Tampere University

Lauri Koskela

PLATFORM INDEPENDENT JOB WORKLOAD MANAGEMENT

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
October 2020

ABSTRACT

Lauri Koskela: Platform Independent Job Workload Management
Master of Science Thesis
Tampere University
Master's Degree Programme in Information Technology
October 2020

Short-lived job workloads running on cloud platforms can utilize the scalability in cloud computing. A job only requires computing resources for the time it is executing. The platforms can dynamically and automatically provision resources for jobs when needed. Similarly, the platforms can release the resources when there are no jobs running. This brings cost benefits: cloud consumers only pay for the time the jobs are running and use the computing infrastructure.

Managing jobs and the cloud infrastructure automatically in software applications can be challenging. Job management services on cloud platforms are complex, and using them requires a considerable amount of information about the jobs and the platform. Applications that do job management must handle this information, even though it is not necessarily their concern. Additionally, this information and the job management in general are platform-specific. The job management code only works with the explicitly supported platforms. This makes testing and developing the job management code locally difficult.

This thesis studies solutions for easier programmatic job management. The main contribution of this study is a new job management system that can be used to run and manage jobs independently of any single cloud platform job service. The new system decouples the specifications of the jobs from the applications that manage them. This decoupling significantly simplifies job management code and makes it work with multiple job management services. The platform-independence makes job management code easy to develop and test, as identical code works both in production cloud environments and local development hosts. The new system is evaluated by interviewing software developers who have used it. The interviews indicate that the system makes developing job functionality easier.

Keywords: job workloads, cloud, platform independence, dynamic resource provisioning

TIIVISTELMÄ

Lauri Koskela: Alustariippumaton työohjelmien hallinta
Diplomityö
Tampereen yliopisto
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Lokakuu 2020

Pilvialustoilla ajettavat lyhytkestoiset työmuotoiset sovellukset voivat hyödyntää pilvilaskennan skaalautuvuutta. Laskentaresursseja voidaan varata töiden suoritusta varten dynaamisesti ja automaattisesti. Vastaavasti resurssit voidaan vapauttaa, kun töitä ei ole ajossa ja resursseja ei käytetä. Tämä on kustannustehokasta: pilviresurssien käyttäjiä laskutetaan vain siltä ajalta, kun resurssit ovat olleet käytössä.

Töiden ja laskentainfrastruktuurin hallinta ohjelmallisesti voi olla haastavaa. Pilvialustojen tarjoamat töiden hallintapalvelut ovat monimutkaisia, ja niiden käyttäminen vaatii huomattavan määrän tietoa sekä töiden sisäisestä toteutuksesta että pilvialustan toiminnasta. Töitä hallinnoivan sovelluksen pitää käsitellä näitä tietoja, vaikka ne eivät välttämättä kuulu sen vastuulle. Töiden hallinta ohjelmallisesti on lisäksi alustakohtaista. Ohjelma, joka hallinnoi töiden suoritusta, toimii vain niillä pilvialustoilla, joille ohjelmaan on rakennettu tuki. Tämä voi tehdä ohjelman paikallisesta kehityksestä ja testaamisesta vaikeaa.

Tässä diplomityössä tutkitaan ratkaisuja töiden ohjelmallisen hallinnan helpompaan toteutukseen. Diplomityön tuloksena toteutetaan uusi töiden hallintajärjestelmä, joka mahdollistaa töiden ajamisen ja hallinnan alustariippumattomasti. Uusi järjestelmä eriyttää työohjelmien määrittelyn ja konfiguraation töitä hallitsevista sovelluksista. Tämä eriytys yksinkertaistaa töiden hallinnan toteutusta huomattavasti. Samalla se mahdollistaa toteutuksen toiminnan useiden eri pilvialustojen kanssa. Tämä alustariippumattomuus helpottaa töiden hallintakoodin kehitystä ja testausta, sillä sama toteutus toimii sekä pilviympäristöissä että paikallisella tietokoneella kehityskäytössä. Uuden järjestelmän hyödyllisyyttä arvioidaan haastattelemalla sitä käyttäneitä kehittäjiä. Haastattelujen perusteella uusi järjestelmä tekee töiden hallintatoiminnallisuuden kehittämisestä helpompaa.

Avainsanat: työohjelmat, pilvilaskenta, alustariippumattomuus, dynaaminen resurssien varaaminen

PREFACE

This thesis was done for Bitwise Oy to solve issues with job management in the Wisehockey product. The research and implementation were done in spring 2020, and the thesis itself was written in the following summer and autumn.

I would like to thank Bitwise for the opportunity to write my thesis on this topic. The idea came from real problems that me and my colleagues have faced when developing the Wisehockey product. Developing a solution was interesting and rewarding, knowing that a successful solution would help in developing the product in new directions. Thank you to all my colleagues who gave guidance, feedback and suggestions along the way.

I would also like to thank my thesis supervisor, Professor Kari Systä, for invaluable comments and guidance along the whole process. Finally, I am grateful to my friends and family, especially Ellinoora and Unto, for supporting me during this summer and throughout my studies.

In Tampere, 24th October 2020

Lauri Koskela

CONTENTS

LIST OF FIGURES

LIST OF SOURCE CODE LISTINGS

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
AWS	Amazon Web Services
BaaS	Backend as a Service
CPU	Central Processing Unit
EC2	Amazon Elastic Compute Cloud
FaaS	Functions as a Service
gRPC	A remote procedure call framework developed by Google
GUID	Globally Unique Identifier
HPC	High-Performance Computing
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
JSON	JavaScript Object Notation
NIST	National Institute of Standards and Technology
PaaS	Platform as a Service
REST	Representational State Transfer
SaaS	Software as a Service
UML	Unified Modeling Language
YAML	YAML Ain't Markup Language, a data serialization language

1 INTRODUCTION

One of the most prominent promises of cloud computing is cost-effectiveness. Typically, cloud computing costs are directly proportional to the amount of resources used. With dynamic cloud computing resource provisioning, the resources can be reserved only when they are needed and released afterwards. Cloud computing consumers only pay for the time the resources are running. The term elastic computing, often associated with cloud computing, refers to this: resources are scaled horizontally and elastically based on the current demand.

Short-lived application workloads, *jobs*, can also take advantage of the dynamic cloud resource provisioning. Unlike a constantly running server application waiting for requests from clients, a job application does one piece of work and exits after it has finished. It only reserves resources for the time it is running. Dynamic computing resource provisioning can be used to adjust the amount of computing resources based on the number of jobs that are active. When there are no jobs running, all computing resources can be released to achieve cost savings.

The objective of this thesis is to explore solutions for easier job workload management in cloud environments. The aim is to solve previously experienced difficulties when utilizing current job management systems on public cloud platforms. The focus is on the development of applications that use these job management systems to run jobs in the cloud. Notably, ways to manage jobs in a platform-independent manner — such that the applications can work in multiple cloud environments — are studied.

This thesis was conducted for the Wisehockey project at Bitwise Oy. Wisehockey is a real-time sports analytics platform that uses tracking data gathered from sports players and equipment to automatically calculate statistics and generate visualizations. In the Wisehockey system, the need for running job applications in the cloud is tied to gathering and processing tracking data from sports events. Real-time data processing jobs are started when needed, usually some time before the scheduled start time of a sports event. Cloud computing resources should be dynamically allocated for the processing jobs only when needed.

Services and capabilities for running job workloads exist in most cloud platforms already. These job management services generally allow utilizing the dynamic provisioning capa-

bilities of the platform, such that computing resources are allocated for the jobs only when they are running.

In Wisehockey, the processing jobs are run and managed automatically by special applications. They launch jobs based on known sports event schedules. These “controller applications” use the APIs (Application Programming Interfaces) of job management services to run jobs. However, using these APIs introduces several challenges to the development, testing and maintenance of the controller applications. Two of the main issues are complexity and mixing of concerns in the job management operations, and dependence to the job management service or platform.

To launch a job, a specification of the job configuration and implementation has to be provided to the job service API. The specification may include details such as the job application version, computing resource requirements, internal configuration values for the job, and cloud platform configuration. All the details must be known inside the controller application, even though they are not necessarily in its scope of responsibility — for instance, an internal configuration value of the job is only a concern of the job itself.

In this situation, to change these job specifications, the controller application implementation or its own configuration would have to be changed. Independently developing and deploying the job application may be difficult. For example, to change to a newer version the job, the specific version number would have to be changed inside the controller application. In Wisehockey, the jobs and the controller applications are developed by different teams. Each development team should be responsible for the configuration of only their own applications. Therefore, the job specifications and the controller applications should be decoupled from each other so that they can be individually configured and developed.

The second challenge relates to the controller applications being coupled to the used cloud platform. This leads to reduced portability, as the applications can only be run on the specific platform. Development and testing are also made more difficult. The software may have to be developed, debugged and tested in a special environment, where regular tools for these tasks cannot be used. Depending on the complexity of the platform, developing the controller applications might also require specific knowledge and skills from the developers. If the applications were less coupled to the platform, no platform-specific expertise would be needed during development, and the same application could be run on many cloud platforms.

As a solution to these challenges, this thesis proposes a new job management system that uses one or more existing cloud platform job services to run jobs. An abstract job management interface is provided to client applications, and the applications are decoupled from any individual job service. The proposed system separates the configuration and definition of jobs from starting and managing them. A job is declaratively defined at deployment time, and a service that wants to run a job just refers to this definition when

launching a job.

This separation allows controller services to be completely platform-independent. When starting a job, a service does not have to specify any details about where or how the job is run. All platform-specific information is included in the job definition and hidden from the service. The same controlling code works everywhere, including production cloud environments and local development machines.

As the proposed system utilizes the existing job management solutions in cloud platforms, many aspects about job management and cloud resource provisioning do not have to be considered in it. For example, job scheduling policies and dynamic resource provisioning are already handled by the other systems. This existing functionality is utilized to achieve the benefits discussed above.

This thesis describes the background, design and implementation for the new job management system and evaluates the benefits gained from adopting it. In Chapter 2, cloud computing and its benefits and patterns are studied. Job workloads are examined in Chapter 3. In Chapter 4, the Wisehockey system, the job management challenges in it, and possible solutions to the challenges are examined. Chapter 5 considers the design and implementation of the new job management system. The results are evaluated in Chapter 6. In Chapter 7, the results are summarized and some future research and development possibilities are presented.

2 CLOUD COMPUTING

Cloud computing is a model where computing resources are provided remotely from a large pool to consumers on demand. Cloud services can be rapidly provisioned with minimal human interaction between the cloud providers and consumers. [15][41, p. 2] Cloud has an important role in many organizations — in a 2020 survey about cloud adoption, 81% of organizations reported to use cloud computing [29].

The key advantages of cloud computing adoption include reduction or elimination of upfront computing infrastructure investments, and increased scalability. Organizations can easily scale the computing infrastructure based on the load demand by provisioning more resources from the cloud platform. Instead of investing in the infrastructure in advance and preparing for the future scaling requirements, cloud resources can be provisioned just when they are needed. Similarly, resources can be released when they are not needed. [15][48, p. 253]

This chapter will introduce cloud computing as a concept and inspect some patterns and models commonly associated with it. Then, virtualization is discussed as an important enabling technology in cloud computing. Containers and the container orchestration platform Kubernetes are examined in more detail. Finally, two architectural patterns related to cloud computing, microservices and serverless computing, are briefly introduced.

2.1 Cloud Computing Definition

A widely used definition for cloud computing comes from the National Institute of Standards and Technology (NIST) in the United States. This definition lists five characteristics for cloud computing: on-demand self-service, broad network access, resource pooling, rapid elasticity and measured service. [41, p. 2]

Having cloud resources available on-demand and without required human interaction with the provider allow cloud consumers to provision cloud resources whenever they are needed. This enables easy scalability, which in turn may lead to cost-savings as discussed above. Because human interaction is not required, automating the resource scaling is possible. [15][41]

With cloud computing, the usage of the cloud services is measured, and service costs are

proportional to the usage. Instead of preparing for the computing resource usage ahead of time, a cloud consumer can provision resources on-demand and release them when they are not needed anymore. The costs would be proportional to this usage time. Usage measurements can also be used by the providers for internal service optimization and control. Reporting and monitoring capabilities can be offered for both the cloud provider and the consumers. [15][41, p. 2]

Rapid elasticity is another characteristic that is tied to cost-savings. Elasticity allows the cloud consumer to horizontally scale the cloud resources based on the current demand. Horizontal scaling refers to the possibility to increase and decrease the number of instances of the same resource. For example, elasticity allows for automatically provisioning more web servers during a traffic peak so that the increased load can be handled. [41, p. 2][51, p. 18]

Broad network access refers to the availability of cloud services over networks using standard protocols and mechanisms. This minimizes the locational constraints and reduces the need for specialized devices or software when using the services. [41, p. 2][48, p. 245]

The final characteristic of the five, resource pooling, allows the cloud service providers to efficiently manage and run the services. Typically, cloud services are multi-tenant: hardware resources are shared between multiple different customers, with the resources of each consumer being isolated from the other consumers. Pooled cloud resources are dynamically assigned to the consumers based on their demands. Frequently, pooling and multi-tenancy is achieved by the use of virtualization, which will be further discussed later in this chapter. [15][41, p. 2]

In addition to these characteristics, the NIST definition of cloud computing lists three different cloud service models and four different deployment models for cloud infrastructure. These will be examined in the following sections.

2.1.1 Cloud Service Models

A cloud service model represents an abstraction level of services a cloud provider can offer. In the cloud computing definition, NIST lists three different cloud service models: *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) and *Software as a Service* (SaaS) [41, p. 2]. The higher-level service models include the lower level models or depend on them. [51, p. 28] Multiple other service models, such as *Database as a Service* and *Storage as a Service* have been defined, but these can be considered as specializations of the first three models [15].

IaaS represents the lowest level of abstraction for cloud computing: IaaS services allow cloud consumers to provision and control hardware infrastructure such as server resources, storage, and networks. Cloud consumers can then use these resources for

their own computing needs and run their own software. IaaS provides a high level of control for the customer, as well as good portability and compatibility with existing software and other systems. However, the consumers are responsible for the infrastructure administration. [8][15][41, p. 3]

Instead of plain computing infrastructure, PaaS services provide computing platforms prebuilt by the cloud provider. These platforms do not necessarily directly expose the underlying infrastructure. Instead, the infrastructure and some software components, such as the operating system, databases, or frameworks, are provided as computing environments that are used by cloud consumers to run applications. PaaS can be considered more abstract than IaaS, as infrastructure management is abstracted away, and consumers only interact with the platform. Unlike with IaaS, consumers do not have direct control over the infrastructure, and managing it is a responsibility of the provider. The applications must be built to support the specific platform, which reduces portability and compatibility. [8][15][41, p. 2]

In the SaaS model, the cloud provider also provides the hosted application to the consumers. The software may be run on an IaaS or a PaaS service, but this is abstracted away from the customer. The consumers use the application over a network connection, and do not have to install or maintain it. [8][41, p. 2][51, p. 29]

2.1.2 Cloud Deployment Models

The cloud deployment models form the final categorization in the NIST cloud computing definition. These models describe who owns the cloud resources, and who the resources are provided to. The four listed deployment models are public, community, private, and hybrid clouds.

A public cloud is available for anyone to use, and it is provided by a third-party cloud provider organization. The users of a public cloud may be organizations or individual consumers. The services are generally widely accessible over the Internet. Typical examples of public cloud IaaS and PaaS providers include the Amazon AWS, Google Cloud Platform and Microsoft Azure. Consumer cloud services, such as storage services Dropbox and Google Drive can also be considered to be public cloud services. [51, p. 36]

A private cloud provides services for multiple consumers in a single organization. A typical example of a private cloud would be a large enterprise providing cloud services for smaller organizations and teams in the enterprise. Because a private cloud must not be accessible from everywhere and by anyone, the cloud services might be provided over an internal network rather than the Internet. [41, p. 3][51, p. 15]

The community cloud model is similar to the private cloud model, but the cloud services are provided to a community of consumers that share interests or concerns, such as

regulatory or policy requirements. The cloud resources may be owned by the community members or a third party. [41, p. 3]

These models can also be combined to form a hybrid cloud, which consists of multiple clouds of same or different types. For instance, a hybrid cloud might consist of a private cloud and a public cloud. The private cloud can be used as the primary environment, and the public cloud could be used to provision more resources during high resource demand. [8][41, p. 3][51, p. 33]

The cloud deployment models are orthogonal to the cloud service models introduced earlier [51, p. 30]. For example, a private cloud might provide IaaS services to software development teams and SaaS services to users inside the organization.

