computers

MDPI

Article

Computational and Communication Infrastructure Challenges for Resilient Cloud Services

Heberth F. Martinez, Oscar H. Mondragon, Helmut A. Rubio and Jack Marquez *

Faculty of Engineering, Universidad Autonoma de Occidente, Cali 760030, Colombia; heberth.martinez@uao.edu.co (H.F.M.); ohmondragon@uao.edu.co (O.H.M.); harubio@uao.edu.co (H.A.R.)
* Correspondence: jdmarquez@uao.edu.co

Abstract: Fault tolerance and the availability of applications, computing infrastructure, and communications systems during unexpected events are critical in cloud environments. The microservices architecture, and the technologies that it uses, should be able to maintain acceptable service levels in the face of adverse circumstances. In this paper, we discuss the challenges faced by cloud infrastructure in relation to providing resilience to applications. Based on this analysis, we present our approach for a software platform based on a microservices architecture, as well as the resilience mechanisms to mitigate the impact of infrastructure failures on the availability of applications. We demonstrate the capacity of our platform to provide resilience to analytics applications, minimizing service interruptions and keeping acceptable response times.

Keywords: resilience mechanisms; fault tolerance; computational platform; kubernetes; microservices

Citation: Martinez, H.F.; Mondragon, O.H.; Rubio, H.A.; Marquez, J. Computational and Communication Infrastructure Challenges for Resilient Cloud Services. *Computers* **2022**, *11*, 118. https://doi.org/10.3390/computers11080118

Academic Editor: Paolo Bellavista

Received: 30 April 2022
Accepted: 25 July 2022
Published: 29 July 2022

1. Introduction

Ensuring services and cloud infrastructure operation in the face of unexpected events is challenging. Modern solutions, such as those based on microservices architectures, which use a simplified application structure and separate them into essential and mostly reusable functions, help to improve application performance, scalability, and fault tolerance.

Technologies such as software containers support service packaging, deployment, and orchestration. These technologies comply with resiliency requirements such as high availability, scalability on demand, load balancing, and traffic limitation. In addition, computing and network infrastructures must be allocated efficiently and dynamically, and must adapt to changing conditions [1]. Network virtualization technologies, distributed storage systems, and resource allocation solutions in heterogeneous environments can help to ensure this.

Resilience at both the network and application levels is typically achieved by applying redundancy mechanisms via hardware and software. Consequently, infrastructure has gradually evolved to include cloud-based, hybrid, and distributed approaches. Advances in cloud computing, containers, virtualization, DevOps, replication, distributed databases, and network technologies have involved moving toward the greater use of software to manage workloads and traffic intelligently, and to provide better support for resilient services [2].

In this work, we identify the challenges we must consider to provide resilience to services and infrastructure in the cloud. We describe the technologies used to guarantee the resilience of applications based on mechanisms that aim to ensure that the services and the computational and network infrastructure are available. This paper makes the following contributions:

- A revision of the communication technologies and computational infrastructures that provide fault-tolerance mechanisms needed to support resilient cloud services.
- A description of the challenges that need to be addressed to design resilient cloud services over computational infrastructures.

- The design and implementation of a resilient platform approach based on a microservices architecture and the mechanisms to provide resilience to infrastructure and services.
- An evaluation of the viability of efficiently leveraging our platform to deploy data analytics applications.

The remainder of this paper is organized as follows. In Section 2 we provide background information on computational and communications mechanisms to improve the resilience of cloud systems. Section 3 describes the resilience challenges faced by cloud infrastructure. In Section 4 we present our approach to a software platform to facilitate the construction of resilient applications. Section 5 describes the mechanisms we designed to ensure resilience in the proposed platform. Section 6 validates the platform through a test scenario. Section 7 discusses related work, and Section 8 concludes the paper.

2. Background

Computational and communication resilience mechanisms leverage various technologies. Important resilience challenges are associated with the interconnection between LANs and WANs, where a wide range of technologies are now available. In addition, computational resilience is accomplished using infrastructure that continues to support the transmission and processing of data under failover scenarios. This section discusses the technologies that can support resilience at both the communication and computational levels.

2.1. Communication Technologies for Resilience

2.1.1. Low-Power Wide-Area Network

A Low-Power Wide-Area Network (LPWAN) is a wireless access technology that provides coverage from a few to tens of kilometers, depending on the environment (urban or rural). LPWANs are optimized for low energy consumption and long range. They are designed to allow for the connectivity of many devices that transmit small amounts of data, mostly via up-links. LPWANs support wireless networks where services require low data transmission rates. There are several different LPWAN technologies, the most popular of which are Long-Range Wide-Area Networks (LoRaWANs), Sigfox, and Narrow-Band Internet of Things (NB-IoT).

LoRaWAN is an open MAC layer standard released by the LoRa Alliance and supported over PHY wireless layer LoRa technology. It was proposed for emergency text-based communication networks [3] and wide coverage emergency location networks in critical environments where 3/4G cellular connectivity is not available [4].

SigfoX is a cellular connectivity proprietary solution for the Internet of Things. It was designed for low-speed communications using ultra-narrow-band technology in the unlicensed sub-1Ghz band. It achieves high-reliability percentages for transmitting narrow-band data over considerable distances in urban areas [5]. It is proving to be an even more robust technology than LoRa [6].

NB-IoT is a 3rd Generation Partnership Project (3GPP) standard. It offers better coverage than 4G in deep indoor or remote areas, although there are trade-offs between the battery life, coverage, and responsiveness [7]. Unlike other LP-WANs, NB-IoT is born conditioned by the LTE architecture and must coexist with this technology without the introduction of modifications to the cellular network structure, and architecture [8].

2.1.2. Self-Organizing Network

A Self-Organizing Network (SON) aims to minimize the costs related to the life cycle of a wireless network by eliminating the need for manual configuration during deployment and operation. The responsibilities of a SON involve automatically planning, configuring, managing, optimizing, and healing the network, increasing the network's reliability, performance, and resilience. The main SON technologies are Mobile Ad hoc Networks (MANETs), Wireless Mesh Networks (WMNs), and 3GPP-long Term Evolution (3GPP-LTE).

A MANET is a type of wireless network in which the nodes are interconnected without a centralized infrastructure and are also free to move randomly, causing corresponding changes in the network topology. Each node of the MANET network behaves like a router as it forwards traffic from other nodes to another specific node in the network. Although MANET routing protocols are primarily used for mobile networks, they can also be useful for stationary node networks that lack infrastructure [9]. Likewise, the concept of a MANET can be successfully applied to many existing wireless technologies, such as satellite and cellular networks, to increase their resilience [10].

In WMN, wireless nodes are arranged in a mesh topology. However, not all of the nodes behave as routers, and some of them are stationary. Only the intermediary nodes between the origin and the destination are routers that forward data and work cooperatively to make decisions on route prediction based on the actual network topology [11]. As proposed in [12], a WMN enables broadband wireless communications in the absence of existing infrastructure, which is ideal for first-response situations during emergencies as it can provide the interoperability, scalability, and performance required in cases where the areas involved are considerable in size and a mesh network topology is the most suitable. 3GPP-LTE, also known as 4G, is an evolution of the Universal Mobile Telecommunications System (UMTS). It implements the SON paradigm, as this is one of the most promising areas for an operator to save Capital Expenditure (CAPEX), Implementation Expenditure (IMPEX), and Operational Expenditure (OPEX), and can simplify network management through self-directed functions (self-planning, self-deployment, self-configuration, self-optimization, and self-healing) [13]. A clear example of SON applications related to resilient mobile networks is autonomous Cell Outage Detection (COD), which is a prerequisite for triggering fully automated self-healing recovery actions after cell outages or network failures [14].

2.1.3. Transport Layer Multi-Homing

Multi-homing connects a host to more than one network, meaning that a multi-homed host requires multiple interfaces with multiple IP addresses. Depending on the destination, it is possible to route data simultaneously through more than one network to increase the connection performance. If a single link fails, packets can still be routed through the remaining links to increase network reliability. Standardized protocols for the TCP/IP model transport layer proposed for multi-homing include Stream Control Transmission Protocol (SCTP) and Multipath Transmission Control Protocol (MPTCP).

SCTP supports data transfer over a network on a single IP or multiple IPs using multi-homing, not for load balancing but for redundancy, thereby enabling transparent failover between redundant network paths [15]. SCTP is a solution that is capable of noticeably improving network robustness to concurrent failures, while achieving delay behavior comparable to traditional TCP, even without modifications to the default protocol settings [16].

