future internet

MDPI

Article

Spot Market Cloud Orchestration Using Task-Based Redundancy and Dynamic Costing

Vyas O'Neill * and Ben Soh

Department of Computer Science and Information Technology, La Trobe University, Melbourne 3083, Australia; b.soh@latrobe.edu.au

* Correspondence: v.oneill@latrobe.edu.au

Abstract: Cloud computing has become ubiquitous in the enterprise environment as its on-demand model realizes technical and economic benefits for users. Cloud users demand a level of reliability, availability, and quality of service. Improvements to reliability generally come at the cost of additional replication. Existing approaches have focused on the replication of virtual environments as a method of improving the reliability of cloud services. As cloud systems move towards microservices-based architectures, a more granular approach to replication is now possible. In this paper, we propose a cloud orchestration approach that balances the potential cost of failure with the spot market running cost, optimizing the resource usage of the cloud system. We present the results of empirical testing we carried out using a simulator to compare the outcome of our proposed approach to a control algorithm based on a static reliability requirement. Our empirical testing showed an improvement of between 37% and 72% in total cost over the control, depending on the specific characteristics of the cloud models tested. We thus propose that in clouds where the cost of failure can be reasonably approximated, our approach may be used to optimize the cloud redundancy configuration to achieve a lower total cost.

Keywords: cloud computing; cloud services orchestration; cloud microservices architecture; distributed computer systems; cloud reliability

Citation: O'Neill, V.; Soh, B. Spot Market Cloud Orchestration Using Task-Based Redundancy and Dynamic Costing. *Future Internet* **2023**, *15*, 288. https://doi.org/10.3390/fi15090288

Academic Editors: Jerry Chou and Wu-Chun Chung

Received: 3 August 2023
Revised: 23 August 2023
Accepted: 25 August 2023
Published: 27 August 2023

1. Introduction

Cloud computing has become ubiquitous in the enterprise environment as its on-demand model provides technical and economic benefits for its users [1]. This on-demand model is made possible by the virtualization and containerization of resources at each level of the cloud infrastructure [2]. Cloud computing users demand a level of reliability, availability, and quality of service commensurate with their application domain [3,4]. The virtualization of resources in cloud environments enables the cloud operator to rapidly respond to faults, changes in demand, and maintenance requirements, redeploying resources as needed [2].

Established approaches to cloud systems' virtualization have used virtual machines as the basis of their operation [5]. Recently, however, the industry has been increasingly moving towards a container- and microservices-based architecture for cloud systems in order to realize the benefits of lighter resource footprints and a more granular level of control in all phases of the software development life cycle [3].

The benefits of using a microservices-based architecture include the ability to provide a finer level of reliability of cloud systems components by managing reliability at the microservice container level [6]. In [6], we proposed a novel approach to reliability in microservices-based cloud architectures using Task-Based Redundancy (TBR). Our approach modeled the reliability of the system with respect to the marginal cost of adding additional redundant microservice containers in comparison with an expected cost of failure, both of which were fixed parameters [6].

There are several existing costing models in the cloud computing market, including pay-as-you-go (PAYG), on-demand (OD), and spot market (SM) [7]. Of these, spot market costing varies according to the specific market conditions at the time of usage which cannot be predicted in advance [7]. The dynamic costing model used in a spot market thus results in fluctuations in the usage cost per unit time, and by extension, fluctuations in the cost of providing redundancy for critical microservice containers. These fluctuations present an opportunity for cost optimization which would not otherwise exist in a fixed system.

In this paper, we consider the effect that such fluctuations may have on the relationship between the cost of redundancy and the cost of failure and propose a TBR-based orchestration algorithm that takes this into account when determining the level of redundancy in the cloud. We apply the idea of task-based, microservice-level reliability to a market-based cloud ecosystem, where the marginal cost of adding cloud services and the expected cost of failure are parameters dictated by the open market.

In Section 2, we survey related works. In Section 3, we discuss our model for reliability in cloud systems through an application of TBR. Then, in Section 4, we present our proposed algorithm for cloud orchestration with respect to the spot market conditions and reliability requirements. In Section 5, we present our experimental methodology and the results of simulations comparing our proposed approach with a control approach based on a fixed reliability requirement. Finally, in Sections 6 and 7, we discuss the observed results and draw conclusions.

2. Related Work

In this section, we review existing approaches and related work in virtualization and containerization, microservices architectures, and resilience of cloud systems.

2.1. Virtualization and Containerization

Virtual machines (VMs) are an established method of virtualizing cloud-based assets [5]. A VM is a self-contained unit comprising a guest operating system image, supporting libraries, and the applications required to provide service [8]. While VMs provide the advantages of virtualization such as isolation, transferability between physical assets, and platform independence, a significant amount of the resources they consume are wasted on content common to multiple instances, such as operating system components [8].

Increasingly, the cloud computing industry is adopting container-based models to address the shortcomings of VMs [2]. A container, as opposed to a VM, comprises only those components necessary to provide the individual characteristics of the application, while operating system binaries and other support infrastructure are shared (Figure 1) [2]. Containers "provide OS-level virtualization by leveraging kernel features to isolate processes and define system usage limits for resources such as CPI, memory, disk I/O and network" [9].

These efficiencies are manifested as performance improvements over VMs which are significant at scale, as shown in several recent studies that compared VMs to containers in various applications, including the classic cloud computing use cases of artificial intelligence [10], big data [11], edge computing [12], and databases [13].

The disadvantage of container-based systems is that they are limited in their flexibility to support multiple types of heterogeneous operating systems [2].

2.2. Microservices Architecture

Cloud systems based on VMs, which, by their nature, are self-contained units capable of independent operation, have in the past tended toward more monolithic system architectures [2]. However, with the trend toward containerization in the infrastructure, a concurrent trend is being seen in the industry moving toward microservices-based system architectures [3].

