



**Politecnico
di Torino**

POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

**Allowing prototyping of
applications running on
heterogeneous HW through a
multi-tenant platform based on
cloud microservices**

Relatori

Urgese Gianvito

Risso Fulvio Giovanni Ottavio

Pignata Andrea

Candidato

Fanuli Giuseppe

April 2024

Abstract

Neuromorphic engineering is an emerging field that aims to develop physical implementations of brain-inspired systems using electronic circuits. These circuits, drawing inspiration from the brain's architecture, offer several advantages over the traditional von Neumann architecture. These advantages include improved power consumption, enhanced adaptability and learning capabilities, real-time processing, and fault tolerance. As a result, neuromorphic technologies hold the potential to replace conventional approaches in various domains. However, it is important to note that existing technologies for neuromorphic computation, such as chips and boards, are still predominantly in the prototype stage. Consequently, they are often expensive and not readily available on the market. This limitation currently impedes the widespread adoption of these technologies in the development of IoT and industrial use cases. This thesis project aims to find a solution for this problem allowing users to prototype applications on a cloud platform based on microservices, guarantying a user-friendly access to the system through a web application accessible by a common browser, and run such applications on a heterogeneous hardware platform composed by several boards for neuromorphic and general-purpose computation. The platform allows tenants to develop, build, run, test and compare the results of their application on different boards to provide an overview of the state-of-the-art reached by such technologies. In the design stage, the user can take advantage of the flexibility and general-purpose nature of the platform, running different tools and application that he may need for his goals. Once the application is ready, the platform allows the user to deploy and run it on real HW, to verify the correctness of the results and check parameters like running time and power consumption. This is possible through a job scheduling system, which orchestrates all the resources by assigning incoming jobs to the available nodes according to the user needs. The user can interact with the platform through a user-friendly interface by means of well-known applications such as Visual Studio Code. This IDE is one of most used and powerful environments for developing software thanks to its integration with some features able to create a fully customizable interface which best address all the use cases and needs of the user. The infrastructure relies on two major open-source technologies: Kubernetes and Slurm. The former orchestrates the microservices to provide everything needed by the user to use the

platform in the most user-friendly and effective way, the latter is used to orchestrate all the heterogeneous boards in order to assign the “jobs” created by the user and provide them the results of the computation. By combining these two technologies, it is possible to manage a distributed system without taking care the place of the various physical resources and managing a multi-layer infrastructure, completely transparent to the users, that improve the isolation between resources needed for the computation and resources needed to expose the cloud services.

Contents

List of Figures	7
1 Introduction	9
1.1 Goal	10
2 Background	13
2.1 Neuromorphic Engineering	13
2.1.1 Neuromorphic Solutions	14
2.1.2 Software Modules	15
2.2 Prototyping Platform	15
2.2.1 Board as a Service	16
2.2.2 Available market solutions	16
2.2.3 Heterogeneous Hardware	17
2.3 Cloud Computing	18
2.3.1 Commercial Products	18
2.3.2 Virtualization	19
2.4 Crownlabs	20
2.5 Kubernetes	21
2.5.1 Kubernetes Resources	21
2.5.2 Kubernetes Components	23
2.6 Service Mesh	24
2.6.1 Istio Service Mesh	25
2.7 Slurm	26
2.7.1 Slurm architecture	27
2.7.2 Slurm Job	27
3 Materials and methods	29
3.1 Design	29
3.1.1 Microservice-based architecture	29
3.1.2 Kubernetes as Backend	30
3.1.3 System resources	31
3.1.4 Visual Studio Code as developing platform	31

3.1.5	Istio Service Mesh	32
3.1.6	Slurm as HPC cluster	32
3.1.7	User developing process and job submission	33
3.2	Implementation	36
3.2.1	Container-based instances	36
3.2.2	Infrastructure	38
3.3	Journey	38
3.3.1	Architectural choices	39
4	Results and discussion	41
4.1	Testing conditions	41
4.2	User workflow	42
4.3	Slurm Testing	44
5	Conclusion	49
5.1	Future works	50
	Bibliography	53

List of Figures

2.1	Contaners vs Virtual Machines	20
2.2	General overview of components of a Kubernetes cluster [25]	24
2.3	Service communication before <i>Istio Service Mesh</i>	26
2.4	Service communication after <i>Istio Service Mesh</i> [26]	26
3.1	CRDs and Kubernetes operators of the cluster[27]	30
3.2	Kiali dashboard	33
3.3	Slurm architecture	34
3.4	User interaction with the coding platform	35
3.5	Job creation process	36
3.6	User Namespace overview	37
4.1	The architecture used for testing purposes	42
4.2	VsCode instance accessible by the browser	43
4.3	A view of the tasks allowing users to run their script to a board hosted by Slurm	44
4.4	Output of <code>sinfo</code> command. All the nodes are in <i>IDLE</i> state, waiting for new jobs	45
4.5	Output of <code>squeue</code> command. The status field (ST) has the value <i>PENDING</i> (PD) for the job named <code>example</code> and the <i>NODELIST</i> field is empty since no nodes have been allocated yet.	45
4.6	A view of the command <code>scontrol show job</code>	46
4.7	The <code>htop</code> command executed on the <i>jts03</i> node. The CPU 1 has been allocated for running the Python script named <code>example.py</code>	47
4.8	The content of the file <code>slurm-48.out</code> . The file contains the prints of the executed script. The filename is the default name provided by Slurm and it can be modified at configuration time.	48

Chapter 1

Introduction

Over the years, with the advent of the *Cloud Computing*, computation is moving more and more to data centers where a lot of computing resources are gathered together in order to address the various needs of the customers in the form of services to be consumed. On the other hand, with the advent of the *Internet of Things (IoT)*, some of the computation is moving to smart devices, equipped with low-power chips able to perform simple tasks [1].

Trends suggest that industries investing in various sectors such as automotive, renewable energies but also RF/microwave communications, are moving to the utilization of these embedded devices for performing specific kinds of computation which strongly require the usage of sensors for getting data and antennas for communications [1].

In addition, new computational paradigms are emerging which require special needs to be addressed and new forms of devices to be used.

One of these new types of computation paradigms is represented by *Neuromorphic Computation*. Neuromorphic engineering is a relatively young field that takes inspiration from different subjects of science such as neurology, psychology, biology, physics, mathematics, computer science, and electronic engineering to design and develop artificial neural systems and machine learning algorithms simulating the functioning of the human brain.

Devices supporting this kind of computation are mainly prototypes from the research world, so transferring these technologies from a research environment to a production environment is not so simple due to the market cost of the devices themselves. Consequently, there is a growing demand to develop technologies that allow us to work without physically owning the hardware devices. This is where the Cloud Computing paradigm comes into play.

The cloud environment tries to expose those HW devices as virtual devices to the various users, allowing them to use the resources *as a service*. The cloud service model can be summarized as *BaaS, Board as a Service*, stressing that a board of

the system becomes a service for users who are allowed to use it without effectively owning the hardware itself. The strategy of BaaS is inspired by the Platform as a Service (PaaS) paradigm that is commonly adopted in the cloud domain.

There are already several solutions available on the market that expose special boards as virtual service to the customers such as *NengoEdge* [2], *Cadence Pro-tium* [3] or *STM32Cube.AI Developer Cloud* [4] developed by *STMicroelectronics* which provides a set of services allowing to optimize, quantize benchmark and deploy user-trained neural network on a STM32 target, the proprietary boards of *STMicroelectronics*. The problem with these solutions consists of the devices that have been integrated, as most of them are solutions developed by a single brand, limiting users to their hardware only and providing just a cheap way.

Having a platform that manages a set of *heterogeneous devices*, where research teams of academies and companies can evaluate a set of different brand solutions before buying them is the motivation behind the project of *Politecnico di Torino* for *EBRAINS-Italy* in which this thesis want to provide a solution and a fully functional design of the platform.

EBRAINS-Italy (European Brain ReseArch INfrastructureS-Italy) will be the Italian node of the European distributed infrastructure of *EBRAINS* which aims to allow the usage of the most advanced technologies of data modeling, computation, and analysis for the neuroscience field to the healthcare sector.

In detail, the goal of *Politecnico* is to create a *Neuromorphic Computing Lab* focused on the design of a platform suitable for the development of new *Neuromorphic solutions* [5].

1.1 Goal

As mentioned before, the goal of this thesis project is to develop a cloud infrastructure allowing users to use physical devices through the internet, hence without effectively owning them, a sort of *BaaS (Board as a Service)*.

The system adheres to the design outlined by the *CrownLabs project* [6], providing a platform structured around micro-services. This platform enables students to engage in practical exercises and examinations within secure virtual environments. The cloud environment I'm developing has been designed considering the advancements made in related technologies to ensure its alignment with the current state of the art. The platform relies on a collection of micro-services that improve the maintainability and reliability of the infrastructure in case of failures.

Kubernetes [7] is the main tool used as a container and micro-service orchestrator for developing cloud services by providing also storage management as well as user accounting and request authentication. The main service I'm developing with this project consists of an integrated platform where users can design their own code in

the most user-friendly and effective way. Within this platform, users are allowed to test their code on physical boards to evaluate such devices.

These physical resources (e.g. prototypes of boards) are managed by *Slurm* [8], an open-source project of an HPC (High-Performance Computing) cluster. Thanks to Slurm it is possible to allocate specific resources to the various users in an exclusive way, queue incoming requests if there are no available resources, and provide the users with information about the HW state (e.g. power consumption, executing time, ...) during the job execution.

Chapter 2

Background

This thesis project was born with the need to create a cloud infrastructure to allow users to test their algorithms on physical boards that may be really expensive and hard to find on the market due to their prototypical nature. The system is thought of as PaaS (Platform as a Service) since users can design, code, build, and test their algorithms directly using the facilities offered by the infrastructure. It consists of two different clusters. The former handles and orchestrates all the micro-services needed to expose a user-friendly and easy-to-use system to the users; the latter handles all the physical resources (boards, computing nodes, CPUs, FPGA, GPUs) and orchestrates them to run jobs issued by the users.

In this chapter, I will introduce all the technologies and concepts adopted for the development of the platform to discuss in the following chapter how these tools have been integrated for the purposes of this thesis project.

2.1 Neuromorphic Engineering

This thesis project started with the goal of allowing researchers and engineers to develop new algorithms related to a rather new engineering sector, the *Neuromorphic Computing*.

Neuromorphic Computing is a new field of Computer Engineering that simulates the way our brain processes information and decides by using electronic circuits implemented in very large-scale integration technology. This emerging field is characterized by its multidisciplinary nature and its focus on the physics of computation, driving innovations in theoretical neuroscience, device physics, electrical engineering, and computer science [9].

As support for the research and development of Neuromorphic Computing, specialized devices are extensively employed. These devices are designed specifically to execute algorithms related to this field, alongside specialized software modules

that facilitate the development process for such algorithms.

2.1.1 Neuromorphic Solutions

Regarding physical devices, several solutions coming from different brands are already available on the market. These devices called also *Development Kit*, have been created to allow developers to use special hardware created ad-hoc for testing and developing new technologies.

Regarding the Neuromorphic computation, there are different solutions available on the market.

BrainChip

BrainChip is an Australian-based company, a worldwide leader in edge AI on-chip processing and learning. BrainChip engineered several technologies regarding the *Artificial Intelligence (AI)* world. Examples of these technologies are represented by the *Akida processors*, ultra-low-power neuromorphic processors that mimic the human brain to analyze only essential sensor inputs at the point of acquisition—processing data with unparalleled performance, precision, and reduced power consumption [10].