MPTCP offers a way to establish communication between hosts rather than between interfaces as in TCP. This protocol allows for redundancy and enables inverse multiplexing of resources, thus increasing TCP throughput through the sum of data traffic from available link-level connections called sub-flows [17]. We can implement load balancing with MPTCP to distribute traffic within networks, maximize reliability and throughput, minimize response time and avoid overloading of the systems [18].

2.1.4. Programmable Network

Programmable Network (PN) technology is a new paradigm in networking that is increasing in popularity as it allows for converting hardware problems into software problems. PN systems have great flexibility because devices and network behavior can be defined entirely by software that runs on general-purpose hardware. In a PN, engineers can re-program the network infrastructure instead of re-building it physically to provide a wide

array of functions or services. Technologies in this category include Software Defined Networks (SDNs), Software Defined Radio (SDR), and Network Function Virtualization (NFV).

SDR refers to radio communications systems where application-specific integrated circuits (ASICs) perform analog radio signal processing. In contrast, baseband digital signal processing is performed by software running on a general purpose processor (GPP) [19]. SDR enables greater resilience of wireless communication links by dynamically programming their runtime properties, such as carrier frequency, modulation type, and packet size, making them less vulnerable to physical layer attacks [20].

SDN is an emerging architecture that centralizes network management by abstracting the data routing process (control plane) from the data forwarding process (data plane). The control plane consists of one or more devices called network controllers, which are considered the brain of the SDN network, while data plane tasks are carried out by SDN switches [21]. This innovative architecture offers the following benefits:

- Lower cost: They typically cost less than their hardware equivalents because they run on off-the-shelf servers rather than expensive single-use devices. In addition, their deployment requires fewer resources since they allow several functions to be executed on a single server. Resource consolidation requires less physical hardware, resulting in lower overhead, power, and footprint costs.
- Greater scalability and flexibility: Network infrastructure virtualization allows network resources to be scaled up or down as needed without adding another piece of proprietary hardware. SDN offers tremendous flexibility that enables the self-service deployment of network resources.
- Simplified management: SDN results in an overall infrastructure that is easier to operate since highly specialized network professionals are not required to manage it.

These benefits mean that SDN-based resilient networks are feasible solutions for fast disaster recovery network services at large scales [22,23].

NFV was proposed by the European Telecommunications Standards Institute (ETSI) as a new network architecture concept that virtualizes network node functions over open computing platforms, formerly carried out using proprietary dedicated hardware technology. NFV improves infrastructure scalability and agility by allowing service providers to deliver new network services and applications on demand without requiring additional hardware resources [24].

NFV can be combined with SDN to create sophisticated network resilience strategies, thus simplifying resource reallocation and minimizing network recovery time. Together, these approaches can offer flexibility in terms of controlling architecture components, allowing for smart usage of the network resources, optimizing intelligent traffic steering, and increasing network reliability in general [25,26].

Resilient systems require higher network flexibility; hence, networks are being softwarized via paradigms such as SDN and NFV. These solutions reduce the demand for specialized network hardware devices by extracting the inherently distributed control plane of forwarding network elements, such as switches and routers, to a logically centralized control plane.

2.2. Computational Infrastructure for Resilience

Ensuring the operation of services, networks, and computing infrastructure in the face of events threatening resilience is challenging. Modern solutions such as those based on microservices architecture [27], which create a simplified structure for applications by separating them into essential and mostly reusable functions, can help to improve application performance, scalability, and fault tolerance. Technologies such as software containers support the packaging, deployment, and orchestration of services, and comply with resiliency requirements such as high availability, scalability on demand, load balancing, and traffic limitations. In addition, the computing and network infrastructure must be allocated efficiently and dynamically and adapt to changing conditions; this is supported

by network virtualization technologies, distributed storage systems, and resource allocation solutions in heterogeneous environments.

2.2.1. Software Containers: Docker

The purpose of using containers is to provide the ability to run multiple processes and applications separately, to make better use of infrastructure while maintaining the security of separate systems [28]. Docker technology uses the Linux kernel and its features, such as cgroups and namespaces, to segregate processes to run independently and safely. Containers such as Docker offer an image-based deployment model, which allows an application or a set of services with their corresponding dependencies to be shared across multiple environments. This approach also automates the implementation of the application (or combined sets of processes that constitute an application) in the container environment, leaving improvement features such as portability, lightness, and self-sufficiency in each application involved.

2.2.2. Container Orchestration

Container orchestration allows clusters to be formed, which enables management functionalities, resource planning, scalability, load balancing, monitoring, and service discovery. The most widely used container orchestration platforms are Docker Swarm [29], Apache Mesos [30], Nomad [31], and Kubernetes [32]. Kubernetes has become the orchestrator par excellence, and the most important cloud providers have integrated it into their platforms through solutions such as Google Kubernetes Engine (GKE), Amazon Elastic Kubernetes Service (EKS), and Azure Kubernetes Services (AKS). Kubernetes allows for managing Linux containers in private, public, and hybrid cloud environments and is used to manage microservice architectures by most cloud service providers [33].

2.2.3. Fault-Tolerant Service Communication

Under the microservices framework, it is common to find applications that are connected through the network, mainly via a cascade architecture. This highlights the need for tools that can constantly monitor the network to mitigate errors or failures at any time. This is where the mechanisms responsible for making decisions in the event of a failure, in terms of actions related to load balancing and service scalability, are most important.

2.2.4. Clusters Management for Resilience

In cloud computing systems, processes are needed to guarantee the stability of the deployed services. These processes usually monitor the status of the resources belonging to a particular cluster. In the case of Kubernetes, these resources can be monitored by a third party if the appropriate connector is created. Two of the best-known tools for performing this task are Rancher [34] and Gardener [35].

Rancher is a tool that allows for the total administration of one or more Kubernetes clusters without discriminating the creation origin. It supports EKS, AKS, and GKE clusters, among others. When creating applications within a cluster managed by Rancher, there is a large selection of pre-configured services of the most frequently used technologies, such as storage, communication, and data processing services. It is currently the management tool of choice for production environments, due to its integration with monitoring services and alert displays for each deployed resource. Rancher can also manage multiple clusters over the network in a single interface, which allows the user to employ different cloud providers and unify application deployments in a single site, and creates an additional layer of security between resources.

Gardener is a project that was developed natively in Kubernetes, which allows for the management of all the resources deployed within Kubernetes via the project's API. Gardener can be seen as a parallel work environment to the main Kubernetes platform, since the installation provides a very similar structure regarding programs and services.

When implementing the service, there are different frameworks that are adaptable to almost any cloud provider.

3. Resilience Challenges

This section describes the challenges that must be considered to provide resilience for services and infrastructure.

3.1. Network Infrastructure

3.1.1. Fault-Tolerant Service Communication

A resilient system should offer fault-tolerant communication mechanisms with the goal of minimizing the impact on the quality of service. To this end, considerations such as limiting the number of requests that can be made during failure periods, time limits on responses to requests, avoiding making requests to services that are not responding for some reason, and limiting the number of concurrent requests to a service should be taken into account.

Replication is also essential for obtaining fault-tolerant services. In active replication, each client request is processed by all the servers. Client requests are assigned to non-faulty servers using a mechanism for coordinating the replies. In passive replication, there is only one server (called the primary) that processes client requests. After processing a request, the primary server updates the state on the other (backup) servers and sends back the response to the client. If the primary server fails, one of the backup servers takes its place. It is a less costly scheme in terms of redundant processing and communication and is, therefore, more prevalent in practice [36].

3.1.2. Network Monitoring

A major function of network monitoring is the early identification of trends and patterns in network traffic and devices. Network administrators use these measures to determine the current state of a network and then implement changes so that the observed condition can be improved [37].

Accurate and efficient network monitoring is essential for network analysis to detect and correct performance issues. Continuous network monitoring can help to identify potential problems before they occur and therefore ensure that the network operates according to the intended behavior. This means that network administrators can proactively solve problems or enable faster failure recovery in order to improve network resilience.

3.1.3. Network Assessment

The main factors impacting network performance are network latency, congestion, infrastructure parameters (QoS, filtering, routing), and network node health. Network evaluation involves reviewing and analyzing network statistics represented by these factors. This qualitative and quantitative process that measures and defines the performance level of a given network allows administrators to optimize the network service quality with subsequent decisions [38].