2.2 Virtualization

Virtualization is perhaps the most important technology enabling cloud computing. Conceptually, virtualization refers to decoupling the physical servers and computing resources from their users by introducing an abstraction level of virtual resources. Most often virtualization is associated with the concept of virtual machines: running one or more virtual hosts, each with their own operating system and software, on a single physical host. However, other resources such as networks and storage can also be virtualized. [15][55, p. 89]

Virtualization is used to implement the pooled resource characteristic of cloud computing efficiently. It allows sharing physical servers and other cloud resources between multiple customers. For example, multiple virtual servers, potentially from different consumers, can be run on a single physical server. This resource consolidation increases server utilization and therefore efficiency. Because virtual servers are not tied to the physical servers they are running on, virtual servers can be run in a pool of physical servers. [15][46][51, pp. 5–6]

Furthermore, virtualization allows easier management of the resources. For instance, provisioning a virtual server does not require physical access to the hardware and can be automated. Virtual resources can be replicated, and their state can be stored via snapshots. Virtual resources can also be migrated to other physical hosts or even data centers, often without even disrupting the operation of the resource. [13, p. 6][15]

2.2.1 Containers

Software containerization is a virtualization technology that has gained popularity alongside cloud computing adoption. Container technology is operating-system level virtualization: applications are run in isolated environments, containers, under the same operating

system. Unlike traditional virtual machines, which contain a complete operating system and all software running on it, containers share the operating system kernel on the host they are running on. This allows efficiently running multiple isolated applications on a single server without resource overhead from running multiple operating systems. Because an operating system does not have to be started when launching containers, they are substantially faster to start up compared to traditional virtual machines. [15][26][46]

Containers are packaged as container images, which contain applications alongside all their dependencies, such as shared libraries and frameworks. Everything needed at runtime, except the operating system kernel, is included in the container image. This makes containers highly portable: because all the required software is delivered within the application container, it can be deployed anywhere where the container platform is available. Libraries and other required software do not have to be installed in advance, and the applications are shipped with their specific library and framework versions. Multiple containers running on the same host may include and use different library versions without any conflicts with other containers. [15][26]

Docker [12] is currently the most prominent container platform [47]. It uses the container features of the Linux operating system to provide a container runtime. Additionally, it has features for managing containers, building and managing container images, and tools for using the platform. Docker also provides functionality for distributing container images: container registries are servers that store and serve container images.

Because of the encapsulated and host-independent nature of Docker containers, they have been increasing in popularity among PaaS solutions. A PaaS provider can offer a platform for running arbitrary Docker containers, and consumers can run any software in the container without having to know anything about the hosts the containers are running on. [14] This gives greater flexibility compared to more traditional platforms, where the application must be built to support the specific platform.

Running large numbers of containers on pooled servers causes significant management overhead. Container orchestration solutions have been developed to make running large container systems easier. One of these solutions is Kubernetes, which will be examined next.

2.2.2 Container Orchestration and Kubernetes

Container orchestration refers to building and managing systems consisting of many containers. The orchestration solutions automate the deployment, management and monitoring of containers, and allow defining how multiple containers communicate and cooperate. Additionally, the orchestrators provide features for container networking, scaling and fault tolerance. [33][47]

Kubernetes is a commonly used container orchestration platform [9][33]. It is designed for running and orchestrating containers in server clusters. It provides all the common orchestration listed above such as automated deployments and scaling, networking and service discovery. The popularity of Kubernetes has steadily grown over the past years [11][28][56]. Many cloud providers now have managed Kubernetes offerings that integrate with other cloud platform features and make Kubernetes adoption easier [9].

Kubernetes runs containers in a server cluster consisting of worker servers called *nodes*. Each node runs Kubernetes infrastructure daemons, a container runtime, and the containers scheduled to the particular node. When running in a cloud environment, nodes are typically virtual machines provisioned on the cloud platform. Nodes have limited CPU and memory capacities that are calculated based on the actual CPU and memory resources of the machine. [60]

A pod is the smallest schedulable unit in Kubernetes. Typically, a single instance of a single application is encapsulated as a pod. It consists of one or more containers and their configuration, and optionally storage volumes, metadata and some Kubernetes-specific configuration such as scheduling policies. The containers in a pod may specify CPU and memory requirements and limits. Kubernetes schedules a pod to run on a node based on the resource requirements and scheduling policies such that all pods are guaranteed to get the resources they request. [9][28][61]

Kubernetes is designed to be used *declaratively*. Instead of directly creating and removing pods, users declare the *desired state* of the pods running in the cluster. Special components in Kubernetes, controllers, run constantly in the background and compare the desired state to the actual state of the pods running in the cluster. When these states differ, the controllers make changes and try to make the actual state match the desired state. The desired state is declared by creating special objects via the Kubernetes API. [58] For instance, a Deployment object declares a template for a pod, and specifies the amount of these pods that should be running in the cluster. The deployment controller in Kubernetes notices when a Deployment is declared, and creates pods based on the template so that the desired state is achieved. When the actual state of the pods diverges from the desired state — for example, due to a pod dying due to an error, or changes being made in the Deployment object — the controller keeps making changes so that the desired state is reached. Practically, this means creating and deleting pods. [28]

Kubernetes supports elasticity and horizontal scaling on multiple levels. First, pods can be automatically replicated based on usage metrics. This can be used to automatically create new pods during a high demand and remove them when the demand is lower. In addition to this, Kubernetes supports automatically scaling the server cluster. When new pods are requested, but there is no space for them in the cluster, Kubernetes can automatically expand the cluster by provisioning new nodes. Combining automatically scaling

pods and the cluster results in completely automatic elasticity on both the infrastructure and software level: during high demand, Kubernetes can request more pods to handle the load, which in turn may result in new nodes automatically being provisioned to fit the new pods. When the additional resources are no longer needed, Kubernetes can remove excess pods and nodes to remove unused capacity from the cluster. [28]

Because managing systems consisting of many containers is straightforward and efficient with Kubernetes, it is regularly used to run applications composed of many small independent services. This kind of pattern is called the microservice architecture, and it is studied in the next section.

2.3 Microservices

A software system following the microservice architecture is composed of many small loosely coupled service applications. The services work together to implement the full system functionality. Compared to a monolithic software architecture, where the software is built as a single application containing all the functionality, microservices enable creating more scalable applications and can make developing and deploying them simpler. [46] Microservices are often associated with cloud computing and the term *cloud native*, due to them being suitable for fully taking advantage of the benefits of cloud computing [52].

Microservices should be highly cohesive and have a narrow scope of responsibility. They should be able to be independently developed and deployed without requiring changes to other microservices. Because microservices are loosely coupled and typically communicate with each other over the network using standard protocols, they can be implemented using different technologies and programming languages. The implementation technologies for each microservice can be chosen based on the suitability for the task and the competence of the development team. [46]

Microservices can be independently scaled both horizontally and vertically to achieve optimal resource usage. Multiple instances of a frequently used microservice can be run side-by-side, whereas a microservice with less usage might only need one or a couple of instances. This horizontal scaling can be managed automatically according to demand and service load to achieve elasticity. Independent vertical scaling allows giving more computing resources to demanding microservices even if horizontal scaling cannot be used. For instance, a microservice doing heavy data processing might require greater amounts of memory and CPU resources compared to services with lighter functionality. [46]

2.4 Serverless Platforms

The serverless paradigm has gained popularity in the past years as a computing model that falls between the PaaS and SaaS models. Serverless platforms aim to remove the need for infrastructure and server process management. PaaS services already hide the infrastructure management at some level, but serverless platforms take the idea further by allowing software deploys without knowledge of server instances or processes. Additionally, they regularly offer a pay-per-use model for computing. [16][50][54]

Currently, the concept of serverless is commonly used synonymously with *Functions as a Service* (FaaS). It refers to running small pieces of code, functions, on a serverless platform. The functions are triggered by events such as incoming web requests. The serverless platform only runs the functions when triggered, and the underlying services can be shut down during idle times. Billing is based on the executions of the functions, which results in the pay-per-use model. Unused functions do not incur costs. [16][31][54]

Serverless functions are generally stateless and short-lived. One function invocation cannot persist any data for subsequent invocations without the use of an external data storage service. A FaaS architecture may not be the best choice for a long-running and stateful computing task. [49]

Backend as a Service (BaaS) is another computing pattern that is considered to be a part of the serverless paradigm alongside FaaS. BaaS refers to full software components and services offered by third-party providers. Using BaaS services can reduce development costs, as commonly used components offered via BaaS services do not have to be implemented in applications. [50]

Serverless computing fits the cloud computing characteristics well. Measured usage can be seen in the pay-per-use billing, as explained above. Rapid elasticity is easily achieved with the small functions: as infrastructure management is hidden, resources can be freely scaled based on the number of function requests. Abstracted infrastructure additionally allows running the functions on pooled resources, as the cloud consumers do not need to manage or know anything about the infrastructure.

3 JOB WORKLOADS

Many cloud platforms focus on running *server workloads* or *service workloads*: applications that are long-running and always ready to handle incoming requests from clients [38]. Even the serverless architectures introduced at the end of the previous chapter can be considered services: even though the serverless functions may be completely shut down during idle times, clients can still request the services and the serverless platform starts the functions on demand.

The service workload model does not fit all applications. *Job workloads* are short-lived: unlike services, they are designed to execute one piece of work, and exit once they have completed. They often do not provide services for other applications to use, and rather function independently. Because job workloads only require resources for the time they are running, they can be efficiently scheduled to run on a limited pool of computing resources. In cloud environments, running short-lived job workloads brings cost benefits. The computing resources can be dynamically provisioned for running each job and released when the job finishes. The job only incurs costs for its running time.

Defining the concept of a job is challenging due to the vast differences in various systems that use the term. On an abstract level, a job refers to one piece of work to be done by a computing system [65]. Task is another term which is sometimes used interchangeably with job. However, in some systems a task represents a smaller part of a larger job [66].

For the purposes of this thesis, a job workload refers to any application or process having a finite lifetime. In short, a job will always finish at some point after it has started.

This pattern has similarities to the FaaS model: one execution of a serverless function executes one piece of work, and only requires resources for its execution time. During idle times, resources can be released. One function execution can be thought of as a job or a task, and in fact, utilizing FaaS services is a common way for implementing job functionality in many cloud platforms.

Job workloads have been traditionally used as the primary way of computing in many systems, such as mainframe computers [36] and high-performance computing (HPC) systems [64]. Jobs are still an essential part of computing, and considerable research is still conducted into job scheduling and other job characteristics [34][35][64]. Numerous job management systems and job schedulers have been created and are still in use today.

This thesis does not consider most of this research. Instead, the focus is on the general job usage patterns and in the use of existing job management systems, especially those found on public cloud platforms.

This chapter will further examine general concepts about job workloads. Various types of jobs are introduced, and patterns about job management systems are studied. Finally, a handful of job services in public cloud platforms are shortly examined as examples of existing job management systems. The use of FaaS platforms for running jobs is also examined more in depth.

3.1 Job Types

In this thesis, jobs are divided into two broad categories based on their interaction with users or other processes: interactive jobs and batch jobs. Additional types and categorizations can be defined based on other aspects, such as triggering reasons or specific use cases.

Batch jobs are possibly the most typical example of job workloads. They are jobs that run without user interaction and are designed to process a finite amount of data. [10][27][45] Batch jobs are often used in repetitive and heavy computing and data processing tasks, such as transaction processing tasks or reporting processes. They are the primary way of running computing processes in high-performance computing (HPC) and grid computing systems. [30][36]

Because batch jobs can be run without user interaction, they can be scheduled to run at some time in the future. For example, batch jobs can be specified to run only when sufficient computing resources are available for them, such as during nighttime when the resources are not actively used by other tasks. Alternatively, jobs can be scheduled to run at a specific point in time in the future or repeatedly at some intervals. These kinds of jobs are called timed jobs or *cron jobs*, named after the *cron* scheduling daemon used in UNIX systems. [27]

Interactive jobs interact with users or other processes during execution. These jobs can receive inputs and emit outputs. In HPC systems, users can run interactive applications by running interactive jobs. A time slot is given for an interactive job, and during that time a user can utilize the HPC system resources by running their applications. The interactive job exits when the time slot ends. [39]

Another example of an interactive job, more relevant for this thesis, is a real-time data processing job that receives data from an external source. The job processes the incoming data, until all the data has been handled or some other condition for stopping is reached. Because the job depends on the external data source, it cannot be run independently, so it is an interactive job instead of a batch job.

3.2 Job Management Systems

Jobs are typically run and managed using job management systems, which are also called workload management systems. They are responsible for scheduling, running, controlling and monitoring the jobs. Users can submit and control jobs using interfaces provided by the management systems. [17]

Job management systems are often discussed in the context of grid computing and high-performance computing systems. A job management system is responsible for monitoring the available resources in the computing system and for allocating resources for each job. Jobs are scheduled to run based on their resource requirements, the available resources in the computing system, and policies specific for the computing system [17][32]. However, the primary functionality of job management systems is also applicable in other kinds of computing systems, such as cloud computing or traditional servers.

The various job management systems differ in their computing platform support, support for different job types, and how the jobs can be defined and implemented, along with many other characteristics. Even though the basic functionality, managing and running jobs, is similar on an abstract level, two job management systems can be completely different due to their different use cases and platform support. Some of these characteristics are examined next.

As stated above, many job management systems have to be able to interact with the underlying computing infrastructure to analyze and control the available computing resources. Therefore, the management systems have to be integrated with the computing platforms. [7] For example, in HPC computing systems the job management systems are integrated with the computing cluster. Similarly, many public cloud providers have their own job management solutions that integrate with the cloud platform features [1][19][43]. This cloud integration allows the job management to be made elastic: the elastic infrastructure in the cloud platform can be used to run the jobs and change the amount of available resources based on the amount of submitted jobs.

In many job management systems, a job is submitted by specifying all details about it to the system. These details include the program or script to run, configuration values, input data, computing resource requirements, and when and where the job should be run. However, some job management systems allow creating predefined *job definitions*, which act as reusable templates for creating *job instances* [3][62]. The definition includes almost all details about a job, excluding any instance-specific ones such as input data. A job instance is created based on a job definition when a job is triggered. Job definitions are helpful when creating reusable jobs because multiple job instances can be created using the same job definition. For the same reason, they are used with repeatedly scheduled jobs. Job definitions are always specific to each job management system and computing

platform, because of the high variance in their features and capabilities.

3.3 Job Workloads on Cloud Platforms

Integrating job management systems with cloud platforms has clear benefits. The advantages of cloud computing, such as access to large numbers of computing resources and elasticity, can be utilized with job workloads similarly to service workloads. For example, worker servers can be automatically horizontally scaled based on the number of scheduled jobs. This is why most cloud providers have their own job management system offerings.

When using IaaS cloud services, and provisioning servers and other computing resources from the cloud, any job management system can be installed and used on the infrastructure. Compared to running the jobs on traditional server infrastructure, this solution has the benefit of easy server provisioning and large amounts of available resources. However, as the job management system is not necessarily integrated with the cloud platform, automatic scaling may not be possible.

The job management systems offered by cloud providers usually integrate with the other PaaS and IaaS services that provide computing resources. The job management systems dynamically provision resources for the jobs using the other services. Another typical solution is to use the FaaS offerings in the platform to run serverless functions as jobs.

In the following sections, the FaaS usage for jobs is first studied. Then, some job management systems in the largest public cloud platforms are further examined. The job functionality of Kubernetes is also studied as an example of a cloud provider agnostic platform.

3.3.1 FaaS and Job Workloads

Job workloads, as defined in the beginning of this chapter, have similarities to Functions as a Service. Both patterns consider running relatively short-lived processes with a finite lifetime. Both also allow for more efficient resource utilization by only requiring the resources for the duration they are running.

FaaS can be considered as a way for implementing job workloads. The function definition describes the implementation of the job. The job execution can be launched by triggering a function via an event or a web request. The same implementation can be used multiple times by just triggering the same function again, which makes this kind of system usable for repeated tasks. Using a FaaS solution may be sufficient for simple situations where short-lived jobs need to be executed. However, a FaaS platform typically does not offer all the necessary features of a job management system, such as job retries.

Some job management systems can use serverless functions when executing jobs: instead of running a program or a script they trigger an execution of a function [21]. These systems implement the job management features that are not present in the FaaS platform. Running jobs this way allows easily utilizing the elasticity and other benefits of serverless architectures.

The drawback is that the limitations of FaaS apply to the job workloads. Long-running, stateful, or interactive jobs may be difficult or impossible to implement on the FaaS platforms. Additionally, the resource and time limits in the FaaS services limit the size of the computing tasks that can be executed. For example, in Amazon's AWS cloud platform, the FaaS service AWS Lambda has a memory usage limit of 3008 megabytes and an execution time limit of 15 minutes for each function invocation [2]. Running resource-intensive and long tasks may not be possible in AWS Lambda due to these limits. For running these kinds of jobs, AWS provides the AWS Batch service, which will be examined next.

3.3.2 AWS Batch

AWS Batch [1] is a batch computing service in Amazon's AWS cloud platform. The service manages jobs and schedules them to run on EC2, the computing infrastructure platform of AWS, while automating all infrastructure management related to scaling and provisioning.

Docker containers are used as job implementations. Each job can specify requirements for CPU, memory, and other computing resources. Containers are scheduled on virtual machine instances that are automatically provisioned from EC2 based on the amount of resources required. Separate job definitions must always be created before a job instance can be launched. The definition includes details such as the container to run, the resource requirements and other job configuration parameters. Job instances are created based on job definitions. Most of the definition parameters can be overridden when submitting a job, if necessary. [3]

AWS Batch is exclusively designed for running batch jobs. Interactive jobs are not supported. When jobs are submitted, they first go into a job queue, from where they are scheduled to run. Failed jobs can be retried automatically. AWS Batch has support for a wide variety of batch computing features, including dependencies between multiple jobs, array jobs for efficient parallel tasks, and multi-node jobs for utilizing parallel computing systems.