UAB NeuroTechnologijos

UAB Neurotechnologijos is a Lithuanian company that operates on the branch of AI technologies. It produces boards, pluggable inside a common slot of PCIe, for supporting the developing process of neural networks. These boards are powered by the *NM500* chip [11], a neuromorphic chip with 576 neurons engineered by *NEPES* [12]. This chip provides a re-configurable and low-power multiple recognition engine for storage devices with high-speed contextual recognition, uncertainty management, and hypothesis generation for more robust decisions.

SynSense

SynSense is the a supplier of neuromorphic intelligence and application solutions [13]. Based on the experience of the Research and Development department of the University of Zürich and the ETH Zürich, it provides intelligent application solutions that combine *neuromorphic sensing and computing*. SynSense offers solutions for edge computing applications by providing ultra-low power consumption, ultra-low latency inference ASICs and IP blocks, as well as full-stack application development services. In addition, it is the only company which involves both sensing and computing technologies.

Regarding hardware solutions, SynSense provides different products concerning different fields of application such as *SPECKTM*, a fully event-driven neuromorphic

vision SoC, and *XYLOTM*, a programmable, ultra-low power neuromorphic chip for low-dimensional signal processing.

SynSense offers a variety of software modules that best fit the characteristics of their devices such as *ROCKPOOL*, *SINABS*, *SAMNA*, *TONIC*.

2.1.2 Software Modules

Several libraries and packages are available to support developers using such technologies.

snnTorch

snnTorch is a Python package that extends the capabilities of *PyTorch*, a package that enables fast, flexible experimentation and efficient production through a user-friendly front-end, distributed training, and ecosystem of tools and libraries [14]. *snnTorch* performs gradient-based learning with spiking neural networks by taking advantage of PyTorch’s GPU-accelerated tensor computation and applying it to networks of spiking neurons [15].

GeNN

Nowadays GPU computation is becoming more and more a key point of workstations and edge computing devices. *GeNN* (*GPU-enhanced Neuronal Networks*) is a framework which aims to facilitate the use of graphics accelerators for computational models of large-scale neuronal networks aiming to speed up the computing process. The GeNN library an open source project written in C++ which generates code to accelerate the execution of network simulations on NVIDIA GPUs through a flexible and extensible interface, which does not require in-depth technical knowledge from the users. [16]

2.2 Prototyping Platform

The new computing paradigms that are emerging nowadays require the usage of special devices, optimized to execute the corresponding algorithms most effectively. In most cases these devices reside with the end users or at least in a room close to them, necessitating developers themselves to handle the maintenance and management of these devices. To facilitate and improve the development process of new algorithms using such devices, companies are moving towards the usage of *Prototyping Server Farms*.

A prototyping server farm is a controlled-temperature room where several devices

have been placed together to be accessed remotely from all around the world allowing users to perform software development, software-driven verification, hardware-software validation tasks, and various other tasks as needed [17].

2.2.1 Board as a Service

The necessity for the creation of these prototyping server farms is to improve the company's productivity by delegating to whoever handles the server farm infrastructure the maintenance and management of the hardware. This is a big advantage for those companies with limited resources since they can evaluate their work by using remote devices, which are expensive, without effectively owning them.

Several device manufacturers have developed their cloud infrastructure to address such problems by allowing customers to access the *Prototyping Platform* they've developed allowing them to utilize their proprietary boards as a service.

2.2.2 Available market solutions

Below is a brief presentation of the main Prototyping Platform available on the market.

STM32Cube.AI Developer Cloud

The *STM32Cube.AI Developer Cloud* [4], developed by *STMicroelectronics* is a set of services allowing to optimize, quantize, benchmark, and deploy trained neural networks on STM32 microcontroller targets. Users can directly upload the neural network model they want to use or load a subset of the neural network model *ZOO*, developed by STM32.

Protium Enterprise Prototyping

Prototyping platforms are well-known and used in the System-on-chip (SoC) and Application Specific Integrated Circuit (ASIC) development world. In such a context, developers design their hardware using Hardware Description Languages (HDLs) and then request the production of their chips on silicon to foundries. This process is very expensive; HDLs usually allow developers to test their projects via software simulators, checking the correctness of all their components before manufacturing.

Another possibility is to deploy their design on re-configurable chips, known as Field Programmable Gate Array (FPGAs). Hardware deployed on FPGAs performs way faster than any software simulation, but those pieces of hardware are pretty expensive.

With this in mind, prototyping platforms such as Protium Enterprise by Cadence [3] allow Pre-silicon prototyping for system verification (to check if the hardware is

working as intended) and to start development of software compatible with such work-in-progress chip before being even manufactured. In this way, hardware developers can save money, as malfunctioning designs can be corrected before production, and save time, allowing them to hit the market with their hardware + software frameworks faster.

NengoEdge

NengoEdge is a prototyping platform (still in beta version on March 2024) developed by *Applied Brain Research* [2]. The NengoEdge solution's purpose is to provide users with a development framework where high-performance and low-power AI applications can be designed with no-code tools in the cloud. Then the framework supports easy deployment procedures of those solutions to edge devices. Typical application targeted by this framework are: *Keyword spotting*, *Automatic speech recognition*, *Text-to-speech*, *Natural language processing* and *Signal processing*

2.2.3 Heterogeneous Hardware

All the previously mentioned platforms rely on cloud services for exposing to the customers the computational resources of the boards they've implemented within their Prototyping Server Farm. These boards may belong to the same manufacturer, as the solution proposed by STMicroelectronics, or belong to different manufacturers and be created with different architectures to address the various needs of developers.

Having a platform that integrates a heterogeneous set of computational resources, allows users to choose which device best fits their needs and develop their algorithms most effectively.

Devices that may be integrated with these kinds of platforms can be devices created ad-hoc for addressing specific research fields.

Nvidia

An actor that operates in the sector of AI technologies is *Nvidia* with the family of *NVIDIA Jetson* boards. These boards are integrated systems for the edge computing process with a very tiny form factor and high performance. Each Nvidia Jetson board is a *SOM (System on Module)* which includes CPU, GPU, memory, power management, and high-speed interfaces. Nvidia provides different solutions and configurations for its boards, accounting for the different needs of the customers. Available products are *Jetson Orin* family, *Jetson Xavier* family, *Jetson TX2* family and *Jetson Nano* family [18].

2.3 Cloud Computing

Officially born in 2002 with the creation of Amazon Web Services, but with its bases founded in the late '60, Cloud Computing nowadays is the main solution adopted by companies who want to expose their digital services to customers.

The National Institute of Standards and Technology (NIST) defines cloud computing as a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [19].

This cloud model has three different service models:

- **SaaS** - *Software as a Service*. The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, except for limited user-specific application configuration settings.
- **PaaS** - *Platform as a Service*. The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.
- **IaaS** - *Infrastructure as a Service*. The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer can deploy and run arbitrary software, that can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).

2.3.1 Commercial Products

The IaaS solution allows companies to create their own fleet of cloud services without effectively owning the underlying hardware. This helps those companies that cannot create their on-premise servers due to problems of space, costs, and effort to manage the whole architecture.

Several solutions can be found on the market.

Amazon Web Services (AWS)

Amazon offers a collection of cloud computing products able to meet all the needs of the customers. Over 200 products are available such as *Amazon Elastic Compute Cloud (EC2)* which offers the possibility to the users to rent virtual machines and run whichever application they want and *Amazon Simple Storage Service (S3)* which instead offers a *Storage as a Service*, a virtual storage space users can use to store their private data.

Other competitors are **Microsoft Azure** and **Google Cloud Platform**.

2.3.2 Virtualization

Virtualization is the main concept behind cloud computing and it allows a more efficient and powerful utilization of the underlying physical hardware. Virtualization uses software to create an abstraction layer over computer HW that allows the HW elements of a single computer—processors, memory, storage, and more—to be divided into multiple virtual computers, commonly called virtual machines (VMs). Each VM runs its own operating system (OS) and behaves like an independent computer, even though it is running on just a portion of the actual underlying computer [20].

Virtual Machines

As explained above, Virtual Machines (VMs) are the result of the virtualization process. They are highly used for virtualizing servers in a distributed system (e.g. in a datacenter) avoiding the problem of managing several physical servers for accounting different jobs. The most widespread approach is the *One app per server* rule which consists of having a single VM running a specific application (e.g. Firewall, Nat, or whatever other application). This approach increases the process isolation and the system reliability in case of failures and by concatenating several of these VMs it is possible to create the macro-service they are built for.

Containers

Containers are another tool of the virtualization process. A container is a **lightweight virtualization** of resources and the operating system itself is responsible for handling them. Containers are the cloud-native solution for services engineered for the cloud environment.

A container is a sort of isolated space that creates a virtualized environment lighter than the normal virtualization (used for VMs).

The *Figure 2.1* compare the architecture of a fully virtualized environment (a VM) and a lightweight virtualized environment (a container). In the case of VMs,

the virtualization process is handled by the *hypervisor*, a software that runs over the host Operating System (OS) and it is involved in intercepting (trap) the system calls coming from the guest OS and translate it for the host OS. In this way, the VMs are unaware of being virtualized and behave as a normal system. On the other hand, containers are handled directly by the host OS which is involved in providing the proper isolation to them. This involves fewer overheads to be managed and, as a consequence, fewer resources to be used for handling those instances.

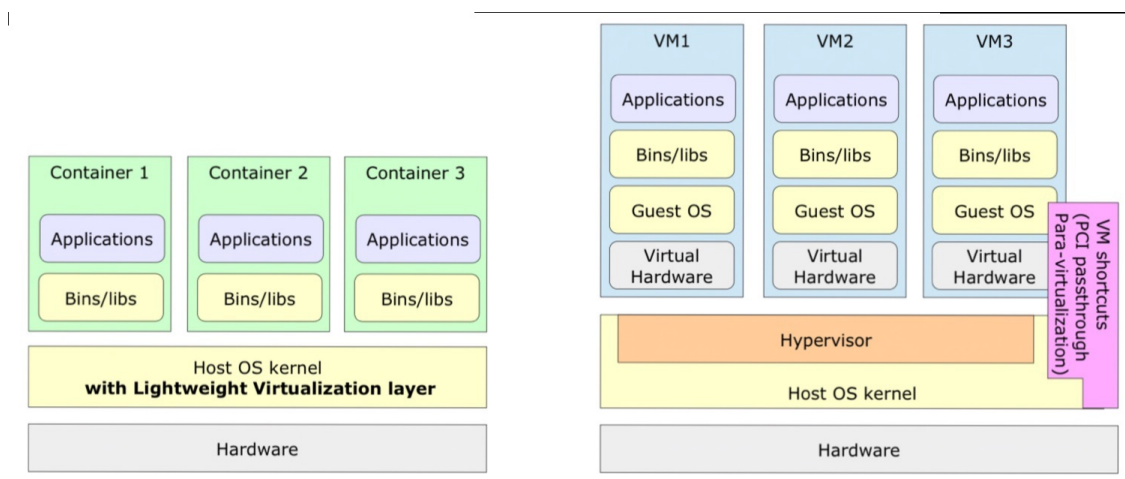


Figure 2.1: Containers vs Virtual Machines

2.4 Crownlabs

This thesis project is inspired by the *CrownLabs project*, a platform created at Politecnico Di Torino during the pandemic years to make available to the students of Politecnico a virtual environment where they can practice with educational tools and take exams (e.g. coding exams) [6]. CrownLabs platform is based on the cloud computing paradigm and it heavily relies on the containerization of its micro-services to improve the reliability of the system.

Crownlabs services

The Crownlabs system relies on *Kubernetes*, a powerful tool used for managing, deploying, and orchestrating cloud micro-services. The Crownlabs platform allows users (students mainly) to use virtual environments accessible as web applications through the browser. These virtual environments are containerized operating systems integrated with the *noVNC* web client which offers the possibility to access the virtual Desktop from the web without having a client installed on the end-user computer [21].