Network assessment is an effective method for identifying performance gaps, enhancement opportunities, and network functionality. The information gathered during a network assessment can assist administrators in making critical IT infrastructure decisions.

3.1.4. Fault-Tolerant Networking

The core components for improving fault tolerance include diversity, where some diverse fault-tolerance options result in the backup not having the same capacity level as the primary component. The redundancy can be imposed at a system level, which means an entire alternate computer system is in place in case a failure occurs. Replication involves multiple identical versions of systems and subsystems and ensures their functions always provide similar results.

In fault-tolerant networking, the network must adapt to continue operating without interruption in the case of failures that involve a loss of connection between nodes or their shutdown. It can be addressed with two different approaches: one is to consider making nodes more robust to failures. The other is to design a network topology with redundant intermediate nodes and connections [39].

3.1.5. Data Flow Management

The practice of capturing and analyzing network data traffic and routing it to the most appropriate resources based on priorities is known as data flow management.

When considerable volumes of data are handled in the network, traffic-prioritization mechanisms allow for efficient bandwidth management and better distribution of node processing loads, thus preventing vital information from being lost in transit due to network saturation, either due to increased traffic or node loss [40].

3.2. Computational Infrastructure

3.2.1. Performance Isolation

Technologies should be used that provide the necessary mechanisms to package microservices and mitigate the performance degradation resulting from their concurrent execution on shared computational resources. In view of this, developers should use virtualization solutions or software containers. Candidate technologies for microservices packaging include Docker (software containers) [41], Linux LXD Containers (Containers for OS Partitioning) [42], Singularity Containers (containers for scientific workflows) [43], Shifter Containers (containers for HPC applications) [44], and KVM or Xen Virtual Machines (lightweight virtual machines) [45,46].

3.2.2. Provision of Lightweight and Loosely Coupled Microservices

Loosely coupled, lightweight services must be provided that can be reused for the construction of different applications. These microservices come in packages (e.g., virtual machines and containers) with all the system libraries and dependencies they need to run. They can be selected from public repositories, which facilitates the composition of services in a fast and flexible way. Many of the tools and technologies for containerization and virtualization provide public repositories. Examples of these are Vagrant Cloud [47], Image Server for LXC/LXD [48], Docker Hub [49], Singularity Hub [50].

3.2.3. Portability and Integration with Version Control Systems

The packages must be easily exportable between different computing infrastructures. In the same way, creating different versions of the packages must be allowed. For this purpose, we can use repositories based on GIT or Apache Subversion (SVN). Such repositories could contain, among others, provisioning files (e.g., Dockerfile, Vagrantfile, Manifest files) and images of the packages.

3.2.4. Live Migration and Rolling Updates

The system must provide the capacity for live migrating containers to other physical nodes, along with service rolling updates. The system must provide the functionality needed to update the services and their configuration parameters without affecting the service availability. This is one of the capabilities that support fault tolerance [51].

3.2.5. Redundancy and Load Balancing

The services to be implemented should be redundant as a means of improving fault tolerance. When the platform is faced with high traffic demand, the system must distribute the load among the different containers using various balancing algorithms, such as round-robin; for fairly load distribution; least connection, where requests are assigned to the servers with the least active connections; and by source, where requests are distributed

based on the source IP address. To achieve this, implementing a system with a frontend that receives requests and multiple backends to fulfill them is suggested.

3.2.6. Performance Monitoring and Scalability

Based on the demand for a service, the system must allow for horizontal scaling (i.e., adding more service instances). Horizontal scaling can be done manually or through the use of autoscaling [52], which requires monitoring of the instances (health check) and the resources they use (i.e., CPU, memory, bandwidth) to make real-time decisions about the resources assigned to each microservice. The creation of decision-making systems for scalability can be as simple as defining rules triggered depending on readings of resource consumption, or may involve complex solutions that rely on artificial intelligence algorithms [52,53].

3.2.7. Distributed Storage

For large volumes of information, the system must supply mechanisms for efficient access to the data. The infrastructure should provide distributed file systems with redundant storage mechanisms and fault tolerance. We must consider the hierarchy of storage levels offered by these systems (i.e., RAMDISK, SSD, HDD, external storage) to provide intelligent data storage techniques. Previous works [54,55] leverage this hierarchical storage infrastructure and propose intelligent approaches to optimize data placement on modern distributed file systems, such as HDFS (Hadoop Distributed File System) used to allocate big data applications.

3.2.8. Efficient Planning and Resource Allocation Systems

Mapping containers to physical nodes is a complex problem, in which the aim is to maximize the performance of the applications (i.e., their execution time) while taking into account factors such as interference reduction (i.e., performance isolation), maximization of the use of computational resources, use of bandwidth, and reduction of power consumption. This requires the implementation of intelligent resource allocation algorithms. In the case of cooperative microservices, efficient resource allocation mechanisms must be implemented that consider the locations of the data and reduce the communication time between services.

3.2.9. Standardized Service Communication Interfaces

The use of lightweight service communication interfaces based on standardized protocols such as HTTP for service invocation is recommended. The use of application programming interfaces such as those supported in the REST (Representational State Transfer architecture) favors the efficient exchange of messages under high load conditions. Modern microservices-based architectures use this type of communication interface. The Web services community is increasingly using REST in place of heavier and more complicated approaches such as SOAP (Simple Object Access Protocol) [56].

3.2.10. Interoperability with Multiple Cloud Providers (Multicloud)

Resilient systems must allow for connectivity and interaction between the services deployed by different cloud providers while complying with quality of service standards. There is a trend to build services leveraging microservices-based architectures in a multicloud environment where microservices solutions have to be portable and interoperable, facilitating integration across independent cloud providers and preventing vendor-lock issues generated for proprietary services [57].

4. A Resilient Platform Approach

We implemented a software platform (see Figure 1) to leverage the construction of resilient applications based on the microservices architecture defined in [58]. This platform offers developers and users the mechanisms needed to build and use resilient services through standardized interfaces. With this in mind, our platform is deployed on a Ku-

bernetes [59] orchestration cluster, which provides auto-scaling, load balancing, service discovery, and self-healing, among other features. Our resilient service platform consists of the following components:

- API gateway node: This contains a service that allows the platform to receive data from external sources, such as user devices or access gateways. JSON was chosen as the data representation format in order to provide a standardized mechanism to send data to the platform
- Master node: This contains the control plane components needed to manage the Kubernetes cluster. It provides the Kubernetes API and allows for scheduling and cluster data management.
- Worker nodes: These host the pods, which group the containers and other resources used by applications. The master node manages the pods and worker nodes
- Access gateway: This allows IoT nodes to send data to the platform. For this work, we used a LoRaWAN gateway.
- Broker: This implements a publish/subscribe communication mechanism between the platform microservices. This functionality was implemented using Zookeeper and Kafka frameworks.
- ResCity Database: This provides storage services to the platform microservices.

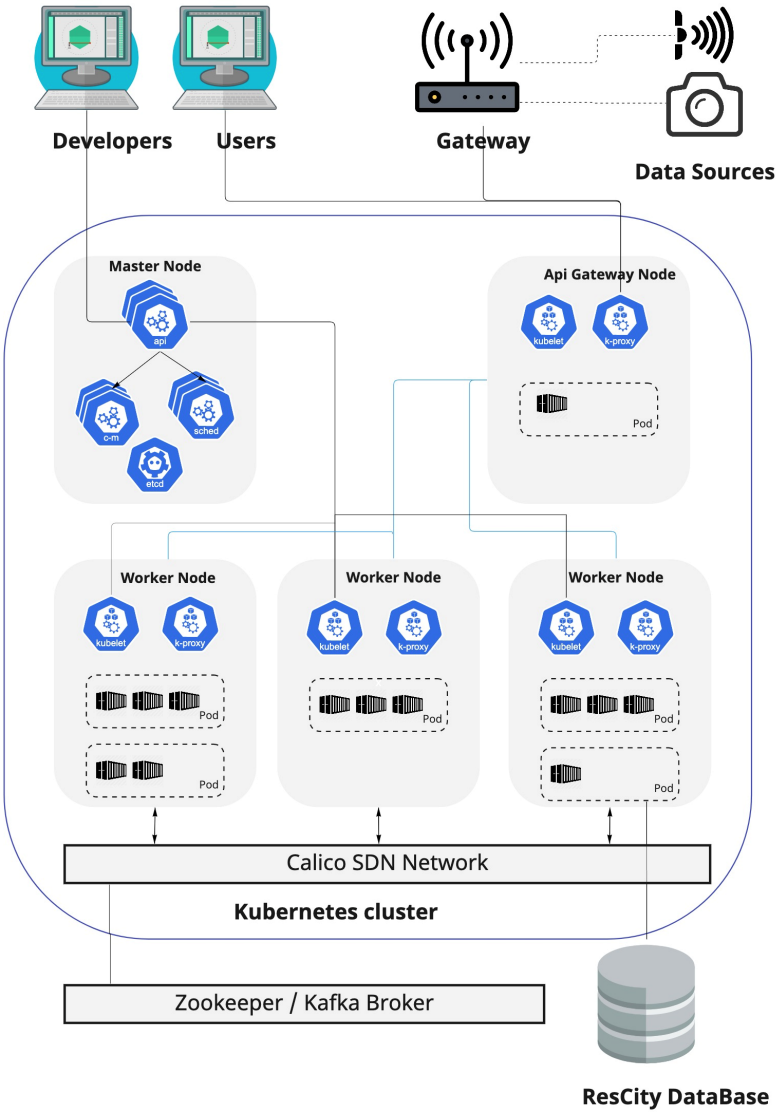

Figure 1. Platform for provisioning resilient services.