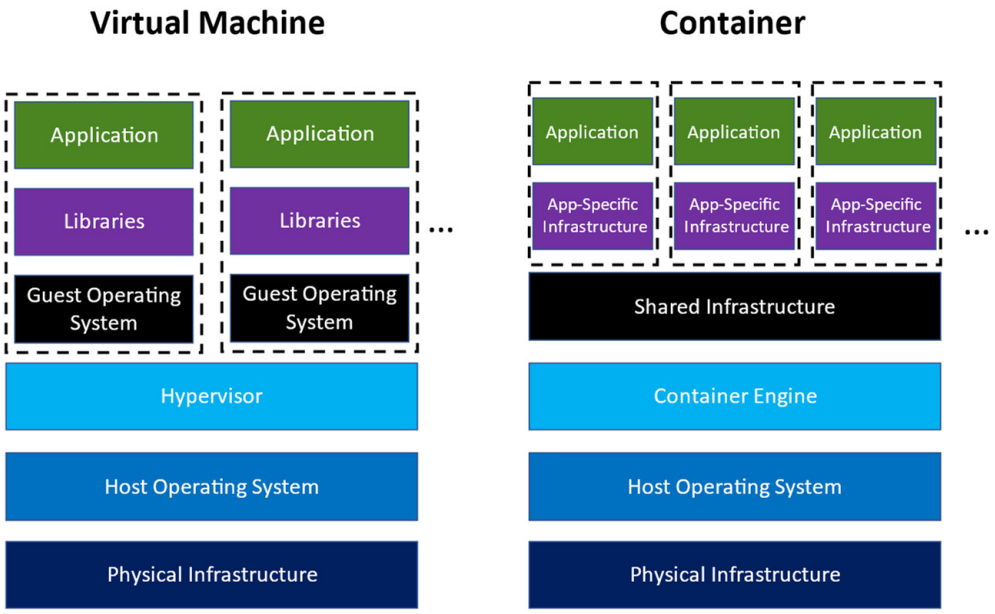

Figure 1. Comparison of virtual machine (**left**) and container (**right**) architectures.

Microservices architecture "is a cloud-native architecture that aims to realize software systems as a package of small services, each independently deployable on a potentially different platform and technological stack, and running in its own process while communicating through lightweight [APIs]" [14]. Microservices architecture "aims to decompose a monolithic application into a set of independent services which communicate with each other through open APIs or highly scalable messaging" [15].

In [3], the authors identify several advantages of the microservices architecture, including simplification and acceleration of deployment through modularity, language and programming framework heterogeneity, and simplification of correctness and performance debugging. In [16], the author notes cost reduction, quality improvement, agility, and decreased time to market as benefits of the microservices architecture.

The increasing use of microservices is, in part, based on their correspondence with the containerization model at the infrastructure level, with "each microservice accommodated in a single container" [3].

A key innovation in the microservices architecture is that of the stateless container. While stateful containers also exist, in which state information is kept in the container between calls to the microservice, stateless microservice containers are designed such that no state information is kept in the container between calls to the microservice [17]. Any stateful information required to supply the desired service is abstracted out of the microservice into a separate, persistent repository such as a database [17].

Stateless containers are favored by cloud orchestrators due to their simpler synchronization and the ability to restart and initialize them from a clean state [17]. They thus "facilitate the architectural decomposition of microservices into functional, stateless components supported by persistent data storage with defined scope and intention" [6] (Figure 2).

One of the advantages of stateless containers is that different instances of the same container are functionally and statefully equivalent [6]. This property of stateless containers makes them particularly suitable for at-scale replication and use in redundancy-based fault tolerance and reliability strategies [6].

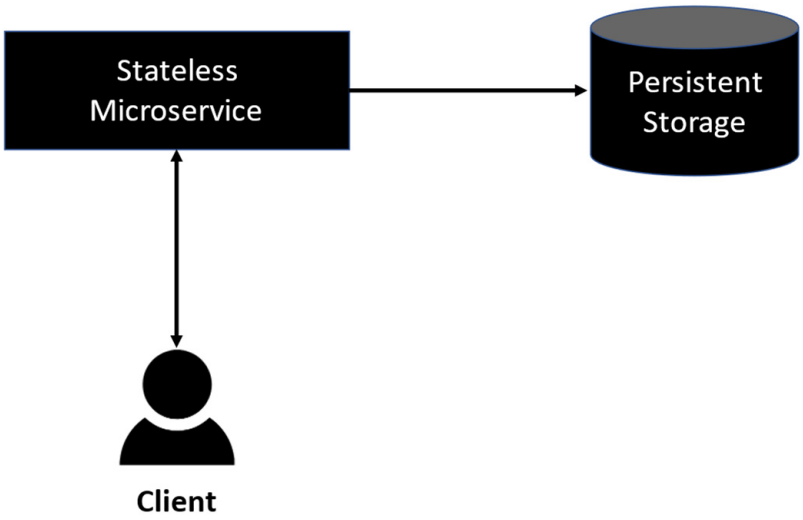

Figure 2. Stateless container architecture.

2.3. Resilience of Cloud Systems

Cloud systems generally have resilience requirements that define their required behavior with respect to various parameters such as fault tolerance, reliability, quality of service, and availability [18].

Resilience is defined as "the capacity of a system . . . to remain reliable, dependable, failure tolerant, survivable and secure in case of any malicious or accidental malfunctions or failures that result in a temporal or permanent service disruption" [18]. Reliability is defined as "the probability that the system will be functioning at a given time and capable of providing service" [6,19].

Various approaches exist to improve the resilience of cloud systems at different levels of the cloud infrastructure. For example, in [4], the authors propose a cloud systems architecture that automatically configures the fault detection and fault recovery methods in the context of particular service level requirements, while in [18], the authors propose to map component failures to different layers in the cloud architecture so that the component failures propagate from the lower layers to the application layer if not handled.

Broadly speaking, the different approaches to cloud resilience can be classified into three categories (Figure 3): failure forecasting, protection, and recovery, with protection being further divided into replication-based and checkpointing-based approaches [6,18,20]. Checkpointing, where periodic snapshots are taken to enable rollback and recovery from errors, is inherently applicable to stateful systems; thus, the replication strategy is most appropriate for stateless microservice containers [6].

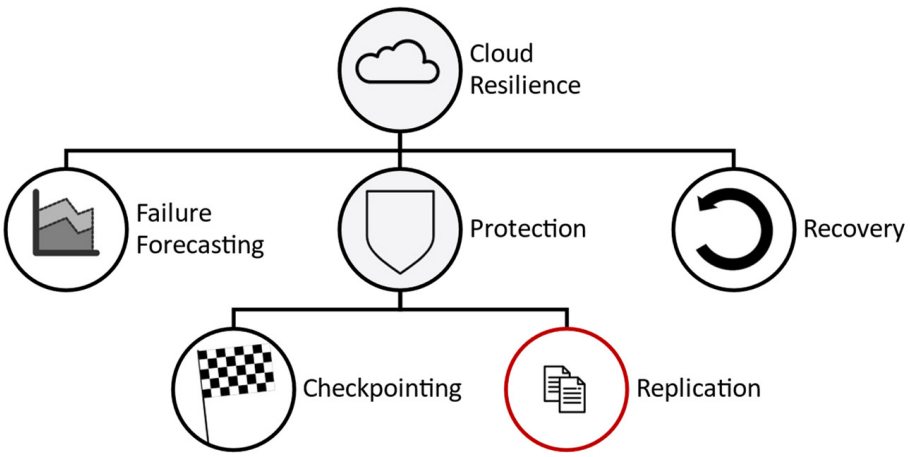

Figure 3. Classification tree of approaches to cloud resilience.