The other goal of CrownLabs is to deliver Politecnico exams to the booked students by integrating these virtual environments with *Respondus Lockdown Browser* [22], a custom browser that locks down the testing environment within a learning management system.

2.5 Kubernetes

Kubernetes, also known as k8s, is an open-source system developed by Google and maintained by *Cloud Native Computing Foundation*. Kubernetes is the main technology used in the cloud world for deploying, scaling, and orchestrating micro-services.

Orchestration happens thanks to a set of operators that continuously observe the state of the various containers (or more in general resources) deployed and compare such state with the desired state defined in the various configuration files by the developers. In this way, it is possible to manage all the resources by scaling, creating, updating, and deleting them according to the various needs or actual state of the system.

2.5.1 Kubernetes Resources

Kubernetes relies on a huge set of built-in resources as well as user-defined resources. Below, is a brief introduction to the most important one.

Node

A node is a physical machine (or even a virtual one such as a VM) that hosts the Kubernetes system. A node can be a *Control Plane Node* or a **Worker Node**.

A control plane node is the main actor of the cluster (a server) and it is responsible for the coordination of the other nodes (worker nodes).

A worker node is instead the node responsible to host and run the containers.

Namespace

Users that consume cloud services are called *tenants*. To provide isolation and security for each tenant's data, the concept of *Namespace* has been developed.

A Namespace is a portion of the cluster where resources deployed inside obey to the rules associated with that Namespace, hence providing strong isolation and more security since the access to such resources is restricted to who has the proper role defined for that Namespace.

A resource can belong to a given Namespace (in this case it is called *namespaced*, or be *cluster-wide* if it is accessible from the the whole cluster.

Pod

Pods are the smallest deployable units of computing that you can create and manage in Kubernetes. It can be composed of one or more containers with shared storage and networks. Containers within the same pod are tightly coupled and they work together to provide a given service. Consider a scenario when a container hosts an HTTP server. This server receives external traffic, and there is a requirement to filter this incoming traffic using a proxy server: the proxy server will be hosted by another container within the same pod. The proxy server will be hosted by another container within the same pod. In this case, the proxy server container is called *sidecar container*.

Service

A Service is a method for exposing a network application that is running as one or more Pods in your cluster [23].

There are different types of services:

- **ClusterIP** - It is the default service type and it exposes the Service on a cluster-internal IP. This kind of service is accessible only within the cluster and if you want to make the application accessible from the public internet you need to create an *Ingress* or *Gateway* resource.
- **NodePort** - It creates a static port where services of this type can be exposed to the node's IP where they are deployed. In this way, Pods referring to this kind of service are accessible from the public internet by referring to the physical node IP and the exposed port.
- **LoadBalancer** - It exposes the Service externally using an external load balancer, a component that distributes new connections across the server to homogenize the load of each pod deployed.

Deployment and ReplicaSet

A *ReplicaSet*'s purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods to increase the system reliability in case of failures of one or more Pods[24]. Consider the case of a failing Pod. The condition given by the ReplicaSet is no longer met. To satisfy the desired state, the failed pod is deleted and another one is created automatically.

A ReplicaSet is defined through fields. This field contains a label that specifies the pod to be referred to and the number of replicas desired for that pod. A *Deployment* instead provides a declarative way for defining the ReplicaSet.

Custom Resource Definition

Kubernetes works using resources such as Volumes, Pods, Services, and so forth. These resources are handled and monitored by the so-called *operators*. Operators are the core components of Kubernetes and they are responsible for making the desired state of each resource actual in the cluster.

Kubernetes has its own set of built-in resources but allows developers to create their own as well as the proper operators for managing them. These resources are called *Custom Resource Definition (CRD)*.

2.5.2 Kubernetes Components

The following section illustrates the most important Kubernetes components.

etcd

As described in the previous paragraphs, Kubernetes is an orchestration platform of containers. Orchestrating containers means managing a complex workload and Kubernetes uses the *etcd* component for simplifying this task. The *etcd* is an open-source, distributed key-value storage system that facilitates the configuration of resources, the discovery of services, and the coordination of distributed systems such as clusters and containers. It provides a single, consistent source of the truth about the status of the system at any given point in time such as the current state of the cluster, the desired state, configuration of resources, and runtime data.

ApiServer

APIServer is another key concept of Kubernetes. It provides both external and internal interfaces with the Kubernetes cluster utilizing a collection of REST APIs. This means that the only way to interact with the cluster is through the *APIServier* which is responsible for parsing and validating each call and submitting the requests to the *etcd* component described above to update the cluster state with the new requests.

Scheduler

The scheduler is the component involved in the selection of which node (or nodes) has to be used for hosting a given resource (e.g. Pod). *Kube-scheduler* is the default Kubernetes scheduler and runs as a part of the control plane.

Kube-scheduler selects an optimal node to run newly created or not yet scheduled (unscheduled) pods according to the specification provided for each pod such as CPU resources or the amount of memory. The scheduler filters out any node that doesn't meet those requirements.

Controller Manager

The *Controller Manager* is responsible for matching the current cluster state with the desired one by running the *reconciliation loop* acting on the default Kubernetes resources. Then it communicates with the *APIServer* to create, update, or delete such resources.

Kubelet

Kubelet is the agent, deployed on each node, responsible for scheduling the container within a pod on the node and reporting the status of the creation to the *APIServer*.

The *Figure 2.2* shows the general architecture of a Kubernetes cluster, highlighting the core components belonging to the system. It is important to notice the *Kubelet* agent running on each worker node of the cluster, the *APIServer* and the *etcd* component running on the master node as well as the *scheduler*, responsible to assign the assign new pods to the various worker nodes.

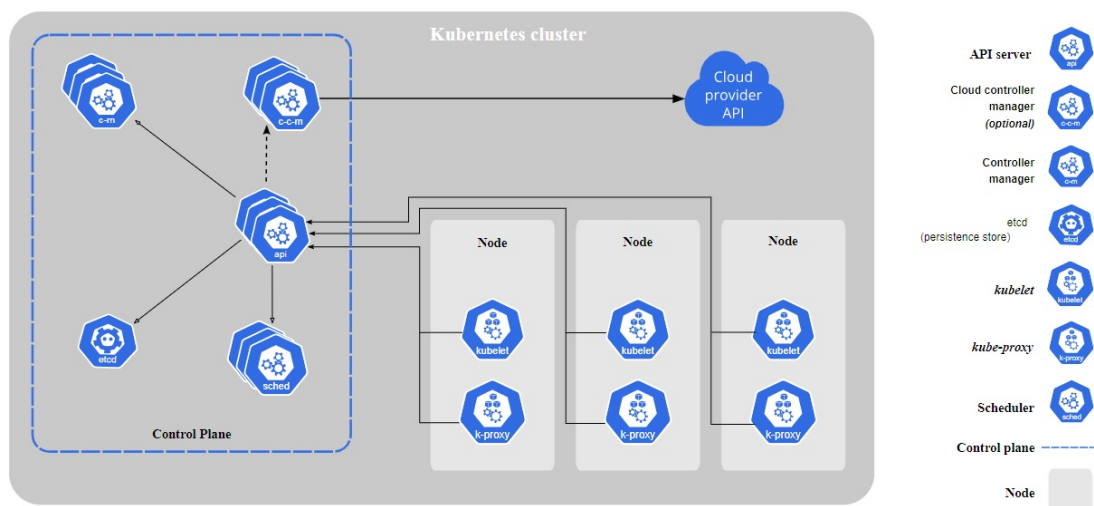


Figure 2.2: General overview of components of a Kubernetes cluster [25]

2.6 Service Mesh

Modern applications, delivered through the cloud computing paradigm, are generally organized as a collection of micro-services that implement a specific business logic. These micro-services are interconnected with each other to deliver the bigger service that the entire infrastructure aims to provide. To facilitate the interconnection of micro-service, the *service mesh* comes into play.

A *Service Mesh* is an additional layer that can be added to the system. This layer provides some important features like observability of the system, security, and traffic management without adding additional code to the service already implemented. Service mesh is widely used in distributed environments such as the Kubernetes-base clusters. This is because deploying distributed services becomes increasingly complex as the size of the cluster grows, mainly due to the increasing complexity of the Service-to-Service communication. Considering this aspect, introducing features like service discovery, load balancing, failure recovery, metrics, and monitoring will enhance the observability and management of the system.

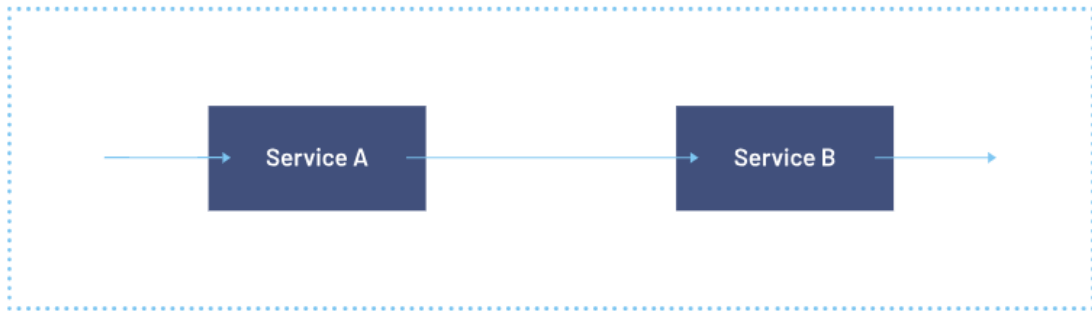
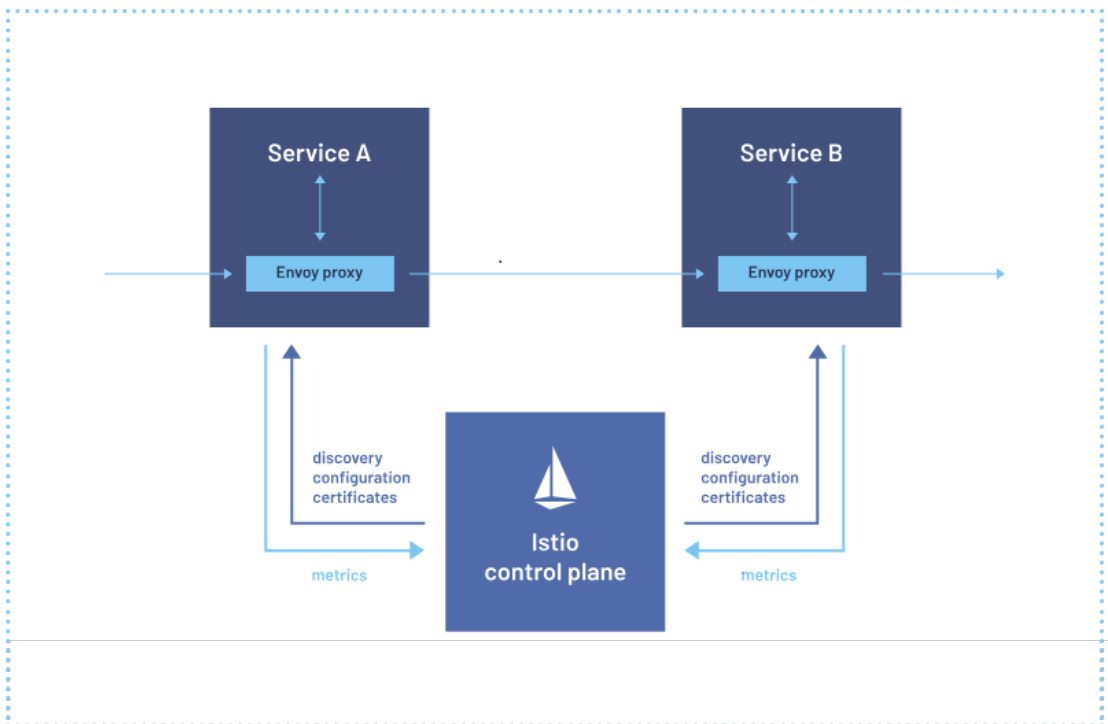
2.6.1 Istio Service Mesh

Istio is an open-source service mesh that layers transparently onto existing distributed applications [26]. Istio Service Mesh is composed of two different components: **control plane** and **data plane**. The *data plane* represents a communication between services. It is needed to track communications within the mesh to comprehend the nature of the traffic being transmitted and to make decisions based on the kind of traffic, its origin, and its destination. The *control plane* instead takes your desired configuration, and its view of the services, and dynamically programs the proxy servers, updating them as the rules or the environment changes.