5. Platform Resilience Mechanisms

In order to provide resilience to the applications and infrastructure, we incorporated into our platform a number of capabilities, including fault-tolerant network and service communication, access control, and scalability mechanisms. We describe these mechanisms below.

5.1. Traffic Management and Fault Tolerance

We propose several mechanisms to deal with traffic management and fault tolerance problems in resilient systems and evaluate them on our platform, which is described in Section 4.

5.1.1. Network Infrastructure Adaptation and Fault Tolerance

When discussing fault tolerance and traffic management, there are several factors to consider, such as high availability, load balancing, and the definition of alternatives to guarantee that the deployed services do not suffer from intermittency. Kubernetes already incorporates mechanisms to address these aspects in the network. Similarly, it includes external tools for the deployment of a complete SDN network. In this way, we can integrate tools within resilient services to provide high availability. The use of SDN makes it possible to provide elasticity to traditional network models and to address failure scenarios more efficiently. Since the network architecture must allow traffic to be redirected from one host to another in the case of failure, a mesh infrastructure is proposed.

A stable network with the SDN paradigm guarantees that all the containers of the microservices in Kubernetes are connected. Kubernetes has compatible tools such as Flannel and Calico that are responsible for solving this problem. Various aspects should be evaluated to decide which tool to use, such as customization of the network configuration parameters. Calico supports the provision of additional rules that allow us to manage the network flow between the containers, its compatibility with future versions of Kubernetes, and the existence of active forums that can support the tool development team.

To include Calico in resilient services, it is necessary to deploy a group of resources within Kubernetes, which deploys a series of containers and services that allow for constant network monitoring. This process is done through a YAML configuration file, which is available from the official Calico website. In Calico, the design of network policies is based on the construction of YAML files which contain instructions on what to do in the event of network abnormalities or other situations such as high demand or blockages by the destination microservice. The configuration file in Listing 1 shows an example of how to configure a network policy that allows for the transfer of data between containers with a specific label.

Listing 1. Calico network policy.

```
apiVersion: projectcalico.org/v3
kind: NetworkPolicy
metadata:
  name: allow-tcp-6379
  namespace: production
spec:
  selector: color == 'red'
  ingress:
  - action: Allow
    protocol: TCP
    source:
      selector: color == 'blue'
    destination:
      ports:
        - 6379
```

5.1.2. Fault-Tolerant Services Communication

Kubernetes allows the infrastructure to adapt to failures and high demand for services. Unfortunately, keeping track of each of the applications deployed within the containers is a difficult task, and external tools can help to solve this problem. Fault tolerance libraries such as Resilience4J [60] include different patterns that are capable of monitoring and making decisions in scenarios that affect the resilience of applications. On the other hand, we incorporated Kafka in our platform which implements a publish/subscribe mechanism between microservices and natively integrates fault-tolerance, replicating Kafka topics and providing high availability.

5.2. Perception, Communication, and Distributed Computing

The computational and communications mechanisms that support the transport and processing of data from multiple sources are described below.

5.2.1. Microservices Interconnection

Interconnecting microservices is an important task. Since we can deploy any service in a resilient platform regardless of the programming language, we establish the following rules for their interconnection.

- Access to the platform: An access control mechanism must be designed to control all requests to a resilient platform.
- Distribution of services: In view of the variety of services that can be hosted on a resilient platform, we must devise a way to group the set of services that make up the microservice, based on the resources deployed.
- Naming service: Each of the resources that can be deployed in Kubernetes requires access to the network, which means there will be an IP for each resource. For this reason, it is important to design a DNS service that can manage names, in order to avoid the problem arising from the use of dynamic IPs by Kubernetes.
- Availability of resources: The platform must have the capacity to grow in terms of the available resources, so that if more services need to be deployed in the future, this will not be an impediment.
- The platform must provide mechanisms to facilitate the access to data sources.

Various components were incorporated into the platform based on the proposed rules for interconnection of the microservices.

API Gateway

The use of a single gateway for all services allows us to control each path that the data can take. In a resilient platform based on services, the API gateway must allow interaction with all of them when required. This offers many advantages for the management of these services. When centralizing the information, we must devise strategies aimed at a single resource to scale, maintain and manage the tool.

Namespaces

Namespaces allow a set of resources to be grouped in Kubernetes. Each microservice can be separated into small groups of resources containing everything necessary to deploy an application. This offers advantages for maintenance of the platform, as it is possible to modify the resource configurations without affecting external services. On the other hand, being isolated also separates the network segment to which the resources belong, thus avoiding the need for services to share information. To prevent this issue, it is necessary to set internal DNS policies for the set of Kubernetes resources.

DNS

To install a DNS service that is capable of maintaining connections between all the deployed microservices, coreDNS is used. This is an open-source tool that allows a flexible

DNS service to be deployed between the containers in Kubernetes. It offers a basic configuration that allows us to link all Kubernetes namespaces and interconnect those of interest to the application.

Data Access Controller

A data access controller provides the mechanisms to allow microservices the access to data by incorporating authorization and security checks.

5.2.2. Scalability

Another important factor to take into account is the high level of availability of services. To meet this requirement, various solutions are proposed.

Load Balancer

Kubernetes containers can be treated as replicable elements to meet the demand for services. The replica factor or ReplicaSet allows the instances to be multiplied in a fixed way for a given set of containers. We can do this operation manually when deploying the resources through the replicas parameter.

Horizontal Pod Autoscaler (HPA)

This tool allows us to automatically create service instances, to adapt to CPU consumption and other system metrics. As the number of containers multiplies, the need for a load balancer arises in order to distribute the requests to each. Kubernetes uses round-robin as the default load balancing algorithm, based on the generation of shifts and information queues to serve the requests in each instance. It is important to remember that sufficient resources must be available to carry out deployment on the platform nodes when multiplying the instances.

Metrics Collection

In order to efficiently support the scalability and distribution of tasks, the platform collects key service and infrastructure metrics, leveraging the Prometheus [61] framework integrated to Grafana [62] for metrics visualization.

6. Results

To test our platform and its resilience mechanisms, we designed an application that uses input data received from multiple cameras. The system receives as input images of the faces of people and detects the proper use of face masks. We deployed this application on our platform to supervise compliance with health measures implemented in public places in a city, working as a tool to mitigate infections with the new coronavirus (COVID-19).

6.1. Scenario

A service was designed to monitor and notify users when COVID-19 protective measures were not respected. Video cameras work as the data sources to be processed on our platform, as depicted in Figure 1. The intent is to use this system in crowded places, to detect the proper use of facial masks and keep a record, helping to enforce biosafety regulations. We simulated a camera installed at the exit of the laboratory facilities of a university comprised of 30 laboratories with a capacity of 30 students each. In rush time, approximately 1000 students are expected to exit the laboratories in a period of 5 min. Figure 2 shows the implementation made on the platform.