Replication of redundant components is the core principle upon which reliability and fault tolerance strategies are based [19]. Through replication, individual component failures can be masked and tolerated, thereby preventing a failure of the entire system [19].

We propose that replication is particularly suitable for improving the resilience of stateless microservice containers based on the low cost of replication and user agnosticism with respect to which particular microservice container will provide them with their desired service [6]. Thus, provided that sufficient redundancy exists to cover the temporary shortfall, failed components can be seamlessly replaced with no loss of user experience.

Given a suitable algorithm for determining the level of redundancy, microservice containers can be spawned and shut down accordingly to achieve the desired level of reliability. Various approaches exist in the literature to guide such choices of replication and redundancy levels in cloud systems, including the classic, static redundancy level approach [19], adaptations of the Mean-Time-To-Failure (MTTF) metric [21], Petri Nets [22–24], and Markov chains [25], among others.

2.4. Existing Spot Market Cloud Implementations

Several existing services provide a spot market for cloud resources which have been studied in the literature. In [26], the authors considered the life cycle of spot instances (SIs) across three different regions in the Amazon EC2 cloud service. They found that SIs were most reliable in the first 20 to 30 min after deployment.

In [27], the authors explored a method for optimizing the usage cost of spot market-based cloud resources using a Long/Short-Term Memory recurrent neural network to predict the spot price.

Similar services to the Amazon EC2 spot market are provided by Google [28] and on the Azure cloud platform [29]. In [30], the authors provide an exhaustive survey of spot pricing in the cloud ecosystem, from both a computation and economics perspective.

3. Task-Based Redundancy in Cloud Systems

This section presents Task-Based Redundancy (TBR), a general redundancy technique that forms the basis of our approach to managing reliability in complex cloud systems.

3.1. Background to Task-Based Redundancy

Existing approaches generally assume the need for a particular level of reliability as an axiom, rather than determining it from an analysis of the domain-specific conditions applicable at runtime.

Previously, we proposed an approach using TBR [6,31] as a method of modeling and managing the reliability of a complex cloud system by decomposing its subcomponents according to the tasks (or services) a system must be able to reliably accomplish for the user.

Our approach links the source of the reliability requirement, the predictive model for microservice container failure, and the compound nature of the cloud services under the responsibility of the cloud orchestrator [6].

The advantage of this approach is that it closely aligns the reliability requirement, and its consequent replication strategy, to the specific service outcomes expected by the user of the cloud system. With modern, enterprise-grade cloud applications generally comprising many microservice components, possibly even hundreds, to create the final solution, each microservice container may be found to be a dependency for several independent tasks [32].

In the classic approach to reliability, each microservice container is simply replicated to a certain desired level of replication. The level of replication is determined by what the Service-Level Agreement (SLA) defines as an acceptable value for the probability of system failure in a given time interval, such as in "nine-nines reliability" (where the probability of failure in the interval is 1.0×10^{-9}), which we used as the control for our experiments.

In contrast, the TBR approach considers the reliability of the system not with respect to redundant microservice containers per se, but with respect to the system's wholistic

ability to complete a task comprising several microservices. From a software engineering perspective, TBR may be considered an application of use case analysis and microservices-oriented decomposition to reliability analysis [6,33].

In the TBR approach, once microservices are grouped according to tasks, the same probabilistic models which are applied to components in general reliability analysis are applied to these tasks, with each microservice making up an in-series or in-parallel component of the task's reliability function according to its configuration [6].

3.2. Applying Task-Based Redundancy to Cloud Microservices

Our proposed model for applying TBR to cloud microservices uses three tiers of abstraction: the task tier, the microservice tier, and the container tier (Figure 4). A task is a set of actions that depend on the acceptable functioning of one or more microservices. In our model, microservices are abstract, stateless images that can be instantiated into as many containers as are necessary at runtime. These containers are the instantiated redundant instances that actually provide the service to the end user.

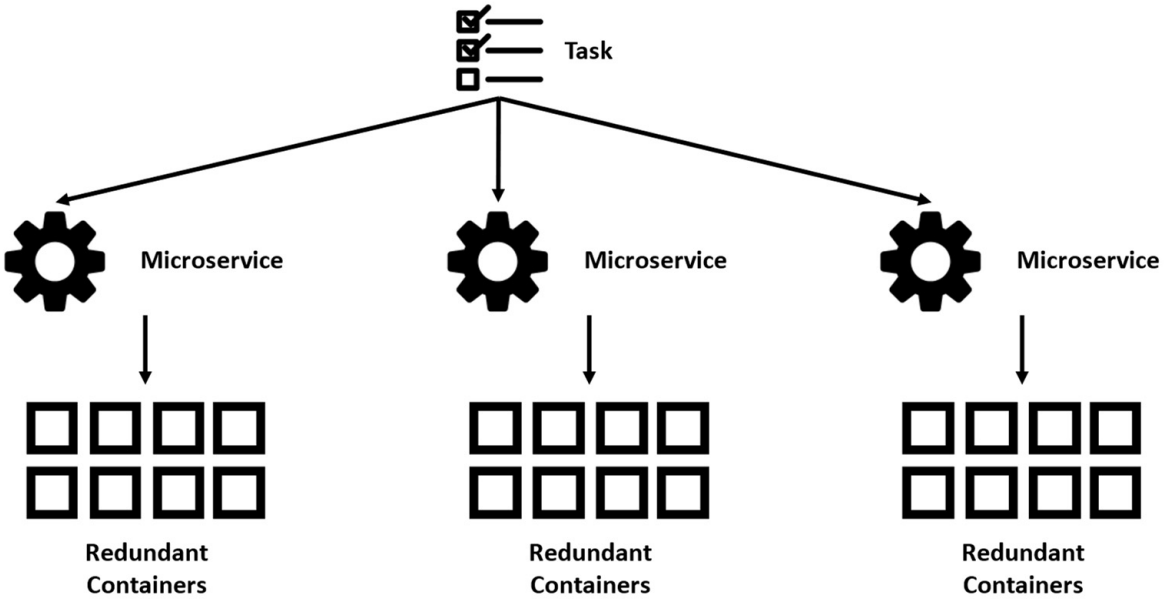

Figure 4. Our Task-Based Redundancy (TBR) model for cloud microservices. The diagram shows a single task composed of several microservices. The microservices shown may also be dependencies for other independent tasks in the cloud system.