Proxy servers are essential components of the Istio service mesh. Typically, a proxy server intercepts network traffic to perform certain operations, such as firewalling, before forwarding it to the intended destination. In the context of Istio, proxy servers manage the network traffic produced by the services within the mesh, enforcing the rules specified in the Istio configuration.

Another key component is the so-called *Envoy Proxy*. It is a component that can be injected (manually or automatically) in a Pod during its creation as support for the Pod communication. The *Envoy Proxy* is the component which is responsible to allow the Service-to-Service communication.

The *Figure 2.3* and *Figure 2.4* show the difference between a service communication without and with using the *Istio Service Mesh*. In the former case, the services directly communicate with each other, without performing any action on that traffic, since in the latter case the service communication happens through the *Envoy proxy* which applies the configuration rules defined with *Istio*.

Figure 2.3: Service communication before *Istio Service Mesh*Figure 2.4: Service communication after *Istio Service Mesh* [26]

2.7 Slurm

One of the goals of the platform I'm developing is to allocate and orchestrate a set of physical resources (mainly neuromorphic boards) allowing users to run their algorithms on such boards for testing them.

Slurm is the system used for accounting for the scheduling management job. It is an open-source cluster management system, fault-tolerant and highly scalable, which manages resources on small or large Linux clusters.

It has three key functions:

- **Resource orchestration** - It allocates in an exclusive or non-exclusive way computing nodes and resources to the users for a certain amount of time.
- **Job monitoring** - It provides a framework for starting, executing, and monitoring parallel jobs in a set of allocated nodes.
- **Job queuing** - It orchestrates access to the resources by managing a queue of pending jobs to be allocated.

2.7.1 Slurm architecture

Similarly to Kubernetes, Slurm relies on a distributed architecture divided into control nodes, which orchestrate jobs and resources, and worker nodes, which effectively execute the jobs. Below is an overview of the most important components.

slurmctld

slurmctld is the centralized daemon that orchestrates the jobs to be submitted. It can be scaled to improve reliability and have a backup node in case of failures.

slurmd

slurmd is the daemon installed on compute nodes which waits for incoming jobs, executes those jobs, returns status, and waits for more work. It can be compared to a remote shell providing fault-tolerant and hierarchical communication.

2.7.2 Slurm Job

A *Slurm Job* represents the task being executed within the Slurm system. The definition of Jobs happens in a declarative way by creating a simple bash file (`.sh`) containing the script to be executed and a list of configuration flags describing the HW that must be allocated to execute such a job. With the *sbatch* command, it is possible to submit the defined job to the system, delegating the control nodes to allocate the requested resources.

After the job has been defined, Slurm assigns it a specific *State*, which describes the lifecycle of the job. Possible states are:

- **PENDING**: Job is awaiting resource allocation.
- **RUNNING**: Job currently has an allocation and the worker nodes are executing it.
- **SUSPENDED**: The job has an allocation, but the execution has been suspended and CPUs have been released for other jobs.

- **COMPLETING:** The job is in the process of completing. Its execution is finished and the system is freeing the allocated resources.
- **COMPLETED:** Job has terminated all processes on all nodes with an exit code of zero.

Chapter 3

Materials and methods

In this chapter, I'm going to present how I designed and developed the system. I divided the developing process into two parts: the former related to the design of the system, where I defined the general architecture of the platform, and the latter is the implementation of the component I defined.

In addition, the third section of this chapter describes in detail the steps I made during the development process, highlighting the modifications made along the way.

3.1 Design

This section describes and analyzes the design choices made during the development of the system.

3.1.1 Microservice-based architecture

Nowadays different developing approaches have been used for designing new systems, obeying the new requirements needed for the cloud environment. Monolithic services are no longer used and the micro-services approach took their place.

According to that, in this thesis project, I created an architecture fully based on micro-services. This approach enhances the reliability and maintainability of the system, while also enabling more efficient resource management, thereby optimizing the sizing of the physical servers hosting it.

To implement this kind of architecture, I used Kubernetes, a micro-service and container orchestrator, which allows the creation of an operator-based infrastructure. The *operators* are its core components and they can manage the cluster resources from within the cluster itself.

At the same time, the interaction with the Kubernetes cluster happens through the *Kubernetes APIServer* which offers a collection of endpoints for creating, deleting, and more in general using all the services exposed by the cluster.

3.1.2 Kubernetes as Backend

The core components of the platform are the so-called operators. These kinds of components run indefinitely inside the cluster, taking care of and reacting to the changes in the associated resources by performing some jobs according to the type of events that occurred.

Some already built-in operators manage the default Kubernetes resources such as Pod and Deployments (see section 2.5 for more details), as well as some user-defined operators which are associated with particular user-defined resources called *Custom Resource Definition (CRD)*. This project heavily relies on these custom resources where many of the entities composing the system are mapped as CRDs. The *Figure 3.1* illustrates the operators and CRD belonging to the system. Through the *APIServer* (see section 2.5.2), users can manage these resources by accessing the corresponding REST APIs. This enables them to create, delete, and update those CRDs to trigger the corresponding operator for performing some jobs according to the implemented logic.

Given this architecture, the final user can easily use those services using APIs (called from a client running on the browser), while a developer, who wants to manage those resources from the inside, can use a powerful command line tool called `kubectl`.

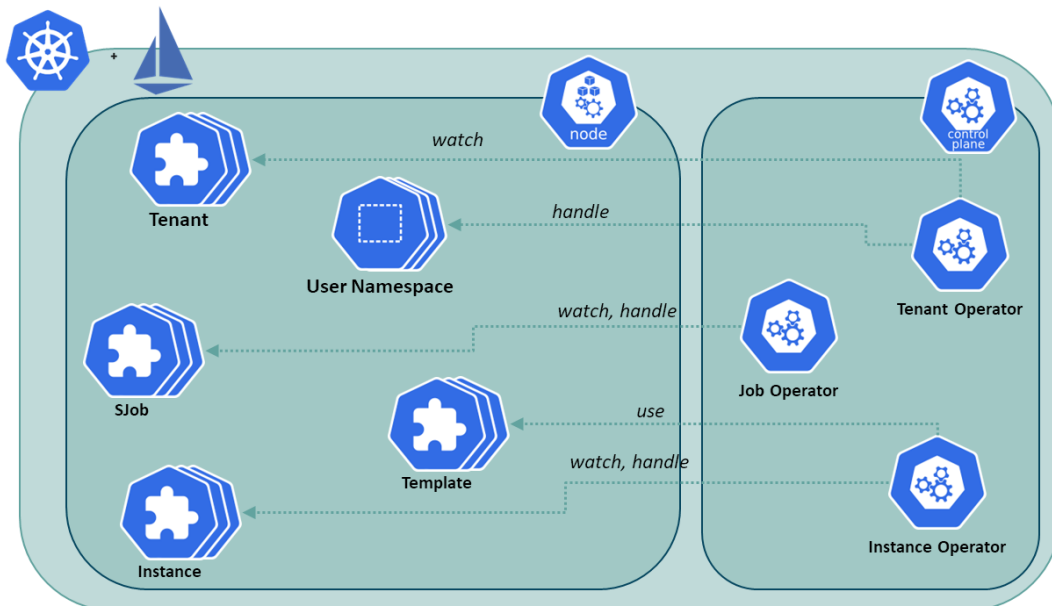


Figure 3.1: CRDs and Kubernetes operators of the cluster[27]

3.1.3 System resources

As explained in the previous paragraph, the platform relies on a set of CRDs that define the business logic behind the system. A brief description follows.

Tenant

A *Tenant* is the resource describing a platform user. It contains all the user information needed for using the system such as ID, name, and the Namespace (see 2.5.1) it refers to. A Tenant resource is created when a user signs up to the platform and the operator responsible for handling its lifecycle is the *Tenant Operator*.

Instances

An *Instance* describes a running environment. It is created following the specification described in a *Template* object and belongs to a given user Namespace.

An Instance resource contains all the useful information regarding the actual state of the deployed object (e.g. a Pod) and its lifecycle is managed by the *Instance Operator*. In the architecture I deployed, users can interact with an Instance using a web interface (see VsCode Instance in section 3.2.1).

Templates

A *Template* is the resource that describes a model of an Instance. It contains the configuration for creating a given Instance such as the image to be used, the services to be created, the environment variable needed for running the application, and so forth.

A Template must be created a priori by the system administrators to be used by users who want to create the related Instances.

SJobs

The *SJob* is the resource describing a Job to be issued to the Slurm cluster. Such resources can be created by users who want to test their algorithms on real HW. Each SJob contains the configuration needed to submit the job to Slurm as well as the commands to launch the algorithm and the algorithm itself as explained in the next section.

3.1.4 Visual Studio Code as developing platform

As described before, this thesis project has a double goal: exposing computational resources to the end-users and allowing them to develop their code on an integrated platform that is connected to those resources and allows users to easily issue their jobs.

There are several IDE (Integrated Development Environment) available on the market but for this thesis project, I chose to implement *Visual Studio Code (VsCode)*. It is one of the most flexible coding environments with several features and is fully customizable. It supports the major programming languages and has several extensions useful for improving the coding experience.

The solution I adopted consists of the containerization of *code-server*, a web-based version of VsCode developed by Coder which exposes the whole VsCode environment through a web interface accessible from a simple browser [28].

The code-server has been customized to allow users to create SJobs to be submitted to Slurm. It is done by a special task defined inside *vscode* which sends requests to the Kubernetes APIServer, calling the proper API for creating such resource.

3.1.5 Istio Service Mesh

Even if Kubernetes allows the orchestration of containers and microservices, it does not provide a powerful tool for: 1) handling the communications between pods; 2) monitoring and analyzing the workload of each service deployed; and 3) observing what happens in the cluster in a more high-level view. This is required when you need to analyze the system to find bottlenecks reducing the performance of the system.

For this platform, I adopted a helpful mechanism for managing the whole system with a high-level view: the *Service Mesh*.

A *Service Mesh* is an additional layer you can add to your system which provides some important features like observability of the system, security, and traffic management without adding additional code to the service already implemented. There are several third-party service mesh providers and for this project, I chose **Istio Service Mesh** (see Section 2.6.1 for more details).

In addition, Istio provides a way to visualize the implemented Service Mesh using a web-based graphical user interface called **Kiali** (*Figure 3.2*). This add-on allows developers to monitor the service communication most effectively.

3.1.6 Slurm as HPC cluster

The main purpose of the project is to export computational resources hosted on-premise (e.g. neuromorphic boards). These boards are organized in a separate cluster from Kubernetes, which is specialized in performing job scheduling, job queuing, and node management in a distributed and highly available system.

This Heterogeneous cluster is powered by Slurm.

Users can develop their algorithm on the platforms made available by the Kubernetes cluster and, through special commands, send their code to the Slurm cluster to be executed by Slurm according to the requirements specified at submission time.