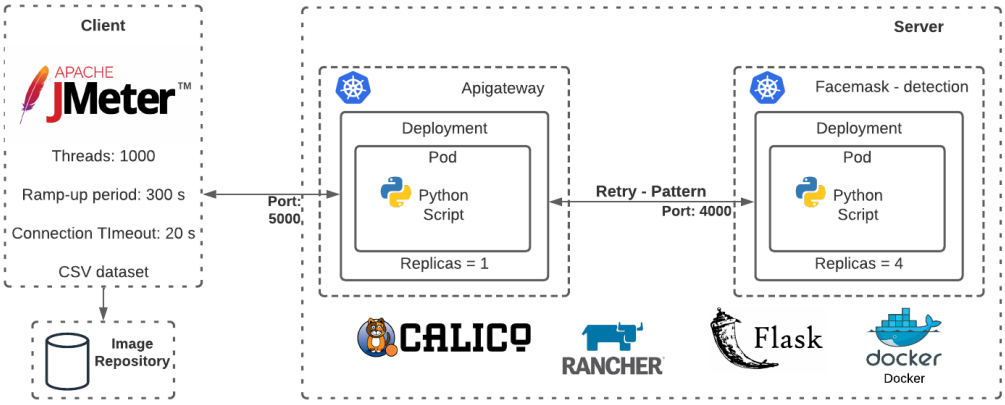

Figure 2. Test scenario.

6.2. Experimental Setup

To run our experiments, we used a two-node Kubernetes cluster. One node was configured as the master node and the other as a worker. Each node has 64GB of RAM, an Intel Xeon CPU E5-2650 v4 with 24 cores, 2TB of HDD, and Ubuntu 18.04 as the operating system. For our deployment, we used Kubernetes 1.17, Docker 1.19.3, and Python 3.7.

We defined a number of requirements for each implemented Python script that the Docker container needs. Listing 2 shows a list of requirements for each microservice, and Listing 3 shows the basic configuration in a Dockerfile to create the container for the facemask service. The Dockerfile for the Apigateway is similar, only changing the exposed port to 5000.

Listing 2. Requirements for Python applications.

```
opencv-contrib-python==4.5.4.58
numpy==1.21.5
requests==2.18.4
click==8.0.4
Flask==2.0.3
itsdangerous==2.1.2
Jinja2==3.1.1
MarkupSafe==2.1.1
Werkzeug==2.0.3
```

Listing 3. Dockerfile for the facemask service.

```
FROM python:3.7
LABEL maintainer="Heberth Martinez"
RUN apt-get update
RUN apt install libgl1-mesa-glx -y
RUN mkdir /app
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
EXPOSE 4000
```

6.3. Deployment

Our microservices-based architecture uses Kubernetes for container orchestration. Kubernetes allows the separation of developed applications into small groups of containers and provides tools to enable management and making decisions during application failures.

A resource group is defined during service deployment in a Kubernetes cluster, allowing deployed containers to be monitored. As shown in Figure 2, we deployed our application using two Kubernetes services. One service runs an ApiGateway that receives client requests and routes them to a service running the facemask detection algorithm. Each service routes the traffic to a set of pods, which are "logical" host grouping Docker containers. Pods can be replicated inside a single multicore server or across multiple servers. We implemented a Retry pattern in the ApiGateway service configured to retry a request to the facemask service a maximum of 10 times with a period of 10 s between retries in case of request timeout the internal logic described in Listing 4. Our platform leverages the implemented resilience mechanisms to scale the number of pods according to the demand. We use CPU utilization as the metric to decide the number of active pod replicas. For our application, an acceptable maximum response time for a request is 20 s.

Listing 4. Apigateway pseudocode.

```
start
    while app is runing
        init Flask app
        define route "facemask"
            receive data from the client
            send request to facemask service on port 4000
            response data to client with status code 200 if not exist
                send message error "error" and status code 500
        define route "facemask-retry"
            implement retry pattern
            receive data from the client
            send request to facemask service on port 4000
            check response and retry request 10 times each 10 seconds if not is valid
            response data to client with status code 200 if not exist
                send message error "error" and status code 500
        expose port 5000
end
```

For the facemask detection service, we implemented a Python Flask application that receives the request from the client with a JSON body that contains the image URL to process. The service downloads the image and processes it with the help of a neural network, and returns a JSON with objects detected information. Listing 5 shows the internal logic of the service.

Listing 5. Facemask pseudocode.

```
start
    while app is runing
        init Flask app
        define route "detect"
            receive data from the client
            download image from url send by client
            load image with the help of opencv module
            load neural network with the help of opencv module
            load configuration file and labels file with the help of opencv module
            process image with neural network
            create JSON response with all object detected, confidence value and label detected
            send JSON response to client with status code 200 if not exist
                send message error "error" and status code 500
        expose port 4000
end
```

6.4. Test Cases

We designed three test cases to validate the fault tolerance capability of our platform. We used JMeter tool [63] to simulate users concurrently sending requests to the service. We configured Jmeter with 1000 threads (users) and a specific ramp-up period for each case. Each thread pulls an image from a 100-image repository and sends the request to the API gateway service, which is routed to the facemask service. We used JMeter to simulate users concurrently sending requests to the service. The purpose of each test case and the expected results are described below.

- Case 1: The ability of the system to keep instances alive in the face of malfunctions. One of the characteristics of a fault-tolerant system is to launch new instances or restart existing ones in case of failure. In this experiment, we sought to validate the capacity of our platform to maintain deployed resources in the presence of development-level and architecture-level failures. For this case, we used a Jmeter ramp-up period of 5 min and set the number of replicas for the facemask deployment to four. As shown in Figure 3, Kubernetes Replicasets help to keep a specified number of pods alive. In the first test, we deleted the pods of the facemask deployment. In this case, Kubernetes took care of making the service available again and restarted the pods in approximately 4 s. We ran the test with and without using the retry pattern. We got 0.5% of requests with errors when not using the retry pattern and no errors when using it.

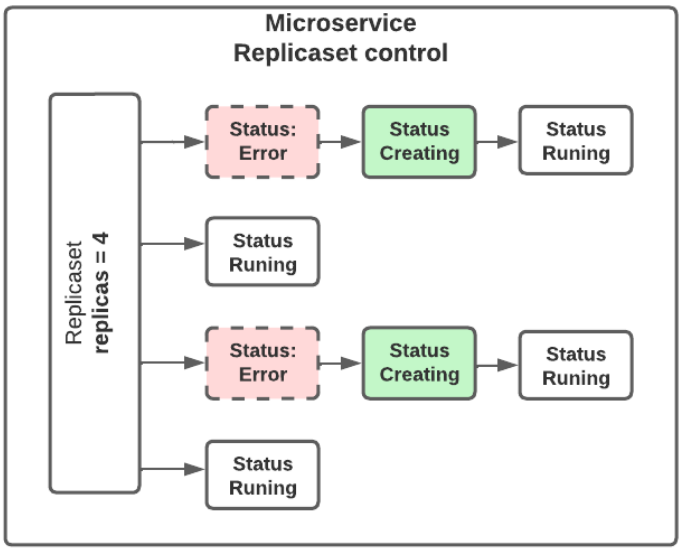

Figure 3. Microservice ReplicaSet control.

In the second test, we ran the experiments in the presence or absence of failures and using or not the retry pattern. In the case of failures, we injected a 20 s service interruption and got 10% of requests with errors when not using the retry pattern. We should note that the retry pattern allows us to keep the error percentage at 0 percent. However, as expected, the maximum response time increases during the failure. Table 1 shows the request response times and the percentage of error in each of the evaluated cases.

Table 1. Performance of the facemask service with 1000 users and induced failures.

Case	Average(s)	Min(s)	Max(s)	%Error
No retry pattern and no induced failures	1.6	1.1	2.1	0
Retry pattern and no induced failures	1.7	1.2	2.3	0
No retry pattern and induced failures	1.1	0.5	2	10
Retry Pattern and induced failures	10.2	1.5	17.4	0

- Case 2: The ability of the system to create new instances in the presence of many different requests. We assessed the ability to balance the resource load to ensure that the system responded successfully to demands from multiple users. In this case, we configured load balancing policies to allow the number of active instances to be increased or decreased by considering the number of user requests. To achieve this, we used the HPA Kubernetes services, which allowed the platform to respond to a high volume of requests and improved the response times and the number of simultaneous requests it could process. HPA will enable requests to be evenly distributed across replicas, defining the total instances and the metric to be considered for tuning on the platform. Figure 4 shows how the HPA works.