3.3.3 Google Cloud Tasks

Google Cloud Tasks [19] is a task queuing and job management service in Google Cloud Platform. It is designed for running short background jobs with guaranteed execution and automated retries. The service runs jobs by making HTTP requests to a web service that

implements the job.

When submitting a job to a queue, the target URL and optional payload data are specified. Additionally, a scheduling time can be set to defer the execution of the task to some point of time in the future. When the job is executed, Cloud Tasks makes an HTTP POST or PUT request to the defined URL and includes the payload data in the request body. When the web service has received and processed the request, it must respond with an HTTP status code corresponding to the result. If an error status is returned, the job is retried.

Because Cloud Tasks relies on HTTP requests, there are limitations to the maximum execution times of the jobs. Generally, a job must finish in ten minutes or it is considered to have failed. This makes Cloud Tasks not suitable for heavy and long-running computing jobs.

Google Cloud Tasks can integrate with serverless offerings in Google Cloud or other cloud platforms to increase the efficiency of running jobs. Any serverless platform that supports HTTP services can be used to implement the job handlers. Additional integration with the App Engine serverless platform in Google Cloud is available. This provides easier and more efficient scaling management and longer maximum job execution times.

3.3.4 Kubernetes Jobs

Kubernetes ships with a Job controller, which can be used to run containerized job workloads in Kubernetes clusters. The Job controller supports both batch jobs and interactive jobs. [9][59]

In Kubernetes, a job is specified and launched by creating a Job object. The Job object contains a template for a pod. The Job controller creates a pod based on this template and waits it to complete. Whether the job succeeded or failed is determined by the container exit status code. The controller tries to make sure that the job finishes: if the container exits with an error status code, the execution can be automatically retried. Similarly, if the created pod fails, for example due to a server failure, the job controller automatically creates a new pod. When the job finishes successfully, no more new pods are created, and the Job object is marked finished.

A Job object in Kubernetes consists of the pod template and some job-specific configuration values. The pod specification includes the container to run, configuration values for the container, and platform configuration. The job specific configuration values include an optional limit for job retries, a deadline for the job execution, and options for running jobs consisting of multiple pods running in parallel. There is no built-in support for separate job definitions and instances in Kubernetes. All job details must be directly specified in the Job objects. [59]

Because the jobs are run as regular pods in the cluster, Kubernetes supports running all kinds of applications as jobs. The only requirement is that the containers will exit when the executions complete. This versatility allows easily running interactive jobs. In particular, the job applications can act as servers and network traffic can be forwarded to them. Connecting to job instances from a web browser or other applications is therefore possible.

Jobs can utilize the existing cluster scaling capabilities in Kubernetes. When the Job controller creates a pod, but it cannot be scheduled to run in the cluster due to insufficient computing resources, the cluster autoscaler can automatically provision a new node in the cluster. When the job is finished, and the pods created by it are removed, the cluster autoscaler can then remove excess nodes. This can be used to automate creating a dedicated node for each job in the cluster.

For running timed jobs, Kubernetes provides a CronJob controller. It utilizes the Job controller and provides a way to specify repeated jobs using the scheduling syntax of the cron scheduler. A CronJob definition contains the schedule specification and a *job template*. During the scheduled times, the CronJob controller creates a new Job objects based on the job template. Effectively, the job template acts as a job definition, and the CronJob controller schedules new job instances based on it. [62]

The Job controller in Kubernetes is quite low-level in its functionality. There is no built-in support for complex scheduling policies, job queuing, and many other commonly needed job management features. Submitting a job to run in Kubernetes is a complex operation, which requires creating a detailed Job object definition. However, the Job controller can be used by other job management systems and tools to run jobs. This is what will be presented in this thesis in Chapter 5.

4 JOB WORKLOAD MANAGEMENT IN THE WISEHOCKEY SYSTEM

This thesis is commissioned by Bitwise Oy to improve the job workload management patterns in the Wisehockey product. The Wisehockey system extensively uses cloud computing technologies and job workloads, which were examined in previous chapters.

This chapter will first briefly introduce the system and its architecture in those respects that relate to this work. Then, the job workloads in Wisehockey and requirements for them are inspected in more detail. Finally, the difficulties and problems with the current job management patterns are analyzed, and possible solutions for these problems are studied.

4.1 Overview of the Wisehockey System

Wisehockey is a real-time sports tracking and analytics system. It is used to track the positions and movement of athletes and sports equipment, and to produce statistics, analytics, and visualizations based on the tracking data. [67]

Position tracking is achieved via a Bluetooth-based tracking system installed in the sports venues. Athletes and equipment carry Bluetooth tags that communicate with locator antennas installed in the venue ceiling. Real-time positions of the tags are determined based on the data received by the locators. [67]

The collected position data is streamed over the Internet from the venues to applications running in the cloud. All data processing is done in the cloud in real time, and statistics and analytics are available to users with a latency of a few seconds.

Wisehockey is run on Google Cloud's public cloud platform, and it is built following the microservice architecture pattern. The next sections examine how Wisehockey's microservices are run in a multi-tenant environment, and the general cloud and container technologies in Wisehockey.

4.1.1 Multi-Tenant and Multi-Instance Applications

In SaaS applications, where the consumers directly use software provided by the cloud provider, there are two different approaches for providing isolation between customers. The objective of the isolation is to make sure that each customer can only access their own data and customize the application to their own needs. The two approaches are called multi-instance and multi-tenant applications [37]. The Wisehockey system contains aspects of both approaches.

As a general concept, multi-tenant software refers to same application instances being shared by multiple users, customers or clients. A tenant is typically a client or a group of them, and there might be multiple users for one tenant. For instance, a client organization using the application might be a single tenant, but the organization could have multiple separate users for the application. The tenants see only their own data and may have a customized view of the application. The application isolates the tenants internally, so that they have no access to other tenants' data. [37]

An alternative solution for providing applications for multiple tenants is running a separate application instance for each tenant. This multi-instance model can be efficiently implemented using virtualization such that the application instances are running on shared server resources [37]. Running separate application instances for each tenant must not be confused with horizontal scaling and elasticity. In a multi-instance model, each tenant instance uses separate configuration and data, and each instance can be independently replicated and scaled horizontally.

Compared to this multi-instance model, multi-tenant applications are more efficient due to the per-tenant overhead being minimized. For example, instead of running separate database instances per tenant, the data can be stored in a single database instance with per-tenant isolation being done on the data level. However, multi-tenant applications are more complex and more difficult to develop, as the application must handle the tenant isolation and per-tenant customization capabilities. [37]

The architecture of the applications in Wisehockey contains both multi-tenant and multi-instance aspects. This architecture is depicted in Figure 4.1.

Some of the microservices in Wisehockey are designed to be multi-tenant. These include microservices whose functionality requires handling data from multiple tenants, such as external APIs and internal management services. In contrast, some microservices, such as real-time data processing applications, are implemented following a multi-instance model. Dedicated instances of these are run for each tenant. For instance, dedicated data transfer services are run for each sports venue. This allows easy per-tenant configuration, and simplifies guaranteeing performance, as multiple tenants do not share the same application resources.

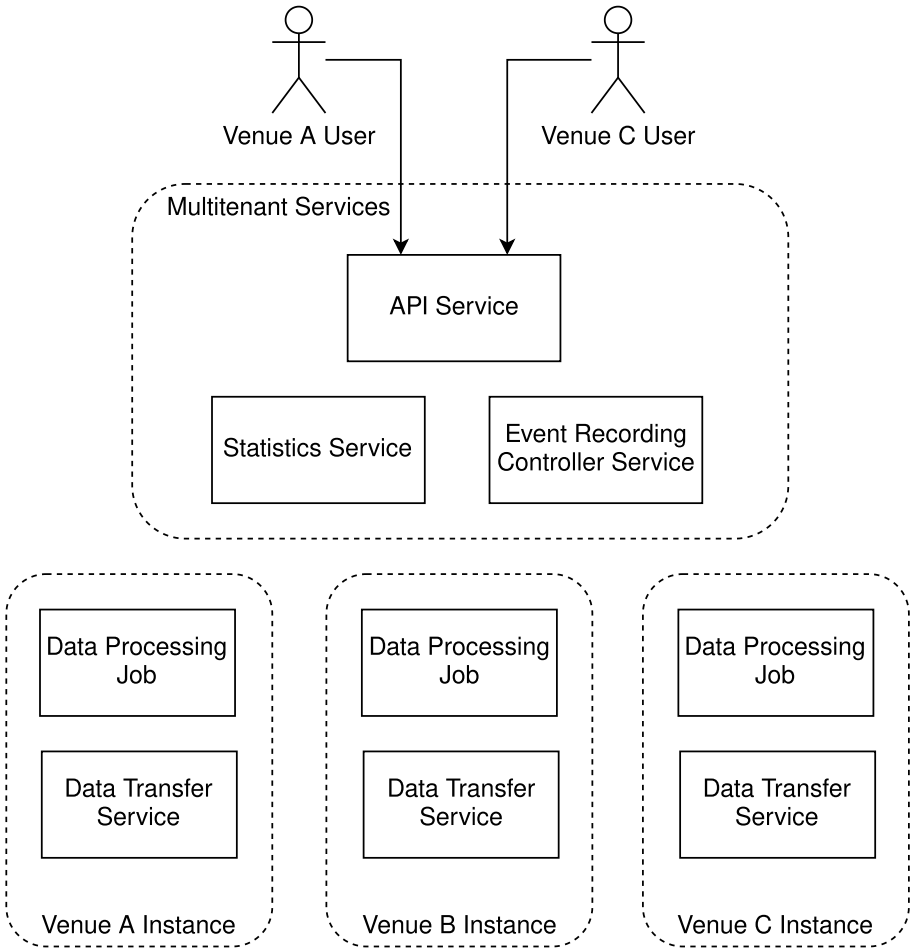

Figure 4.1. *Multi-tenant and multi-instance microservices in Wisehockey. Users for different sports venues use shared multi-tenant API services. Venue-specific instances are run for some services.*

4.1.2 Cloud Technologies

The computing infrastructure of the Wisehockey system consists of public cloud infrastructure, and servers and networks deployed to the sports venues. This thesis only considers the cloud infrastructure, as the job workloads are run in the cloud.

Currently, all cloud infrastructure in Wisehockey is provided by Google Cloud Platform. A wide range of IaaS and PaaS products are utilized to run Wisehockey's services. Perhaps the most important Google Cloud service used in Wisehockey is Google Kubernetes Engine [20], which is a managed Kubernetes offering in Google Cloud. Other Google Cloud Platform features used include data storage and database services, such as Google Cloud SQL [18], which provides a managed PostgreSQL database.

Wisehockey utilizes Kubernetes to run all its cloud applications in containers. Kubernetes allows efficiently running the microservice containers in a server cluster. It also provides tooling for easy scalability, simple independent microservice deployments and updates, and monitoring capabilities. For managing the configuration of the various Kubernetes

resources in multiple Kubernetes clusters and cloud environments, the packaging tool Helm [24] is used in Wisehockey. Helm creates reusable and configurable packages from collections of Kubernetes resources. These packages can be utilized to install Kubernetes resources with varying configuration into one or more Kubernetes clusters. In Wisehockey, all applications are packaged and installed with Helm. Multi-instance applications are deployed by creating a separate package installation for each instance.

In Google Kubernetes Engine, the cluster nodes are created as part of *node pools*. Each node pool defines a node configuration that determines what kind of virtual servers will be provisioned as nodes. The size of a node pool determines how many of these nodes with the same configuration will be created. The size can be adjusted to scale the cluster horizontally. Creating multiple different node pools allows creating different types of nodes in the same cluster and scaling the various types independently. Kubernetes Pods can be instructed to only run on specific node pools. This is used to ensure that a pod with special resource requirements is placed on a node that can provide those resources.

The cluster autoscaling functionality in Kubernetes, introduced in Section 3.3.4, works with node pools. The cluster autoscaler watches for pods and their node preferences and adjusts the node pool sizes accordingly. In Wisehockey, this is used to dynamically provision new nodes for pods that cannot fit into the other nodes in the cluster. When the pods exit and there are excess resources in the cluster, the node pool size is decreased automatically.

A large part of the cloud computing costs in Wisehockey are caused by reserved computing resources, practically Kubernetes nodes and other virtual servers. In Google Cloud, virtual machines are billed per second and the price is proportional to the number of virtual CPUs and the amount of memory in the virtual machine. To avoid excessive costs, the computing resources should be reserved only for the time they are needed. In practice, this can be achieved by utilizing the cloud elasticity and dynamic resource provisioning.

Using job workloads is a way of taking advantage of the elasticity in the cloud platform services, as discussed. In the next section, the usage of jobs in Wisehockey is examined in detail.

4.2 Job Workload Use Cases

In Wisehockey, many subsystems are inherently suitable for job-based computing because their functionality is related to timed events in the real world. For instance, the statistics processing for an ice hockey match is only run while the match is being played and data is being gathered. Hockey matches have a defined start and end, and the start and end of the data processing should align with these.

As discussed previously in Chapter 3, running processes as jobs is desired to achieve

more efficient resource usage and thus lower costs. In the Wisehockey system, two different scenarios for job workloads can be observed: jobs that are run during live sports events, and batch jobs that are run separately from any other events.

4.2.1 Live Processing

The Wisehockey system collects and processes the tracking data during the sports events with a strict latency requirement. This is done by several applications working in cooperation. Each of these applications have a narrow scope of responsibility and the design follows the microservice pattern, even though the applications are not necessarily service applications. Examples of these applications are:

- An application streaming tracking data from a sports venue to storage in the cloud.
- An application that receives tracking data from a sports venue and calculates live sports statistics from it.
- An application that receives tracking data from a sports venue and generates live visualization video from it.

The data processing in Wisehockey is mostly implemented as stateful stream processing applications [25]. The tracking data is streamed from the venues to the applications as individual data points. Each data point contains a timestamp and the position of an entity. To determine the movement of each entity, and to calculate statistics and other outputs, the applications keep internal state. The state is updated based on each incoming data point and then used to produce the statistics results. The emitted results are then stored and provided to users. This architecture is depicted in Figure 4.2 below. The produced results depend on the internal state of the application. To keep the internal state consistent and to produce correct results, the application must receive every generated data point from the data source. The processing cannot be distributed to multiple instances of the application running in parallel.

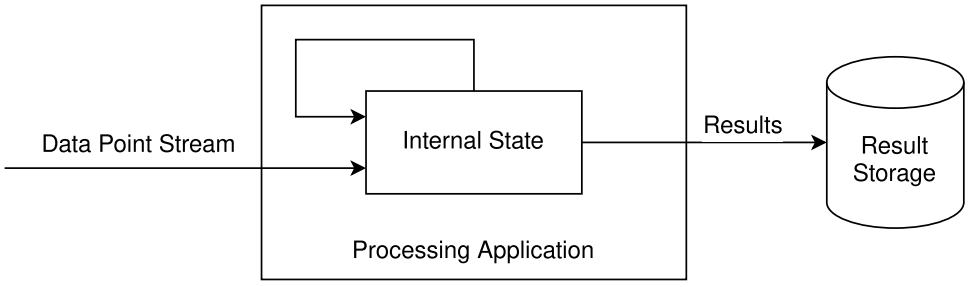

Figure 4.2. *Stateful data processing application architecture*

The data processing has a latency requirement: the statistics and other results must be available as soon as possible after the data points for the respective time period have been received. All processing applications must be guaranteed adequate computing resources

so that they are able to produce the results quickly enough in all situations. In practice, this is done by allocating each application enough CPU, memory, and other resources such as graphics accelerators and storage. The applications are replicated for each venue, as described above, and the resources must be guaranteed separately for each instance.

Currently, some of these data processing applications are run as always-running services. This is easy to implement: the service is always ready to accept tracking data and process it, and no management for starting and stopping the application is needed. However, this solution has a significant disadvantage: the service is always running and reserves computing resources, even when no data is streamed, and the application is idle. A typical ice hockey rink has two or three matches per week, each taking about three hours. A continuously running data processing application for this kind of rink is idle 95 % of the time:

$$1 - \frac{3 \cdot 3\,\mathrm{h}}{7 \cdot 24\,\mathrm{h}} \approx 95\%$$

Running the applications as jobs that are started only when they are needed, practically when a match is starting, solves this inefficiency problem. When combined with dynamic resource provisioning via automatic scaling, the idle time removal leads to cost savings, as servers can be released when they are not needed.

Some processing applications in Wisehockey are already run as jobs. When a sports event is starting, a microservice responsible for managing the processing uses the Kubernetes Job API to launch the processing applications as jobs. The applications exit when they have completed the data processing for the entire event. Kubernetes cluster autoscaling handles automatically provisioning and releasing node servers in the Kubernetes cluster.

Currently, the data processing applications expect the data to be streamed to them over a network connection. The data processing jobs must be able to act as servers listening for incoming data, and other microservices must be able to connect to them. Therefore, live processing applications are run as interactive jobs. The data processing cannot be run by itself without interaction with other services or applications. However, Wisehockey also contains some batch jobs that are run independently of any other service.

