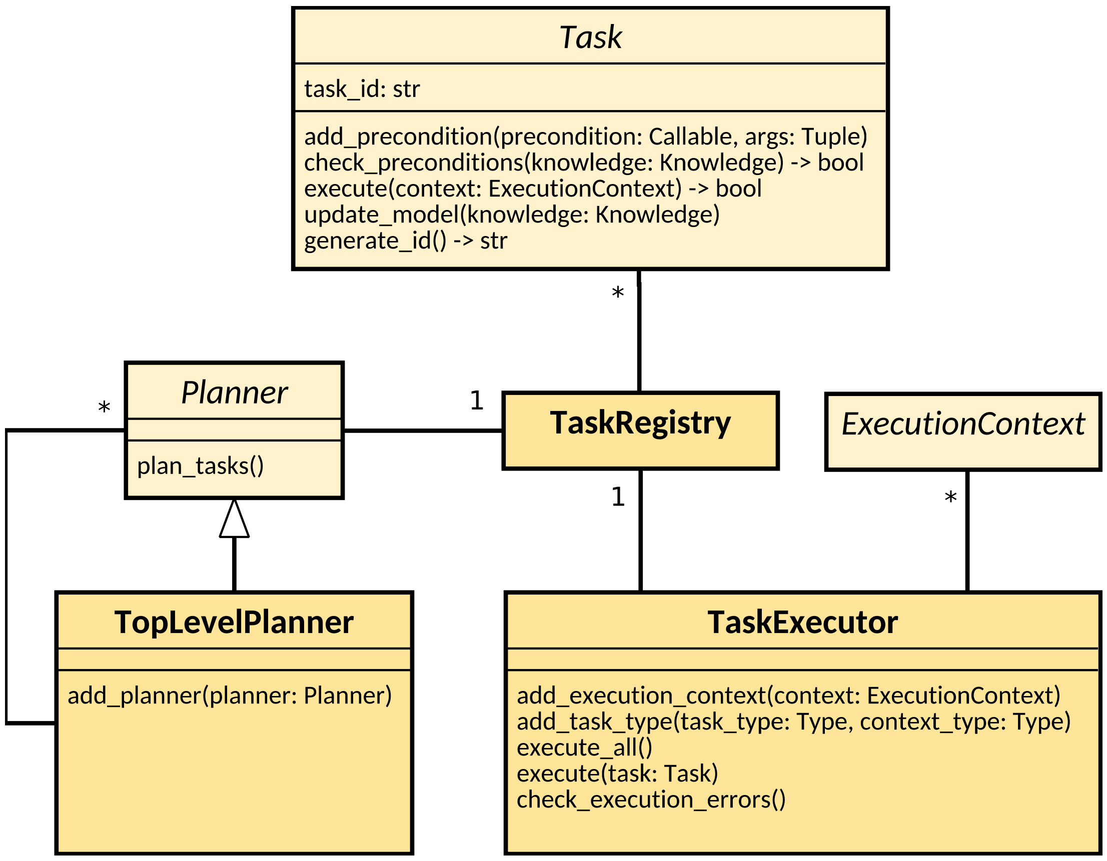

Task
task_id: str
add_precondition(precondition: Callable, args: Tuple)
check_preconditions(knowledge: Knowledge) -> bool
execute(context: ExecutionContext) -> bool
update_model(knowledge: Knowledge)
generate_id() -> str
*
Planner
plan_tasks()
*
1
TaskRegistry
ExecutionContext
1
*
TopLevelPlanner
add_planner(planner: Planner)
TaskExecutor
add_execution_context(context: ExecutionContext)
add_task_type(task_type: Type, context_type: Type)
execute_all()
execute(task: Task)
check_execution_errors()

Figure 3.3: A UML class diagram for the planning and execution phases. Only the main classes and the most important methods are shown.

are created in anticipation of incoming client connections. These clients are reported to the cloud controller so that it would create the instances that would serve them (even though there are no clients to connect to those instances at the moment).

- **A new client connects.** The client controller assigns one of the virtual clients of this type to the newly connected client and sends the client ID to that client. If the instances for that virtual client have already been created, it sends their IP addresses to the client immediately. If the number of virtual clients of a certain type drops below the threshold, a new virtual client is created.

  The client controller keeps an open communication channel to each of the connected clients. Through this channel, it periodically controls whether the client is still connected. If the client is latency-sensitive, the client controller also periodically asks it about its current location.

- **A client changes its location.** Location of the client is represented as a pair of float values that serve as client's coordinates. These coordinates are periodically updated. If after a coordinate update a client falls into the "area of responsibility" of a different datacenter than the one on which its instances are currently deployed, the client controller will register a "client location change" event. This event will be subsequently sent to the cloud controller (which may in response start the process of instance redeployment, data migration, etc.).

- **A client disconnects.** This event is detected if the communication channel between the client and the client controller breaks. If the client reconnects within a short time, nothing happens. Otherwise, the client record in the system gets turned back into a virtual client. If the number of virtual clients of a particular type rises above the threshold, a virtual client gets deleted. The client controller still preserves the information about all clients that have connected before, including their IDs.

- **A client reconnects.** This means that after a relatively long disconnection, a client with an already assigned ID connects again. Client controller assigns the reconnected client to one of the virtual clients.

- **An application gets deleted.** The client controller closes all the communication channels to the clients of that application, and removes the application from its model.

- **A dependency change request comes from the cloud controller.** If this request is associated with a virtual client, the address gets stored in the model. Otherwise, the client controller forwards the address of that dependency to the client through the corresponding communication channel.

3.5.5 Statefulness support

The instances deployed in our cloud are allowed to use any external (located outside of our cloud) data storage if needed. In that case, we cannot guarantee that these instances will have a fast access to those data all the time (since the developers of an application decided to use some services located outside of our cloud, it is up to them to ensure that those services are available). For some applications, like the running example 2 from section 2.3.2, this can be enough. This application does not have any external clients, and thus is not latency-sensitive. Also, its probe is a long-running operation, and data access does not significantly contribute to its response time. Having the data stored outside of our cloud is not likely to be an issue for this application.

However, for the latency-sensitive applications with external clients, data access latency may be crucial. For this reason, we implement our own mechanism that allows us to deal with stateful applications and data migration. This mechanism is built on top of MongoDB [36].

MongoDB is a distributed document database that has a built-in sharding mechanism. This mechanism divides the database into several *shards* - database servers that can run in different locations, while being connected together into a single distributed database. MongoDB uses this mechanism for load balancing between the servers of the database. When sharding mechanism is used, each collection of documents stored in the database is split into *chunks*. The documents are divided into chunks by the value of a particular key contained in them: each chunk is associated with a particular range of values, and a document belongs to the chunk if and only if the value of the key falls within its range. A chunk is always located in a particular shard of the database. Sometimes, MongoDB may move a chunk to a different shard for load balancing purposes.

Load balancer is the component of MongoDB responsible for splitting the collections into chunks, determining the range of keys for the chunks, and moving the chunks between the shards. For the purposes of statefulness support in our framework, we disable the load balancer, and take its functions under our control.

The main idea behind our solution to statefulness support is to use the MongoDB sharding mechanism for data migration. We split the collections into chunks, assigning a single chunk of the collection to each client. This chunk is supposed to be located in a shard that runs in a datacenter in which the instances that serve the client are deployed. When a client changes its location and is reconnected to different instances, its chunks are moved to a shard in the new datacenter.

Using the MongoDB sharding mechanism allows us to avoid any concerns about data integrity and availability during data migration, since the implementation of chunk movement in MongoDB already takes care of them.

There are two main modules responsible for the statefulness support in our framework: the **statefulness controller** that runs as a part of cloud controller, and the **mongo agent** included in the control middleware library.

The statefulness controller is responsible for execution of the tasks related to the control over the data location: sharding collections, splitting them into chunks, moving the chunks.

A mongo agent gives a cloud instance an access to a chunk, while making it look like the instance works with a complete collection. It implements the same interface that a mongo collection does, and thus can be used in completely the same way. Internally, it contains an instance of a mongo collection to which it forwards all the requests. Before doing that, it specifies that those requests should be executed only on the documents with a particular value of the sharding key (that corresponds to a particular client). Since all the documents that have a specific value of the sharding key belong to a the same chunk, this effectively limits the scope of those requests to a single chunk. This allows us to achieve data isolation between different clients.

Overall, this implementation solves all 3 challenges outlined in section 2.7: persistent data storage, data migration, and access control.

3.6 IVIS platform integration

The IVIS platform is a web application with a frontend written in ReactJS and a backend written in Node.js.

Developing this integration required extending both backend (adding communication with the framework) and frontend (extending GUI in order to allow users to create jobs deployable within our cloud)

3.6.1 Main concepts of the IVIS platform

In order to explain how the integration works, it is necessary to lay out the core concepts around which the IVIS platform is built. Those concepts are the following:

- A **signal** represents a piece of information, usually coming from a sensor (e.g. a temperature reading). The signals are grouped into **signal sets**. The incoming values of signals are stored as records in the IVIS SQL database.
- A **task** is essentially a piece of code that specifies a data processing procedure. A task definition can contain multiple **parameters**. A parameter can be either a primitive (e.g. string, integer, float) or a signal set. The code of the tasks is written in Python.
- A **job** is an executable instance of a task. A job must be parametrized by the values of all parameters of its task. Jobs can be exectuted manually (from the IVIS GUI), or automatically (repeatedly, with a specified interval).
- A **template** is a piece of Javascript code that defines the rules for data visualization. In the same way as tasks, a template can be parametrized by primitive types and/or signal sets.
- A **panel** provides a view on the data stored in the database. The data are visualized according to a particular visualization template.
- A **workspace** is an element that groups multiple visualization panels together.