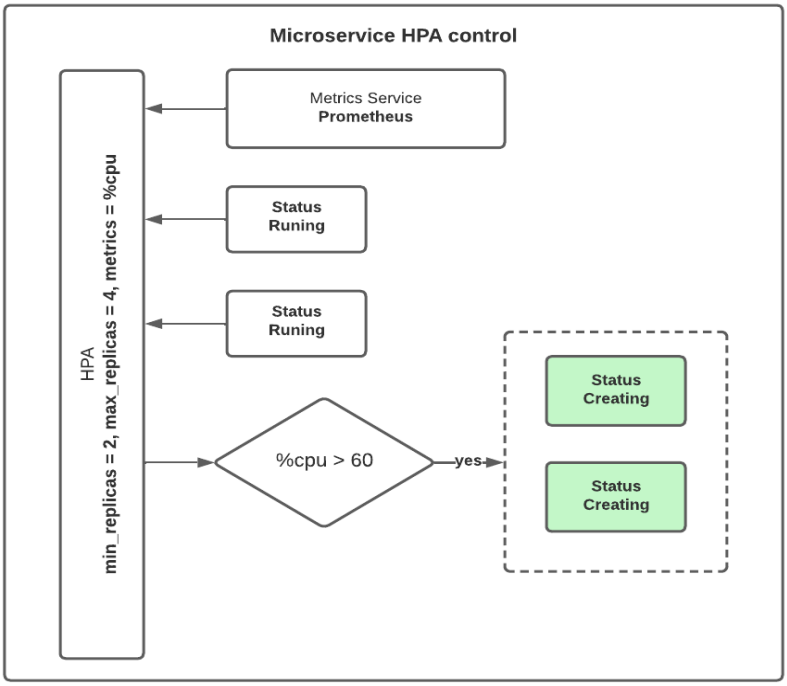

Figure 4. Microservice HPA control.

For this case, we used a Jmeter ramp-up period of 2 min and set the number of replicas for the facemask deployment to two. We used the CPU utilization as the metric to scale the service to four instances. We set the maximum CPU utilization to 60% and verified that Kubernetes created the additional instances when the CPU utilization threshold is exceeded in less than 2 min while processing the 1000 requests. This traffic represents a bigger demand than expected during the rush time (1000 users in 5 min) and the platform quickly responds by autoscaling the pods.

- Case 3: The ability of the system to route traffic in a controlled manner. Communication channels were established using Calico's network policies to define routes for the most important services, such as the API gateway and facemask detection, to improve the security of data delivery between microservices. To validate this operation, we sent data in compliance with the routes defined using the network policies and observed the platform's proper operation. When attempting to bypass a service by sending it via an incorrect path, the platform restricts the passage of information. Figure 5 shows the network policies to enter data into the platform and connect internally to the microservices. The network policies, in this case, allow the creation of a bridge between both namespaces and find the services by using their names and not the IP addresses, avoiding connectivity problems when IP addresses change. Each policy is created with a configuration file shown in Listing 1.

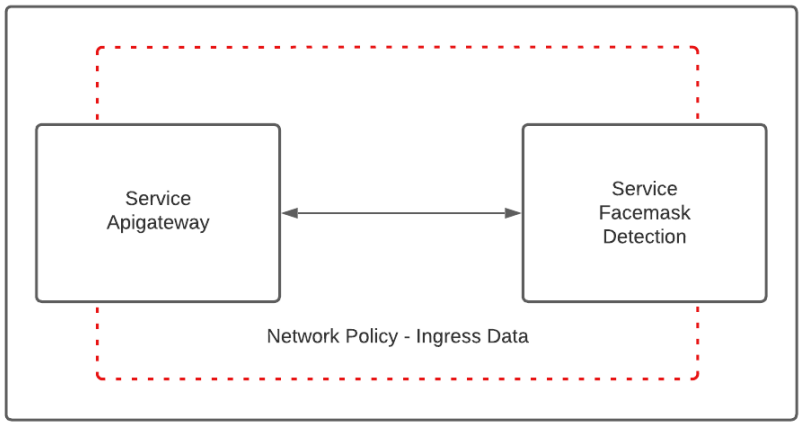

Figure 5. Network policy.

7. Related Work

We propose a novel distributed computing platform that leverages resilience mechanisms to provide application fault tolerance. Although we found a number of works related to fault tolerance on cloud computing systems, few of those seem to tackle the specific problem of providing fault tolerance to high-latency analytics applications. Our platform is based on a microservices architecture that allows the distribution of high-latency analytics applications based on the rapid provision and reuse of self-contained services.

In [64] Javed et al. present an architecture for fault-tolerant IoT services deployed on edge and cloud infrastructures. This approach is based on integrating Kafka and Kubernetes technologies to provide a mechanism to tackle hardware and network failures. The platform is tested by deploying a surveillance camera system. Our approach leverages fault-tolerance capabilities integrated within Kafka and Kubernetes to provide a completely fault-tolerant pipeline, and was tested using high-latency analytics applications. Additionally, our approach incorporates resilience mechanisms such as the autoscaling component to adapt infrastructure to increased workload automatically.

Torres et al. [65] propose a framework that leverages state-of-the-art cloud technologies to support the deployment of artificial intelligence services with high transmission latency and high bandwidth consumption on edge-cloud infrastructures. They focus on partitioning and distributing Deep Neural Networks (DNN) to improve response times. Our approach proposes a platform based on a microservices architecture that allows the partitioning of high-latency analytics applications, improving performance. In addition, our approach provides a set of predefined and reusable microservices that facilitates the fast construction and deployment of new resilient applications.

Given the memory size and processing capability of today's commodity machines, it is inevitable to run distributed machine learning (DML) on multiple machines [66]. Wang et al., propose a scalable, high-performance, and fault-tolerant Distributed Machine Learning (DML) network architecture on top of Ethernet and commodity devices. BML builds on BCube topology and runs a fully-distributed gradient synchronization algorithm. In our approach the goal of the Kubernetes orchestration cluster is also to process machine learning datasets, node/link failures were simulated, and fault tolerances and traffic management were achieved to keep services performance.

One of the most critical challenges of the Internet of Things (IoT) is to provide real-time services. Bakhshi et al., propose in [67] a comprehensive SDN-based fault-tolerant architecture in IoT environments. In the proposed scheme, a mathematical model called Shared Risk Link Group (SRLG) calculates redundant paths as the main and backup non-overlapping paths between network equipment. When comparing the suggested scheme to two policies for constructing routes from source to destination, it is found that the proposed method improves service quality parameters such as packet loss, latency, and packet jitter while reducing error recovery time. In our work, SDN technology provides a fault-

tolerant network architecture to support the Kubernetes cluster, offering redundant paths to guarantee that all the containers of the microservices in Kubernetes remain connected. This setup can also be used for IoT applications with nodes prone to failure.

Fault-tolerant frameworks are also applicable for fog computing. Zhang et al., propose in [68], a model based on a Markov chain to use it as a fault-tolerant system. This system allows them to predict the number of fog nodes that can fail and change them to reduce delays and costs. This approach is focused only on nodes, whereas our work is focused on microservices that can be used with resilience mechanisms to reduce the impact of different types of failures.

Tang in [69] proposes a scheduling algorithm that integrates fault tolerance and cost reduction for applications in the cloud. Tang's model is based on the DAG model, which considers task priorities to select the best VM for executing the entire application. At the same time, the scheduling algorithm tries to find a backup to support the task that is not running correctly in the VM previously selected. The goal of our approach is not to schedule or allocate applications. Still, it is to provide a infrastructure with some resilient mechanisms to support the execution of applications.

8. Conclusions

In this paper, we provide a first evaluation of the computational and communications infrastructure requirements that must be met by a resilient cloud system. We present the design and implementation of a microservices-based platform that provides resilience to applications and evaluate it by deploying and testing a high-latency analytics application. Resiliency features such as hot migration and modification, redundancy, autoscaling, optimal computing and storage resource allocation systems, and load balancing services are crucial for this type of platform. In regard to the computing infrastructure requirements, the need for performance isolation mechanisms based on virtualization solutions and software containers was evidenced. These solutions also allow for the provision of lightweight, loosely coupled microservices and their distribution through public repositories. In terms of the communications infrastructure requirements, resilient solutions must support fault-tolerance mechanisms and traffic management in order to minimize the impacts on the quality of service when failures occur. Services should use lightweight, standardized communication interfaces based on protocols such as HTTP. In the same way, the use of protocols and technologies with low energy consumption is imperative. Performance isolation in the network and adaptation of the network infrastructure to failure, supported by efficient monitoring systems, are also required features. Trying to meet these requirements, we proposed a computational platform and demonstrated its ability to provide resilience to high-latency analytics applications. Our microservice-based platform, which leverages current container and orchestration technologies, allowed us to quantify the impact of different resilience mechanisms.