4.2.2 Batch Jobs

In addition to the live processing done during the matches, some processes such as backups, data validation and archival are run as batch jobs, independently of any match schedules. These batch jobs are triggered either by predefined schedules or dynamically by other applications. Unlike the live processing jobs, the batch jobs do not have strict latency requirements, and all input data is available when they are started. They can run as quickly as possible within the resource constraints given to them.

The batch jobs may still require relatively large amounts of resources. For example, a nightly position data validation job in Wisehockey needs substantial amounts of memory depending on the size of the input data. The same advantages of dynamic provisioning of the computing resources as described above apply here. Provisioning resources when the batch job is started and releasing them after it has finished leads to direct cost savings.

In the Wisehockey system, most of the batch jobs are implemented as separate applications that are scheduled to run with Kubernetes CronJobs. Some non-scheduled batch jobs are managed similarly to the live processing services: various microservices can use the Kubernetes API to create Job objects.

4.3 Difficulties with Current Job Management

The current model for launching and managing live processing jobs has several difficulties and issues. This is the main reason for why the previously described inefficient always-running approach for data processing is still partially in use. Job management has been so inconvenient to implement and administer that it has been more cost-effective to just run the services all the time and try to minimize the costs using other approaches.

The issues relate to the pattern where jobs are launched and controlled by software in Wisehockey. There are two notable situations where a live processing job needs to be started: when a sports event is starting, or when a user starts collecting data from a sports practice. The first situation could in theory be handled by timed jobs, as the sports event schedules are known in advance. However, in the practice situation this is not possible, as users could start practices at any time. Therefore, the jobs are launched and controlled on-demand by Wisehockey's microservices. Currently, these microservices would use the Kubernetes API to create a Kubernetes Job object, and Kubernetes would manage the actual execution of the Job.

These microservices that run and manage jobs are called *controller applications* through the end of this thesis. Two main difficulties in the development of these applications can be identified: complexity and mixing of concerns in job management, and platform-dependence. First, the actual operation for creating and running a Kubernetes Job is relatively complex and difficult to implement. The controller application must know a considerable amount of detail about the job application and its configuration just to start an instance of the job. These details include the container image of the application, any configuration for the application itself, Kubernetes-specific definitions and platform-specific resource requirements. Moreover, because the job applications are tenant-specific, the configuration values have a lot of variance between individual sports events and venues. A simplified example of launching and managing a Kubernetes job from C# code is presented in Appendix A. Even though the example code sets a minimal amount of properties for the job, it requires dozens of lines of code.

All these details about a job fall outside the scope of the controller application. If the main responsibility of the service is to start data processing during a sports event, it should not need to know which job version should be run, or how much CPU or memory it requires. The controller application should only be responsible for domain-specific configuration values that only it can provide. For instance, the controller could specify the starting timestamp and other details about the specific sports event that should be recorded, and all other details for the job should be specified somewhere else.

The second main issue is platform-dependence. An application that uses the Kubernetes API to start jobs is dependent on the Kubernetes platform. Running the job management functionality in the application requires Kubernetes, which can make local testing difficult. Special development environments have to be created so that the service can be run during development. Furthermore, depending on the implementation technologies, development activities such as debugging and profiling are more difficult or even impossible when running in Kubernetes. The development and debugging tools might not work with the applications running inside a specialized test environment.

In addition to these technical difficulties, implementing this kind of job management with Kubernetes requires advanced knowledge about the platform from the developer. The developer must know cloud-specific options that must be configurable to successfully run jobs in the cloud environment. This ties to the issue with scopes of responsibility discussed above. Application development and cloud operations teams might be completely separate in an organization, and an individual developer might have little to no expertise regarding the cloud environment their own applications are running in. Therefore, building detailed support for a cloud platform can be a challenging task.

Platform-dependence additionally reduces software portability. When the application is required to run on a new platform, explicit support for the new platform must be added to the application. This can be a considerable effort if the old and the new platform differ drastically. In a microservice architecture, where multiple microservices might have the same platform-dependence, the porting must be done separately for each microservice, which increases the amount of work even further.

The difficulties are not special to the Kubernetes platform. Starting and managing jobs can be complex regardless of the job management system used. Of course, the complexity varies due to job management systems providing different APIs and requiring different kinds and amounts of information to start a job. Similarly, the platform-dependence is an issue regardless of the actual platforms or job management systems used.

In Wisehockey, there are multiple controller applications that need to launch and manage jobs. This pattern repeats in many places in the system because functionality is tied to timed sports events, as described above. The controllers are developed by multiple different teams and developers. A solution for the difficulties discussed here would make

it easier for an individual developer to create new job workloads and develop existing ones in the Wisehockey system.

4.4 Evaluated Solutions

Various solutions for the difficulties listed above were briefly evaluated. The possibilities of adopting other systems for managing the job workloads, or even replacing jobs with other computing paradigms were studied, before landing on the solution implemented in this thesis.

When evaluating the solutions, a few key constraints were considered. In addition to the issues identified in job management, the underlying objective of lowering the cloud infrastructure costs was kept as a guideline. Only solutions that would result in lowering the costs would be acceptable. Additionally, the existing interactive job applications for stream processing were considered to be relatively immutable. A considerable amount of research and development has been done on the applications, and substantial re-architecting was not acceptable at this time.

One solution for making job management easier is to not do it at all, and to use some other computing paradigm instead. Multiple different alternatives were considered here, but none were selected due to the constraints identified above. The simplest option would be to just run the data processing applications constantly, like explained above in Section 4.2.1. This solution has been already explored previously and has the issue of large infrastructure costs. The applications would reserve resources and incur costs constantly. As the main objective is to lower the costs, this solution is not acceptable.

Another possible solution for avoiding job management would be to re-architect the applications to use a more efficient computing paradigm. Building the application as serverless functions (FaaS) would make scaling the infrastructure very easy and cost-effective. However, this could be difficult or even impossible with the inherent statefulness in the application logic. The applications would have to be re-architected to remove the statefulness from the application, and store the processing state in some external storage, so that the individual function executions could remain stateless.

Using a more scalable stateful stream processing platform, such as Apache Flink [25] or Amazon Kinesis Data Analytics [4] could be a better alternative solution. These platforms allow creating stateful stream processing applications that can be dynamically scaled based on the amount of incoming data. For instance, in Apache Flink, the applications can be implemented as serverless functions that get the current calculation state and the stream input data as their parameters [6]. The platform manages the state storage and resource scaling automatically. When no data is being streamed, the platform can completely shut down the functions and release the computing resources. This closely resem-

bles the functionality of traditional FaaS platforms. Neither FaaS nor another streaming platform was not chosen as a solution here, because re-architecting the existing applications is not an option.

Job management systems alternative to Kubernetes Jobs were also studied. These included the ones introduced earlier in Chapter 2, AWS Batch and Google Cloud Tasks. Because substantial changes to the job applications were not possible, some additional requirements for the job management systems were identified. A suitable system must be able to run arbitrary programs as the jobs, as rewriting the application to run with a specific job management system is not acceptable. The system must also be able to run interactive jobs and provide full networking capabilities to the applications, which currently act as servers while they are being run.

Neither of AWS Batch nor Google Cloud Tasks fulfil the requirements. Google Cloud Tasks is completely unsuitable for this situation, as its model for making web requests as tasks does not support stateful stream processing at all. AWS Batch could run the existing processing applications in Docker containers, but it does not support interactive jobs. Additionally, AWS is not currently used in Wisehockey, and switching to another public cloud provider was considered unfeasible. Google Cloud Platform currently does not provide a similar solution to AWS Batch.

Submitting a job in both of these systems is simpler than a similar operation in Kubernetes. AWS Batch supports job definitions, and only a reference to a definition and some scheduling parameters are required when submitting a job. In Google Cloud Tasks, only a target URL and scheduling parameters are required.

Even though other job management systems would provide simpler operations, switching to another system would not resolve the issue of platform-dependence. Staying with running the jobs on Kubernetes was thus considered as the best option. Kubernetes has good support on Google Cloud, and jobs on Kubernetes are well suited to the required use cases. Using the automatic cluster scaling with Job objects is already in use and it works well.

To make job management with Kubernetes and other platforms easier, and to remove the dependency to any specific platform, a new solution for job management was designed as the main result of this thesis. It was influenced by the design of other job management systems. For instance, it builds on a concept of job definitions similar to those used in AWS Batch. The proposed solution is examined in the next chapter.

5 DESIGN AND IMPLEMENTATION

The difficulties with job management identified in the previous chapter were used to design a new job management system. The objective of the new system is to enable managing job workloads in a platform-independent manner, while requiring minimal information about the jobs when managing them. Achieving these objectives results in easier and quicker development and testing of applications that run and control jobs.

The new system is named *Mission*. This name is used throughout the end of this thesis when referring to the system.

Mission is designed to contain no actual job workload scheduling or management logic in itself. Instead, it uses existing job management systems, like Kubernetes or cloud platform job services, to run jobs. Mission acts as an abstraction layer hiding the use of one or more actual job management systems and allows clients to manage jobs platform-independently. Jobs are defined as platform-specific in Mission, but this specificity is hidden from the users. In this chapter, the job management systems and cloud platforms used by Mission are collectively referred to as just “platforms”.

The focus in Mission is on job workloads that are run repeatedly, triggered by client applications, time-based scheduling, or some other reasons. The objectives of the system are not applicable to one-off jobs: if a job submitted by a user is to be run only once, platform independence or the complexity of the submission does not matter. The job and all the required details about it would have to be defined as platform-specific in Mission in any case. In this situation, the user could just submit the job directly to the platform used. Therefore, these kinds of jobs are ignored in the design and implementation of Mission.

This chapter first studies the requirements and the design of the new system. Then, the designed system architecture is presented. Finally, the system implementation is examined in depth.

5.1 Design

The issues and difficulties with the current job management in Wisehockey, listed in the previous chapter, were used to identify main requirements for the new system. These requirements form the basis for the high-level design of Mission. The extensive use of

separate job definitions and job instances was identified as a possible solution to many of these issues early on, and it also had a major impact on the overall design of the system.

Mission was designed and implemented as a completely independent system, even though it is currently only used in Wisehockey. The design was driven by the job workload usage patterns in Wisehockey, but the implementation contains no Wisehockey-specific features or components. Mission could be used in any system that contains similar usage of short-lived jobs.

In the following sections, the main requirements for the system are presented, and the overall design and architecture of the system are examined. Additionally, the chosen implementation technologies and reasons for choosing them are shortly studied.

5.1.1 Requirements

Requirements were identified for the system by analyzing the difficulties and issues with the current job management solutions. The four main requirements identified are:

1. Users of the system should be able to be submit jobs with minimal information about them.
2. Multiple job management systems should be supported and usable with a unified API.
3. The new system should be cross-platform.
4. Jobs should be easy to launch from any program implemented with any programming language.

The first requirement results from the issue of complex job submissions. The new system should allow submitting jobs with minimal information, such as a reference to a job definition and optional input data. Any internal configuration for the job, such as computing resource constraints, or platform-specific details such as scheduling policies, should not be needed when submitting a job. It should be as simple as possible for an application or a user to make job submission requests.

The second requirement is derived from the issue of platform dependence. The new system should support running job workloads using any number of existing job management systems. The details about which job management system or cloud platform is used should be hidden from the user of the new system. An abstract and unified API should provide a way of managing jobs regardless of the actual platforms used. Additionally, the system should be easily extensible so that support for new platforms and job management systems can be added in the future.

A special case of the previous requirement is the ability to run jobs on local machines. Operating systems on the development hosts should be supported as platforms. This en-

ables local running, testing, and debugging both the job applications and applications that manage jobs without special development environments. In this situation, any local development tools can be leveraged. This might not be possible if an external job management system was needed during development.

The new system should itself be able to run on as many platforms and operating systems as possible. This ties to the previous requirement: platform support must be extensive so that the wide support for job management systems can be utilized. Most common operating systems must be supported so that the new system can be run in all necessary cloud platforms and local development machines. Running the new system in containers should be possible so that it can be easily run on many PaaS and container platforms.

The final requirement considers the usability of the new system in various applications. The new system should be designed so that it can be used in a heterogeneous microservice environment where each service and job may be implemented using different technologies and programming languages. Interfaces should use common and standardized protocols instead of any language specific ones.

5.1.2 Jobs and Job Definitions

To minimize the amount of information needed to submit a job, the concept of job definitions (introduced previously in Section 3.2) is extensively used in Mission. To run any job, a separate definition must be created in advance. If the actual platform used to run the job does not support separate job definitions, Mission provides this functionality. The use of job definitions is depicted in Figure 5.1.

The job definitions are platform-dependent, as explained in Section 3.2. In Mission, each job definition has two properties, `type` and `name`, in addition to the platform specific properties that actually describe the job. The type property identifies the platform which the job is defined for. The name property is a unique name, which is used to refer to the job definition when submitting a job based on it. The platform specific properties contain all information required by the platform so that a job can be submitted, and other configuration values specific to the job or the platform. For example, these properties could include a reference to the application or script used as the job implementation, configuration such as credentials or database connection details, and platform configuration such as computing resource requirements.

Job definitions allow decoupling job management from the job implementations. Jobs are defined separately from the client applications that submit the jobs (1). The definitions can also be modified without changes in the client applications. For example, a job implementation version can be updated by modifying the job definition, without making any changes to the job management code in client applications.

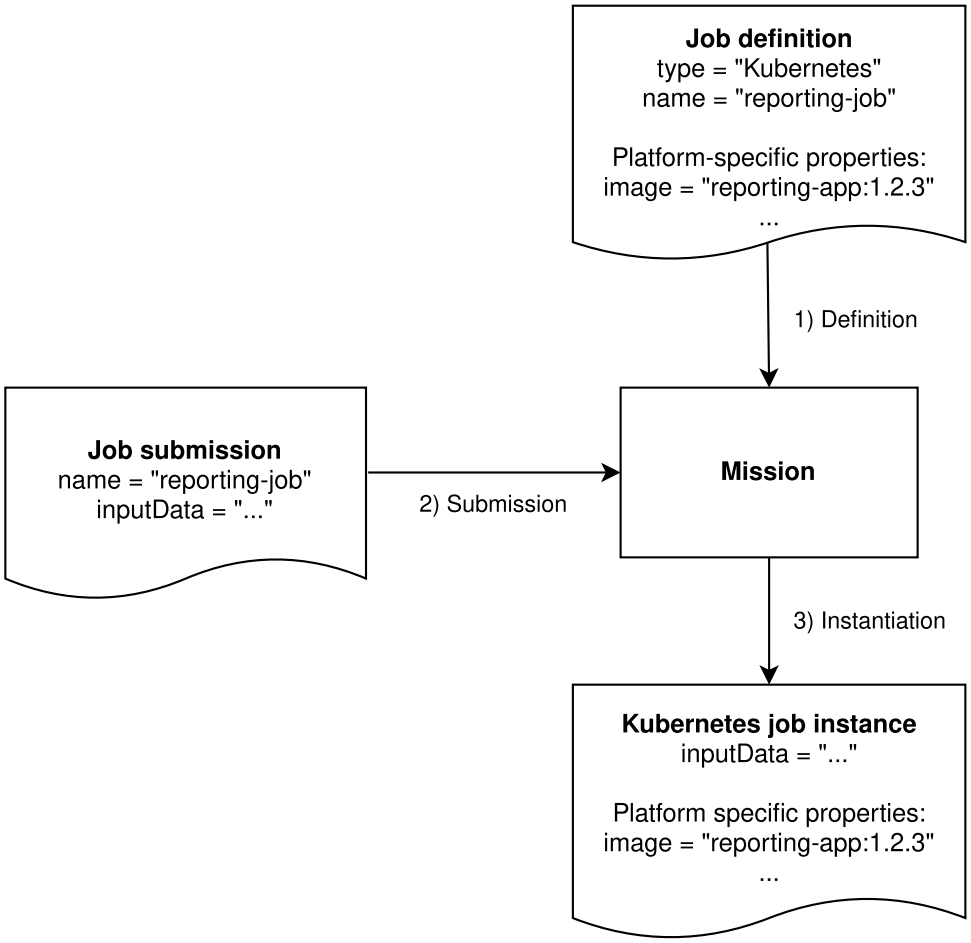

Figure 5.1. *The use of job definitions in Mission. A platform specific definition is first created (1). The definition is referenced when submitting a job (2). Mission instantiates jobs on the target platforms based on the definition and submission (3).*

When a job is submitted (2), the client only provides a name of a previously created job definition, and optionally input data for the job. No other information is required. Multiple jobs can be created using the same job definition. Submitting a job is decoupled from the actual job management system. The client does not know how the job is defined or implemented when submitting it.

Mission creates a job instance based on the received submission (3). It finds the definition the submission refers to and determines what platform should be used to run the job based on the type of the job definition. The implementation details of the job are read from the definition, and input data for the job are read from the submission. Finally, Mission submits the job to the actual job management platform.

This extensive use of job definitions directly satisfies some of the requirements identified in the previous section. First, the information required when submitting a job is minimal, as explained above. Second, because only the name of the job definition is required, any number of job management systems can be used with a single management API. Creating the job definitions is still platform-dependent, but that is completely decoupled

from any client application or user that launches jobs.