In TBR, the reliability of the system with respect to some task T is modeled using a failure function, $F_T(t)$, which defines the probability that, due to the failure of all the redundant components, the system will enter the failure state (defined as the state where it is unable to execute T) before time t [31].

Applying this method to our proposed cloud microservices model (Figure 4), we develop increasingly compound failure functions at each tier, culminating in the failure function $F_T(t)$ at the task tier. The failure functions at each tier are series/parallel arrangements of the failure functions of the relevant dependencies in the preceding (lower) tier.

At the lowest tier (the container tier), we define a uniform failure function $F(t)$ for each of n container instances of some microservice [6]. $F(t)$ reflects some real-world assessment of the failure behavior of the microservice container.

In our empirical testing, discussed in Sections 5 and 6, we elected to use the exponential function due to its constant hazard rate, reflecting the general case where no additional information is known about the propensity of a particular container to fail [6]. However,

any suitable function may be chosen, perhaps even an approximation of statistical failure behavior in cloud systems with a monitoring history available.

Having selected some function for $F(t)$, we can determine the failure function, $F_{MS}(t)$, at the next superior tier, which defines the probability of failure of a microservice comprising n redundant containers, as in (1) [6]. $F_{MS}(t)$ is found to be the product of the failure functions of the containers because the microservice will cease to provide service *iff* all redundant containers fail. While the failure functions for each microservice container, $F_i(t)$, are identical by virtue of the replication of the same microservice image, in order to adjust for the different times of instantiation of each container, the argument t is offset by t_{0i}, where t_{0i} is the time of instantiation of the microservice in terms of the global system time, as in (1).

$$F_{MS}(t) = \prod_{i}^{n} F_i(t - t_{0_i}) \tag{1}$$

Finally, we find the failure function of the task with respect to the failure functions $F_{MSm}(t)$ of its constituent microservices. In our example, we assume a simple sequence of calls to microservices (i.e., all microservices are required), yielding an in-series arrangement. Thus, we find the failure function of the task to be as in (2), expanded to (3) [6].

$$F_T(t) = 1 - \prod_{j}^{m} (1 - F_{MSj}(t)) \tag{2}$$

$$F_T(t) = 1 - \prod_{j}^{m} \left(1 - \prod_{i}^{n} F_i(t - t_{0_i}) \right) \tag{3}$$

4. Our Proposed Approach

This section presents our approach to managing the reliability of cloud-based systems using spot market cloud orchestration based on TBR and dynamic costing.

4.1. Dynamic Costing of Cloud Resources

There are several different costing models used by cloud service providers to charge their customers. In [7], the author identifies three costing models typical of the cloud computing market: pay-as-you-go (PAYG), on-demand (OD), and spot market (SM).

As per [7], the PAYG model uses a combination of a fixed fee and a variable cost per use of the cloud resource [7]. Under the PAYG model, the fixed fee component can be considered a down payment to ensure that the cloud resource will be available when required [7]. Equation (4) shows the total cost of a PAYG cloud resource, where f represents the fixed component, v represents the variable component, and τ represents the usage time.

$$C_{PAYG}(\tau) = f + v\tau \tag{4}$$

In contrast, the OD model, which is still based on a per-use charge, does not guarantee the availability of a resource [7]. Thus, it is suitable for tasks which are delay-tolerant [7]. Equation (5) shows the total cost of an OD resource.

$$C_{OD}(\tau) = v\tau \tag{5}$$

The complement to the OD model is the SM model, in which the cloud provider sells off access to idle resources that are not demanded by OD customers at a particular time [7]. SM resources are sold at significant discounts, but executing tasks may be interrupted at any time and the resource reallocated to an OD customer if demanded [7].

The total cost of a spot market resource is determined by the nature of the market pricing, i.e., whether the price per unit of time is set at the time of purchase, or whether it is variable, recomputed at each time slot used over the course of the task in accordance with the market fluctuations. In our model, we propose the latter, more general case, giving

the total cost of an SM resource as in (6), where S is a set of time slots in which the SM resource was used, and $\Phi(t)$ refers to the spot market price of the resource during the time slot t.

$$C_{SM}(S) = \sum_{i}^{|S|} \Phi(t_i),\ t_i \epsilon S \tag{6}$$

Thus, in contrast with the PAYG and OD models, the cost of resources in an SM model is dynamic, driven by various market forces including scarcity, demand, and global system congestion [7].

4.2. Quantifying the Cost of Failure

The task of achieving a reliable system is, in its essence, a trade-off between the cost of redundancy and the expected cost of failure. We define the expected cost of failure as the financial cost of a catastrophic system failure multiplied by the probability of that failure occurring. The addition of redundant components increases the cost of production (or usage, in the cloud context) while decreasing the probability of the failure occurring.

Applied to our proposed model in Section 3, we can define the expected cost of failure with respect to some task T before time t, $E_T(t)$, in terms of Ω, the financial cost of failure, and $F_T(t)$, the failure function of task T [6].

$$E_T(t) = \Omega F_T(t) \tag{7}$$

In closed systems, particularly electronic and physical systems, changing the level of component redundancy is often not possible or feasible after production (e.g., adding an additional engine to an existing airplane). Thus, in the design phase, a particular level of reliability is determined, possibly by contract stipulation, and the system is designed accordingly. Should some externality arise that varies the probability of failure, or indeed the cost of a catastrophic failure, it is not possible to take advantage of the change or mitigate it, as the case may be, by changing the level of redundancy.

However, reconfiguration of software systems post-deployment is much more feasible, with stateless cloud microservice containers being particularly suitable for this. Supposing the probability of failure to remain unchanged, if the cost of a catastrophic failure can be quantified at a reasonable frequency, the cloud system can be dynamically reconfigured to add or remove stateless microservice containers, keeping the expected cost of failure to an acceptable level and reducing the cost of idle redundancy.