Author Contributions: Conceptualization, O.H.M.; Investigation, O.H.M. and H.A.R.; Methodology, O.H.M., H.A.R. and J.M.; Software, H.F.M. and J.M.; Supervision, O.H.M.; Validation, H.F.M. and J.M.; Visualization, J.M.; Writing–original draft, H.F.M., O.H.M., H.A.R. and J.M.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All data were presented in main text.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Abdullah, M.; Iqbal, W.; Bukhari, F.; Erradi, A. Diminishing returns and deep learning for adaptive CPU resource allocation of containers. *IEEE Trans. Netw. Serv. Manag.* **2020**, *17*, 2052–2063. [CrossRef]
2. Shahid, M.A.; Islam, N.; Alam, M.M.; Mazliham, M.; Musa, S. Towards Resilient Method: An exhaustive survey of fault tolerance methods in the cloud computing environment. *Comput. Sci. Rev.* **2021**, *40*, 100398. [CrossRef]
3. Pueyo Centelles, R.; Freitag, F.; Meseguer, R.; Navarro, L.; Ochoa, S.; Santos, R. A LoRa-Based Communication System for Coordinated Response in an Earthquake Aftermath. *Proceedings* **2019**, *31*, 73. [CrossRef]
4. Sciullo, L.; Trotta, A.; Felice, M.D. Design and performance evaluation of a LoRa-based mobile emergency management system (LOCATE). *Ad Hoc Netw.* **2020**, *96*, 101993. [CrossRef]
5. Mikhaylov, K.; Stusek, M.; Masek, P.; Fujdiak, R.; Mozny, R.; Andreev, S.; Hosek, J. Communication Performance of a Real-Life Wide-Area Low-Power Network Based on Sigfox Technology. In Proceedings of the ICC 2020—2020 IEEE International Conference on Communications (ICC), Online, 7–11 June 2020; pp. 1–6.
6. Oliveira, L.; Rodrigues, J.J.; Kozlov, S.A.; Rabêlo, R.A.; Furtado, V. Performance assessment of long-range and Sigfox protocols with mobility support. *Int. J. Commun. Syst.* **2019**, *32*, e3956. [CrossRef]
7. Adhikary, A.; Lin, X.; Wang, Y.E. Performance Evaluation of NB-IoT Coverage. In Proceedings of the 2016 IEEE 84th Vehicular Technology Conference (VTC-Fall), Montreal, QC, Canada, 18–21 September 2016; pp. 1–5.
8. Mangalvedhe, N.; Ratasuk, R.; Ghosh, A. NB-IoT deployment study for low power wide area cellular IoT. In Proceedings of the 2016 IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC), Valencia, Spain, 4–8 September 2016; pp. 1–6.
9. Hinds, A.; Ngulube, M.; Zhu, S.; Al-Aqrabi, H. A review of routing protocols for mobile ad-hoc networks (manet). *Int. J. Inf. Educ. Technol.* **2013**, *3*, 1. [CrossRef]
10. Miao, Y.; Sun, Z.; Wang, N.; Cruickshank, H. Comparison studies of MANET-satellite and MANET-cellular networks integrations. In Proceedings of the 2015 International Conference on Wireless Communications Signal Processing (WCSP), Nanjing, China, 15–17 October 2015; pp. 1–5.
11. Akyildiz, I.F.; Wang, X.; Wang, W. Wireless mesh networks: A survey. *Comput. Netw.* **2005**, *47*, 445–487. [CrossRef]
12. Yarali, A.; Ahsant, B.; Rahman, S. Wireless mesh networking: A key solution for emergency & rural applications. In Proceedings of the 2009 Second International Conference on Advances in Mesh Networks, Athens, Greece, 14–19 June 2009; pp. 143–149.
13. Jorguseski, L.; Pais, A.; Gunnarsson, F.; Centonza, A.; Willcock, C. Self-organizing networks in 3GPP: Standardization and future trends. *IEEE Commun. Mag.* **2014**, *52*, 28–34. [CrossRef]
14. Zoha, A.; Saeed, A.; Imran, A.; Imran, M.A.; Abu-Dayya, A. Data-driven analytics for automated cell outage detection in Self-Organizing Networks. In Proceedings of the 2015 11th International Conference on the Design of Reliable Communication Networks (DRCN), Kansas City, MO, USA, 24–27 March 2015; pp. 203–210. [CrossRef]
15. Caro, A.L.; Iyengar, J.R.; Amer, P.D.; Ladha, S.; Heinz, G.J.; Shah, K.C. SCTP: A proposed standard for robust internet data transport. *Computer* **2003**, *36*, 56–63. [CrossRef]
16. Iyengar, J.R.; Amer, P.D.; Stewart, R. Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths. *IEEE/ACM Trans. Netw.* **2006**, *14*, 951–964. [CrossRef]
17. Arzani, B.; Gurney, A.; Cheng, S.; Guerin, R.; Loo, B.T. Deconstructing MPTCP performance. In Proceedings of the 2014 IEEE 22nd International Conference on Network Protocols, Raleigh, NC, USA, 21–24 October 2014; pp. 269–274.
18. Manzanares-Lopez, P.; Muñoz-Gea, J.P.; Malgosa-Sanahuja, J. An MPTCP-Compatible Load Balancing Solution for Pools of Servers in OpenFlow SDN Networks. In Proceedings of the 2019 Sixth International Conference on Software Defined Systems (SDS), Rome, Italy, 10–13 June 2019; pp. 39–46.
19. Giannini, V.; Craninckx, J.; Baschirotto, A. *Baseband Analog Circuits for Software Defined Radio*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2008.
20. Almoualem, F.; Satam, P.; Ki, J.; Hariri, S. SDR-Based Resilient Wireless Communications. In Proceedings of the 2017 International Conference on Cloud and Autonomic Computing (ICCAC), Tucson, AZ, USA, 18–22 September 2017; pp. 114–119.
21. Feamster, N.; Rexford, J.; Zegura, E. The road to SDN: An intellectual history of programmable networks. *ACM SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 87–98. [CrossRef]
22. Mas Machuca, C.; Secci, S.; Vizarreta, P.; Kuipers, F.; Gouglidis, A.; Hutchison, D.; Jouet, S.; Pezaros, D.; Elmokashfi, A.; Heegaard, P.; et al. Technology-related disasters: A survey towards disaster-resilient Software Defined Networks. In Proceedings of the 2016 8th International Workshop on Resilient Networks Design and Modeling (RNDM), Halmstad, Sweden, 13–15 September 2016; pp. 35–42.
23. Nguyen, K.; Minh, Q.T.; Yamada, S. A Software-Defined Networking Approach for Disaster-Resilient WANs. In Proceedings of the 2013 22nd International Conference on Computer Communication and Networks (ICCCN), Nassau, Bahamas, 30 July–2 August 2013; pp. 1–5.
24. Herrera, J.G.; Botero, J.F. Resource allocation in NFV: A comprehensive survey. *IEEE Trans. Netw. Serv. Manag.* **2016**, *13*, 518–532. [CrossRef]
25. Machado, C.C.; Granville, L.Z.; Schaeffer-Filho, A. ANSwer: Combining NFV and SDN features for network resilience strategies. In Proceedings of the 2016 IEEE Symposium on Computers and Communication (ISCC), Messina, Italy, 27–30 June 2016; pp. 391–396.