Mission does not have any requirements for the job application implementations. The jobs simply must be able to run using a job management system that Mission supports. However, when the jobs themselves are cross-platform and not tied to any specific platform or job management system, the platform-independent nature of Mission can be fully taken advantage of. The same job can be run on multiple platforms without any changes to the job itself or any application that manages the job executions.

5.2 Software Architecture

Mission is designed to run as an independent application that provides job management services to other applications. It leverages existing platforms and job management systems to run jobs and contains no job scheduling capabilities in itself. Submitted jobs and their status information are stored in an external database. The system architecture and architecture design decisions are examined in this section.

The system architecture is described here using the C4 architecture description model. The model itself is first briefly introduced. Then, the high-level architecture of the system and the design and structure of the application are discussed and visualized using C4 diagrams. Finally, some implementation technology choices are explained before moving on to describing the system implementation.

5.2.1 C4 Architecture Model

The C4 model is used to create hierarchical architecture diagrams. In the diagrams, five main element types are used. From the most abstract to most specific they are: *software systems*, which are independent systems that may interact with each other, *people*, who use the software systems, *containers*, which are independently deployable parts of a software system, *components*, which form containers, and *code elements*, which are used to implement components. The containers in the C4 model do not necessarily represent software containers, such as Docker containers. A container in C4 can be any individually runnable application or service. [53]

A typical C4 architecture description consists of four types of hierarchical diagrams that use the elements listed above. The most high-level diagram is the system context diagram (Figure 5.2), which depicts a software system in relation to people and other software systems. It is used to visualize how and by whom the system is used, and how it may use other external software systems. The next diagram type, the container diagram (Figure 5.3), shows the structure of the software system in terms of containers. All individual applications, services and data stores are shown in this diagram in relation to each other. The people and external software systems may still be included in this diagram. [53]

Component diagrams (Figure 5.4) visualize the component-level structures of containers. People, other containers, and external software systems can be included in the diagram to show how they relate to the components. If the system contains multiple containers, a component diagram can be created for each one of them. The final and most specific diagram type is the code diagram (Figure 5.5), which is used to describe the implementation of a component. These diagrams are made using UML class diagrams or other models that can describe the internal structure of a software. Typically, code diagrams are only made for the components that are the most important for understanding the system architecture and design. [53]

The first three diagram types are used in the following section to describe the system architecture of Mission. A code diagram is presented later when describing the internal implementation of the plug-in interface.

5.2.2 System Architecture

The context diagram for Mission is shown in Figure 5.2. It reflects the general design guidelines examined above: people or other systems use Mission to manage jobs, and it uses external job management systems to actually run the jobs. The figure also shows the decoupling of job definitions and job instances: administrators define the jobs separately from any system users.

The system was designed as an independent application. This model was chosen to achieve maximum decoupling of job management from any other applications. The use of the various job management systems and the job definition handling is provided in a separate application that provides services usable with common network protocols. This makes the system usable from almost any other application or service, regardless of their implementation technologies.

An alternative design could have been to implement the system as a software library. The library could encapsulate the functionality of the system and provide an abstraction layer for using the actual job management systems. The library would be included in the applications that control jobs. This solution would not be as portable as a separate application: libraries only work with the one programming language or platform they are built for. In addition, the library solution would move the responsibility of storing job definitions, configuration, and job state storage to the applications including the library. This would result in additional complexity and more difficult adoption.

The C4 container diagram is depicted in Figure 5.3. It shows that the system is simple and consists of just two C4 containers: the Mission service application and a database. The container diagram also shows the implementation technologies, which will be further studied in the next section.

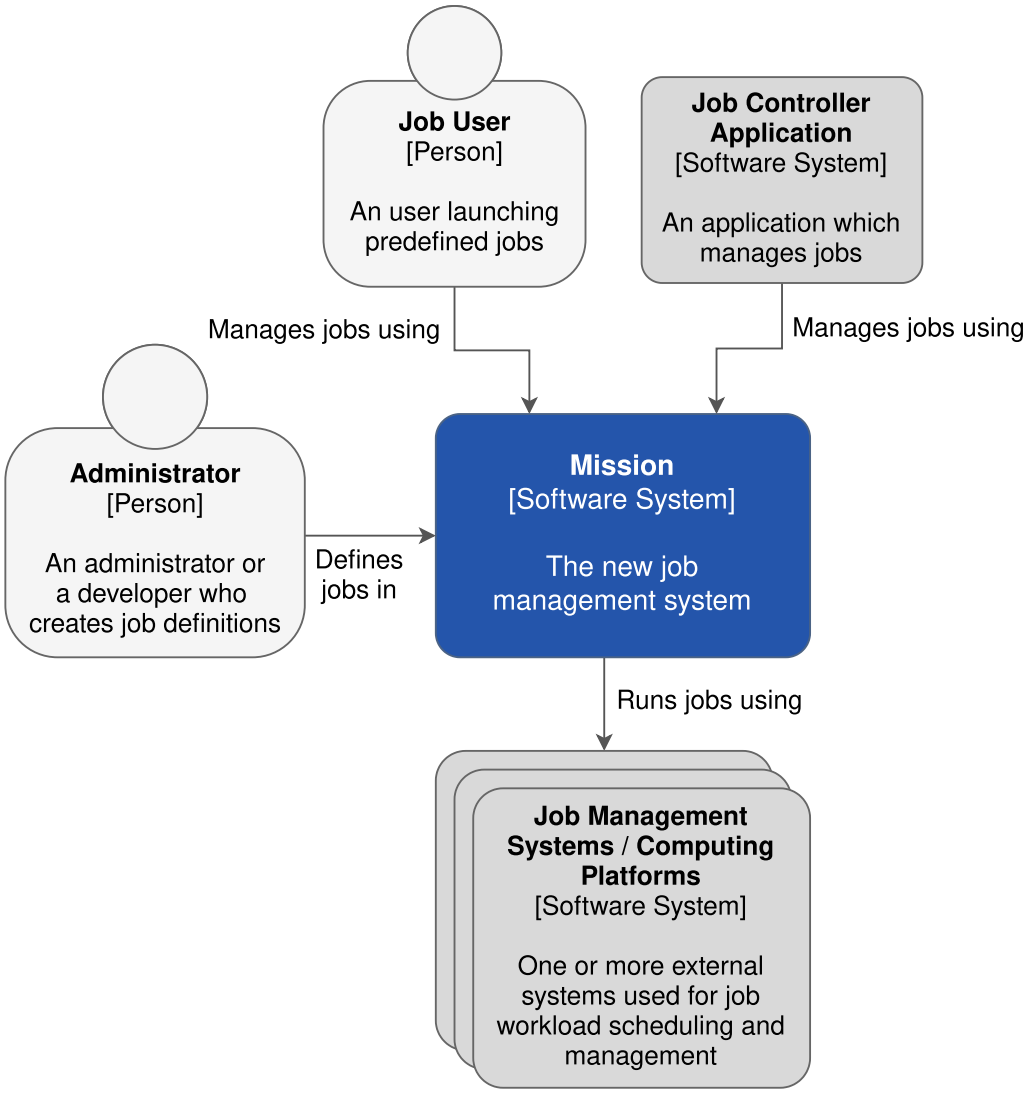

Figure 5.2. *C4 system context diagram for Mission*

As can be observed from the container diagram, users, other systems using Mission, and administrators interact only with the service. The service itself is *stateless* and persists no information. It uses a database internally for storing all state information about the jobs. Multiple instances of the service can be run in parallel, all connecting to the same database. As no information about any previous requests is stored in any specific instance, a client can freely use any instance when making requests. This makes the service horizontally scalable. Instances can be added or removed on demand.

The Mission service was designed to follow a plug-in architecture. Plug-ins are used both to implement support for job management systems and platforms, and to implement ways for providing job definitions to Mission. In addition to the plug-ins, Mission contains shared “core” components, such as the job management API and a database access component. These are collectively referred to as the core service. All platform-specific code is provided in the plug-ins. Adding support for new platforms by implementing new plug-ins is straightforward and does not require changes to the core service.

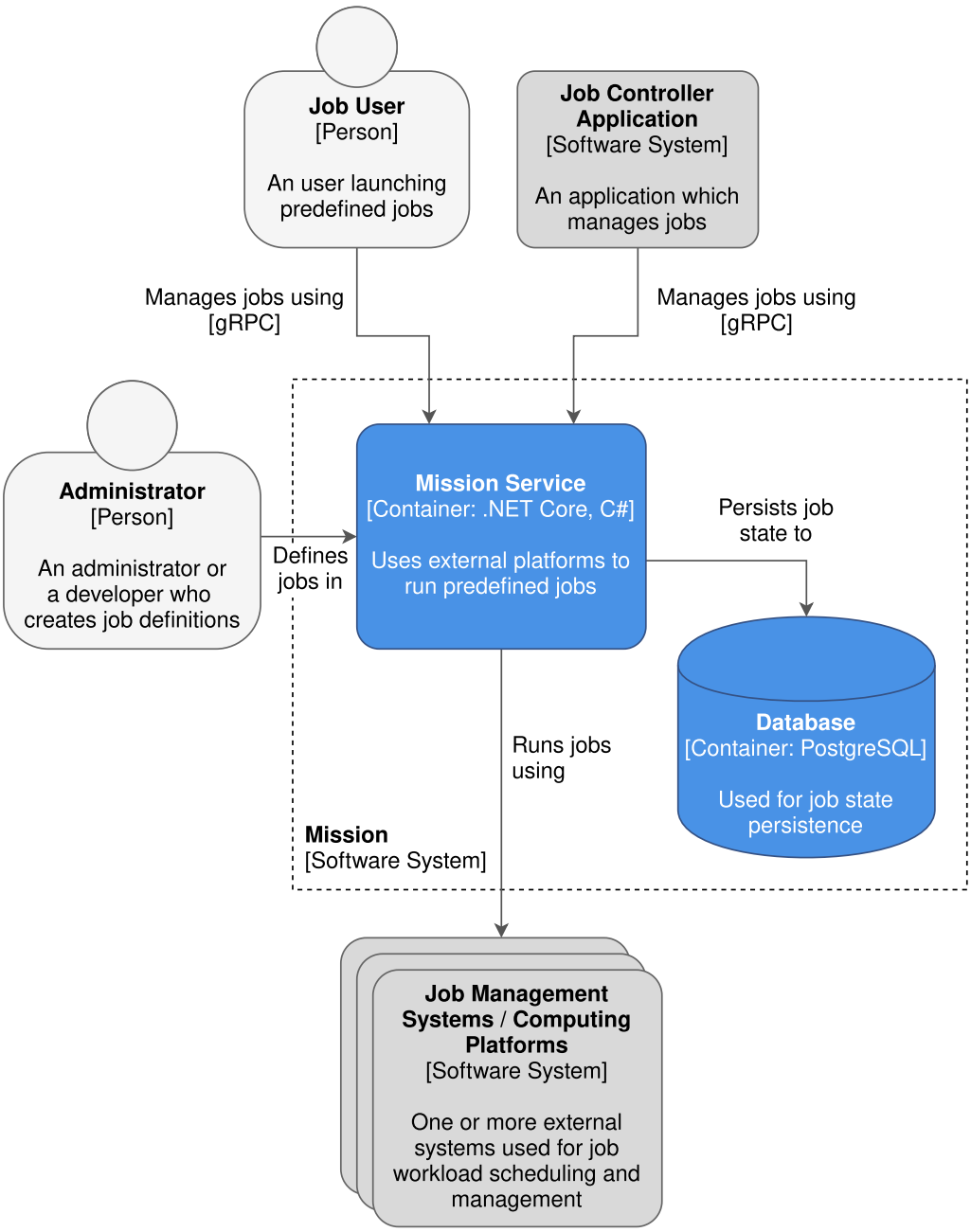

Figure 5.3. *C4 container diagram for Mission*

The component-level architecture is shown in the component diagram in Figure 5.4. In addition to the plug-ins, there are three core components: the API, a runner service and a data access layer. The API component implements the job management API used by users and external systems. The runner service uses the plug-ins to run jobs requested by users. Finally, the data access layer component provides internal functionality for storing and reading job state information from the database.

Platform plug-ins add support for job management systems or other platforms that are capable of running jobs. Each platform plug-in introduces its own type of job definition tailored for the specific job management system the plug-in uses. These job definition

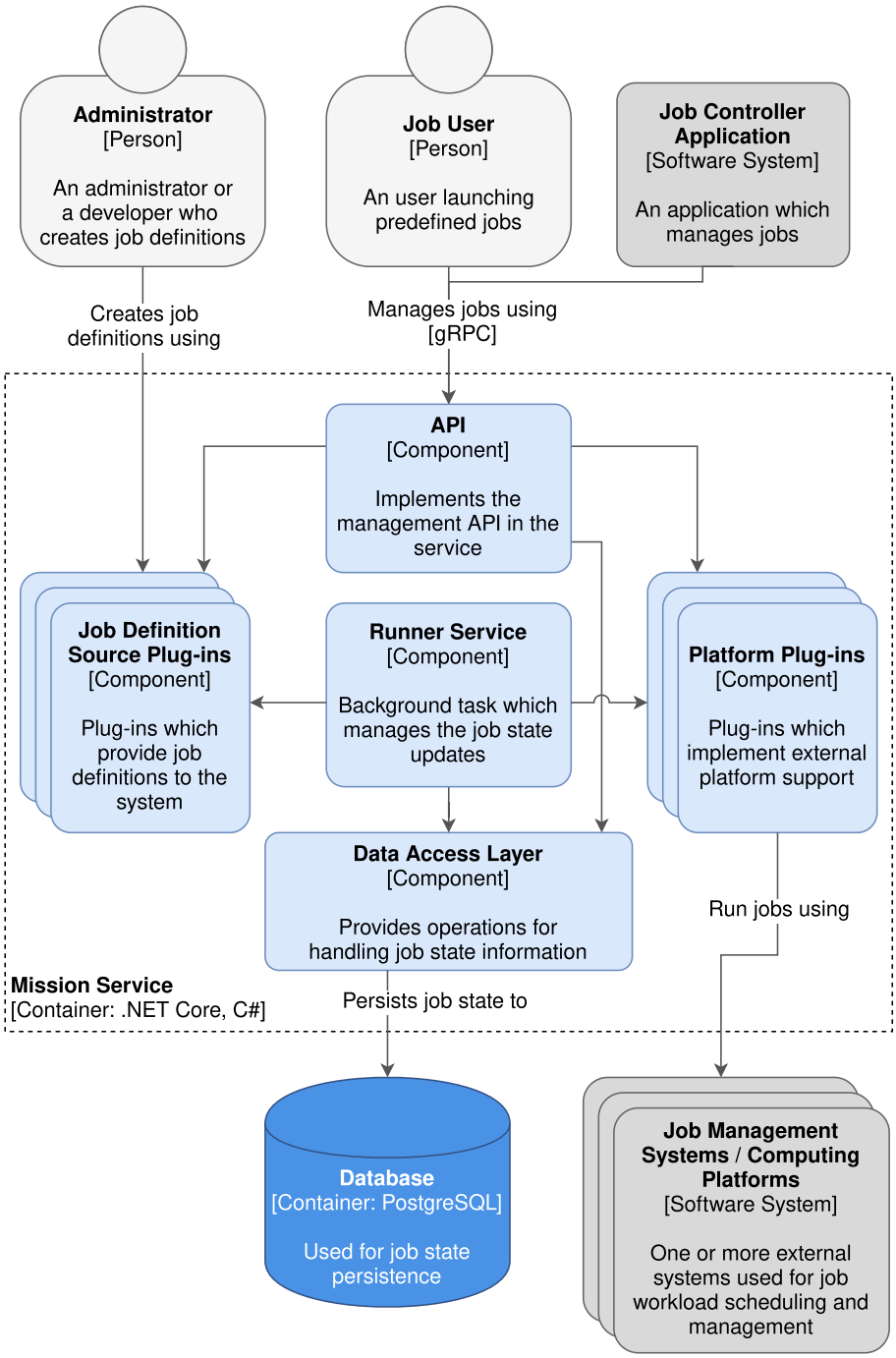

Figure 5.4. *C4 component diagram for the Mission service*

types vary based on the platform capabilities and configuration options. The core service uses the type of a definition to determine which plug-in should handle the jobs created based on it.

Job definition source plug-ins implement ways to provide job definitions to Mission. The plug-in architecture helps in supporting various storage solutions and deployment patterns for the job definitions. For example, in a distributed installation the job definitions must be available for multiple servers, whereas in a local development installation the file

system is sufficient for job definition storage. A job definition source may support multiple different job definition types or be specific to a platform plug-in. Platform-specific job definition sources are useful with platforms that already provide some way to manage job definitions or have existing tooling for managing declarative resources. These plug-ins are examined in depth later in this chapter.

The API component does not directly launch jobs. Instead, when an API call for launching a job is made, the API only persists the job into the database. The runner background service notices the new job, and runs it using a platform plug-in. This decouples the potentially slow process of starting a job from the API call: the API client gets a response as soon as the new job has been stored.

Both types of plug-ins implement generic contracts specified by the core service. Using these contracts, the core service can utilize the plug-ins without special handling for any of them. The contracts will be further examined later when the service implementation is studied in depth.