In [6], we proposed a cloud orchestrator that added new microservice containers to the cloud until it reached an optimum configuration where the marginal cost of adding a new microservice container outweighed the benefit to the expected cost of failure. We showed empirically that this approach achieved a lower overall cost (the sum of the running cost and cost of catastrophic failures) than a fixed reliability approach [6]. However, this assumed a fixed model in which both the cost of catastrophic failure and the marginal cost of adding another microservice were constant. In this research, we propose a dynamic SM model based on free market forces which varies both these parameters.

4.3. Proposed Spot Market Orchestration Algorithm

Our proposed algorithm for cloud systems orchestration in a spot market context adjusts the redundancy of microservice containers in the system according to a periodically updated reassessment of market conditions (Algorithm 1).

Algorithm 1: Our proposed algorithm for Spot Market Orchestration for task T

```
1:  procedure OrchestrateCloud (Microservices, t)
2:      Delete failed microservice containers.
3:      Ensure each microservice has at least one non-failed container
4:       MaxUtility ← ∞
5:      while MaxUtility > 0 do
6:          SelectedMicroservice ← null
7:          MaxUtility ← 0
8:          for each ms ∈ Microservices do
9:              if U_ms(t) > 0 and U_ms(t) > MaxUtility then
10:                 SelectedMicroservice ← ms
11:                 MaxUtility ← U_ms(t)
12:             end if
13:         end for
14:         if MaxUtility > 0 then
15:             Replicate SelectedMicroservice
16:         end if
17:     end while
18:     while MaxUtility > 0 do
19:         SelectedContainer ← null
20:         MaxUtility ← 0
21:         for each ms ∈ Microservices do
22:             for each c ∈ ms.Containers do
23:                 if U'_c(t) > 0 and U'_c(t) > MaxUtility then
24:                     SelectedContainer ← c
25:                     MaxUtility ← U'_c(t)
26:                 end if
27:             end for
28:         end for
29:         if MaxUtility > 0 then
30:             Safely shut down SelectedContainer
31:         end if
32:     end while
33: end procedure
```

The algorithm permits two quantities to vary at runtime: the marginal cost of redundancy (i.e., the price of adding an additional microservice container at the current spot market price), Φ, and the cost of catastrophic failure, Ω. In response, the algorithm updates the level of redundancy in the cloud system until the reduction in the expected cost of failure is outweighed by the marginal cost of redundancy.

Similar to how each microservice may have a different failure function, as in (1), our approach considers that the marginal cost of each microservice may also be different, reflecting the particular characteristics and resource usage profile of the microservice. Thus, we define the marginal cost of redundancy in terms of each microservice independently as $\Phi_{MS}(t)$.

Given that each microservice contributes differently to the failure function of a task, $F_T(t)$, as in (2), and has a different marginal cost, $\Phi_{MS}(t)$, at each iteration of the algorithm, we replicate the microservice which provides the greatest improvement for the lowest cost.

We thus define a utility function, $U_{MS}(t)$, which quantifies the benefit of replicating a particular microservice container MS, as in (8), where $E(t)$ is the expected cost of failure given the present configuration of the cloud, $E'(t)$ is the expected cost of failure supposing an additional microservice container of type MS is added to the cloud, as per (7), and $\Phi_{MS}(\mathrm{t})$ is the marginal cost of adding the microservice container [6]. With the cost of catastrophic failure also being variable, this is expanded as in (9).

$$U_{MS}(t) = E(t) - E'(t) - \Phi_{MS}(\mathrm{t}) \tag{8}$$

$$U_{MS}(t) = \Omega(t)F_T(t) - \Omega'(t)F'_T(t) - \Phi_{MS}(\mathrm{t}) \quad (9)$$

From (9), the replication of microservice MS is worthwhile *iff* $U_{MS}(t) > 0$. Furthermore, the values of $U_{MS}(t)$ for each microservice in the task can be used to select which microservice to replicate at each iteration of the orchestration algorithm, selecting the microservice with the greatest utility, i.e., $MAX(U_{MS}(t))$.

Having used (9) to add redundancy to an under-replicated cloud configuration, we must also consider the complementary situation in which market forces leave the cloud in an over-replicated state with respect to the cost of failure, such as late at night when there are fewer users and consequently the financial cost of a failure during this period of time may be less.

Thus, we define a complementary utility function, $U'_C(t)$, which determines the improvement in overall expected cost if a redundant microservice container c is safely shut down and removed from the cloud (10).

$$U'_C(t) = \Omega(t)F_T(t) - \Omega'(t)F'_T(t) + \Phi_{MS}(\mathrm{t}) \quad (10)$$

Whereas $U_{MS}(t)$ is defined with respect to a microservice because any added microservice will, by definition, spawn as a new component, $U'_C(t)$ is defined with respect to a specific container, as the probability of failure for redundant containers of a particular microservice may differ depending on their instantiation time.

Equation (10) is similar to (9) in that it calculates the cost of failure before and after removing the container, but with the sign reversed on the spot price, representing the cost savings achieved by removing the superfluous container from the system.

Thus, Algorithm 1 consists of three key steps: pruning of failed microservice containers (thereby enabling the effective redundancy of the task to be calculated), addition of redundant microservice containers using (9) until there is no further utility, and removal of surplus containers using (10) until there is no further utility, as the case may be.

5. Empirical Testing

This section presents our experimental method and results.

5.1. Simulation Environment

To empirically test the performance of our proposed algorithm, we extended the open-source project Cloud Reliability Simulator, which we had previously developed in [6], adding the logic of our proposed algorithm (Algorithm 1) as a new *Orchestrator*.

Cloud Reliability Simulator is built using Python 3. It uses a discrete time simulation methodology to model the failure of cloud microservice containers [6,34]. The key components of the software are the *Simulator*, *Cloud*, *Microservice*, *MicroserviceContainer*, and *Orchestrator* classes (Figure 5).

The Simulator class is responsible for running the simulation environment, determining when to cause *MicroserviceContainers* to fail according to their failure probability functions. It also manages the collection and aggregation of data from the simulation. The *Cloud* class represents a particular cloud configuration being tested, which in our case is the analog of a task. The *Microservice* and *MicroserviceContainer* classes represent their namesakes in our model, as per Figure 4. Finally, the *Orchestrator* class drives the simulation, with its children implementing the desired orchestration functionality. In our case, we implemented the functionality of Algorithm 1 in the *SpotMarketOrchestrator* class.