3.6.2 The process of cloud job deployment

Before we have developed the integration, the jobs in IVIS could have been executed only locally - on the same machine on which the IVIS platform runs. We have added an option to deploy jobs in the cloud. If a user checks "Run in Kubernetes cloud" checkbox while creating the job, the job description will be sent to the IVIS interface of the framework (exposed in the cloud controller process). There, this description gets transformed into an application descriptor which is subsequently submitted to the assessment controller in a regular way.

An application descriptor for a job contains a single component with a single probe. The docker image for the container corresponding to this component can be specified in IVIS GUI. Alternatively, the default image can be used instead. The code of the probe is the same as the code of the job's task specified in IVIS GUI. The parameter values of the job are written to the standard input of the probe (in the same way as it is done with regular IVIS jobs).

Once the assessment controller receives the application descriptor, it proceeds with the assessment of the application in a regular way. After the initial (isolation) measurements are completed, the framework sends the results of those measurements to the IVIS server. These results include the values of the response time for several different percentiles that can be guaranteed, as well as the mean throughput. The user can look at those values in IVIS GUI, and then can specify their own QoS requirement for the job (related to either response time or throughput). Then, this QoS requirement gets through the process of application review. If the review is successful, the job gets deployed into the production cloud.

Once deployed, the specified probe can be invoked either manually, with a click of the ”Run” button, or automatically with a time trigger set in the IVIS GUI. In other words, both ways to run jobs provided by the IVIS platform work with the cloud-based jobs as well.

The process of deploying an IVIS job into the framework is shown in section 4.1.3.

3.6.3 Communication with the framework

The IVIS server communicates with our framework through the gRPC interface exposed by the cloud controller process. The IP address and the port on which that interface runs have to be set in the IVIS configuration. Until a user attempts to deploy a job into the cloud, IVIS does not communicate with the framework at all.

The requests to deploy a job, to specify its QoS requirements, or to get its measurement status are forwarded to the assessment controller. Requests to run a probe on an already deployed instance are forwarded to that instance.

On the side of the IVIS server, we have extended the exposed REST API with several methods. The cloud instances can use the new methods in order to report successful and failed runs, download current state of the job, or execute IVIS runtime requests. Since IVIS does not allow external entities to access its REST API without an access token, this token needs to be generated first. When an IVIS user generates an access token, this token is sent to the framework. Afterwards, the access token is sent to every IVIS job instance deployed in the cloud. The need for an access token has two consequences for the interactions between IVIS and the framework:

1. No IVIS jobs can be submitted to the framework until an access token is generated. If an IVIS user attempts to do that, the job will be immediately rejected.
2. If at least one job is already running in the cloud, changing an access token may break the running jobs. A user will get a warning if they try to generate a new access token in this case.

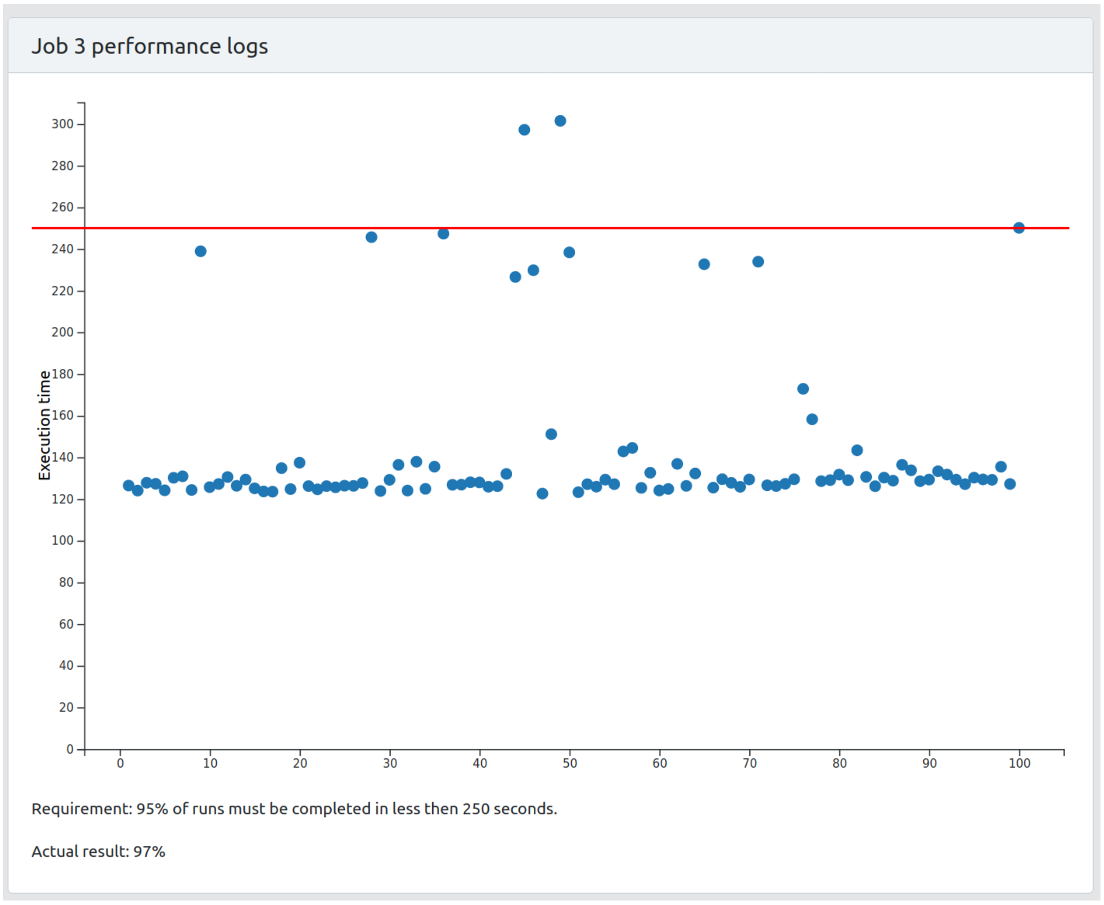
3.6.4 Visualization

The IVIS platform allows users to create visualizations of the data produced by jobs. We have decided to use those capabilities for visualization of the performance data that we collect from the cloud instances.

When a job gets deployed into the cloud, three additional IVIS entities are created automatically for the job. Those objects are:

1. A signal set that stores the values of response times of every run of the job.
2. A visualization template that describes how the data from that signal set should be visualized.
3. A panel that visualizes those data according to the template.

All panels that are created in this way are grouped into a separate workspace that gets created automatically along with the first cloud job. Browsing through the panels in that workspace allows IVIS users to observe the performance of different cloud instances over time. An example of such visualization is shown on figure 3.4.



Requirement: 95% of runs must be completed in less then 250 seconds.

Actual result: 97%

Figure 3.4: Performance visualization of an IVIS job deployed in cloud.

4. Usage of the framework

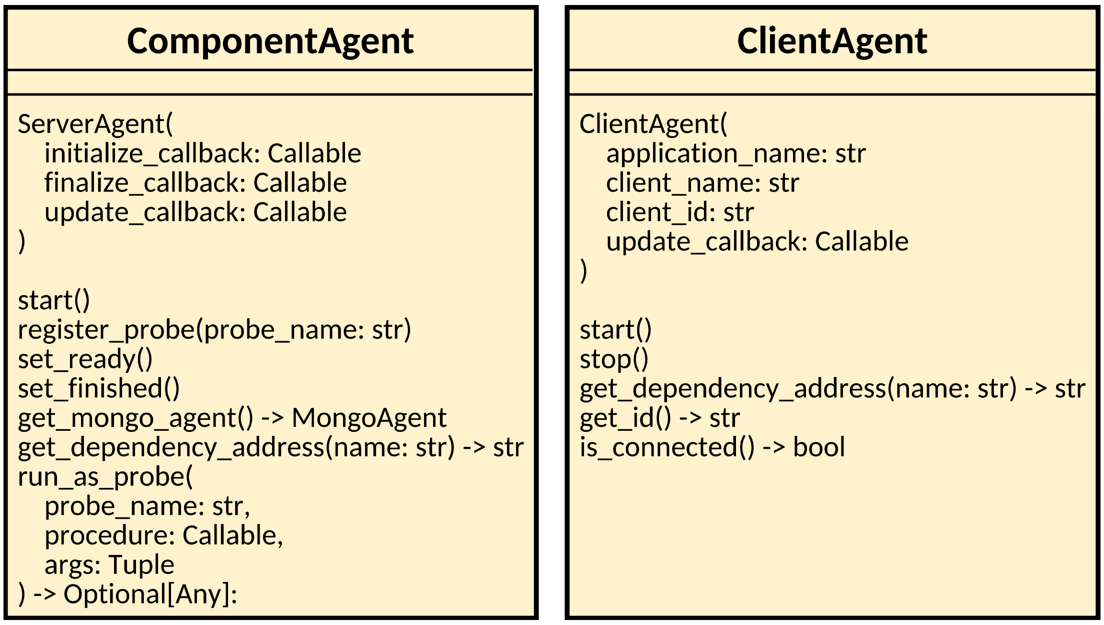
This chapter covers the usage of the framework from three different angles:

1. How to write applications deployable with the framework and how to deploy them.
2. How to run the framework, and how to prepare the insfrastructure on which the framework can run.
3. How to change the framework configuration, and add new modules and extensions to it.