5.2.3 Technology Choices

The actual implementation technologies were chosen based on the system design requirements, previous experience, and the expertise of the developers in the Wisehockey team. Even though the system was completely implemented by the author, multiple people will likely maintain and further develop it in the future. Therefore, technologies that are already used in the Wisehockey project were preferred. Some of the chosen technologies can be seen in the C4 diagrams in the previous section.

At least two of the main requirements for the system listed previously directly affect the possible implementation language choices. First, the system must be cross-platform, and consequently the implementation language must support all the most common platforms. Second, the system must provide interfaces over common network protocols. The implementation technology must support the chosen communication protocols.

The gRPC remote procedure call framework [23] is used as the main implementation technology for the external API of the service, due to the support for strongly-typed API contracts in the protocol. GRPC uses the Protocol Buffers format to serialize data and to define the remote call interfaces. The use of Protocol Buffers enables the use of strictly typed interfaces, while still having wide platform and programming language support. Code for API clients can be generated from the API definition for many programming languages, which makes using the API straightforward. GRPC is also used in Wisehockey for communication between microservices.

The plug-in architecture introduces constraints in the possible choices of programming language and implementation platform. Implementing new plug-ins and adding support

for new job management systems should be easy. Preferably, plug-in implementation should be possible by third parties and with no changes to the core service. Languages with a rich library ecosystem would be preferable to guarantee that the language has adequate support for all kinds of different job management systems. For instance, a client library that encapsulates the REST API of Kubernetes would be helpful when implementing a Kubernetes plug-in. Kubernetes has official library support for six different languages: Go, Python, Java, C#, JavaScript and Haskell [57]. Other languages may have community-maintained libraries, but those are not always as up to date as the official ones.

Most microservices in the Wisehockey system are implemented with the C# programming language on the .NET Core platform [42]. In addition, the ASP.NET Core web framework [5] is used. This is a cross-platform solution provided by Microsoft for building applications and web services. .NET Core has a rich library ecosystem and wide support for various communication protocols, due to the .NET platform having a long history as the main development platform for the Windows operating system.

C# and .NET Core are used as the main implementation technologies for Mission, due to the previous experience and existing expertise regarding them. They fulfill all the technology requirements listed above. Additionally, dynamically loaded plug-ins are relatively straightforward to implement in .NET Core [44], which is beneficial for the planned plug-in architecture in Mission. GRPC is also natively supported in the ASP.NET Core framework [40].

PostgreSQL [63] is used as the external database for Mission, mainly for the same reasons as the other technologies. It is widely used in the microservices of Wisehockey, and there is lots of experience with it in the development team. PostgreSQL is straightforward to run locally, which should make running Mission easy during local development. Google Cloud also has a managed PostgreSQL service that can be used in the cloud environments. The simple data model in Mission is suitable for a relational database.

5.3 Implemented System

The new system was implemented following the design described in the previous sections. Thus far, only minimal functionality has been implemented. For instance, the platform support is still quite narrow. However, the system is already in production use and it is proven to work.

In the next sections, the various subsystems and components of the implemented service are examined in detail.

5.3.1 Job Management API

Mission is used by client applications via its job management API. The API is defined and implemented using gRPC and Protocol Buffers. It consists of five simple operations: list all job definitions, list job instances, create a new job instance, get a job instance with its ID, and abort a job instance. Client applications can use this API to launch jobs using predefined job definitions, check the state of previously started jobs and abort running jobs if necessary.

The operation for creating a job instance requires a job definition name as a parameter. The name must correspond to a previously defined job definition. Additionally, the client may specify the job input values as key-value pairs of strings.

Job instances returned by the API include the status of the job, which may be one of five values: scheduled, running, succeeded, failed, or aborted. When a job is created, it is first in the scheduled state. Mission then submits the job to the actual job management system, and once the job is started there, the job is moved to the running state. When the job finishes, it is either moved to the succeeded or failed state depending on the reason of the exit. Jobs are marked as aborted when they are aborted either by the user or the job management system, or if some other condition occurs such that the exit reason for the job cannot be determined.

The API does not include operations for creating and managing job definitions. Using job definition sources is currently the only way to manage job definitions. This is a deliberate design choice made due to security considerations. As job definition management is not exposed in the API, a user who can access it cannot directly create arbitrary and potentially malicious job definitions. Instead, job definition sources must be used, and using those should require a higher level of access.

5.3.2 Dynamically Loaded Plug-ins

Mission is extensible via dynamically loaded plug-ins. Built-in functionality in .NET Core for dynamically loading libraries is used to implement the plug-in functionality. Third-party users and developers can extend Mission without making changes to the core service by creating new plug-ins and loading them to their own Mission instances.

The plug-ins are provided in Mission as .NET Core library files. The paths to the libraries are listed in the service configuration. A plug-in may itself load further libraries if necessary. For example, using client libraries for various job management systems in the plug-ins is possible, even if the core Mission service does not already include them.

When the Mission service starts up, it loads all configured plug-ins. Initialization routines in each plug-in are called to allow the plug-ins to register their own dependencies and

to initialize themselves. The plug-ins also register their implementations for plug-in contracts into the core service. The core service only communicates with the plug-ins via the contract interfaces and does not contain any plug-in specific code. The plug-in contracts are examined next.

5.3.3 Plug-in Contracts

The plug-in interface contracts are defined as C# interfaces in a separate library. This library is referenced by both the core service and each plug-in. Interfaces for both platform and job definition source plug-ins are defined in the same library, but a plug-in may choose to implement only a subset of them. The interfaces, along with samples of how plug-ins can implement them, are depicted in Figure 5.5 as a UML class diagram. The implementations are examined in detail in later sections.

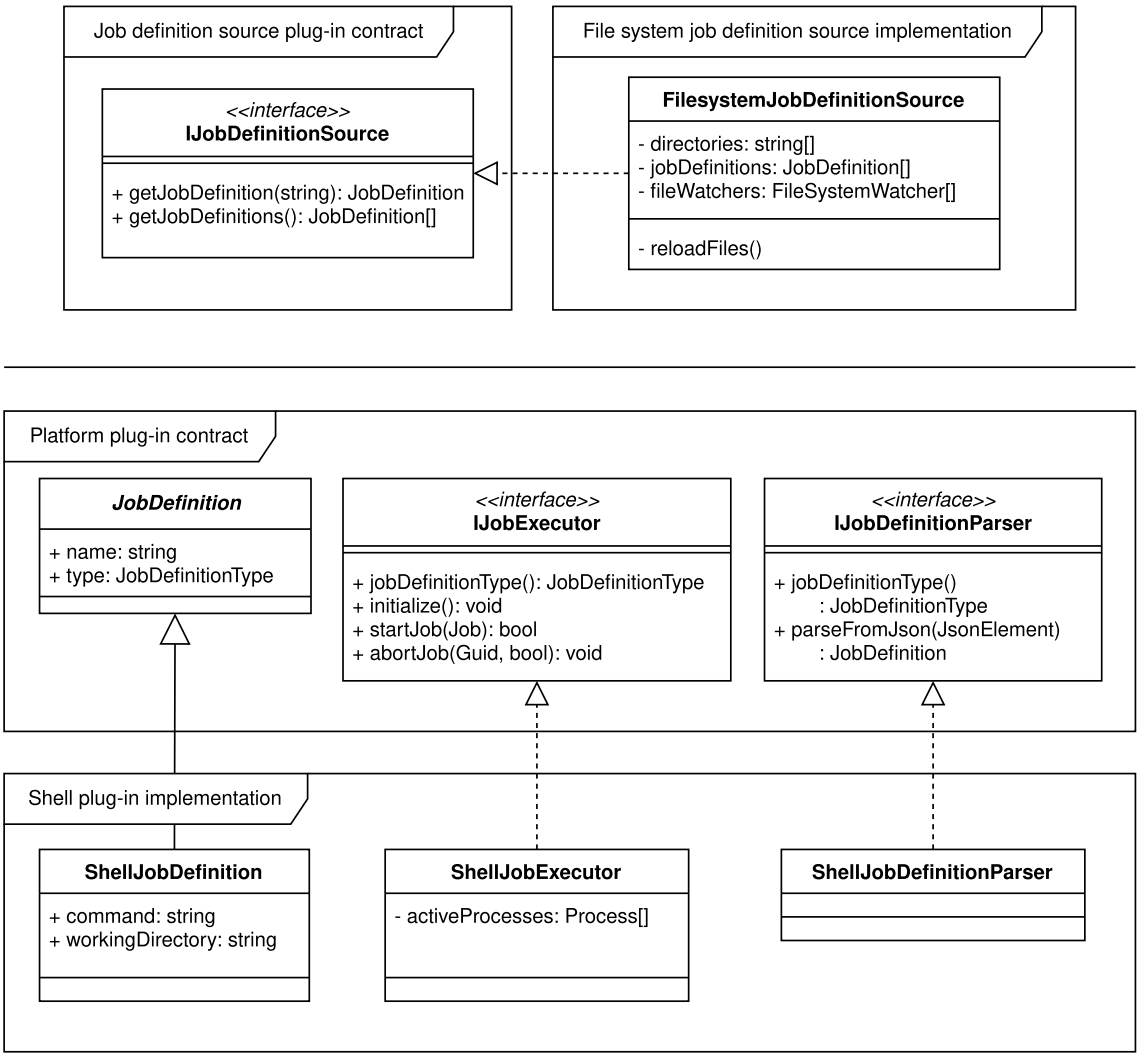

Figure 5.5. *C4 code diagram for Mission plug-in contracts, and sample plug-in implementations*

The JobDefinition abstract class is specialized by each platform plug-in to describe its

own type of job definition. The base class only contains the type and name of the job definition. Subclasses must then include the platform-specific fields required for defining a job on the particular platform.

The IJobExecutor interface is the primary contract for the operation of a platform plug-in. Every platform plug-in must implement this interface. It contains methods that are used by the core service for starting and aborting jobs. Additionally, it contains an initialization method, which is called by the core service on startup. It allows the plug-in to start any background processing operations or do other initialization steps. A job executor is always specific to one job definition type and specialized JobDefinition subclass. When multiple platform plug-ins and job executor implementations are loaded, the core service uses this job definition type information to determine which executor should be used to run a requested job.

A platform plug-in can optionally implement the IJobDefinitionParser interface. An implementation of this interface can be used to convert JSON-formatted data to instances of the specialized JobDefinition subclass in the plug-in. The parser implementation defines the structure of the data to match the structure of the job definition class. Job definition parsers are used in Mission to create job definitions from JSON data without being coupled to any specific job definition type.

Job definition source plug-ins implement the IJobDefinitionSource interface. It contains two simple operations: listing all job definitions known by the source and getting a single job definition by its name. The core service uses these simple functions for gathering and finding job definitions from all active job definition sources. A job definition source plug-in can use job definition parsers from loaded platform plug-ins to support arbitrary job definition types.

Currently, two platform plug-ins and one job definition source plug-in have been developed for Mission. These plug-ins are inspected in detail in the following sections.

5.3.4 Shell Platform Plug-in

The shell plug-in is primarily intended for locally running jobs during development and testing. It runs jobs as operating system processes. A job is defined as a command string, which is executed in the standard shell of the current operating system. The shell plug-in implementation is depicted in Figure 5.5.

A shell job definition in JSON format is presented below in Listing 5.1. In addition to the name field present in all job definitions, the shell definition contains two fields: `command` and `workingDirectory`. The first describes the command that will be run in a shell when the job is executed. The second field sets the working directory that will be used for the shell.

```
1 {
2     "name": "example-shell-definition",
3     "command": "make && ./example-job-application",
4     "workingDirectory": "./app/"
5 }
```

Listing 5.1. *An example shell job definition*

When the shell plug-in launches a job, it launches a new shell process as a child process of the Mission service. This shell then runs the job command. Input values specified are passed to the job via specially named environment variables. The plug-in waits for the shell process to exit and reads the exit status code. The exit code is used to determine whether the job succeeded or failed. Aborting a running shell job is simply done by sending a termination signal to the process.

The shell plug-in is suitable for running jobs on local hosts during development and testing. Because any command can be run as a job, the job execution can include repetitive development tasks, such as compiling the job application every time before running it. Similarly, the jobs can be defined to run an application inside a debugger. The same development and debugging tools can be used as when developing any other application.

Because the shell plug-in does not use any actual job management system, and the jobs are run as subprocesses of the main Mission service, the plug-in has some shortcomings. When the Mission service shuts down, any jobs running under the shell plug-in are immediately stopped and aborted. Job retries are also unsupported. Therefore, the shell plug-in is not suitable for use in any production environment where the completion of the jobs must be guaranteed. The other platform plug-in implemented at this point, the Kubernetes plug-in, is more appropriate for production usage.

5.3.5 Kubernetes Platform Plug-in

The Kubernetes plug-in uses the Job functionality in Kubernetes to run jobs. This plug-in can be used to run jobs in various cloud environments, as Kubernetes in itself is available on many cloud platforms.

When the Kubernetes plug-in is used, the Mission service is typically run inside a Kubernetes cluster, and the plug-in can manage Job objects by utilizing the Kubernetes API. To run a job, the plug-in just creates a new Job object using the Kubernetes API. Kubernetes then manages the execution of the job. To abort a job, the plug-in can just delete the Job object.

In the job definitions handled by this plug-in, it should be possible to specify any property that can be specified in Kubernetes Job objects. To achieve this, the job definition type specified in the plug-in utilizes object definitions from the Kubernetes API. The structure

of the job definition and a Job is very similar, and anything that can be specified in a Job can also be specified in the job definition.

Kubernetes allows extending its API by adding custom object types. These custom objects can then be stored and managed in the Kubernetes clusters similarly to managing built-in objects, such as Jobs or Pods. Mission's Kubernetes plug-in defines a custom Kubernetes object type, JobDefinition, which is used to describe and store Kubernetes job definitions in Kubernetes clusters. Developers and administrators create JobDefinition objects in Kubernetes, and the Kubernetes plug-in reads them using the Kubernetes API. Hence, the plug-in acts as a job definition source plug-in as well as a platform plug-in.

A YAML-formatted Kubernetes job definition (JobDefinition object) is presented in Listing 5.2. The job definition is based on a Kubernetes sample job [59], and matches the job started by the sample code in Appendix A. Kubernetes resources are presented in YAML format by convention.

```
1  apiVersion: mission.bitwise.fi/v1alpha1
2  kind: JobDefinition
3
4  # Metadata is automatically added to all Kubernetes objects.
5  # It contains the name of the object, along with other identifying
6  # details.
7  metadata:
8    name: sample-kubernetes-jobdefinition
9
10 # The job specification. The contents of this spec field match the JobSpec
11 # object type in the Kubernetes API. When a new Job object is created based
12 # on this job definition, this spec is directly set as the value for
13 # the spec field in the Job.
14 spec:
15   template:
16     spec:
17       containers:
18       - name: pi
19         image: perl:latest
20         command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
21       restartPolicy: OnFailure
22
23   # If the job fails, retry it at most 4 times
24   backoffLimit: 4
```

Listing 5.2. *An example Kubernetes job definition*

When a job is started, the Kubernetes plug-in extracts the `spec` field from the JobDefinition object. It then creates a new Job object using this specification. Possible input values are added to the Job as environment variables.

As the job execution is managed by Kubernetes, jobs are not disrupted by restarts of the Mission service. On startup, the Kubernetes plug-in in Mission fetches all jobs started by

itself from the Kubernetes API and synchronizes their state to the database.

5.3.6 File System Job Definition Source Plug-in

The file system job definition source plug-in reads job definitions from JSON files residing on the local file system. This plug-in is designed to provide an easy way to share job definitions in source control systems alongside the job implementations. It supports all job definition types that can be parsed from JSON.

When Mission starts, the file system plug-in reads a list of directories from the service configuration. The plug-in then finds any files named `mission.json` or `*.mission.json` (star acting as a wildcard) from these directories or any of their subdirectories. Job definitions are read from these files. The plug-in watches for changes in these files, and automatically reloads changed job definitions.

A job definition file contains one JSON object with one property, `definitions`, which is a list of objects. Each of these objects corresponds to one job definition, and has two fields: `type` and `spec`. The type property is used by the plug-in to determine which job definition parser to use. The spec property is the actual JSON definition, which will be passed to the job definition parser. An example job definition file is shown in Listing 5.3.

```
1  {
2    "definitions": [
3      {
4        "type": "shell",
5        "spec": {
6          "name": "shell-job-1",
7          "command": "echo 1" ,
8          "workingDirectory": "."
9        }
10     },
11     {
12       "type": "shell",
13       "spec": {
14         "name": "shell-job-2",
15         "command": "echo 2",
16         "workingDirectory": "."
17       }
18     }
19   ]
20 }
```

Listing 5.3. *An example job definition file for the file system plug-in, containing two shell job definitions*

Because the plug-in is configured to look for definition files in a set of directories instead of specific files, sharing the files in source control repositories is straightforward. Each developer in a team can configure their own Mission installation to look for definitions

in the root directory of a repository. The file system plug-in then finds all job definitions defined under the directory. When new definition files are created in the repository, they will be automatically loaded with no additional configuration.

5.3.7 Client Libraries

One of the requirements for the system listed in Section 5.1.1 was that jobs should be easy to launch using any programming language. The Mission gRPC API can be used with any language that has gRPC support. However, using the API can be made easier by creating client libraries that wrap it and add higher-level abstractions. These libraries can provide interfaces designed in the style of each programming language.