We also added a *SpotMarketProvider* to represent the dynamic market behavior during the simulation. The *SpotMarketOrchestrator* queried the *SpotMarketProvider* regularly to determine the applicable spot price for the simulated cloud computing resource, which in our simulation was modeled as the execution time of the *MicroserviceContainer* classes.

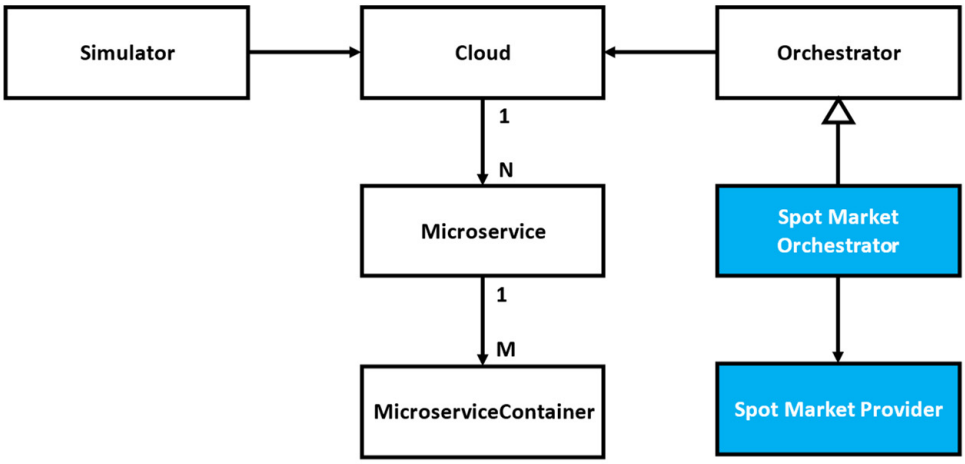

Figure 5. High-level architecture of the open-source Cloud Reliability Simulator, showing our proposed spot market functionality.

5.2. Experimental Design

Several randomized experiments were conducted under different conditions to empirically test our proposed orchestration algorithm (Algorithm 1) in comparison with a control orchestrator (Algorithm 2). The control orchestrator implemented a "naïve" algorithm that added or removed containers such that, at any given time, the probability of each microservice operating reliably approximated "nine-nines reliability" (99.9999999%).

Algorithm 2: The orchestration algorithm we used as a control for task T

```
procedure OrchestrateCloud (Microservices, t)
    Delete failed microservice containers.
    Ensure each microservice has at least one non-failed container
    for each ms ∈ Microservices do
        while P_failure(ms, t) ≥ 1.0×10^−9 do
            Replicate ms
        end while
        while P_failure(ms, t) < 1.0×10^−9 do
            Safely shut down a container from ms
        end while
    end for
end procedure
```

The same *SpotMarketProvider* was used by both the control and experimental orchestrators so that resultant costs could be validly compared. Only the experimental orchestrator adjusted its behavior in response to market conditions.

The cloud configuration for each of the experiments consisted of two microservices with exponential failure functions of the form $F_{ms}(t) = 1 - e^{-\lambda t}$. All experiments were conducted over 500 samples and repeated $n = 1000$ times, with the spot market conditions varied according to the design of each experiment.

In our experiments, we selected a value of 1 for the parameter λ as this gave a 99% probability of failure within 500 samples at the rate of 100 samples per second, thus approximating the exponential distribution within the execution time constraints of the simulator.

In each experiment, the orchestrators were run at two frequencies: once at the sampling rate, modeling a system where failures are immediately detected and remedied, and again at a lower frequency of once per ten samples, modeling a system where diagnostics are run periodically to detect failures.

We used a simplified model of a spot market in the experiments which was composed of two parameters. These parameters gave the marginal cost of running a microservice, $\Phi_{MS}(t)$, according to (11), where $\Phi(t)$ referred to a global "spot price" per second of

execution and α referred to a static multiplier assigned to each microservice to represent its resource usage intensity per unit time.

$$\Phi_{MS}(t) = \alpha\Phi(t) \tag{11}$$

5.2.1. Experiment 1: Varying the Cost of Failure

The objective of this experiment was to compare the behavior of the experimental and control algorithms in a market with a constant spot price of 1.00 but varying costs of failure from 1 to 100,000 per second (Figures 6 and 7). This scenario models the expected behavior of an enterprise cloud serving users for whom the majority of activity, and thus the highest impact of failure, occurs during business hours. The *SpotMarketProvider* in this experiment varied the cost of failure according to the time of day, with the peak between 09:00 and 17:00.