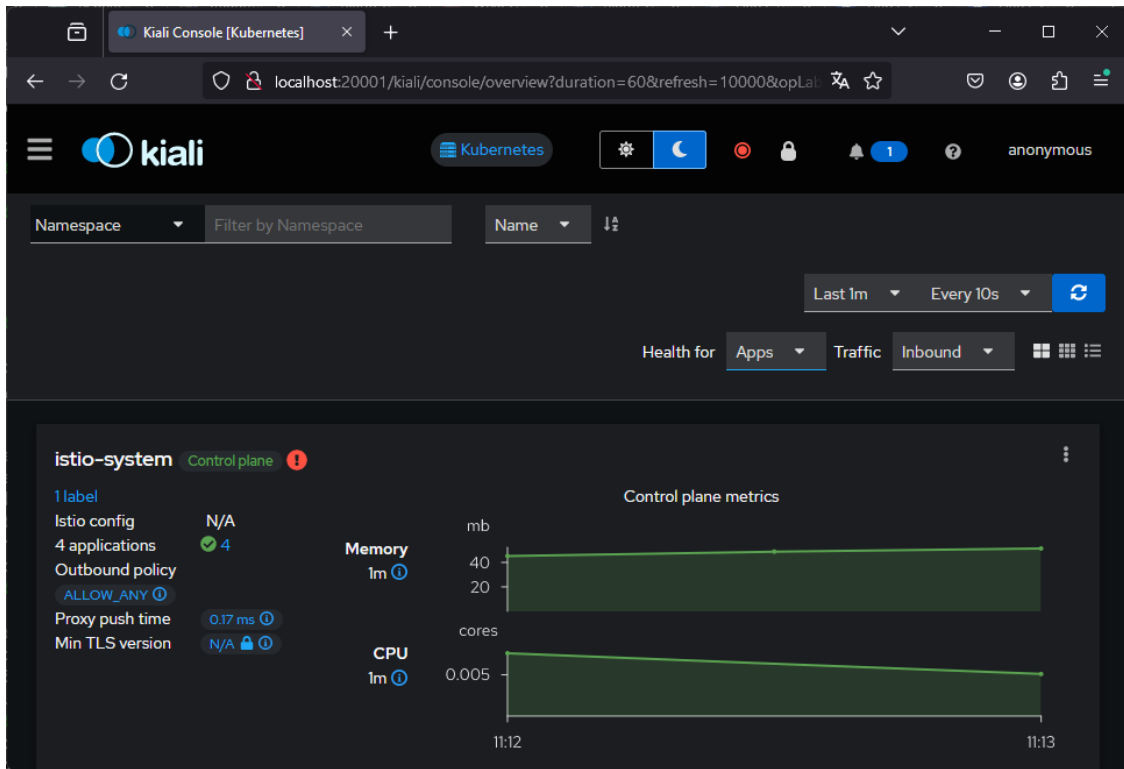


Figure 3.2: Kiali dashboard

The communication between Slurm and Kubernetes happens through some endpoints exposed by a web server running on the *Slurm control plane*. REST APIs exposed by this web server can store the bash file coming from Kubernetes, already configured with the necessary setup, and submit it to the nodes using special commands such as `sbatch` or `srun`. These commands schedule or queue the job if resources are unavailable. The *Figure 3.3* shows an overview of the Slurm cluster I implemented. The master node implements both the Slurm logic (`slurmctld` daemon) and the logic which allows the communication with the Kubernetes cluster by means of the web server. Each worker node, instead, executes the `slurmd` daemon which contains the Slurm logic for the worker nodes.

3.1.7 User developing process and job submission

This paragraph shows the way how users can use this platform and interact with the various services offered by the system.

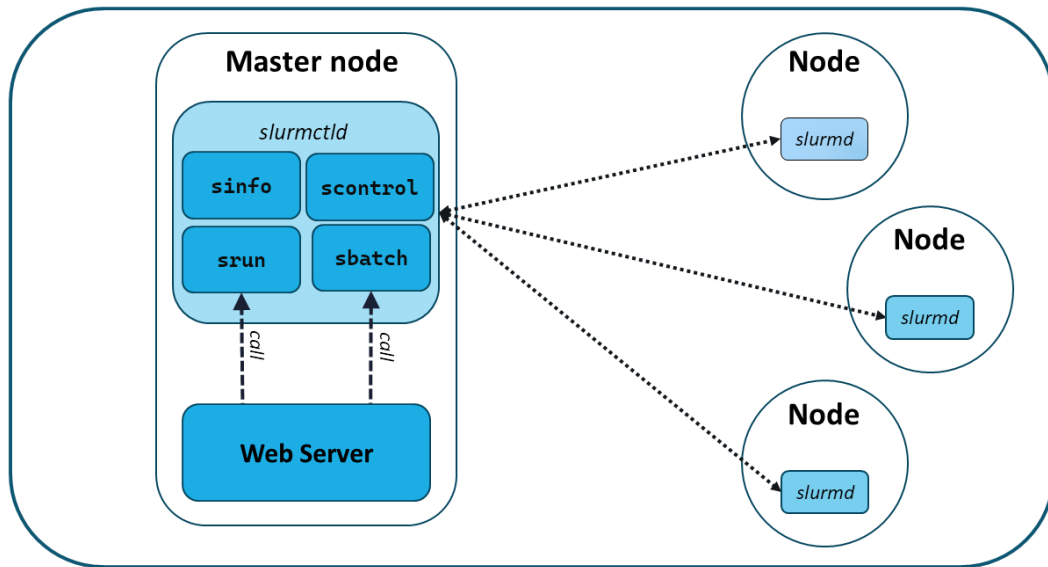


Figure 3.3: Slurm architecture

Coding

As said previously, users are allowed to code in a web-based version of VsCode. They can access the tool directly from the web application by calling the proper API exposed by the APIServer. The APIServer will create a new *Instance* (see 3.1.3), which describes the VsCode configuration, leaving to the Instance Operator the task of creating the VsCode container. The process of the instance creation is depicted in *Figure 3.4*. The instance is created in a user private workspace only accessible by him. Each user is allowed to interact only with its workspace to create projects and store data and files as well as import their custom libraries, tools, and extensions to enhance their development experience.

Job creation

When users want to run their code using some special resources available in the system, they need to create a job by calling the proper *task* directly from VsCode itself. A VsCode task is a user-defined command which can be executed directly from the VsCode environment.

I implemented the task mentioned above as a predefined function within the application, allowing users to execute it most easily. The only duty related to users is to specify which and how many resources they wish to use (e.g. the amount of memory, how many CPUs, which special boards, which GPU).

Job submission

The submission of jobs is handled by the system and it is depicted in *Figure 3.5*. Once jobs are created, the *SJob operator* parses the configuration provided by the user and submits the created bash script to Slurm which is in charge of scheduling and queueing incoming jobs to the physical resources orchestrated.

Results

Once the job is completed, results are provided to the user, together with eventual errors, logs, and statistics about the execution provided by Slurm itself.

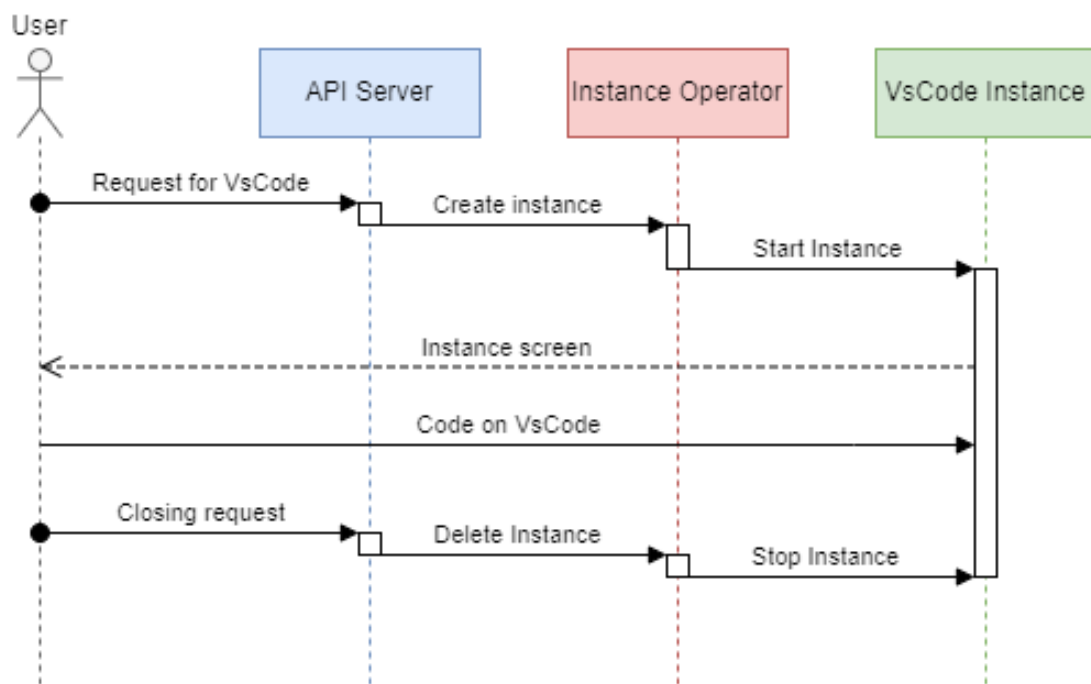


Figure 3.4: User interaction with the coding platform

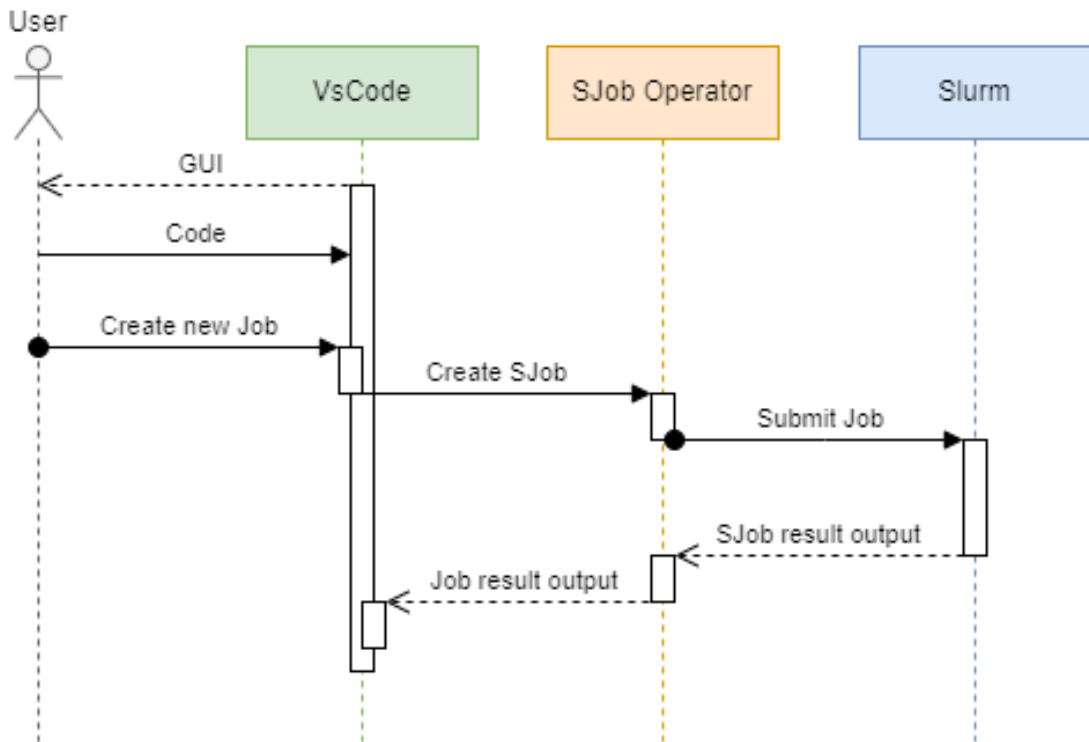


Figure 3.5: Job creation process

3.2 Implementation

This section shows more in detail how the different components have been integrated into the system.