Currently, only a C# client library has been created. The library hides the gRPC API client generated by gRPC tools and provides a more abstract interface for interacting with Mission. The underlying protocol is hidden from the library users, and the library interface is designed to follow C# conventions. Additionally, the library automatically manages connections to the Mission service. A short sample of using the C# client library is shown in Listing 5.4.

```
1  public static async Task LaunchJob(
2          string jobDefinitionName, int inputParameter)
3  {
4      // Connect to a locally running Mission instance
5      var client = new MissionClient(
6              new MissionClientOptions { Address = "https://localhost:5000"});
7
8      // Run a job which requires one input value
9      await client.CreateJobAsync(jobDefinitionName,
10         inputValues: new Dictionary<string, string>
11         {
12             ["input_parameter"] = inputParameter.ToString()
13         },
14         metadata: new Dictionary<string, string>
15         {
16             ["created_by"] = "sample-application"
17         });
18 }
```

Listing 5.4. *Example usage of Mission C# client library*

Client libraries for other programming languages and platforms will be created in the future. Providing ready-made libraries for a wide variety of languages will make adopting Mission easier in applications using different implementation technologies.

6 EVALUATION

The main objective of this thesis was to find a solution for more easily running and managing jobs. This chapter examines how Mission addresses the two main difficulties in previous job management patterns: platform dependence and the complexity of launching jobs.

Mission was adopted for job management in Wisehockey in two distinct situations: to convert previously constantly running applications to run as jobs, and to replace Kubernetes-specific job management with platform independent job management. Developers who have implemented job management code using Mission were interviewed to evaluate the usability and benefits of the system.

In this chapter, Mission is first compared to manual Kubernetes-specific job management code in an example case. Then, the usage of Mission in Wisehockey is studied, and the system is evaluated based on the developer interviews. Finally, the general usability of the system outside of Wisehockey is briefly evaluated.

6.1 General Evaluation

To evaluate how the Mission system helps in job management, a sample of Mission usage in a program is presented here. The C# sample code is shown in Listing 6.1. The functionality is similar to the Kubernetes job management code presented in Appendix A so that comparisons can be made between them. Both sample functions launch a job, wait for it to finish, and return a Boolean value indicating whether the job succeeded or not.

```
1  public static async Task<bool> RunJob(IConfiguration config)
2  {
3      var missionClient = new MissionClient(new MissionClientOptions
4      {
5          Address = config["mission_address"]
6      });
7      var jobDefinitionName = config.GetValue("job_definition_name",
8              defaultValue: "sample-job-def");
9
10     var job = await missionClient.CreateJobAsync(jobDefinitionName);
11
```

```
12      do
13      {
14          await Task.Delay(100);
15          job = await missionClient.GetJobAsync(job.Id);
16      } while (job.Status == JobStatus.Scheduled ||
17               job.Status == JobStatus.Running);
18
19      return job.Status == JobStatus.Succeeded;
20  }
```

Listing 6.1. *A function for launching a job using Mission and waiting it to finish*

Compared to the Kubernetes-specific code in Listing A.1, the code using Mission is significantly shorter and more readable. The Kubernetes-specific function reads multiple configuration values to build the job specification. The container image, Kubernetes namespace name, command, and container restart policy have to be specified in the function. In contrast, the Mission function only specifies the job definition name. When using Mission, no details of the implementation or the configuration of the job itself is present in the managing code.

When Mission is used, the Kubernetes job properties are specified in the job definition. The sample job definition presented earlier in Listing 5.2 can be used here: it defines a job that is identical to the job created by the code in Listing A.1. Because the job is defined outside the managing code, changes to the job definition require no implementation changes, unlike with the Kubernetes sample code.

Mission makes implementing this kind of code simple, compared to using platform-specific libraries. Developers do not have to know how the jobs will be defined, or even which platform or job management system will be used to run them. Adding job management functionality to an application is very straightforward and does not require advanced knowledge about any platform. Separate job definitions allow jobs and the applications launching them to be developed by different teams. Likewise, operations teams can change the configuration of the jobs or their environment without requiring any code changes in the applications that manage jobs.

Mission also clearly addresses the platform-dependence problem: the function in Listing 6.1 contains no code specific to any job management system. The job platform used is determined in the job definition. Running the management code in local development is possible by just configuring a shell job definition to be used. Debugging the job applications is also possible by defining the job to run in a debugger. For example, the following shell job definition launches an application in the gdb debugger in a new terminal window:

```
{
  "name": "job-in-debugger",
  "command": "xterm -e 'gdb ./application'",
  "workingDirectory": "."
}
```

In this kind of simple situation, Mission fulfills the requirements set for this thesis, as shown above. In the next sections, the adoption of Mission in real-world use cases in Wisehockey is studied and evaluated.

6.2 Converting Always-Running Services to Jobs

As previously discussed in Chapter 4, some data processing applications in Wisehockey are implemented as always-running applications. These applications are always ready to receive data streams, and to run data processing and produce results. Running these applications as jobs was previously not feasible due to the difficulties in job management with Kubernetes.

Mission was utilized to convert one of these applications to run as a job. Minimal changes to the application were required. Previously, it kept waiting for a new input stream after previous streams were completely handled. The application was changed to automatically exit after an input stream is closed, indicating that all data for a sports event have been received. This allows the application to act as a job instead of a service: it exits after it has completed its work.

A controller application responsible for managing the data processing flow for sports events was changed to launch these jobs using Mission. The application includes the Mission client library, and uses it to create new processing jobs, list active jobs and abort jobs if necessary. In total, the job management logic is about 30 lines of C# code, and it contains no platform-specific information. The controller manages the jobs identically regardless of the environment. The application can be run both locally and in the Kubernetes environment without any changes.

Two job definitions were created for the processing job. A definition for the Shell platform plug-in was created with the filesystem job definition source. Similarly, a Kubernetes job definition was defined using the job definition source in the Kubernetes plug-in.

The shell job definition is defined in a `mission.json` file in the source control repository of the processing job application. The job command is specified simply as `dotnet run`, which builds and executes the application. Because the job definition is stored in the source control repository, it is automatically shared with all developers who have a local copy of the application source code. To run the controller application and jobs locally, a developer only has to configure the repository path in their local Mission configuration. No special environment setup is necessary.

The Kubernetes job definition is defined as a templated YAML file in the Helm package for the job application. This package is installed in Wisehockey's cloud environments using pre-existing deployment infrastructure. Helm templates allow the same definition to be installed multiple times with slightly altered configuration. In Wisehockey this is

used to install tenant-specific instances of the job definition. Tenant-specific configuration changes can be made to the jobs with no changes in either the controller implementation or its configuration.

To make running the jobs cost-effective, some environment-specific configuration was added to the JobDefinitions. The Kubernetes Jobs were configured to only run on specific nodes in the Kubernetes cluster. A separate node pool was created for these Jobs in Google Kubernetes Engine, and cluster autoscaling was turned on for the pool. When no processing jobs are running, the node pool is scaled down to zero nodes, and it incurs no costs. When a processing job is created, the node pool is automatically scaled up, and new nodes are provisioned. The jobs are then run on these dedicated nodes. After the jobs complete, the nodes are released. Effectively, each processing job only reserves resources for the period it is running.

To estimate the cost savings achieved by converting the data processing to jobs, weekly costs both with the old constantly running services and the new jobs were calculated. Similar usage estimation to that presented in Section 4.2.1 was used: in each venue, there are three sports matches each week, and each match is three hours long. As explained in Section 4.1.1, the data processing is replicated for each venue. These estimations were done for 20 parallel venues. Each data processing application was modeled as requiring two virtual CPU cores and four gigabytes of memory.

In the old model, the data processing applications were constantly running, so the costs of the computing resources are simple to calculate. At the time of writing, in Google Cloud's eu-north1 region, one virtual CPU core (E2 family) costs $0.024016 per hour, and one gigabyte of RAM costs $0.003218 per hour [22]. Running the 20 data processing applications constantly for one week would cost:

$$20 \cdot (7 \cdot 24\,\mathrm{h}) \cdot (2 \cdot 0.024016\,\$/\mathrm{h} + 4 \cdot 0.003218\,\$/\mathrm{h}) = \$\,204.63744$$

Running the processing as jobs, each running only for the duration of each match during the week, would cost:

$$20 \cdot (3 \cdot 3\,\mathrm{h}) \cdot (2 \cdot 0.024016\,\$/\mathrm{h} + 4 \cdot 0.003218\,\$/\mathrm{h}) = \$\,10.96272$$

The cost savings are significant: the cost of running jobs is about 5 % of the cost of running the processing constantly. The costs of the job solution are proportional to the number of matches played: if there is a period when there are no matches in a venue, no costs are incurred.

Mission makes introducing new job usages easy in the Wisehockey system. After the first application conversion to a job, two other applications have also been converted

from constantly running services to jobs. Before Mission adoption, only one processing application in Wisehockey was run as a job using a Kubernetes-specific job management solution. The migration from that job management system to Mission is studied next.

6.3 Migrating from Custom Job Management to Mission

One processing application, run during each sports event in Wisehockey, creates a visualization video from the tracking data. The application generates video frames depicting the movement of entities and renders a video stream from these frames. Video encoding requires a considerable amount of CPU and memory resources. Keeping these video generating applications running constantly was determined not to be a viable option when the feature was first implemented due to the high resource costs. Therefore, a custom solution for running the video processing as jobs in Kubernetes was previously implemented.

The previous solution only worked with Kubernetes. A controller application launched the video generating job as a Kubernetes Job when a sports event was starting. The job itself exited after it received a completion message from the data input stream. To launch the job, the controller made calls to the Kubernetes API, specifying all the details of the jobs. These details were all specified in the configuration of the controller application. To change the job specifications, either the configuration of the application or even its implementation had to be changed. The job management code was similar to the sample code in Appendix A, but it was more complex due to the greater amount of job parameters that needed to be configured.

In addition to the complexity, this solution had the disadvantage of platform-specificity. Job could only be launched in a Kubernetes environment, and testing and debugging the job launching functionality was difficult. Running the controller application locally was not possible without specialized test environments.

This solution was replaced with job management using Mission. Similar job management logic was implemented as described in the previous section. The complex Kubernetes job management code was replaced with much simpler but platform-independent code utilizing Mission. The job configuration and specification were moved to the job definitions and removed from the controller application. As a result, the application is now much simpler, and can be run without changes both locally and in Kubernetes. Changes in the job configuration do not require changes to the controller anymore.

The job examples here show that the objectives of this thesis and designing a new system for managing jobs were met. The underlying objective of using jobs was to reduce the cloud computing resource usage, and thus reduce the cloud costs. The actual purpose of this thesis was to make managing these job applications easier than before, and that

is clearly achieved by Mission. As most of the difficulties in job management are solved, more components in Wisehockey can be implemented as jobs in the future, which aids in keeping the cloud costs low and proportional to the actual resource usage.

6.4 Developer Interviews

To evaluate the usability and benefits of the system, two developers from the Wisehockey project were interviewed. Both interviewees have used Mission when developing job applications or job management services in the previously mentioned use cases. The interviews were conducted as unstructured discussions.

Both interviewees expressed that they had experienced the difficulties with job management before, when developing controller applications. One developer mentioned that they felt that the amount of code required to launch a simple job in Kubernetes was “absurd”. They specifically pointed out that they felt the various Kubernetes API objects required to run a job was very difficult to learn.

The Kubernetes-specificity in Wisehockey’s controller applications was a problem for both interviewees. One developer had had to remove or comment out Kubernetes-specific code on several occasions just to be able to run the application and test unrelated functionality. Both interviewees felt that the setup for a local testing Kubernetes environment was too difficult. One interviewee had sometimes just skipped testing their code changes until the code was deployed to a cloud environment, because local testing was too time-consuming and difficult.

The developers felt that Mission made running jobs significantly easier and the code simpler to understand. The ability to run and test the controller applications locally without a local Kubernetes setup has been useful already. The abstraction level in the Mission API and the client library was regarded to be appropriate. Both interviewees thought that they did not have to know any implementation or other internal details about Mission when using it.

The interviews also revealed some points for improvement. One developer mentioned that they are annoyed by the difficult error handling in the job client libraries, including the client library for Mission. Proper error handling often requires catching multiple different types of exceptions in the code, many of which may not be even documented. This is somewhat specific to the C# language — exceptions are not visible in the interfaces, and it is easy to accidentally omit error handling. In Mission, the situation can be improved by improving the interface documentation.

Another improvement suggestion is related to the job statuses in Mission. One interviewee had had to find out whether the job was still running or finished, without caring about whether the job succeeded or not. With the current API and client library, this has to be

done by comparing to multiple different status enumeration values. As this kind of pattern is often used in Wisehockey, some helper functions or changes in the client library could be made to reduce the possibility of mistakes and to make the code more readable.

6.5 Utility Outside of Wisehockey

The main use case Mission is designed for is running repeated but non-scheduled jobs that are launched from program code. This use case is not specific to Wisehockey. Mission could be used in any system where these kinds of jobs are needed. The platform-independence and reduced job management code complexity are beneficial in all software projects. Because Mission is a completely independent project, it may be released into public use as an open source project in the future.

Mission can be useful even if its platform-independent job management capabilities are not utilized. By default, Kubernetes does not include any support for creating jobs based on a template or a job definition. Mission provides this functionality.

Mission currently has support for only a couple of job management platforms. However, it can be adapted to support other platforms with moderate effort due to its extensible design. This allows the system to be used on a wide variety of cloud platforms and with many job management systems. The two currently supported platforms, Kubernetes and local shell, cover most use cases in Wisehockey, but other projects and systems could require support for additional platforms. Support for running jobs in Docker is planned. This could be useful when testing jobs running in containers locally, without having a Kubernetes environment running.

7 CONCLUSION

The goal of this thesis was to find a solution for easier job workload management in the Wisehockey system. The research focused on a typical case in the Wisehockey system, where an application launches and manages short-lived jobs. The two main issues to solve were the dependency to cloud platforms in the applications, and the complexity of job management due to the jobs needing to be completely specified inside the application.

The proposed solution and the main result of this thesis is a new job management system, named Mission. It was implemented as a service application that acts as an abstraction layer between existing job management services in cloud platforms and the applications that launch and manage jobs. Mission also decouples the job specifications from these applications, and instead requires all jobs to be predefined separately. An abstract API is provided to the clients, and it is used to run and manage job workloads without specifying the underlying job management platform or any job implementation details or configuration.

Compared to managing jobs with platform-specific libraries, job management code utilizing Mission is considerably simpler and shorter. All the job details are removed from the job management code. This allows the jobs and the applications doing job management to be developed and deployed completely separately and by separate teams. Because the job management code is entirely platform-independent, the same code can be run in local testing and in production environments, even though different job management services are used.

Making job management easier allowed the Wisehockey project to migrate to running more real-time processing applications as jobs instead of constantly-running service applications. This led to significant savings in cloud resource costs, as the resources are only reserved for the time they are needed. Previously, running and managing job workloads on the Kubernetes platform posed a few challenges that made developing and testing the job management code very difficult. Therefore, applications were run as jobs in only a handful of cases. Mission solved these challenges and made job management code straightforward and very easy to develop.

There are some clear aspects in which the system can be further developed. First, platform support should be extended so that the platform-independent nature of the system

can be fully taken advantage of. Currently, the system only supports running jobs in Kubernetes and on a local machine. Additional platform plug-ins should be developed for various job management systems and cloud services. Similarly, additional client libraries should be created for a wider variety of programming languages so that the system can be utilized as easily as possible.

In addition to expanding the support for various platforms, some improvements to the Mission system itself could be explored. Granular client-specific access control should be supported so that the system can be used in an environment where all clients are not trusted. The system currently does not have any access control built-in, and a client that can access the job management API can manage all jobs launched by any client. Another new feature possibility is support for time-based job scheduling. Mission could provide a single place for managing schedules for jobs running on a variety of platforms.

REFERENCES

[1] Amazon Web Services, Inc. *AWS Batch — Easy and Efficient Batch Computing Capabilities*. URL: https://aws.amazon.com/batch/ (visited on 06/20/2020).

[2] Amazon Web Services, Inc. *AWS Lambda quotas - AWS Lambda*. URL: https://docs.amazonaws.cn/en_us/lambda/latest/dg/gettingstarted-limits.html (visited on 08/23/2020).

[3] Amazon Web Services, Inc. *Job Definitions - AWS Batch*. URL: https://docs.aws.amazon.com/batch/latest/userguide/job_definitions.html (visited on 08/23/2020).

[4] Amazon Web Services, Inc. *What Is Amazon Kinesis Data Analytics for Apache Flink? - Amazon Kinesis Data Analytics*. URL: https://docs.aws.amazon.com/kinesisanalytics/latest/java/what-is.html (visited on 09/10/2020).

[5] Anderson, R., Roth, D. and Luttin, S. *Introduction to ASP.NET Core*. Microsoft Docs. Apr. 17, 2020. URL: https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core (visited on 09/07/2020).

[6] Apache Software Foundation. *Stateful Functions — Event-driven Applications on Apache Flink*. URL: https://flink.apache.org/stateful-functions.html (visited on 09/10/2020).