4.1 Developing applications for the framework

4.1.1 Integrating control middleware

All the modules of the control middleware library are located in the `middleware` package. There are three essential classes in the package. `MiddlewareAgent` is the class that exposes the gRPC interface through which the framework controls the cloud instances. `ComponentAgent` and `ClientAgent` are the classes through which the application-side code can interact with the framework and its services. A UML diagram of the application-side middleware classes is shown on figure 4.1.



```
ServerAgent(
   initialize_callback: Callable
   finalize_callback: Callable
   update_callback: Callable
)

start()
register_probe(probe_name: str)
set_ready()
set_finished()
get_mongo_agent() -> MongoAgent
get_dependency_address(name: str) -> str
run_as_probe(
   probe_name: str,
   procedure: Callable,
   args: Tuple
) -> Optional[Any]:
```

```
ClientAgent(
   application_name: str
   client_name: str
   client_id: str
   update_callback: Callable
)

start()
stop()
get_dependency_address(name: str) -> str
get_id() -> str
is_connected() -> bool
```

Figure 4.1: UML class diagram of middleware agent classes.

Middleware agent. As it was said in section 3.2, there are two ways to integrate the middleware agent into a component. The first way is to run the midleware agent as the main program of the container. This can be done, e.g. by setting the entry point of the container to the `run_middleware_agent.py` module.

In this case, management and ititialization of instances is fully handled by the framework. Development of a component for the framework in this case consists of two steps: (i) building a Docker image for the component, and (ii) writing the code of the probes and providing it along with the application descriptor.

The framework includes a default Docker image that integrates the middleware agent in the way described above. With this image the first of two steps can be skipped.

Component agent. The second way to integrate the middleware agent is by instantiating and starting the component agent object in the application-side code. Calling the `start` method on that object starts a non-daemon thread that runs the middleware agent. With this option, the probes can be registered as callable procedures.

Using the component agent requires some degree of cooperation on the side of application developer with respect to controlling the lifecycle of an instance. Instance initialization in this case consists of two parts: framework-side and application-side. The application-side code needs to notify the agent when its initialization is completed through the `set_ready` method. The agent will signal the comletion of framework-side initialization by invoking the callback function provided to it at initialization. An instance can be provided to other instances as a dependency only if both parts of initialization are completed.

Instance finalization is also divided in two parts in the same way. The framework will not remove an instance from the cloud until it is finalized by the application-side code.

After an instance is initialized, the IP addresses of its dependencies can be obtained through the component agent. For stateful components, a mongo agent can be obtained after initialization as well. The `MongoAgent` class implements the same interface as the pymongo `Collection` class.

The component agent also provides a way to report operation performance measurements from the application-side code. This function can be used to monitor whether the performance of an actual operation differs from the performance of the probe (in cases when they are not completely the same).

Client agent. An external client must run a client agent in order to be able to connect to the framework. At initialization, a client agent needs to receive a name of an application and a name of a client type. It communicates these values to the framework, so that the framework would know which commponents the client needs to connect to. Starting the client agent and getting dependency addresses from it works in the same way as in the component agent.

If a client wants to use stateful components and to have access to its persistent state after a restart, it needs to save an ID assigned to it by the client controller after the first connection. This ID has to be supplied to the client agent when it is instantiated again after the restart.

4.1.2 Using deployment tool

The purpose of the deployment tool is to provide users with a way to communicate with the framework. It provides the following functions:

- Submitting applications into the framework. An application descriptor file has to be provided in the command. The format of application descriptor files is explained in section 4.1.2.1
- Getting the status of an application with respect to the assessment and the application review processes (i.e. whether the application is still being measured, is already deployed, was rejected because of too strict QoS requirements, etc.).
- Getting the values of the response time at a given percentile or the throughput for a given probe. A name of the component and the application to which the probe belongs to has to be specified. This can be done only if the application has already been measured.
- Submitting QoS requirements for an application that was submitted earlier. This is possible only if the application descriptor of that application was marked as *incomplete*. In that case, the application will not undergo an application review until the requirements are submitted. The requirements are written in a YAML file; its format is described in section 4.1.2.2.
- Deleting applications from the framework.

Listing 4.1 lists the commands responsible for the functions mentioned above.

```
./depltool.py submit application.yaml
./depltool.py status application
./depltool.py get-time 99 application component probe
./depltool.py get-throughput application component probe
./depltool.py submit-requirements requirements.yaml
./depltool.py delete application
```

Listing 4.1: Deployment tool usage

4.1.2.1 Application descriptor

The most important thing that an application descriptor contains is specification of cloud-based components and external clients.

As it was described in section 3.2, one component instance corresponds to one Docker container running in Kubernetes. A template for that container must be specified in the application descriptor. This template is specified in the format of Kubernetes ContainerSpec [37], and thus must contain a name and a reference to a Docker image. Additionally, it can specify command line arguments, environment variables, resource requests and limits, opened ports, etc. The address of the default port on which the middleware agent runs is added by the framework automatically after the submission.

If the container template is not provided, the default Docker image is used for the component.

Apart from the container, component specification includes the component's cardinality (*single* or *multiple*, see section 2.1.2 for explanation), statefulness (*client*, *component*, or *none*, see section 2.7), and specification of its probes and QoS requirements.

Every component must have at least one probe defined. Probe specification may include a reference to a file that contains its code. Every QoS requirement must reference a probe defined on the same component.

An application descriptor is considered complete by default. If a user wants to specify the QoS requirements later, they have to mark it as incomplete by adding `complete: false` to the descriptor.

In the case an application descriptor references a Docker image located in a private Docker repository, a Docker secret must be added to the descriptor. A Docker secret includes Docker username, password, and e-mail.

An example of an application descriptor is shown in the listing 4.2.

```
application: facerecognition
complete: true
components:                         # A cloud-based component:
- name: recognizer
  cardinality: multiple
  statefulness: client # Per-client statefulness
  probes:                # The probe, its code and std input:
  - name: recognize
    codefile: "./recognize.py"
    inputfile: "./input.txt"
  QoSrequirements:
  - type: time        # Response time under 100 ms in 99% of cases:
    probe: recognize
    probability: 0.99
    time: 100
  # Template for the container specified
  # in the standard Kubernetes format:
  template:
    name: container
    image: d3srepo/recognizer:latest
    args: ["-l", "-a"]
    ports:
    - containerPort: 8888
      protocol: UDP
clients:                             # An external client:
- name: client
  latency: true          # The connection is latency-sensitive
  dependencies:
  - name: recognizer     # Needs an instance of a recognizer
dockersecret:
  username: dockeruser
  password: XXXXXXXXXXX
  email: user@example.com
```

Listing 4.2: An example of an application descriptor

4.1.2.2 QoS requirements specification

A QoS requirements file format is essentially a truncated version of the application descriptor format. It references the application and its components, and extends their specification with QoS requirements.

A QoS requirement must specify its type (time or throughput) and the probe it belongs to. Response time requirements must specify a time limit (in milliseconds), and a percentile (as a floating-point number from 0 to 1). Throughput requirements must specify a time unit (second, minute, hour, day), and a number of executions of the probe that should be possible to run in that time.

Listing 4.3 shows an example of the requirements specification.

```
application: example
components:
- name: recognizer
  QoSrequirements:
  - type: time       # Response time under 100 ms in 99% of cases:
    probe: recognize
    probability: 0.99
    time: 100
- name: detector
  QoSrequirements:
  - type: throughput # Throughput at least 20 runs per hour:
    probe: detect
    requests: 20
    per: hour         # possible values: second, minute, hour, day
```

Listing 4.3: An example of the requirements specification

4.1.3 Deploying applications via IVIS

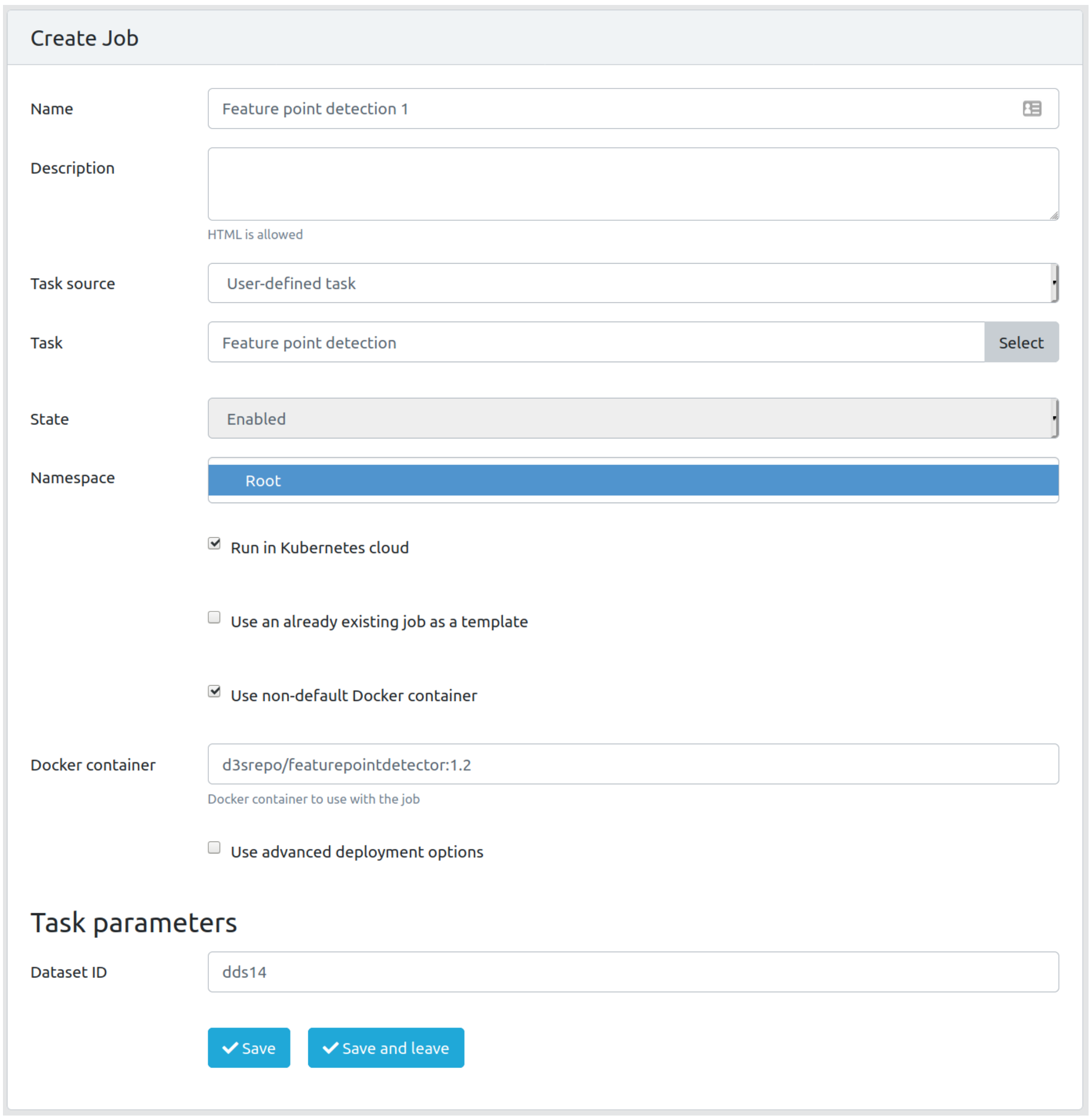
Deploying an application through IVIS GUI is composed of the following steps:

1. **Creating an access token.** This can be done under ”Account / API / Generate access token”. Every time a token is generated, it is automatically sent to the framework.

2. **Creating an IVIS task.** This includes writing the code for the task, and specifying its parameters. IVIS GUI includes a built-in Python code editor that is used for task code specification.

3. **Creating an IVIS job.** By default, the job is deployed locally (as it is normally done in IVIS). In order to deploy it with the framework, the ”Run in Kubernetes cloud” option has to be chosen. In this case, a user can choose a container for the job, and specify advanced deployment options (resource requests and limits, Kubernetes labels for node selector). The values of task-specific parameters are also specified here. Alternatively, a job can be created by using an already existing job as a template. In this case, the task, its parameters, and all other properties except for the name and the description will be copied from the template task. The assessment phase for the new job is skipped in this case, as it is effectively another instance of an already measured component. This step is shown on figure 4.2.

4. **Waiting until the job is measured.** After its creation, a job is sent to the assessment controller, where it gets measured. At this point it is in the *measuring* state. The current state of the job can be observed in the jobs


Create Job

Name	Feature point detection 1
Description	
	HTML is allowed
Task source	User-defined task
Task	Feature point detection
State	Enabled
Namespace	Root
	✓ Run in Kubernetes cloud
	☐ Use an already existing job as a template
	✓ Use non-default Docker container
Docker container	d3srepo/featurepointdetector:1.2
	Docker container to use with the job
	☐ Use advanced deployment options

Task parameters

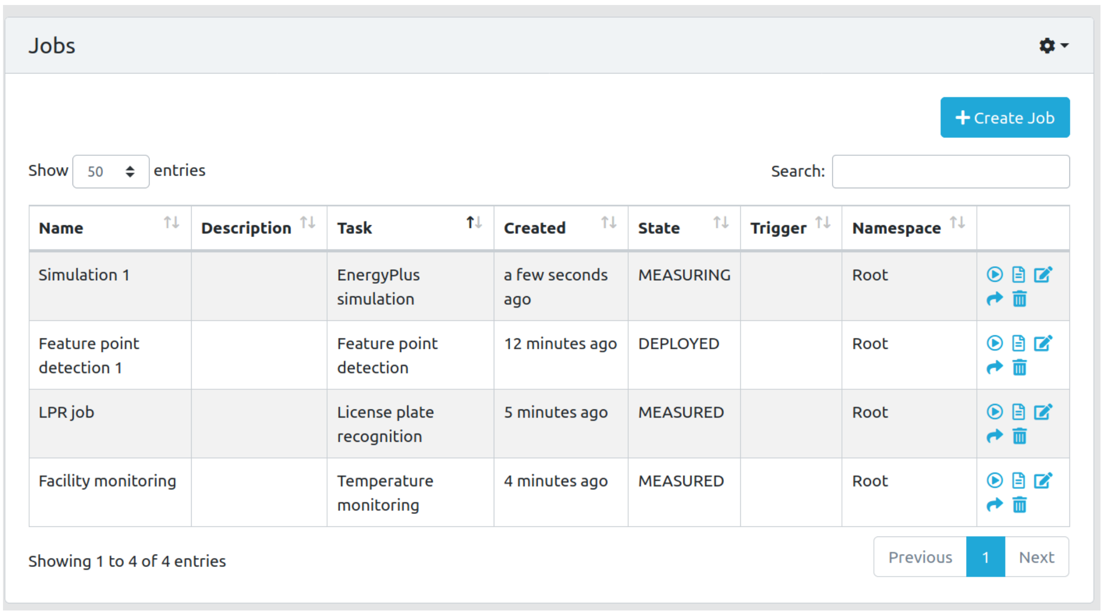
Dataset ID	dds14

Figure 4.2: A cloud job creation panel in IVIS GUI.

table (shown on figure 4.3). A QoS requirement for the job may be specified only after it gets to the *measured* state.

5. **Specifying the QoS requirement.** Once the job gets to the *measured* state, IVIS retrieves its response time statistics and shows them in the job details (see figure 4.4). From those values, a user can estimate which QoS requirements can be guaranteed for the job. In the same panel, a user can specify a response time requirement or a throughput requirement. Once "Save" button is pressed, the requirement is submitted to the framework, and the job undergoes an application review. The job will immediately get to either the *accepted* or the *rejected* state depending on the result of the review. If the job is accepted, the framework will start the deployment and initialization of a cloud instance for the job. Once the instance is up and running, the job will get to the *deployed* state.

6. **Running the job.** If a periodic trigger was specified in the previous step, IVIS will periodically send the requests for job execution to the framework.


Jobs

+ Create Job

Show 50 entries   Search:

Name	Description	Task	Created	State	Trigger	Namespace	
Simulation 1		EnergyPlus simulation	a few seconds ago	MEASURING		Root	
Feature point detection 1		Feature point detection	12 minutes ago	DEPLOYED		Root	
LPR job		License plate recognition	5 minutes ago	MEASURED		Root	
Facility monitoring		Temperature monitoring	4 minutes ago	MEASURED		Root	

Showing 1 to 4 of 4 entries   Previous 1 Next

Figure 4.3: The jobs table in IVIS GUI with several cloud jobs deployed.

Alternatively, a user can run the job at any moment by pressing the "Run" button next to the corresponding entry in the jobs table. When a job run is completed, the instance reports a successful run to IVIS, and sends it the standard output resulting from that run. These outputs can be viewed in the job run logs.

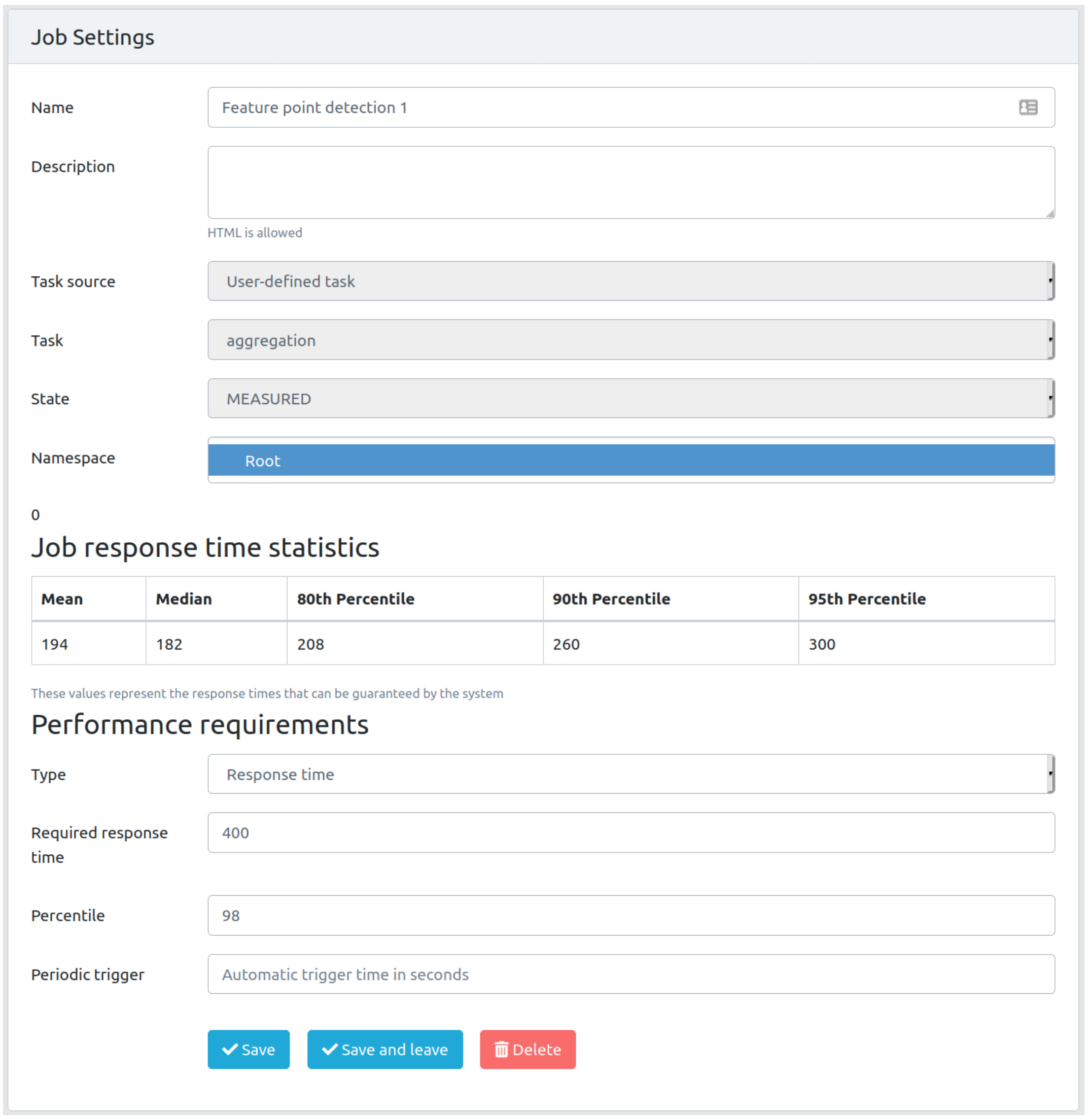
7. **Deleting the job.** After the job is no longer needed, it can be deleted with the "Remove" buttonnext to the job entry. This will delete the job not only from IVIS, but also from the framework. Performance data and the visualization panel corresponding to the instance will stay accessible after the instance removal; they can be deleted separately afterwards.