3.2.1 Container-based instances

Since Kubernetes is the main tool used for deploying and creating micro-services, each resource has been deployed as a container-based instance.

Containers nowadays represent the state of the art for the cloud environment since they are easy to create and manage, allowing the implementation of a dynamic system that continuously creates, moves, and destroys containers according to the requirements to be met.

vscode

VsCode represents the environment used for the coding process. I implemented it as a containerized web application by creating the proper image where I installed the web-based version of the IDE provided by Coder Technologies. [28].

As the default programming language I chose Python, a powerful and high-level

language heavily used for the development of Artificial Intelligence (AI) applications. In addition, I integrate with the environment a set of useful Python modules, such as pyTorch, snnTorch, and numPy, needed for the developing process of AI and Neuromorphic algorithms.

Even if these modules can be integrated directly within the image itself containing the Vscod environment, I chose to attach them to the container as an external volume relying on PVC resources of Kubernetes. This choice was driven by the huge size of these packages which caused an increment of the image size, limiting the advantages provided by the containerized approach. In addition, implementing these packages in a separate volume to be attached during the deployments of instances, allow users to choose which versions of these modules they want to use.

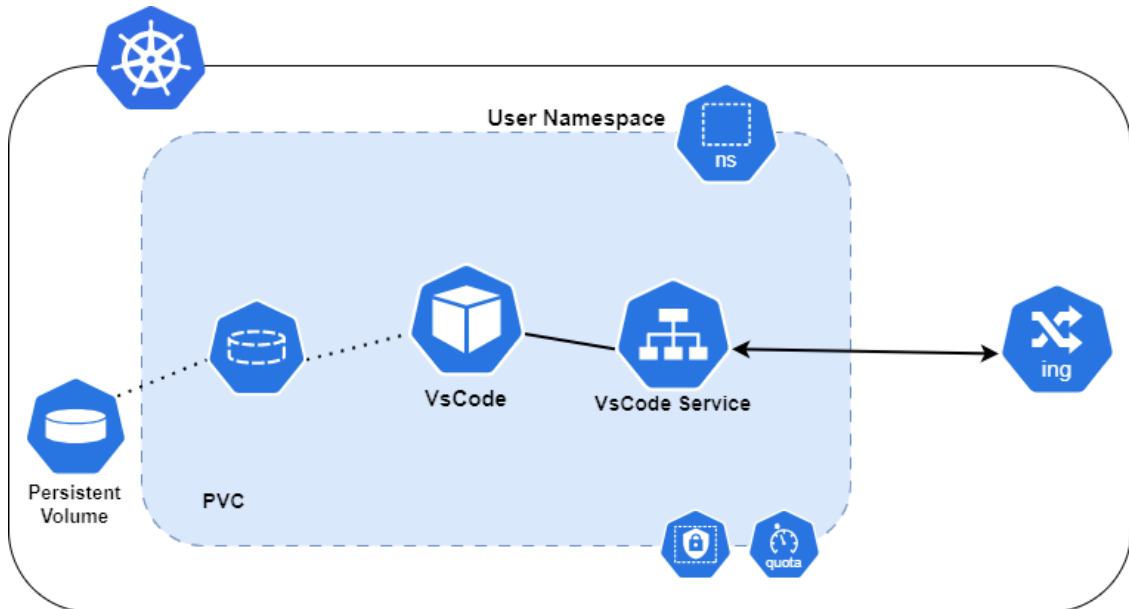


Figure 3.6: User Namespace overview

submitter

The *submitter* is the pod in charge of parsing and sending jobs to Slurm. Slurm exposes a REST API for uploading files and submitting them. What the *submitter* does is to get the Job specification from the SJob resource, parse them by creating the bash file with a proper format of configuration understandable by the Slurm daemon, and call the proper POST API exposed by Slurm, sending the created file.

3.2.2 Infrastructure

Instance operator

The *Instance Operator* is one of the core components of the system. It is the component that reacts to the changes of Instance resources, CRDs which describe the state of the user pods created in the system.

Whenever the user issues a request for creating a Pod (e.g. the user wants VsCode), it creates an Instance object containing all the configuration variables for starting the pod, such as the Template resource the Instance refers to, the user information (provided by the Tenant object), the Namespace where the Instance must be created, PVC to be attached if present and so forth.

Once the Instance has been created, the operator updates the *phase* of the Instance to keep track of the status of the running resource. Every time the phase of an Instance changes, the operator reacts to that change by performing some jobs such as deleting the instance, updating it, or whatever other job has been defined for the operator.

Tenant operator

As well as for the Instance operator, the Tenant operator keeps track of a specific CRD, in this case, a Tenant resource. The tenant operator is responsible for executing various operations, including the creation or deletion of the user Namespace, where all user-related resources will reside. Additionally, it manages user login and logout processes by redirecting these checks to the appropriate pod serving as an authentication server. Furthermore, it associates all the roles linked to the user account with the proper Tenant object.

SJob operator

The SJob operator instead reacts to the changes that happen to the SJob resource, the CRD which describes a user-defined Job to be submitted to Slurm. Whenever a new SJob resource is created, the SJob operator gets the information (configuration) of the SJob and creates a pod, the *submitter* pod described above, for parsing that information, create the bash file with the Job specification for running it and send the created file to the Slurm cluster by calling the proper API.

3.3 Journey

The previous sections of this chapter show the final result of the system I developed for this thesis project. The journey into the development of this distributed system was not so straightforward and in this section, I want to describe the changes I

made to the initially envisioned architecture, leading to the final design.

3.3.1 Architectural choices

Single cluster architecture

The two clusters I developed are based on Kubernetes and Slurm. Even if Kubernetes has been always the first choice to deploy the micro-services, the idea to implement Slurm for managing the job submission to the computing nodes comes in the middle of the development process. In the beginning, my idea was the development of a single cluster running Kubernetes. Within this cluster, I wanted to implement all the business logic needed to expose the IDE (Integrated Development Environment) to the users and provide a way to allocate physical nodes to Kubernetes to run the jobs. The problem with this solution lies in the poor reliability the system could have due to the big complexity introduced for managing such a task. Considering these problems, I figured out that I needed to find a new solution to address them. At this moment Slurm came into play.

Slurm is a tool that allows creating most easily and effectively an *HPC (High Performance Cluster)*, by managing all the computing nodes belonging to the cluster and managing the incoming job to be submitted.

Implementing both Slurm and Kuberntes, creating a *Double cluster architecture*, was the right choice for this project.

Slurm compatibility with the Jetson Boards

Another key problem that I addressed was the compatibility problem between Slurm and the NVIDIA Jetson Nano Boards, implemented as computing nodes.

Installing Slurm on these boards was not so simple since Slurm needs to be compiled for the architecture where it must be executed. NVIDIA Jetson Nano boards rely on an ARM64 architecture, hence it was needed to compile Slurm ad-hoc for such devices.

Due to the limited resources of the Jetson boards, the compiling process took a lot of time and many installation attempts. Additionally, issues arose due to outdated package versions already installed on the boards which needed to be updated.

Istio implementation

As I mentioned in previous sections, Istio is a powerful tool that enables a reliable connection between the various services in the cluster. In the first version of the system, there was no real need to implement such a component since I provided a basic service communication. Problems arose with the connection between the

Kubernetes cluster and the Slurm cluster.

Traffic generated by a Pod running in Kubernetes can only reach services within the same cluster. This limitation implies that establishing connections between different clusters requires additional implementation of components. Istio comes into play for solving this problem. It allows the creation of rules for enabling traffic to flow outside the cluster.

Operator-based cluster

The system I developed highly relies on operators to manage the different events that occur in the cluster such as a new Tenant who wants to log into the system, and the creation of a new Instance rather than a new SJob to be submitted.

Initially, I wanted to manage such interaction through applications running on the system indefinitely. This solution doesn't match with the goals of the Cloud Computing paradigm and does not exploit the potential of the Kubernetes tool.

The implementation of event-driven operators, which react to changes in resources deployed within the Kubernetes cluster, is the optimal solution to adopt for creating a reliable and powerful distributed architecture.

Chapter 4

Results and discussion

The platform created with this thesis project is the beta version of the platform which will be improved in the next future.

As a beta version, the system has been created in *development mode*, without taking into account all the aspects related to a production environment, due to limited resources available for running the system. Despite all, the infrastructure used for testing purposes is as similar as possible to the real one.

4.1 Testing conditions

The system has been developed as two separate clusters, placed in two different IP networks, running *Kubernetes* and *Slurm*.

Kubernetes cluster

The Kubernetes cluster has been hosted on 3 different VMs, virtualized by *Virtualbox*[29], a level 2 Hypervisor that easily allows the creation of VMs by virtualizing the host HW. The cluster for testing purposes contains:

- **1 Control Plane Node** with 4 core and 4 GB of RAM
- **2 Worker Nodes** with 4 core and 4 GB of RAM each

These nodes are under the same IP Network created within Virtualbox.

Slurm cluster

The Slurm cluster has been hosted in a different IP Network than Kubernetes. For testing purposes, it is composed by:

- **1 Control Plane Node** running the *slurmctld* and hosted in a VM created with Virtualbox

- **1 Nvidia Jetson Nano Developer Kit 2GB** with *Quad-core ARM Cortex-A57 MPCore processor, 2GB LPDDR4 Memory, 128-core NVIDIA Maxwell GPU and Gigabit Ethernet port*
- **2 NVIDIA Jetson Nano 4GB**, *Quad-core ARM Cortex-A57 MPCore processor, 4GB LPDDR4 Memory, 128-core NVIDIA Maxwell GPU* carried by *reComputer J101 with Gigabit Ethernet port*

A representation of the two cluster mentioned above is the *Figure 4.1*. The communication between the cluster happen with the HTTP REST protocol as described in the design section 3.1.6.

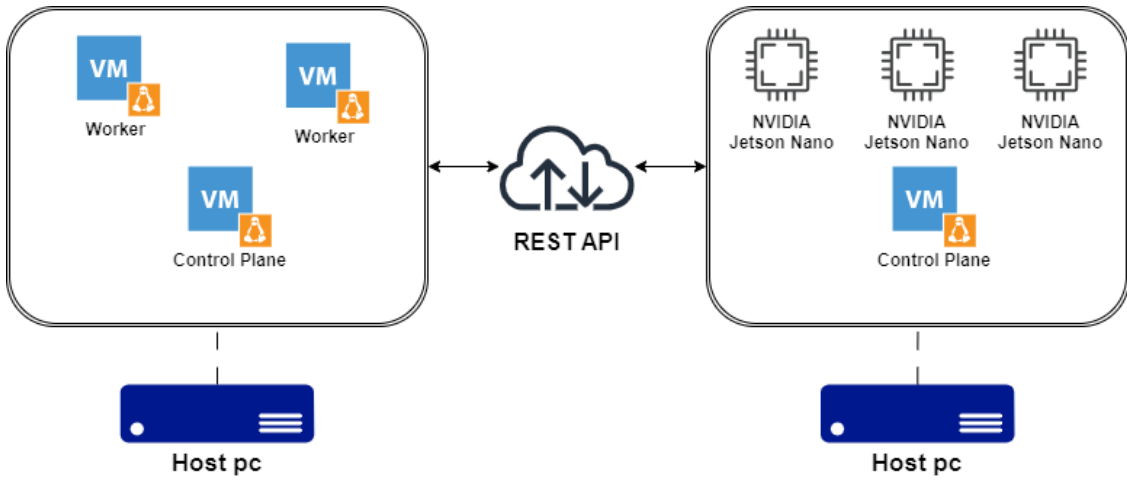


Figure 4.1: The architecture used for testing purposes

4.2 User workflow

The main user interaction with the system is with the integrated developing platform: **VsCode**. As discussed in the previous chapter, I deployed VsCode as a containerized application. Users can consume such service to develop their code directly inside the system.