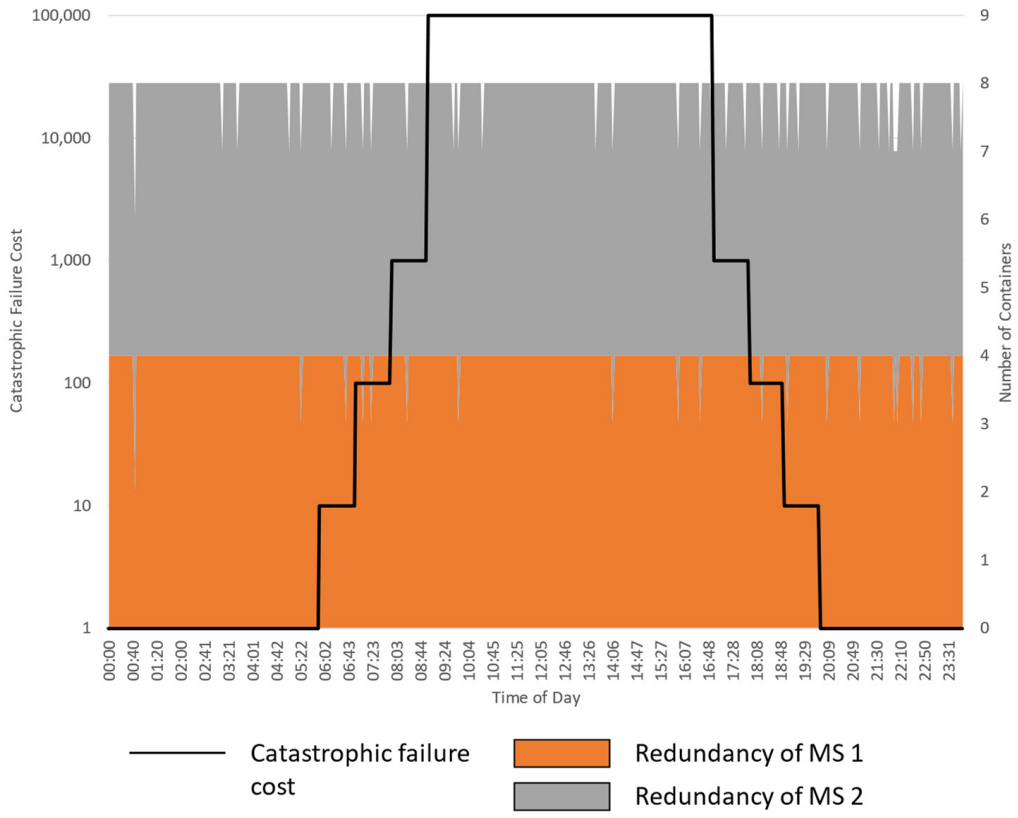

Figure 6. Control algorithm behavior in Experiment 1 with immediate detection and remediation of failures.

Figure 6 shows an example run of the control algorithm, while Figure 7 shows an example run of the experimental algorithm. In both figures, the redundancy of each of the microservices is shown with respect to the cost of catastrophic failure.

The results from each of the immediate and delayed configurations of the orchestrator are given in Tables 1 and 2.

Table 1. Aggregate results of experiments with immediate failure detection and remediation.

	Orchestrator	Mean Running Cost	Mean Actual Cost of Failures	Mean Total Cost
Exp. 1	Experimental	0.94	0.06	1.00
	Control	1.60	0.00	1.60
Exp 2.	Experimental	13,438	3288	16,726
	Control	53,590	0.00	53,590

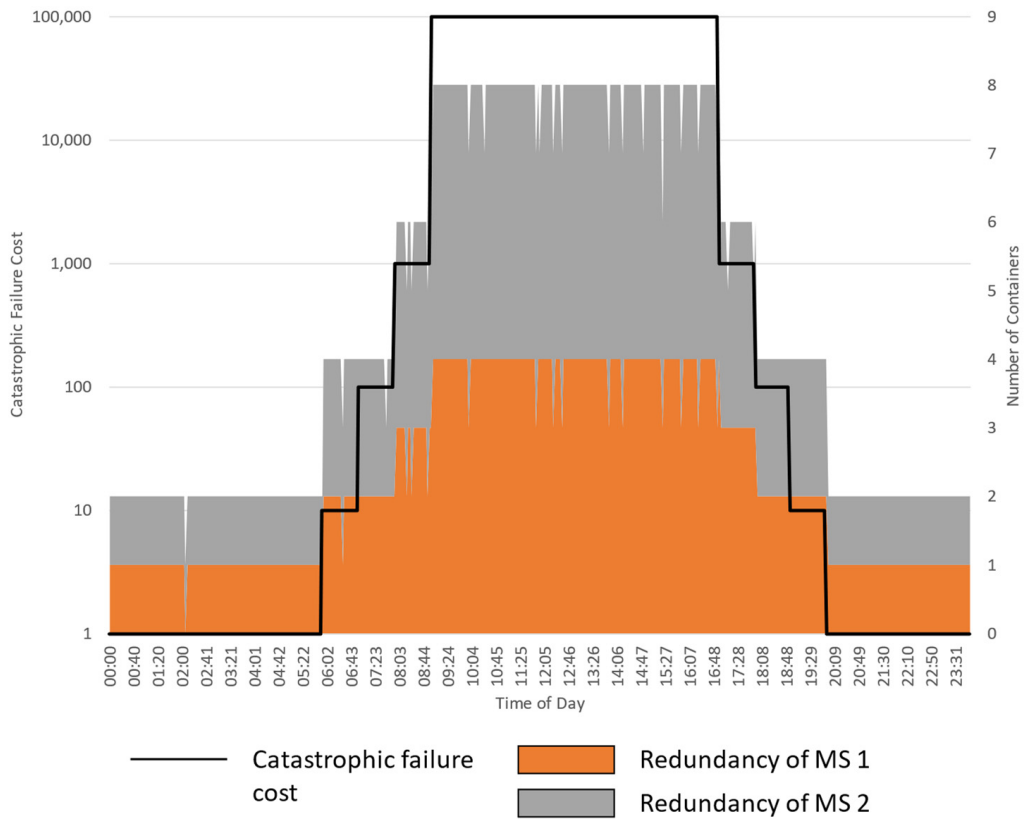

Figure 7. Experimental algorithm behavior in Experiment 1 with immediate detection and remediation of failures.

Table 2. Aggregate results of experiments with delayed failure detection and remediation.

	Orchestrator	Mean Running Cost	Mean Actual Cost of Failures	Mean Total Cost
Exp. 1	Experimental	1.62	0.02	1.64
	Control	3.05	0.00	3.05
Exp 2.	Experimental	27,098	1424	28,522
	Control	102,067	0.00	102,067

5.2.2. Experiment 2: Varying the Spot Price

In this experiment, the cost of catastrophic failure was kept constant while the spot price of execution time was varied. This scenario models the expected behavior of a cloud system which must operate reliably under continuous use while the spot market fluctuates.

The cost of catastrophic failure was set at a constant 100,000 per second, while the spot price was varied in a range from 1 to 100,000 per second. The spot price was varied according to the same schedule and values as the cost of failure in Experiment 1 for comparison.

Figure 8 shows an example run of the control algorithm, while Figure 9 shows an example run of the experimental algorithm. In both figures, the redundancy of each of the microservices is shown with respect to the spot market price.