The performance visualizations can be accessed under "Workspaces / Performance logs".

4.2 Running and maintaining an instance of the framework

Three main processes must be executed in order to run the framework: the cloud controller, the adaptation controller, and the performance data aggregator. By default, these processes assume that they are located on the same machine. However, they can be executed on different machines, if the IP addresses of those machines are specified in the framework configuration.

The cloud controller can run without the assessment controller (in that case new applications cannot be deployed, but the already deployed applications continue to function). Similarly, the assessment controller can run without cloud controller: measuring the submitted applications, but never deploying them. This can be done in order to collect a large measurements database when the application deployment is not needed yet.


Job Settings

Name	Feature point detection 1
Description	
	HTML is allowed
Task source	User-defined task
Task	aggregation
State	MEASURED
Namespace	Root

0

Job response time statistics

Mean	Median	80th Percentile	90th Percentile	95th Percentile
194	182	208	260	300

These values represent the response times that can be guaranteed by the system

Performance requirements

Type	Response time
Required response time	400
Percentile	98
Periodic trigger	Automatic trigger time in seconds

Figure 4.4: A job edit panel in IVIS GUI during QoS requirements specification.

The client controller module also runs in a separate process. Running the client controller is necessary only if support for applications containing clients is required.

4.2.1 Setting up the infrastructure

The framework needs to have an access to two Kubernetes clusters: one for the assessment controller, and one for the cloud controller. The default paths to the Kubernetes configuration files for these clusters can be changed in the framework configuration.

Also, both the assessment controller and the cloud controller must have an access to the Kubernetes overlay network of their respective clusters in order to be able to connect to the running instances. The external clients connecting to the framework also must have an access to the overlay network of the main (production) Kubernetes cluster.

Besides the standard labels added by Kubernetes, every Kubernetes node in both clusters must have the following labels attached:

- A label identifying the hardware configuration of the node. The nodes that have different hardware configurations must have different values of this label. This label is just an abstract identifier and is not required to contain any specific information regarding the node's hardware.
- A label identifying the datacenter to which the node belongs. The nodes in the same datacenter are assumed to have a negligible latency. The framework does not measure the latencies on its own, relying instead on these labels.

If support for stateful applications is required, both clusters must have a sharded MongoDB cluster set up. In addition to the regular requirements to a sharded MongoDB instalation [38], it must to be configured in the following way:

- Every datacenter must contain at least one node with a MongDB shard server. Every replica of that shard must be located in the same datacenter. That node must be labeled with the name of the name of the shard server replica set.
- Every datacenter must contain a node with a mongos interface instance. This interface must be accessible from the Kubernetes overlay network.
- The MongoDB load balancer must be disabled.

The detailed guide on setting up the infrastructure for the framework is available in the digital attachment of the thesis. Additionally, the attachment contains a set of scripts for creation of a virtual simulation of the required infrastructure on a single machine. It creates and provisions the virtual machines for both clusters, and sets all the required properties in an automatic way. The guide to running those scripts is also available in the digital attachment.

4.3 Extending the framework

4.3.1 Configuration files

The settings of the framework can be configured through the configuration files located in the `config` folder. These settings include the IP addresses and ports of all the modules and interfaces, the time limit on the duration of the analysis phase, size of the execution thread pool, and many more. If a value of a setting is present in a configuration file, it overrides the default value set in the source code.

The settings relevant to the middleware library are located in a separate file. It is important to make sure that the middleware settings with which the Docker container was built are the same as those used by the framework.

4.3.2 Extension manager

The implementation of the different submodules of the adaptation loop can be changed with the **extension manager** module. It can be done it two ways: by altering the default implementation of that submodule, or by substituting that submodule with a different implementation that conforms to the same interface. An example is shown in listing 4.4.

```
# First, we need an instance of the extension manager:
extension_mgr = ExtensionManager()

# Now, let us change the objective function for the CS problem:
analyzer: CSPAnalyzer = extension_mgr.get_default_analyzer()
objective = NewObjectiveFunction(analyzer.variables)
analyzer.set_objective_function(objective)

# Adding support for new tasks works in the following way:
executor: TaskExecutor = extension_mgr.get_default_executor()
executor.add_execution_context(CustomExecutionContext())
executor.add_task_type(CustomTask, CustomExecutionContext)

# Changing the monitor for a different implementation:
knowledge: Knowledge = extension_mgr.get_default_knowledge()
monitor = CustomMonitor(knowledge)
extension_mgr.set_monitor(monitor)

# Getting the customized adaptation loop.
# After this call the extension manager will not allow us to
# do any more modifications:
adaptation_loop = extension_mgr.get_adaptation_loop()
adaptation_loop.start()
```

Listing 4.4: An example of usage of the extension manager

All the main submodules of the adaptation loop (monitor, analyzer, planner, executor, knowledge) can be changed and/or modified in this way. Support for an external controller (e.g., similar to the client controller or the statefulness controller) can be added by creating tasks that work with that controller, an execution context and a planner for those tasks, and (if needed) a monitor to receive the updates from that controller.

5. Discussion

It this chapter we discuss the results of our work. First, we show how the implemented framework deals with the specific requirements of the application types it was designed for. Then, we discuss the overall efficiency of the framework, its current limitations, and what those results say about the success of the chosen approach. We also discuss possible solutions to the problems that may need to be addressed in order to turn this proof-of-concept framework into a production-grade software. The chapter ends with an overview of the related work in the field of QoS management in cloud.

5.1 Satisfying the requirements of different application types

In this section we discuss how our framework works for the two running examples described in section 2.3.2. In order to demonstrate the functionality of the framework, we have created our implementations of both of these examples and tested them.

5.1.1 Running example 1: Face recognition

The face recognition example is a client-server application that features a large number of geographically distributed clients. Moreover, these clients are mobile: they can move from one location to another over time. This application has the following specific requirements:

Probabilistic guarantees on the response time. Providing these guarantees is one of the main goals of this work. Our framework makes sure that the performance of the applications gets assessed properly and that the instances are deployed only in the combinations allowed by predictor. Our measurements show that, if accepted by the framework, the SLA for this application remain satisfied over the long term. However, this still does not mean that they will be satisfied for any application or combination of appications, since the predictor we use can give false positive predictions. We discuss this issue and the ways to deal with it in section 5.2.2.

Availability of the recognition servers. When a new client connects to our framework, it needs to get an address of a face recognition server instance. Preferrably, it needs to get it as fast as possible. If there is no such instance running in the cloud when a client connects to the system, the client would need to wait until the framework determines where to deploy that instance, creates it, and initializes it. This may take up to two minutes in the worst case. For this reason, our framework always expects some number of incoming clients and creates instances for them in advance. The number of spare instances to be created can be configured as a fixed number or as a percentage of already conected clients. This way, the clients receive their insances immediately after connection. A situation in which a client needs to wait to get an instance may still happen if too many new clients attempt to connect to the cloud at the same time. However, this problem can be prevented by configuring the number of spare instances properly, and with proper configuration it is not likely happen.

Low end-to-end latency. In order to ensure that the latency between the client and its recognition server is as low as possible, we always connect a client to an instance in the closest data center. This is an optional QoS requirement that can be specified in the application descriptor. It may happen that when a client connects to the framework, there may not be a suitable instance in the closest datacenter. In this case, the client will be connected to a different instance, and the end-to-end latency may initially be higher. However, the framework will start redeployment of that instance during the next execution phase. Thus, the time during which an instance will have a suboptimal latency is minimal. Since minimizing the end-to-end latency is not a strict requirement, we consider it to be satisfied

Data migration. The basic version of the face recognition example presented in section 2.3.2 is a stateless application. However, it can be extended to store some of the user data on the server side (e.g. some statistics about the number of faces recognized, or some of the images with the recognized faces, etc.). In this case, the server-side application will likely need to have a fast access to the stored data. To ensure that, our framework implements a data migration mechanism over MongoDB. This mechanism ensures that the data of the client are moved to the same datacenter where the instance serving that client is located. Since this process is not instantaneous, there may be a (relatively short) period of time when the instance has a higher-latency access to those data. However, as users do not change their geographic locations in an instant, in most cases this means having only a slightly higher data access latency for 10 to 20 seconds after redeployment.

Overall, we can conclude that our framework provides support for all essential requirements of client-server latency-sensitive applications.