Once a user requests a VsCode instance, by calling the corresponding REST API, the proper container will be created in the user Namespace which is only accessible to him.

Then, a new tab on the browser is created redirecting the user to the Vscode instance, allowing him to develop the code implementing the desired application. The *Figure 4.2* shows the VsCode instance on the browser. The Graphical User Interface (GUI) is the same of the desktop-based version. On the left there is the user working directory containing the user files and data. This image shows a user

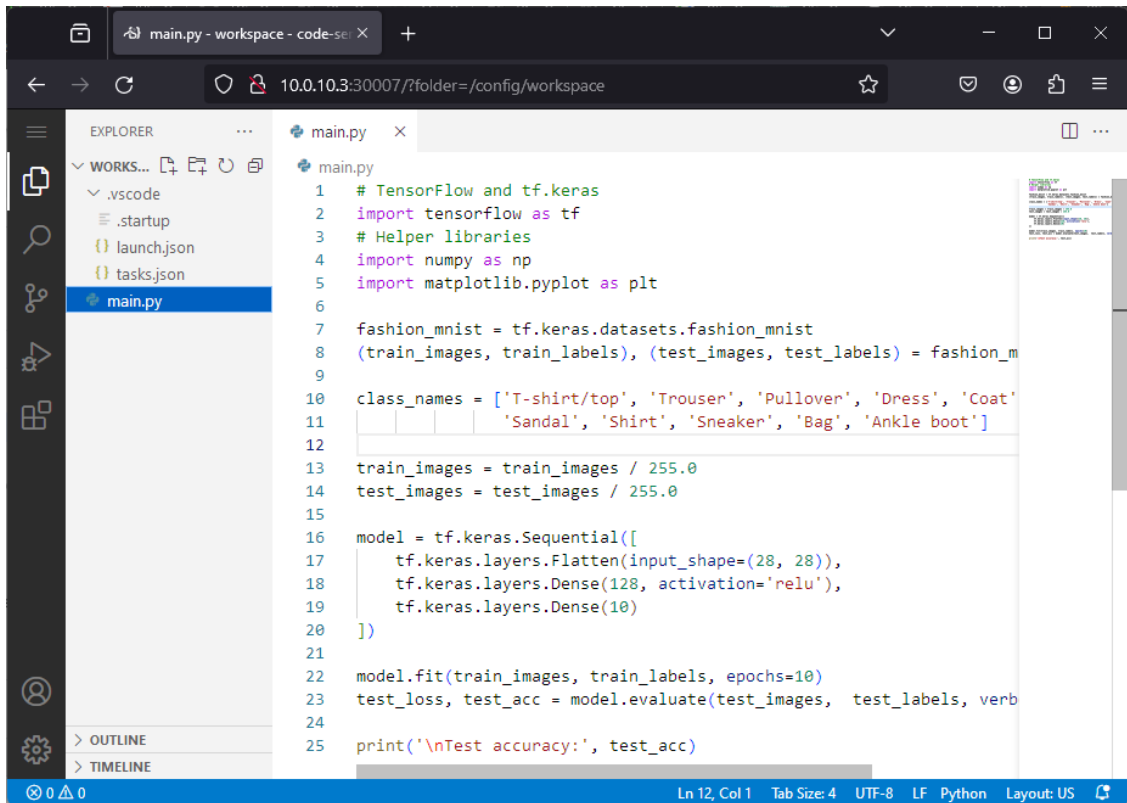


Figure 4.2: VsCode instance accessible by the browser

developing its own Python algorithm. Users are allowed to test their code directly on VsCode by using the built-in terminal. This approach uses the virtual resources assigned to the container at creation time, hence it is suggested to run simple jobs. In addition, users may require to test their code with a physical board having some specific characteristics. In this case, users need to run the predefined *task* in VsCode called *Run job on physical device* as shown in the *Figure 4.3*. The task will issue a Job request to the cluster with all the information needed to use the chosen board.

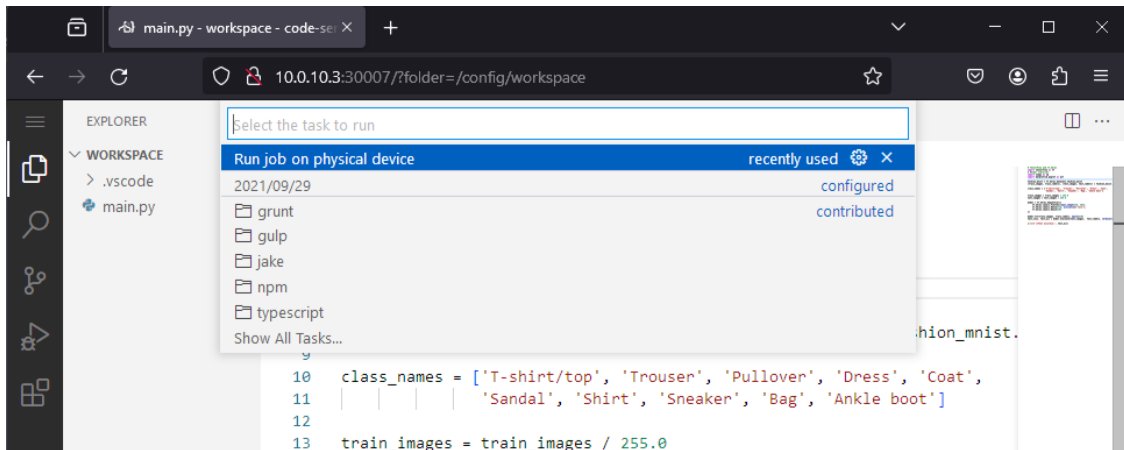


Figure 4.3: A view of the tasks allowing users to run their script to a board hosted by Slurm

4.3 Slurm Testing

After a job reaches the Slurm cluster and the appropriate command for its execution is issued, various scenarios may unfold:

- **Resources are available.** There are available resources for running the job. The Slurm controller allocates the needed resources to the job and executes it.
- **Resources are not available.** The job requests for resources that are not available. This may happen when computing nodes are performing other tasks such as the execution of other jobs. In this case, the incoming job will be queued and executed when resources become available.

The following paragraph describes the submission process and the execution of a test job, illustrating the system’s functionality and the monitoring capabilities available for each job.

Test script overview

The script used for tests is provided by TensorFlow tutorials [30]. It consists of a basic classification of images (clothing images) by using the *Fashion MNIST* dataset which contains 70,000 grayscale images in 10 categories.

Cluster initial state

Before running the test, I’m going to show a brief description of the initial state of the cluster.

By issuing the command `sinfo` it is possible to get the current status of the cluster, highlighting the actual state of the nodes (*DOWN*, *IDLE*, *ALLOCATED*), which partitions are available and which is the current one and other useful info such as the time limit description for each partition.

The *Figure 4.4* shows the output of `sinfo` command issued before starting the test.

```
slurmctld@slurmctld:/storage/example/tensorflow$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
debug*      up       infinite    3    idle jts[01-03]
```

Figure 4.4: Output of `sinfo` command. All the nodes are in *IDLE* state, waiting for new jobs

Job execution

After a user has defined its job on VsCode and the job has been submitted to Slurm via the REST APIs, the web server, hosted on the Slurm control plane, executes the `sbatch` command to run such job.

This paragraph shows what happens to the job after the execution of the `sbatch` command. The job assumes the state of *PENDING*, which means that the Slurm Controller is waiting for resources to be allocated to the job. The *Figure 4.5* shows the output of `squeue` command which prints out all the running and queued jobs.

```
slurmctld@slurmctld:/storage/example/tensorflow$ squeue
      JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
       48      debug  example  slurmctl PD        0:00      1 (None)
```

Figure 4.5: Output of `squeue` command. The status field (*ST*) has the value *PENDING* (*PD*) for the job named `example` and the *NODELIST* field is empty since no nodes have been allocated yet.

Running Job

The job assumes the *RUNNING* state when resources have been allocated for executing the job and it is effectively running.

It is possible to get some information about the running job by issuing the `scontrol show job` command. It prints out all the related information about the job such

as the allocated nodes, the amount of CPU and memory used, and the execution time as shown in the *Figure 4.6*.

It is possible to notice that, for this specific test, the job requested a single node to be allocated (`NumNodes=1`), a single CPU for running the job (`NumCPUs=1`), a single task to be executed (`NumTasks=1`) and just one task assigned to the CPUs allocated (`CPU/Task=1`). In addition, by checking the field `NodeList`, it is possible to notice that the Slurm controller has allocated the node called `jts03` for running the job.

The *Figure 4.7* shows the `htop` command of the allocated node during the job execution, proving that the job has been executed on a single node.

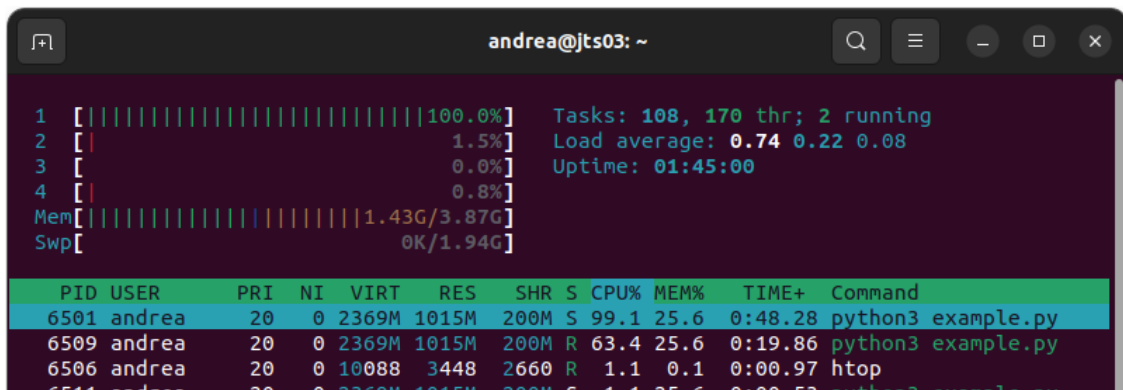
```
slurmctld@slurmctld:/storage/example/tensorflow$ scontrol show job 48
JobId=48 JobName=example
  UserId=slurmctld(1000) GroupId=slurmctld(1000) MCS_label=N/A
  Priority=1 Nice=0 Account=(null) QOS=(null)
  JobState=RUNNING Reason=None Dependency=(null)
  Requeue=1 Restarts=0 BatchFlag=1 Reboot=0 ExitCode=0:0
  RunTime=00:01:11 TimeLimit=365-00:00:00 TimeMin=N/A
  SubmitTime=2024-03-21T11:23:36 EligibleTime=2024-03-21T11:23:36
  AccrueTime=2024-03-21T11:23:36
  StartTime=2024-03-21T11:23:47 EndTime=2025-03-21T11:23:47 Deadline=N/A
  SuspendTime=None SecsPreSuspend=0 LastSchedEval=2024-03-21T11:23:47 Scheduler=Backfill
  Partition=debug AllocNode:Sid=localhost:103178
  ReqNodeList=(null) ExcNodeList=(null)
  NodeList=jts03
  BatchHost=jts03
  NumNodes=1 NumCPUs=1 NumTasks=1 CPUs/Task=1 ReqB:S:C:T=0:0:*:*
  ReqTRES=cpu=1,mem=1M,node=1,billing=1
  AllocTRES=cpu=1,mem=1M,node=1,billing=1
  Socks/Node=* NtasksPerN:B:S:C=1:0:*:* CoreSpec=*
  MinCPUsNode=1 MinMemoryNode=0 MinTmpDiskNode=0
  Features=(null) DelayBoot=00:00:00
  OverSubscribe=OK Contiguous=0 Licenses=(null) Network=(null)
  Command=/storage/example/tensorflow/example.sh
  WorkDir=/storage/example/tensorflow
  StdErr=/storage/example/tensorflow/slurm-48.out
  StdIn=/dev/null
  StdOut=/storage/example/tensorflow/slurm-48.out
  Power=
  TresPerTask=cpu:1
```

Figure 4.6: A view of the command `scontrol show job`

Results

Slurm provides results through an output file created according to the specifications provided during the job configuration. The output file contains the batch script's standard output, hence all the *prints* that the script executes.