[7] Aridor, Y., Domany, T., Goldshmidt, O., Kliteynik, Y., Moreira, J. and Shmueli, E. Open Job Management Architecture for the Blue Gene/L Supercomputer. *Job Scheduling Strategies for Parallel Processing*. Ed. by D. Feitelson, E. Frachtenberg, L. Rudolph and U. Schwiegelshohn. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, 91–107.

[8] Badger, M. L., Grance, T., Patt-Corner, R. and Voas, J. *Cloud computing synopsis and recommendations*. NIST SP 800-146. Gaithersburg, MD: National Institute of Standards and Technology, 2012. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-146.pdf (visited on 05/03/2020).

[9] Burns, B., Beda, J. and Hightower, K. *Kubernetes: up and running : dive into the future of infrastructure*. 2nd ed. Sebastopol, CA: O'Reilly Media, 2019.

[10] A. Cogoluègnes, ed. *Spring batch in action*. Shelter Island, N.Y: Manning, 2012. 479 pp.

[11] Datadog. *8 facts about real world container use*. Nov. 2020. URL: https://www.datadoghq.com/container-report/ (visited on 05/31/2020).

[12] Docker Inc. *Docker overview*. Docker Documentation. Sept. 23, 2020. URL: https://docs.docker.com/get-started/overview/ (visited on 09/24/2020).

[13] Douglis, F. and Krieger, O. Virtualization. *IEEE Internet Computing* 17.2 (Mar. 2013), 6–9. URL: http://ieeexplore.ieee.org/document/6488669/ (visited on 05/24/2020).

[14] Dua, R., Raja, A. R. and Kakadia, D. Virtualization vs Containerization to Support PaaS. *2014 IEEE International Conference on Cloud Engineering*. Boston, MA, USA: IEEE, Mar. 2014, 610–614. URL: http://ieeexplore.ieee.org/document/6903537/ (visited on 09/24/2020).

[15] Erl, T., Mahmood, Z., Puttini, R. and Wise-Martinez, P. J. *Cloud computing: concepts, technology & architecture*. Upper Saddle River, NJ: Prentice-Hall/Pearson, 2013. 487 pp.

[16] Eyk, E. van, Iosup, A., Seif, S. and Thömmes, M. The SPEC cloud group's research vision on FaaS and serverless architectures. *WoSC '17: Proceedings of the 2nd International Workshop on Serverless Computing*. Las Vegas, Nevada: ACM Press, 2017, 1–4. URL: http://dl.acm.org/citation.cfm?doid=3154847.3154848 (visited on 06/28/2020).

[17] El-Ghazawi, T., Gaj, K., Alexandridis, N., Vroman, F., Nguyen, N., Radzikowski, J. R., Samipagdi, P. and Suboh, S. A. A performance study of job management systems. *Concurrency and Computation: Practice and Experience* 16.13 (Nov. 2004), 1229–1246. URL: http://doi.wiley.com/10.1002/cpe.753 (visited on 08/07/2020).

[18] Google. *Cloud SQL: for PostgreSQL, MySQL & SQL Server*. Google Cloud. URL: https://cloud.google.com/sql (visited on 10/07/2020).

[19] Google. *Cloud Tasks Service For Asynchronous Execution*. Google Cloud. URL: https://cloud.google.com/tasks (visited on 06/20/2020).

[20] Google. *Google Kubernetes Engine (GKE)*. Google Cloud. URL: https://cloud.google.com/kubernetes-engine (visited on 10/07/2020).

[21] Google. *Using Cloud Tasks to trigger Cloud Functions*. Google Cloud. June 22, 2020. URL: https://cloud.google.com/tasks/docs/tutorial-gcf (visited on 08/23/2020).

[22] Google. *VM instances pricing*. Google Cloud. Oct. 9, 2020. URL: https://cloud.google.com/compute/vm-instance-pricing (visited on 10/11/2020).

[23] gRPC Authors. *Introduction to gRPC*. gRPC. 2020. URL: https://grpc.io/docs/what-is-grpc/introduction/ (visited on 09/07/2020).

[24] Helm Authors. *Helm*. URL: https://helm.sh/ (visited on 09/12/2020).

[25] Hueske, F. and Kalavri, V. *Stream processing with Apache Flink: fundamentals, implementation, and operation of streaming applications*. First edition. Sebastopol, California: O'Reilly Media, 2019. 292 pp.

[26] IBM Cloud Education. *What are Containers?* IBM Cloud. Aug. 12, 2019. URL: https://www.ibm.com/cloud/learn/containers (visited on 05/24/2020).

[27] IBM Corporation. *What is batch processing?* z/OS concepts. 2010. URL: https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zconcepts/zconc_whatisbatch.htm (visited on 07/06/2020).

[28] Ibryam, B. and Huss, R. *Kubernetes patterns: reusable elements for designing cloud-native applications*. First edition. Sebastopol, CA: O'Reilly Media, 2019. 244 pp.

[29] IDG Communications, Inc. *2020 IDG Cloud Computing Survey Executive Summary*. June 8, 2020. URL: https://cdn2.hubspot.net/hubfs/1624046/2020%20Cloud%20Computing%20executive%20summary_v2.pdf (visited on 10/08/2020).

[30] Indiana University. *About batch jobs*. Jan. 27, 2020. URL: https://kb.iu.edu/d/afrx (visited on 05/31/2020).

[31] Ivan, C., Vasile, R. and Dadarlat, V. Serverless Computing: An Investigation of Deployment Environments for Web APIs. *Computers* 8.2 (June 25, 2019), 50. URL: https://www.mdpi.com/2073-431X/8/2/50 (visited on 06/28/2020).

[32] Jones, J. P. *NAS Requirements Checklist for Job Queuing/Scheduling Software*. NAS High Performance Processing Group, Apr. 1996, 16. URL: https://www.nas.nasa.gov/assets/pdf/techreports/1996/nas-96-003.pdf (visited on 08/22/2020).

[33] Khan, A. Key Characteristics of a Container Orchestration Platform to Enable a Modern Application. *IEEE Cloud Computing* 4.5 (Sept. 2017), 42–48. URL: http://ieeexplore.ieee.org/document/8125559/ (visited on 05/31/2020).

[34] D. Klusáček, W. Cirne and N. Desai, eds. *Job scheduling strategies for parallel processing: 21st International Workshop, JSSPP 2017, Orlando, FL, USA, June 2, 2017, Revised Selected Papers*. Springer, 2018. 188 pp.

[35] D. Klusáček, W. Cirne and N. Desai, eds. *Job Scheduling Strategies for Parallel Processing: 22nd International Workshop, JSSPP 2018, Vancouver, BC, Canada, May 25, 2018, Revised Selected Papers*. Springer, 2019. 147 pp.

[36] Kooijmans, A. L. *Batch modernization on z/OS*. 2nd ed. Poughkeepsie, N.Y: IBM Corp., International Technical Support Organization, July 2012. URL: http://www.redbooks.ibm.com/redbooks/pdfs/sg247779.pdf (visited on 10/04/2020).

[37] Krebs, R., Momm, C. and Kounev, S. Architectural Concerns in Multi-tenant SaaS Applications. *Proceedings of the 2nd International Conference on Cloud Computing and Services Science*. Porto, Portugal, 2012, 426–431.

[38] Larson, P.-Å. and Krishnan, M. Memory allocation for long-running server applications. *Proceedings of the first international symposium on Memory management - ISMM '98*. Vol. 34. Vancouver, British Columbia, Canada: ACM Press, Jan. 1998, 176–185.

[39] Lawenda, M., Okoń, M., Oleksiak, A., Ludwiczak, B., Piontek, T., Pukacki, J., Meyer, N., Nabrzyski, J. and Stroiński, M. Running Interactive Jobs in the Grid Environ-

ment. *Parallel Processing and Applied Mathematics*. Ed. by R. Wyrzykowski, J. Dongarra, N. Meyer and J. Waśniewski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, 758–765. URL: http://link.springer.com/10.1007/11752578_91 (visited on 10/04/2020).

[40] Luo, J. and Newton-King, J. *Introduction to gRPC on .NET Core*. Microsoft Docs. Sept. 20, 2019. URL: https://docs.microsoft.com/en-us/aspnet/core/grpc/ (visited on 09/07/2020).

[41] Mell, P. M. and Grance, T. *The NIST definition of cloud computing*. NIST SP 800-145. Gaithersburg, MD: National Institute of Standards and Technology, 2011. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (visited on 05/09/2020).

[42] Microsoft. *.NET Core intro and overview*. Microsoft Docs. Mar. 26, 2020. URL: https://docs.microsoft.com/en-us/dotnet/core/introduction (visited on 09/07/2020).

[43] Microsoft. *Batch - Compute job scheduling service*. Microsoft Azure. URL: https://azure.microsoft.com/en-us/services/batch/ (visited on 06/20/2020).

[44] Microsoft. *Create a .NET Core application with plugins - .NET Core*. Microsoft Docs. Oct. 16, 2019. URL: https://docs.microsoft.com/en-us/dotnet/core/tutorials/creating-app-with-plugin-support (visited on 09/07/2020).

[45] Minella, M. T. *The Definitive Guide to Spring Batch: Modern Finite Batch Processing in the Cloud*. Berkeley, CA: Apress, 2019.

[46] Newman, S. *Building microservices: designing fine-grained systems*. First Edition. Beijing Sebastopol, CA: O'Reilly Media, 2015. 259 pp.

[47] Pahl, C., Brogi, A., Soldani, J. and Jamshidi, P. Cloud Container Technologies: A State-of-the-Art Review. *IEEE Transactions on Cloud Computing* 7.3 (July 1, 2019), 677–692. URL: https://ieeexplore.ieee.org/document/7922500/ (visited on 05/30/2020).

[48] Rajaraman, V. Cloud computing. *Resonance: Journal of Science Education* 19.3 (Mar. 2014), 242–258.

[49] Roberts, M. *Serverless Architectures*. martinfowler.com. May 22, 2018. URL: https://martinfowler.com/articles/serverless.html (visited on 06/28/2020).

[50] Roberts, M. and Chapin, J. *What is serverless: understand the latest advances in cloud and service-based architecture*. Sebastopol, CA: O'Reilly Media, 2017.

[51] Ruparelia, N. *Cloud Computing*. Cambridge, Massachusetts: The MIT Press, 2016. 280 pp.

[52] Scholl, B., Swanson, T. and Jausovec, P. *Cloud Native: using containers, functions, and data to build next-generation applications*. Sebastopol, CA: O'Reilly Media, 2019.

[53] Simon Brown. *The C4 model for visualising software architecture*. URL: https://c4model.com/ (visited on 08/27/2020).

[54] Stojanović, S. and Simović, A. *Serverless applications with Node.js: using AWS Lambda and Claudia.js*. Shelter Island: Manning Publications, 2019. 330 pp.

[55] Sweeney, J. Virtualization: An Overview. *Encyclopedia of cloud computing*. Ed. by S. Murugesan and I. Bojanova. Chichester, West Sussex, United Kingdom ; Hoboken, NJ: John Wiley & Sons, 2015, 89–101.

[56] The Cloud Native Computing Foundation. *CNCF Survey 2019: Deployments are getting larger as cloud native adoption becomes mainstream*. Mar. 4, 2020. URL: https://www.cncf.io/wp-content/uploads/2020/03/CNCF_Survey_Report.pdf (visited on 05/31/2020).

[57] The Kubernetes Authors. *Client Libraries*. Kubernetes. Aug. 11, 2020. URL: https://kubernetes.io/docs/reference/using-api/client-libraries/ (visited on 09/07/2020).

[58] The Kubernetes Authors. *Controllers*. Kubernetes. Apr. 7, 2020. URL: https://kubernetes.io/docs/concepts/architecture/controller/ (visited on 05/31/2020).

[59] The Kubernetes Authors. *Jobs - Run to Completion*. Kubernetes. May 25, 2020. URL: https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/ (visited on 05/31/2020).

[60] The Kubernetes Authors. *Nodes*. Kubernetes. May 23, 2020. URL: https://kubernetes.io/docs/concepts/architecture/nodes/ (visited on 05/31/2020).

[61] The Kubernetes Authors. *Pod Overview*. Kubernetes. Apr. 21, 2020. URL: https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/ (visited on 05/31/2020).

[62] The Kubernetes Authors. *Running Automated Tasks with a CronJob*. Kubernetes. Aug. 7, 2020. URL: https://kubernetes.io/docs/tasks/job/automated-tasks-with-cron-jobs/ (visited on 08/23/2020).

[63] The PostgreSQL Global Development Group. *PostgreSQL: About*. URL: https://www.postgresql.org/about/ (visited on 10/07/2020).

[64] Warrender, R. L., Tindle, J. and Nelson, D. Job scheduling in a high performance computing environment. *2013 International Conference on High Performance Computing Simulation (HPCS)*. July 2013, 592–598.

[65] Weik, M. H. Job. *Computer Science and Communications Dictionary*. Boston, MA: Springer US, 2000, 845–845. URL: http://link.springer.com/10.1007/1-4020-0613-6_9670 (visited on 08/23/2020).

[66] Weik, M. H. Task. *Computer Science and Communications Dictionary*. Boston, MA: Springer US, 2000, 1737–1737. URL: http://link.springer.com/10.1007/1-4020-0613-6_19098 (visited on 08/23/2020).

[67] Wisehockey. *Real-Time Sport Analytics Platform - Overview of the Wisehockey system*. Jan. 29, 2020. URL: https://wisehockey.com/wp-content/uploads/2020/02/wh_overview_en.pdf (visited on 08/27/2020).

A SAMPLE CODE FOR RUNNING A KUBERNETES JOB

This appendix presents sample code for running a Kubernetes Job from C# code. The job is based on a sample job in Kubernetes documentation [59], and it runs a Perl command calculating an approximation of π. The sample program is shown in Listing A.1.

```
1  // Run a job in Kubernetes. Returns true if the job succeeds,
2  // false if it fails.
3  public static async Task<bool> RunJob(IConfiguration config)
4  {
5      var kubernetes = new Kubernetes(
6          KubernetesClientConfiguration.BuildDefaultConfig());
7
8      var container = new V1Container
9      {
10         Name = "pi",
11         // Read the container image from configuration
12         // to allow changing it without code changes
13         Image = config.GetValue("job:image", defaultValue: "perl:latest"),
14         Command = new List<string>
15         {
16             "perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"
17         }
18     };
19
20     var job = new V1Job
21     {
22         Metadata = new V1ObjectMeta
23         {
24             GenerateName = "sample-kubernetes-job"
25         },
26         Spec = new V1JobSpec
27         {
28             Template = new V1PodTemplateSpec
29             {
30                 Spec = new V1PodSpec
31                 {
32                     Containers = new List<V1Container> { container }
33                     RestartPolicy = "OnFailure"
34                 }
35             },
36             BackoffLimit = 4
37         }
```

```
38      };
39
40      var newJob = await kubernetes.CreateNamespacedJobAsync(
41              body: job, namespaceParameter: config.GetValue(
42                  "job:namespace", defaultValue: "default"));
43
44      // Wait until the job has completed
45      while (true)
46      {
47          await Task.Delay(TimeSpan.FromMilliseconds(100));
48
49          var currentJobStatus = await kubernetes.ReadNamespacedJobStatusAsync(
50                  newJob.Name(), newJob.Namespace());
51
52          if (currentJobStatus?.Status?.Conditions == null)
53          {
54              continue;
55          }
56
57          if (currentJobStatus.Status.Conditions.Any(
58              x => x.Type == "Complete" && x.Status == "True"))
59          {
60              // Job succeeded
61              return true;
62          }
63          else if (currentJobStatus.Status.Conditions.Any(
64              x => x.Type == "Failed" && x.Status == "True"))
65          {
66              // Job failed
67              return false;
68          }
69      }
70  }
```

Listing A.1. *A function for launching a Kubernetes job and waiting it to finish*

The function gets application configuration as a parameter. First, the function creates an instance of the Kubernetes client class. Then, it uses the client to create a Job in the Kubernetes cluster. In this sample code, the job and container objects only contain the bare minimum information required to launch a job. The container image and the Kubernetes namespace are read from the application configuration, to allow changing it without making changes to the source code.

When the job has been created, the function enters a loop and starts to poll the status of the new job. It repeatedly fetches the current status of the job and checks if the job has succeeded or failed. Between each loop iteration, the function waits for 100 milliseconds. When the job finishes, a Boolean value is returned indicating whether the job succeeded or not.

This sample code lacks all error handling. The Kubernetes client functions CreateNames-

pacedJobAsync and ReadNamespacedJobStatusAsync will throw exceptions when errors occur, for instance when the Kubernetes cluster cannot be connected to or the operations fail for some other reason. In a real-world scenario these exceptions should be handled properly.

Notable in this code is the lack of extensibility: if any other settings or properties had to be set in the job or container objects, changes would have to be made in the source code of the function. If the new values had to be configurable, they would also have to be added to the application configuration. Examples of these kinds of values include CPU and memory requests and limits, Kubernetes node selectors, arbitrary configuration values passed via environment variables to the job, job timeouts, and networking configuration. These values are often specific to the environment the jobs are run in and change frequently. These changes all have to be made to the code or configuration of this application, even though they actually are specific only to the job.