5.1.2 Running example 2: Drone footage processing

Drone footage processing example is a long-running, computationally-intensive application. We have implemented this example as an application deployable through IVIS. This application contains only one component. That component has a single probe that fetches an image from a database, detects feature points on it, and stores them back into the database. This demonstrator was initially created to show the applicability of our framework in the context of the AFarCloud project [27], in which it is currently being used.

The specific requirements of this application and the ways in which we address them are listed below.

Guaranteed throughput. This is the main requirement of the application. The framework allows users to specify the throughput requirements and is able to guarantee them. Even though the currently existing predictor implementation does not support prediction of throughput, we have implemented a workaround that allows us to make predictions about it with a constant number of requests to the predictor as is shown in section 3.3.3.1.

Number of instances is determined by the user. In this case, there are no clients that connect to the framework and demand an instance to connect to. Instead, the user that deploys the application (through the IVIS GUI), can create additional instances by creating an IVIS job and specifying the component ID assigned to the already deployed component. This way, the number of instances can be changed manually when needed (e.g. if more drones are deployed, and thus more footage needs to be processed).

Ease of operation execution. Since there are no clients connected to this application, it should be possible to execute the feature point detection operation without being connected to the framework as a client. This is done, again, with the IVIS framework GUI. The user can run the operation manually by clicking the "Run" icon next to the corresponding instance entry, or by specifying a periodic trigger for that instance. In either case, no continuous connection to the framework or to the instance is needed.

Energy efficiency. Even though this is not an explicit requirement of the application, this is something that the maintainers of the cloud infrastructure will potentially be interested in while running many computationally intensive instances at once. For this reason, energy consumption is taken into consideration while making deployment decisions. The framework tries to minimize the number of nodes used so that some of the spare nodes could be turned off, thus reducing energy consumption. We envision that in the future the framework may be also used to control the number of running nodes in the cluster directly. This can be done by creating additional executable tasks for turning the nodes on/off and specifying the conditions under which they should be executed.

5.2 Efficiency of assessment and prediction

The efficiency of our approach depends not only on the efficiency of the framework presented in this thesis, but also on the parts developed outside of its scope, namely the assessment methodology [25] and the predictor module [26]. The analysis of the efficiency of these parts of the approach was already discussed in detail by their respective authors in the cited works. However, we still would like to discuss those results in the context of the approach as a whole, taking into account the requirements that arise when those modules are used as a part of our framework.

5.2.1 Duration of the assessment phase

The currently used assessment methodology uses low-overhead performance measurement methods, such as perf tool [39] and `/sys/block` files. Our implementation of the assessment controller module also does not have any significant overhead: creation of a measurement scenario, deployment of that scenario, and data collection take only a couple of minutes combined. Thus, the main factor determining the duration of the assessment phase is the number of probe executions and the average response time of those probes.

Measurement of a single scenario normally involves 200 to 1000 probe calls. With a probe response time in the order of magnitude of tens of milliseconds (which is the case of our first running example), that means that a scenario can be completed in minutes. With response time of 1-5 minutes, that is several hours to several days. In order to determine whether an application can be deployed in the cloud, we need to run at least one scenario. However, normally we would want to measure several more scenarios to profile the application in order to be able to make meaningful predictions for it. Moreover, since the predictions for different hardware configurations are independent, this number has to be multiplied by the number of hardware configurations that we want to measure the application for.

Altogether, this means that even given the fact that our framework supports parallel execution of measurement scenarios, the assessment phase may take hours or even days for some applications. This is a limitation inherent to our approach that has to be taken into account. However, it still makes sense to deploy the applications containing long-running probes within our framework if those applications are meant to run for weeks or months after their deployment.

For the applications that do not contain any long-running probes, the application review can be performed in several minutes, while the subsequent profiling of its operations takes less than an hour. We consider this to be sufficiently fast.

Adding a new hardware configuration to the cluster may require to measure hundreds of scenarios in order to build the initial performance database of the predictor, which may take up to a week. Consequently, it is better to keep the number of hardware configurations in the cluster not very high. In general, the more homogeneous the cluster under the framework's management is, the more efficiently our approach works.

5.2.2 Accuracy of prediction

One of the most important factors determining the efficiency of the chosen approach is accuracy of prediction. Since implementation and verification of the predictor falls outside of the scope of our work, we can rely on the detailed analysis performed by the creator of the predictor module [26].

In short, that analysis had determined that around 5% of the predictions made on the combinations that were not measured are false positives (cases where predictor predicts that the QoS requirements are going to be satisfied, and they turn out to be violated). The ratio of false positives to true positives gets lower the more data we measure. For this reason, we run measurements in the assessment cluster continually and supply their results to the predictor, making the predictions more accurate over time. However, since the total number of combinations may be very large, in most cases this will not allow us to completely get rid of false positive predictions.

False positives are a major challenge for our approach, since our framework works on the assumption that the predictions are accurate. However, even if an implementation of a predictor that does not give any false positive predictions will never exist, there are still ways to solve that issue within our approach.

In our framework, we have implemented the performance data collection mechanism that allows us to detect when an instance starts violating its SLA. With this mechanism, we can improve the prediction by noticing that a particular combination violates the SLA of one of the instances and ”forbidding” that combination. This can be done, for instance, by implementing a two-layered predictor, where the top layer will only look up the combination in the list of forbidden combinations, and if it is there, will yield a negative result right away. Otherwise, it will forward the request to the second layer, that will carry out the regular prediction process. Since, with our current prediction methods, false positives appear systematically (i.e. there is no randomness involved), this would allow us to mitigate this phenomenon with time. Moreover, forbidding the false positive combinations once we detect them can help to preserve the probabilistic guarantees over the long term even in cases where they will be initially violated due to inaccurate predictions.

5.2.3 Speed of prediction

One prediction takes 10 milliseconds on average. This means that if we perform reconfiguration of the cloud once a minute, we can allow ourselves up to 6000 consequtive calls of predictor. Our tests show that with our current implementation of CSP solver, we can find an optimal deployment for around 1000 instances in that time. Performance of the predictor is a real bottleneck here since, according to our performance measurements, the calls to the predictor take 85-92% of the total time taken by the analysis phase. When we have started the development of the framework, our expectation was that the prediction will work faster and thus will allow us to deal with even larger number of instances. However, even if a faster version of the predictor will never be developed (or, if it turns out that a faster prediction rate is not possible to achieve), there are still ways how to scale the approach far beyond the limits of the current implementation. We discuss our suggested ways to address this problem in section 5.3.1.

5.3 Efficiency of the framework

As the efficiency of the assessment controller and the predictor was discussed in the previous section, here we discuss only the efficiency of the adaptation controller.

The overall efficiency of the adaptation loop is composed of the efficiency of its individual phases. Our performance analysis suggests that the analysis phase will likely be the biggest bottleneck standing in the way of scaling up our approach. For this reason, we discuss this phase separately and propose the ways to deal with that obstacle.

5.3.1 Efficiency of the constraint satisfaction approach to analysis

As it was mentioned in section 5.2.3, the current implementation of the analysis phase based on constraint satisfaction programming and repeated calls to the predictor can deal with situations where up to 1000 instances are deployed into the cloud, with the default time limit on the search of 1 minute. We suggest three approaches that can be taken to scale the approach beyond 1000 instances. These approaches are not mutually exclusive, and implemented together will potentially be able to deal with the number of instances close to the Kubernetes upper limit of 150 000. We discuss these approaches below.

Partitioning the CSP into smaller subproblems. The main problem that pevents us from scaling the constraint satisfaction approach is that it is a sequential, centralized process. The process of finding a solution to a constraint satisfaction problem cannot be parallelized or distributed. However, if we manage to compartmentalize this problem by splitting it into smaller subproblems, we can search for the solutions to those subproblems in parallel. This partitioning can be done, for instance, on the level of datacenter. This way, the analysis phase can be divided into two levels, where the top level will assign the instances to the datacenters, and the bottom level will consist of multiple subproblems assigning the instances to the nodes inside a datacenter. These subproblems will be solved in parallel, which may potentially greatly reduce the time needed to complete the analysis phase.

Increasing the duration of the analysis phase. The duration of the analysis phase can be increased to around 10 minutes. There is one problem that can potentially arise when taking so long to determine the new desired state. In this time, the conditions in the cloud may change so much that the desired state that is being calculated based on the data collected 10 minutes ago becomes obsolete before the calculation finishes. However, this problem is not as significant as it may seem. Let us look at the changes that can happen in the cloud during that time:

1. New applications are deployed into the system. The framework processes the requests for applicaton deployment in the monitoring phase, thus at worst a newly accepted applications will have to wait one whole iteration of the adaptation loop before being deployed. However, as we have discussed in section 5.2.1, the assessment process for an application can normally take hours to days, so adding 10 more minutes on top of that will not make a noticeable difference.

2. New clients connect to the system. A newly connected client needs to get addresses of their dependencies as soon as possible; making clients wait for 10 minutes is not acceptable for most applications. Yet, since our implementation creates some number of spare containers for the clients that may connect in the future, a client does not need to wait for a full iteration of the adaptation loop to get their dependency addresses. The only thing that needs to be taken care of is estimation of the number of spare containers

for each client type that need to be created in order to serve all the clients that may connect during an iteration of the adaptation loop

3. Somewhere in the cloud, an SLA violation is detected. This means that some particular combination of instances was found to be a false positive prediction. After being detected that combination needs to be added to the list of forbidden combinations and the cloud has to be reconfigured in order not to use that combination. However, in order to establish that this combination is a false positive, the framework needs to collect enough performance data from that combination to make sure that this observation is statistically significant. In most cases that will take more than 10 minutes, so adding 10 minutes on top of that (in the worst case) will not change the situation much.