The *Figure 4.8* shows the output file of the script executed as a test for this thesis project.



```
andrea@jts03: ~  
1 [|||||||||||||||||||||||||||||||||100.0%] Tasks: 108, 170 thr; 2 running  
2 [||| 1.5%] Load average: 0.74 0.22 0.08  
3 [ 0.0%] Uptime: 01:45:00  
4 [|| 0.8%]  
Mem[|||||||||||||||||1.43G/3.87G]  
Swp[ 0K/1.94G]  


| PID  | USER   | PRI | NI | VIRT  | RES   | SHR  | S | CPU% | MEM% | TIME+   | Command            |
|------|--------|-----|----|-------|-------|------|---|------|------|---------|--------------------|
| 6501 | andrea | 20  | 0  | 2369M | 1015M | 200M | S | 99.1 | 25.6 | 0:48.28 | python3 example.py |
| 6509 | andrea | 20  | 0  | 2369M | 1015M | 200M | R | 63.4 | 25.6 | 0:19.86 | python3 example.py |
| 6506 | andrea | 20  | 0  | 10088 | 3448  | 2660 | R | 1.1  | 0.1  | 0:00.97 | htop               |


```

Figure 4.7: The `htop` command executed on the `jts03` node. The CPU 1 has been allocated for running the Python script named `example.py`

```

slurmctld@slurmctld:/storage/example/tensorflow$ cat slurm-48.out
2024-03-21 12:23:50.115663: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda driver
s on your machine, GPU will not be used.
2024-03-21 12:23:50.741242: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda driver
s on your machine, GPU will not be used.
2024-03-21 12:23:55.353969: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warn
ing: Could not find TensorRT
2.12.0
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-
idx1-ubyte.gz
29515/29515 [=====] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-
idx3-ubyte.gz
26421880/26421880 [=====] - 3s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-i
dx1-ubyte.gz
5148/5148 [=====] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-i
dx3-ubyte.gz
4422102/4422102 [=====] - 0s 0us/step
2024-03-21 12:24:11.558001: E tensorflow/compiler/xla/stream_executor/cuda/cuda_driver.cc:266]
failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-capable device is detected
2024-03-21 12:24:12.003398: W tensorflow/tsl/framework/cpu_allocator_impl.cc:83] Allocation of
188160000 exceeds 10% of free system memory.
Epoch 1/10
1875/1875 [=====] - 21s 10ms/step - loss: 0.4963 - accuracy: 0.8249
Epoch 2/10
1875/1875 [=====] - 19s 10ms/step - loss: 0.3719 - accuracy: 0.8644
Epoch 3/10
1875/1875 [=====] - 19s 10ms/step - loss: 0.3374 - accuracy: 0.8773
Epoch 4/10
1875/1875 [=====] - 19s 10ms/step - loss: 0.3131 - accuracy: 0.8848
Epoch 5/10
1875/1875 [=====] - 19s 10ms/step - loss: 0.2956 - accuracy: 0.8907
Epoch 6/10
1875/1875 [=====] - 18s 10ms/step - loss: 0.2820 - accuracy: 0.8953
Epoch 7/10
1875/1875 [=====] - 18s 10ms/step - loss: 0.2688 - accuracy: 0.9003
Epoch 8/10
1875/1875 [=====] - 18s 10ms/step - loss: 0.2566 - accuracy: 0.9037
Epoch 9/10
1875/1875 [=====] - 18s 10ms/step - loss: 0.2481 - accuracy: 0.9067
Epoch 10/10
1875/1875 [=====] - 18s 10ms/step - loss: 0.2376 - accuracy: 0.9121
313/313 - 2s - loss: 0.3477 - accuracy: 0.8805 - 2s/epoch - 6ms/step

Test accuracy: 0.8805000185966492

```

Figure 4.8: The content of the file `slurm-48.out`. The file contains the prints of the executed script. The filename is the default name provided by Slurm and it can be modified at configuration time.

Chapter 5

Conclusion

The infrastructure developed in this thesis is a key component of the comprehensive research infrastructure that the Politecnico di Torino team is contributing to the Ebrains-Italy project. This contribution aims to establish *Neuromorphic Computing Lab*, a dedicated facility focused on the study, development, and utilization of cutting-edge neuromorphic technologies.

The core focus of the system developed in this thesis lies in the implementation of key components that a prototyping platform must perform. It incorporates Visual Studio Code (VSCode) as a coding platform, enabling users to develop their applications while also providing an integrated Linux terminal for running and testing these applications within the platform itself.

Furthermore, a critical aspect of the system is to facilitate the capability to test and evaluate developed algorithms on dedicated hardware devices specifically designed to address the paradigm of neuromorphic computing. This functionality allows researchers and developers to validate their algorithms on specialized neuromorphic hardware, ensuring compatibility and optimal performance in real-world deployments.

By seamlessly integrating a robust coding environment, testing capabilities, and dedicated neuromorphic hardware resources, this prototyping infrastructure empowers users to explore the full potential of neuromorphic computing. From algorithm development to hardware testing and validation, the system provides a comprehensive framework for advancing the frontiers of this emerging field.

During the development of the system, the most challenging aspect was the integration of Slurm, the tool responsible for allocating physical devices for the execution of user jobs, with Kubernetes, the platform utilized for orchestrating, deploying, and exposing microservices to the end-users. The solution adopted, which involves a REST Web Server running on Slurm, satisfies the requirements of this beta version of the system. However, further investigation and optimization will be necessary to ensure scalability and efficiency under realistic workload scenarios.

5.1 Future works

The developed system is part of a bigger project, hence in the future several enhancements will be introduced to improve both performance and reliability but also to introduce new functionalities that best fit the user needs.

Some of the future works consist of:

- Improve the connection between Slurm and Kubernetes by switching to using the gRPC protocol.
- Implement other useful tools in parallel to VScode, such as NNI (Neural Network Intelligence), a toolkit that helps users automate *Hyperparameter Optimization*, *Neural Architecture Search*, *Model Compression* and *Feature Engineering*.
- Design a user-friendly interface to allow users to interact with the system most effectively.
- Update the system with a method for getting real-time data from running Slurm Jobs.

Acknowledgements

This research is funded by the European Union - NextGenerationEU Project EBRAINS-Italy (IR0000011, CUP B51E22000150006). I would like to express my deepest gratitude to EDA-ECP research group for their support and guidance, especially to Professor Gianvito Urgese and Andrea Pignata for giving me the opportunity to work on this project.

Bibliography

- [1] Microchip. *Prototyping Embedded Systems*. 2024. URL: <https://www.microchip.com/en-us/about/media-center/blog/2020/prototyping-embedded-systems> (visited on Mar. 27, 2024).
- [2] Applied Brain Research. *NengoEdge*. 2024. URL: <https://edge.nengo.ai/> (visited on Mar. 18, 2024).
- [3] Inc Cadence Design Systems. *Protium*. 2024. URL: https://www.cadence.com/en_US/home/tools/system-design-and-verification/emulation-and-prototyping/protium.html (visited on Mar. 18, 2024).
- [4] STMicroelectronics. *STM32Cube.AI Developer Cloud*. 2024. URL: <https://stm32ai-cs.st.com/documentation> (visited on Mar. 11, 2024).
- [5] Politecnico di Torino. *PoliTo for EBRAINS-Italy*. 2024. URL: <https://www.polito.it/ricerca/luoghi/infrastrutture-di-ricerca/ebrains-italy> (visited on Mar. 11, 2024).
- [6] Politecnico di Torino. *CrownLabs*. 2024. URL: <https://crownlabs.polito.it/> (visited on Mar. 7, 2024).
- [7] Cloud Native Computing Foundation. *Kubernetes*. 2024. URL: <http://kubernetes.io> (visited on Feb. 22, 2024).
- [8] SchedMD. *Slurm*. 2024. URL: <https://slurm.schedmd.com/overview.html> (visited on Feb. 22, 2024).
- [9] Giacomo Indiveri. «Neuromorphic engineering». In: *Springer Handbook of Computational Intelligence* (2015), pp. 715–725.
- [10] BrainChip. *Akida Foundation*. 2024. URL: <https://brainchip.com/akida-foundations/> (visited on Mar. 1, 2024).
- [11] General Vision. *NeuroMem chip*. 2024. URL: <https://general-vision.com/ip-and-chips/nm500/> (visited on Mar. 28, 2024).
- [12] Nepes. *Nepes website*. 2024. URL: <https://www.nepes.co.kr/en/> (visited on Mar. 28, 2024).
- [13] SynSense. *SynSense*. 2024. URL: <https://www.synsense.ai/about-us/> (visited on Mar. 18, 2024).

- [14] The Linux Foundation. *pyTorch*. 2024. URL: <https://pytorch.org/features/> (visited on Mar. 1, 2024).
- [15] Jason K. Eshraghian. *snnTorch*. 2024. URL: <https://snntorch.readthedocs.io/en/latest/> (visited on Mar. 1, 2024).
- [16] Esin Yavuz, James Turner, and Thomas Nowotny. «GeNN: a code generation framework for accelerated brain simulations». In: *Scientific reports* 6.1 (2016), p. 18854.
- [17] Tom De Shutter. *Prototyping Server Farm*. 2024. URL: <https://semiengineering.com/prototyping-server-farms/> (visited on Mar. 19, 2024).
- [18] Nvidia. *Nvidia Embedded Systems*. 2024. URL: <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/> (visited on Mar. 28, 2024).
- [19] Peter Mell, Tim Grance, et al. «The NIST definition of cloud computing». In: (2011).
- [20] IBM. *Virtualization*. 2024. URL: <https://www.ibm.com/topics/virtualization> (visited on Feb. 26, 2024).
- [21] Federico Cucinella. «Mass Scale Lightweight Remote Desktop Environments for Educational Purposes». PhD thesis. Politecnico di Torino, 2021.
- [22] Respondus. *Lockdown Browser*. 2024. URL: <https://web.respondus.com/he/lockdownbrowser/> (visited on Mar. 7, 2024).
- [23] Cloud Native Computing Foundation. *Service*. 2024. URL: <https://kubernetes.io/docs/concepts/services-networking/service/> (visited on Feb. 28, 2024).
- [24] Cloud Native Computing Foundation. *ReplicaSet*. 2024. URL: <https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/> (visited on Feb. 28, 2024).
- [25] Cloud Native Computing Foundation. *K8s Architecture Overview*. 2024. URL: <https://kubernetes.io/images/docs/components-of-kubernetes.svg> (visited on Feb. 28, 2024).
- [26] Istio. *Istio Service Mesh*. 2024. URL: <https://istio.io/latest/about/service-mesh/> (visited on Mar. 6, 2024).
- [27] Kubernetes. *Kubernetes icon set*. 2024. URL: <https://github.com/kubernetes/community/tree/master> (visited on Mar. 5, 2024).
- [28] Coder Technologies. *Coder, code-server project*. 2024. URL: <https://coder.com/docs/code-server/latest> (visited on Mar. 6, 2024).
- [29] Oracle. *VirtualBox*. 2024. URL: <https://www.virtualbox.org/> (visited on Mar. 12, 2024).

BIBLIOGRAPHY

- [30] TensorFlow. *Basic classification: Classify images of clothing*. 2024. URL: <https://www.tensorflow.org/tutorials/keras/classification?hl=it> (visited on Mar. 13, 2024).