26. Volvach, I.; Globa, L. Mobile networks disaster recovery using SDN-NFV. In Proceedings of the 2016 International Conference Radio Electronics & Info Communications (UkrMiCo), Kiev, Ukraine, 11–16 September 2016; pp. 1–3.
27. Dmitry, N.; Manfred, S.S. On micro-services architecture. *Int. J. Open Inf. Technol.* **2014**, *2*, 24–27.
28. Turnbull, J. *The Docker Book: Containerization Is the New Virtualization*; James Turnbull: Brooklyn, NY, USA, 2014.
29. Cérin, C.; Menouer, T.; Saad, W.; Abdallah, W.B. A new docker swarm scheduling strategy. In Proceedings of the 2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2), Kanazawa, Japan, 22–25 November 2017; pp. 112–117.
30. Kakadia, D. *Apache Mesos Essentials*; Packt Publishing Ltd.: Birmingham, UK, 2015.
31. Sabharwal, N.; Pandey, S.; Pandey, P. Getting Started with Nomad. In *Infrastructure-as-Code Automation Using Terraform, Packer, Vault, Nomad and Consul*; Springer: Berlin/Heidelberg, Germany, 2021; pp. 201–236.
32. Luksa, M. *Kubernetes in Action*; Simon and Schuster: New York, NY, USA, 2017.
33. Sayfan, G. *Mastering Kubernetes*; Packt Publishing Ltd.: Birmingham, UK, 2017.
34. Buchanan, S.; Rangama, J. Deploying and Using Rancher with Azure Kubernetes Service. Available online: https://link.springer.com/chapter/10.1007/978-1-4842-5519-3_6 (accessed on 12 July 2022).
35. Franzke, R.; Chandrasekhara, V. Gardener—The Kubernetes Botanist. Available online: https://kubernetes.io/blog/2018/05/17/gardener/ (accessed on 12 July 2022).
36. Kumari, P.; Kaur, P. A survey of fault tolerance in cloud computing. *J. King Saud Univ. Comput. Inf. Sci.* **2021**, *33*, 1159–1176. [CrossRef]
37. Lee, S.; Levanti, K.; Kim, H.S. Network monitoring: Present and future. *Comput. Netw.* **2014**, *65*, 84–98. [CrossRef]
38. Narayan, S.; Graham, D.; Barbour, R.H. Generic factors influencing optimal LAN size for commonly used operating systems maximized for network performance. *IJCSNS Int. J. Comput. Sci. Netw. Secur.* **2009**, *9*, 63–72.
39. Shooman, M.L. *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design*; John Wiley & Sons: Hoboken, NJ, USA, 2003.
40. Singh, J.; Powles, J.; Pasquier, T.; Bacon, J. Data Flow Management and Compliance in Cloud Computing. *IEEE Cloud Comput.* **2015**, *2*, 24–32. [CrossRef]
41. Merkel, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.* **2014**, *2014*, 2.
42. Senthil Kumaran, S. *Practical LXC and LXD: Linux Containers for Virtualization and Orchestration*; Springer: Berlin/Heidelberg, Germany, 2017.
43. Kurtzer, G.M.; Sochat, V.; Bauer, M.W. Singularity: Scientific containers for mobility of compute. *PLoS ONE* **2017**, *12*, e0177459. [CrossRef] [PubMed]
44. Gerhardt, L.; Bhimji, W.; Canon, S.; Fasel, M.; Jacobsen, D.; Mustafa, M.; Porter, J.; Tsulaia, V. Shifter: Containers for hpc. *J. Phys. Conf. Ser.* **2017**, *898*, 082021. [CrossRef]
45. Kivity, A.; Kamay, Y.; Laor, D.; Lublin, U.; Liguori, A. kvm: The Linux virtual machine monitor. In Proceedings of the Linux Symposium, Ottawa, ON, Canada, 23–26 July 2007; Volume 1, pp. 225–230.
46. Fraser, K.; Hand, S.; Neugebauer, R.; Pratt, I.; Warfield, A.; Williamson, M. Safe hardware access with the Xen virtual machine monitor. In Proceedings of the 1st Workshop on Operating System and Architectural Support for the on Demand IT InfraStructure (OASIS), Boston, MA, USA, 11–13 October 2004; p. 1.
47. Thompson, C. *Vagrant Virtual Development Environment Cookbook*; Packt Publishing Ltd.: Birmingham, UK, 2015.
48. Kumaran S, S.; Kumaran S, S. LXC and LXD Resources. *Practical LXC and LXD: Linux Containers for Virtualization and Orchestration*; Apress: New York, NY, USA, 2017; pp. 39–58.
49. Cook, J. Docker Hub. In *Docker for Data Science*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 103–118.
50. Kurtzer, G.M. Singularity. 2018. Available online: https://git.its.aau.dk/CLAAUDIA/docs_aicloud/raw/commit/79165e32ad24cd933dc38bf58c4b60cf6f74f3a0/aicloud_slurm/refs/GMKurtzer_Singularity_Keynote_Tuesday_02072017.pdf (accessed on 1 May 2022).
51. Mirkin, A.; Kuznetsov, A.; Kolyshkin, K. Containers checkpointing and live migration. In Proceedings of the Linux Symposium, Ottawa, ON, Canada, 23–26 July 2008; Volume 2, pp. 85–90.
52. Sami, H.; Mourad, A.; Otrok, H.; Bentahar, J. Fscaler: Automatic resource scaling of containers in fog clusters using reinforcement learning. In Proceedings of the 2020 International Wireless Communications and Mobile Computing (IWCMC), Limassol, Cyprus, 15–19 June 2020; pp. 1824–1829.
53. Yan, M.; Liang, X.; Lu, Z.; Wu, J.; Zhang, W. HANSEL: Adaptive horizontal scaling of microservices using Bi-LSTM. *Appl. Soft Comput.* **2021**, *105*, 107216. [CrossRef]
54. Marquez, J.; Mondragon, O.H.; Gonzalez, J.D. An Intelligent Approach to Resource Allocation on Heterogeneous Cloud Infrastructures. *Appl. Sci.* **2021**, *11*, 9940. [CrossRef]
55. Guan, Y.; Ma, Z.; Li, L. HDFS optimization strategy based on hierarchical storage of hot and cold data. *Procedia CIRP* **2019**, *83*, 415–418. [CrossRef]
56. Jamshidi, P.; Pahl, C.; Mendonça, N.C.; Lewis, J.; Tilkov, S. Microservices: The journey so far and challenges ahead. *IEEE Softw.* **2018**, *35*, 24–35. [CrossRef]
57. de Carvalho, J.O.; Trinta, F.; Vieira, D. PacificClouds: A Flexible MicroServices based Architecture for Interoperability in Multi-Cloud Environments. In Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018), Funchal, Portugal, 19–21 March 2018; pp. 448–455.

58. Solarte, Z.; Gonzalez, J.D.; Peña, L.; Mondragon, O.H. Microservices-Based Architecture for Resilient Cities Applications. In Proceedings of the International Conference on Advanced Engineering Theory and Applications, Bogota, Colombia, 6–8 November 2019; pp. 423–432.
59. Kubernetes. Kubernetes Documentation. Available online: https://kubernetes.io/docs/home/ (accessed on 12 July 2022).
60. Resilience4j. Resilience4j: A Fault Tolerance Library Designed for Functional Programming. Available online: https://github.com/resilience4j/resilience4j (accessed on 12 July 2022).
61. Zhou, Z.; Zhang, H.; Du, X.; Li, P.; Yu, X. Prometheus: Privacy-aware data retrieval on hybrid cloud. In Proceedings of the 2013 Proceedings IEEE INFOCOM, Turin, Italy, 14–19 April 2013; pp. 2643–2651.
62. Grafana. Grafana Labs. Available online: https://grafana.com/ (accessed on 12 July 2022).
63. Halili, E.H. *Apache JMeter*; Packt Publishing: Birmingham, UK, 2008.
64. Javed, A.; Robert, J.; Heljanko, K.; Främling, K. IoTEF: A federated edge-cloud architecture for fault-tolerant IoT applications. *J. Grid Comput.* **2020**, *18*, 57–80. [CrossRef]
65. Torres, D.R.; Martín, C.; Rubio, B.; Díaz, M. An open source framework based on Kafka-ML for Distributed DNN inference over the Cloud-to-Things continuum. *J. Syst. Archit.* **2021**, *118*, 102214. [CrossRef]
66. Wang, S.; Li, D.; Cheng, Y.; Geng, J.; Wang, Y.; Wang, S.; Xia, S.; Wu, J. A Scalable, High-Performance, and Fault-Tolerant Network Architecture for Distributed Machine Learning. *IEEE/ACM Trans. Netw.* **2020**, *28*, 1752–1764. [CrossRef]
67. Bakhshi Kiadehi, K.; Rahmani, A.M.; Sabbagh Molahosseini, A. A fault-tolerant architecture for internet-of-things based on software-defined networks. *Telecommun. Syst.* **2021**, *77*, 155–169. [CrossRef]
68. Zhang, P.Y.; Chen, Y.T.; Zhou, M.C.; Xu, G.; Huang, W.J.; Al-Turki, Y.; Abusorrah, A. A Fault-tolerant Model for Performance Optimization of a Fog Computing System. *IEEE Internet Things J.* **2021**, *9*, 1725–1736. [CrossRef]
69. Tang, X. Reliability-aware cost-efficient scientific workflows scheduling strategy on multi-cloud systems. *IEEE Trans. Cloud Comput.* **2021**. [CrossRef]