4. Clients change their locations. When a client changes its location, it needs to be reconnected to a different instance. Also, for stateful applications, the data need to be moved to the new datacenter. Otherwise, the client will have a higher-latency connection for those 10 minutes. Both of these tasks are carried out in the execution phase. However, as they are not related to the deployment of the instances, they still can be executed quickly if we will make the analysis phase asynchronous.

Making the analysis phase asynchronous. The CSP problem solving that happens in the analysis phase is needed only in order to determine the new deployment plan of the cloud instances. However, creation, deletion, and redeployment of instances are not the only tasks that are carried out in the execution phase. Such things as changing an address of a client's dependency or migrating client's data to a different datacenter can be done without determining the new deployment plan. Thus, in this design, the rest of the adaptation loop will run continuously while the new deployment plan is being calculated. Then, let us say once in 10 minutes, a new deployment plan is produced, and that triggers reconfiguration of the cloud. This method is already partially implemented in our framework: when a new deployment plan cannot be found in a short time, our framework launches a long-term computation of the desired state, and in the meantime the previously determined desired state is used. Making this method the default may probably be necessary in order to scale our approach.

Overall, we can conclude that the constraint satisfaction approach proved to be efficient enough for our purposes. We expect that with several adjustments, this approach can be scaled up to the level of real-life Kubernetes clusters. Even more important characteristics of this approach are its flexibility and open-endedness. It allows developers to easily add additional variables and constraints into the problem, thus providing a way to deal with possible changes of the requirements put on the framework.

5.3.2 Efficiency of monitoring, planning, and execution

Monitoring phase consists of fetching the data from other processes that belong to the framework, querying the Kubernetes API server to list the resources currently present in the cluster, and updating the model based on the results of those queries. The longest step out of those is querying Kubernetes, and that step still can be completed within several seconds even for very large Kubernetes clusters [40]. Thus, it does not present a challenge for scaling the approach.

Planning phase is the shortest phase of the adaptation loop, since it does not include calls to any external entities or any long-running calculations. Comparing two cloud state data structures and creating tasks based on that comparison is a very fast process.

Execution phase can potentially take a lot of time, since for large clusters there can be a lot of tasks, and individual tasks may take up to two minutes to execute. However, this phase can be parallelized very efficiently, since individual tasks are mostly independent of each other. Our implementation takes this observation into account and parallelizes the execution. Moreover, it makes the execution of long-running tasks asynchronous, thus bringing the duration of the phase down to several seconds.

The part of execution phase that deals with statefulness support (and data migration in particular) will likely be less scalabale due to the relatively low speed of chunk migration in MongoDB. For this reason, we think that it is likely that in production environment statefulness support will have to be solved differently. Our solution to statefulness, however, is not a part of our approach to QoS management. Therefore, the fact that it may turn out not to be scalabale does not have any implications about scalability of the approach.

Overall, these three phases work very efficiently, and we do not expect any of them to become a bottleneck while scaling the approach to the level of production-grade software.

5.4 Summary

Taking into account all the results outlined above, we can conclude that the approach taken in this work has a potential to be successfully applied in the production environment.

The performance analysis of our proof-of-concept implementation shows that there are no major unsolvable obstacles standing in the way of scaling up the approach. Some additional research into prediction methods could be needed in order to make prediction more accurate and reduce the ratio of false positives. However, the approach can still be applied even if the accuracy of the prediction methods stays the same. In this case, occasional violations of the SLA may still be observed for some time after application deployment, but they will disappear over time as the database of the predictor grows.

We see two inherent limitations of the approach. First, the application assessment phase can last hours to days for the applications containing long-running probes. This, however, should not be a problem for the applications that are meant to run for months following the initial assessment. Second, the approach requires some degree of cooperation on the side of application developers, since they need to make sure that probes represent the real operations well enough. There may be a need to create a method for assessing how well a probe represents an operation that it stands for.

Also, additional research may be needed in order to establish the limits of applicability of the approach. Currently, it is not known which levels of guarantee can be realistically achieved for various application types (e.g. whether it is possible to provide 99.9% guarantee on operations with typical response time under 100 milliseconds). This would require large-scale testing of different combinations of applications on various hardware configurations in order to collect a statistically significant dataset. One of the goals of this work was to provide a tool that can be used for carrying out such research, and we can conclude that this goal was achieved.

5.5 Related Work

In this section, we overview the related work in the field of QoS-aware cloud application deployment, and compare it with the approach presented in this work.

The **CloudPick** framework [41] targets QoS-aware service deployment in a multi-cloud environment (where multiple cloud providers are available). It looks at the problem from the user side, choosing a provider for every service in their application in a way that optimizes the deployment of the application with respect to the considered QoS metrics (which are latency, reliability, and cost). Thus, it deals with the cloud services that belong to a single user, and does not take into account the impact that different applications running in the same cloud have on each other.

In contrast, we focus on the cloud provider side, and control the deployment of all submitted applications in a QoS-aware way. Additionally, our approach can provide guarantees on performance QoS metrics.

Cloudroid [42] is a framework that supports QoS-aware cloud application deployment for robotic systems. Since robotic systems are often time-critical, the main QoS target of the framework is timeliness. It works by outsourcing the computational tasks run by robots to the cloud and allocating the required computational resources for those tasks on cloud servers. In cases when the QoS cannot be guaranteed due to unavailability of the required resources, it falls back to local computation (using the robot's harware). Thus, there are several levels of QoS in this framework, with the lowest level being guaranteed by the hardware located outside of the cloud. This is different from our approach in which we use the control over deployment of instance combinations instead of relying on resource allocation.

The **Q-Clouds** system [44] manages deployment of virtual machines in a performance-aware way by monitoring the interference between the VMs, and provisioning additional computational resources for them when needed. Thus, it aims to provide the applications with the resources that they need for achieving a desired QoS level. Our approach, in contrast, is based on measuring the impact of interference between the processes on their performance, and providing guarantees on the QoS metrics directly, instead of resource availability. Additionally, we use container-based technologies instead of virtual machines.

The **DDS** framework [43] proposes an approach to a deadline-aware deployment of time critical workloads in cloud. It uses the Earliest Deadline First algorithm that manages the priority of the workloads based on the explicitly stated deadlines. In particular, it deals with the slowdown caused by shared network bandwidth, and employs a dynamic network badwith allocation mechanism.

Thus, this work also focuses on the control over resource allocation, in this case the network capacity. In contrast to this work, our framework takes into account the interference that the collocated workloads have on each other's performance.

Pythia [45] follows an approach similar to ours by relying on pre-assessment of workloads, their characterization, and prediction of their performance. In its pre-assessment phase it measures the contention for system resources between different processes and uses those data to predict the contention for the collocated process combinations that were not measured directly.

In our approach, instead of measuring resource contention, we measure the performance of the workloads directly, and provide guarantees on the individual measured operations.

In general, we can conclude that most of the related work in the field is concerned with the impact of resource availability on the performance of cloud workloads and with dynamic resource allocation. In this work we propose an alternative approach, based on performance measurement, prediction of workload performance, and using that prediction for controlling the wokload deployment in a QoS-aware way.

6. Conclusion

QoS management is an important area of research in the field of cloud computing. Providing and guaranteeing the quality of service is necessary in order to expand the number of use cases in which cloud computing (including the novel architectures such as edge computing and fog computing) can be used. In this thesis, we propose an approach to manage and guarantee performance-related QoS parameters and present a framework that demonstrates this approach.

As we have said in the section 1.1, our main goal was to develop a proof-of-concept framework for providing QoS guarantees. In chapter 2 we have shown that providing strict performance guarantees (similar to the guarantees existing in real-time systems) is not possible in the cloud environment. Instead, we have based our approach on soft probabilistic guarantees which allow us to tolerate unpredictable lags in the software stack.

The core principles of our approach are (i) measuring the performance characteristics of the applications, (ii) predicting the performance of combinations of applications, and (iii) controlling the deployment and the environment of every instance in such a way that its predicted performance does not violate the QoS expected from that instance. While the measurement methodology and the core functionality of the predictor module were developed outside of the scope of this thesis, implementing the rest of the approach is the result of this work.

The central part of the developed framework, the cloud controller, is a self-adaptive system that works on top of Kubernetes. It does not touch most of the functionality of Kubernetes, but instead of relying on the Kubernetes scheduler it makes deployment decisions on its own. These deployment decisions take into account QoS requirements of individual applications, their performance characteristics, and the impact that application instances have on each other's performance.

To demonstrate the ability of this framework to manage multiple QoS parameters, we have focused on two performance metrics: response time and throughput. We have shown that our framework is capable of providing probabilistic guarantees on both of these metrics. Apart from providing probabilistic guarantees, our framework is capable of optimizing the deployment of applications in order to minimize or maximize certain QoS metrics. We have demonstrated it on minimization of end-to-end latencies between the cloud instances and external clients and minimization of the number of physical servers used in datacenters (which we use as a proxy for energy consumption). Thus, both types of QoS management introduced in section 1.1 are supported.

Another important result of our work is integration of our framework with the IVIS platform. This integration extends the capabilities of the IVIS platform by adding an option to deploy the IVIS jobs in the cloud managed by our framework, and to specify the QoS requirements for them. Additionally, we use the IVIS platform for visualization of the performance data collected from the deployed applications, and as a GUI through which the applications can be deployed into the system (in addition to the standard command line deployment tool).

Integration with the IVIS platform allowed us to apply our framework in the context of AFarCloud [27] and FitOptiVis [33] international research projects.

Since the framework is meant to be used for future research into QoS management in cloud, it is designed to be fully configurable and extensible. The framework is composed of replaceable modules and provides multiple ways to extend its functionality.

We can conclude that this framework provides a full proof-of-concept implementation of our approach. The efficiency of this implementation as well as the effectiveness of the chosen approach is discussed in chapter 5.

Apart from being a proof-of-concept for the chosen approach to QoS management, this framework can be used for the future research into managing QoS in cloud environments.

6.1 Future work

There are many directions in which this work could be potentially expanded. They can be broadly divided into the following groups.

Using the framework for additional research into performance of the cloud applications. Creating additional cloud applications for the framework, deploying them, and measuring their performance. This can potentially reveal that the performance of some types of applications cannot be reliably predicted with the currently used methods, which can help to expand the approach.

Extending the existing framework. The extension mechanisms built into the framework make it possible to add support for additional use cases and application types that were not considered during the development of the framework (e.g. more ways to provide statefulness, support for some specific types of external devices, etc.). A more complicated, but still possible task would be adding support for additional QoS metrics (e.g. guaranteed data transfer rate for network-intensive applications). Of course, this may require to extend not only the framework itself, but also the measurement and prediction methodology.

Bringing the framework from proof-of-concept to production stage. Since this framework was always considered to be a proof-of-concept, it does not possess some of the properties that are expected from the production-grade software. In particular, the APIs through which the framework interacts with the outside world (as well as those through which different parts of the framework interact with each other) should be made more secure. More extensive testing should be done in order to improve failure tolerance. User interface may also be improved to make the system easier to interact with. For instance, a GUI dashboard that would show the current and the desired states of the cloud would be beneficial. Also, the production version of the framework should be tested on large-scale real life clusters.

Improving efficiency of the approach. Improving prediction techniques in terms of both speed and accuracy would probably have the largest effect on the framework's efficiency. Also, looking into the ways to compartmentalize the constraint satisfaction process in the analysis phase can potentially improve the scalability of the approach.

Additional research into QoS and deployment in cloud. At the moment, we do not know how much more expensive it is to provide 99% guarantee vs 99.9% guarantee. Therefore, research into estimating the costs of providing the requested QoS guarantees can be very useful. Also, measurement methodology can be expanded in order to be better suited to measuring the performance of applications that can serve a large number of clients at the same time.

Bibliography

[1] Carlos Rodríguez-Monroy, Carlos Arias, and Yilsy Núñez Guerrero. The new cloud computing paradigm: the way to IT seen as a utility. *Latin American and Caribbean Journal of Engineering Education*, 6:24–31, 12 2012.

[2] Abu Zamani, Md Akhtar, and Sultan Ahmad. Emerging cloud computing paradigm. *International Journal of Computer Science Issues*, 8, 07 2011.

[3] Tanweer Alam. Cloud computing and its role in the information technology. *IAIC Transactions on Sustainable Digital Innovation (ITSDI)*, 1(2):108–115, Feb. 2020.

[4] Tiago Oliveira, Manoj Thomas, and Mariana Espadanal. Assessing the determinants of cloud computing adoption: An analysis of the manufacturing and services sectors. *Information and Management*, 51(5):497 – 510, 2014.

[5] L Minh Dang, Md Piran, Dongil Han, Kyungbok Min, Hyeonjoon Moon, et al. A survey on internet of things and cloud computing for healthcare. *Electronics*, 8(7):768, 2019.

[6] Shahla Asadi, Mehrbakhsh Nilashi, Elaheh Yadegaridehkordi, et al. Customers perspectives on adoption of cloud computing in banking sector. *Information Technology and Management*, 18(4):305–330, 2017.

[7] H. Wang, W. He, and FK. Wang. Enterprise cloud service architectures. *Information Technology and Management*, 13:445–454, 2012.

[8] Ling Qian, Zhiguo Luo, Yujian Du, and Leitao Guo. Cloud computing: An overview. In *IEEE International Conference on Cloud Computing*, pages 626–631. Springer, 2009.

[9] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[10] Jin Ho Kim. A review of cyber-physical system research relevant to the emerging it trends: Industry 4.0, iot, big data, and cloud computing. *Journal of Industrial Integration and Management*, 2:445–454, 2017.

[11] A. R. Biswas and R. Giaffreda. Iot and cloud convergence: Opportunities and challenges. In *2014 IEEE World Forum on Internet of Things (WF-IoT)*, pages 375–376, 2014.

[12] H. F. Atlam, A. Alenezi, A. Alharthi, R. J. Walters, and G. B. Wills. Integration of cloud computing with internet of things: Challenges and open issues. In *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 670–675, 2017.

[13] M. Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[14] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing—a key technology towards 5g. *ETSI white paper*, 11(11):1–16, 2015.

[15] A. V. Dastjerdi and R. Buyya. Fog computing: Helping the internet of things realize its potential. *Computer*, 49(8):112–116, 2016.

[16] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[17] Y. Zhang, K. Guo, J. Ren, Y. Zhou, J. Wang, and J. Chen. Transparent computing: A promising network computing paradigm. *Computing in Science Engineering*, 19(1):7–20, 2017.

[18] Ju Ren, Deyu Zhang, Shiwen He, Yaoxue Zhang, and Tao Li. A survey on end-edge-cloud orchestrated network computing paradigms: Transparent computing, mobile edge computing, fog computing, and cloudlet. *ACM Comput. Surv.*, 52(6), October 2019.

[19] Danilo Ardagna, Giuliano Casale, Michele Ciavotta, Juan F. Pérez, and Weikun Wang. Quality-of-service in cloud computing: modeling techniques and their applications. *Journal of Internet Services and Applications*, 5, 09 2014.

[20] Isaac Odun-Ayo, Olasupo Ajayi, and Falade Adesola. Cloud computing and quality of service - issues and developments. 03 2018.

[21] Anton Beloglazov, Rajkumar Buyya, Young Choon Lee, and Albert Zomaya. Chapter 3 - a taxonomy and survey of energy-efficient data centers and cloud computing systems. volume 82 of *Advances in Computers*, pages 47 – 111. Elsevier, 2011.

[22] Mohammadhossein Ghahramani, MengChu Zhou, and Chi Tin Hon. Toward cloud computing qos architecture: Analysis of cloud systems and cloud services. *IEEE/CAA Journal of Automatica Sinica*, 4:6–18, 01 2017.

[23] Object Management Group. Practical Guide to Cloud Service Agreements Version 3.0. February 2019.

[24] Dana Petcu. Service quality assurance in multi-clouds. In Jörn Altmann, Gheorghe Cosmin Silaghi, and Omer F. Rana, editors, *Economics of Grids, Clouds, Systems, and Services*, pages 81–97, Cham, 2016. Springer International Publishing.

[25] Gábor Sándor. Performance assessment of cloud applications. Master's thesis, Charles University, 2020.

[26] Adam Filandr. Latency aware deployment in the edge-cloud environment. Master's thesis, Charles University, 2020.

[27] AFarCloud project website. http://www.afarcloud.eu/. Accessed: 2020-12-10.

[28] Opensfm project website. https://www.opensfm.org/. Accessed: 2020-12-10.

[29] Drone mapping software — opendronemap. https://www.opendronemap.org/. Accessed: 2020-12-10.

[30] Jeffrey Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36:41 – 50, 02 2003.

[31] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *TAAS*, 4, 01 2009.

[32] Lubomír Bulej, Tomáš Bureš, Petr Hnětynka, Václav Čamra, Petr Siegl, and Michal Töpfer. IVIS: Highly customizable framework for visualization and processing of IoT data. In *Proceedings of EUROMICRO SEAA 2020, Portorož, Slovenia.* IEEE, 2020.

[33] FitOptiVis project website. http://www.fitoptivis.eu/. Accessed: 2020-12-10.

[34] gRPC — a high performance, open source universal rpc framework. https://www.grpc.io/. Accessed: 2020-12-10.

[35] Or-tools — google developers. https://developers.google.com/optimization. Accessed: 2020-12-21.

[36] Mongodb — the most popular database for modern apps. https://www.mongodb.com/. Accessed: 2020-12-21.

[37] Container v1 core — kubernetes api reference docs. https://developers.google.com/optimization. Accessed: 2020-12-23.

[38] Deploy a sharded cluster — mongodb manual. https://docs.mongodb.com/manual/tutorial/deploy-shard-cluster/. Accessed 2020-12-23.

[39] Perf Wiki website. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed: 2020-12-12.

[40] Openstack. results of measuring of api performance of kubernetes. https://docs.openstack.org/developer/performance-docs/test_results/container_cluster_systems/kubernetes/API_testing/index.html. Accessed: 2020-12-12.

[41] Amir Vahid Dastjerdi, Saurabh Kumar Garg, Omer F. Rana, and Rajkumar Buyya. Cloudpick: a framework for qos-aware and ontology-based service deployment across clouds. *Software: Practice and Experience*, 45(2):197–231, 2015.

[42] B. Hu, H. Wang, P. Zhang, B. Ding, and H. Che. Cloudroid: A cloud framework for transparent and qos-aware robotic computation outsourcing. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 114–121, 2017.

[43] Yang Hu, Junchao Wang, Huan Zhou, Paul Martin, Arie Taal, Cees de Laat, and Zhiming Zhao. Deadline-aware deployment for time critical applications in clouds. In Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro, editors, *Euro-Par 2017: Parallel Processing*, pages 345–357, Cham, 2017. Springer International Publishing.

[44] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 237–250, New York, NY, USA, 2010. Association for Computing Machinery.

[45] Ran Xu, Subrata Mitra, Jason Rahman, Peter Bai, Bowen Zhou, Greg Bronevetsky, and Saurabh Bagchi. Pythia: Improving datacenter utilization via precise contention prediction for multiple co-located workloads. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, page 146–160, New York, NY, USA, 2018. Association for Computing Machinery.

List of Figures

Listings