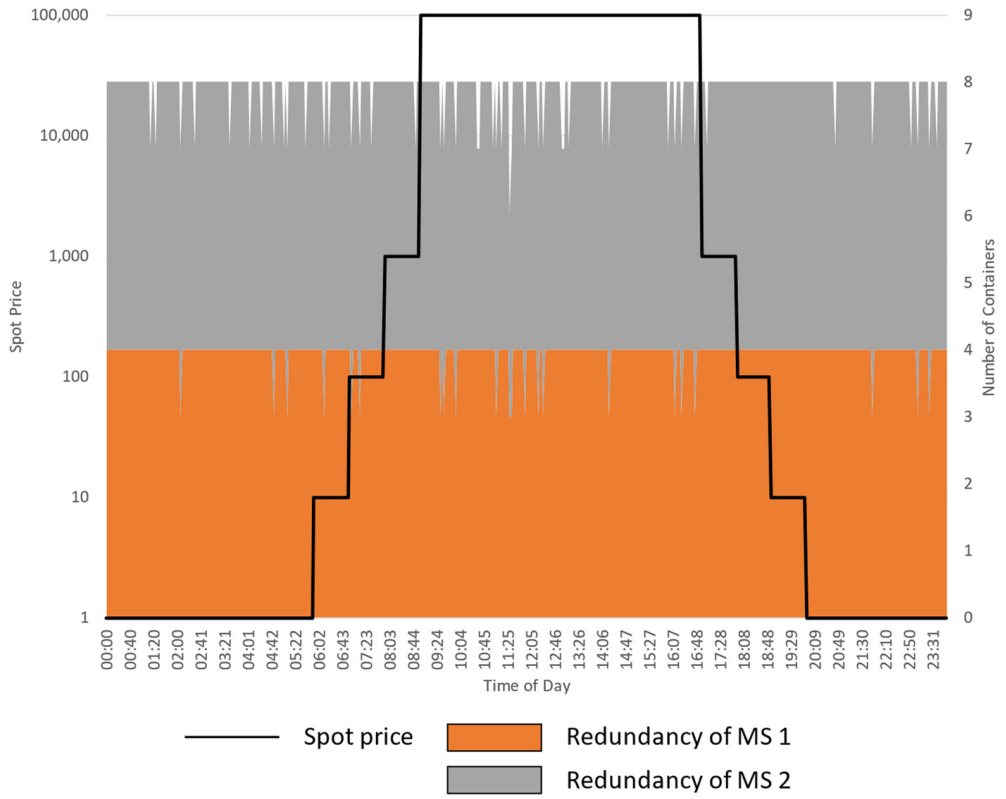

Figure 8. Control algorithm behavior in Experiment 2 with immediate detection and remediation of failures.

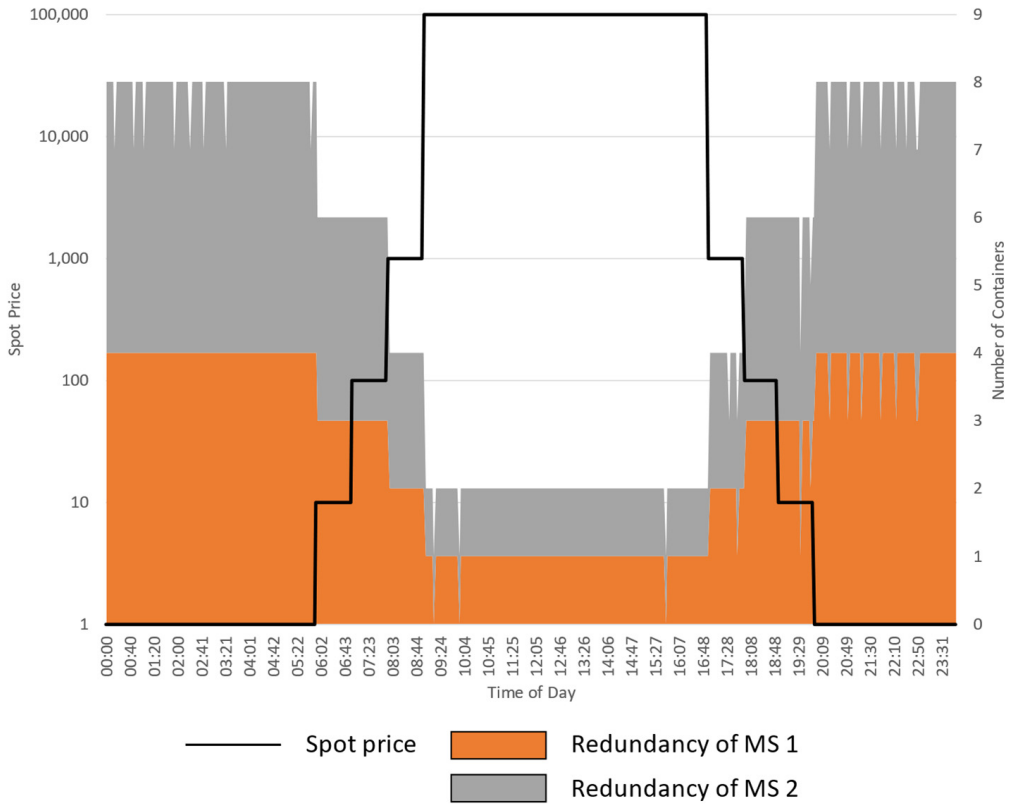

Figure 9. Experimental algorithm behavior in Experiment 2 with immediate detection and remediation of failures.

The results from each of the immediate and delayed configurations of the orchestrator are given in Tables 1 and 2.

6. Discussion

The results of the aggregate experiments given in Tables 1 and 2 demonstrate that our proposed algorithm achieved a lower total cost (the sum of the running and failure costs) across the scenarios tested.

In the first experiment, where the cost of failure was varied, the total cost was reduced by between 37% and 46%, while in the second experiment, where the spot price was varied, the total cost was reduced by between 68% and 72%.

We propose that this improvement is a reflection of the market-following behavior of our proposed algorithm, which consistently maintains the level of redundancy towards an optimum level with respect to the spot price and cost of catastrophic failure.

By the definitions of the utility functions in (9) and (10), we propose that the specific level of redundancy selected by the algorithm is influenced by the relationship between the cost of catastrophic failure and the spot price.

Additionally, assuming the spot market price to be well-known, the performance of our proposed algorithm relative to the control is influenced by the level of accuracy with which it can predict the cost of catastrophic failure. In practice, an incorrect estimation of this parameter may result in a non-optimal outcome, i.e., either excessive or insufficient redundancy in the system.

Another factor we considered was the latency in detecting a failure. As shown in Table 2, a latency of up to 10 samples was introduced to each experiment configuration (the specific latency being probabilistically determined by the time of failure of a container within this range). We made two observations with regard to this parameter.

Firstly, our algorithm accounted for the length of time between orchestrator runs when computing the expected cost of failure. Thus, we observed a modest increase in the running cost and a decrease in the failure cost, possibly the result of a higher level of redundancy used to compensate for the increased likelihood of a failure occurring given the longer time period.

Secondly, the magnitude of the improvement in total cost was greater when a latency was applied. Table 2 shows that even though the control algorithm had a much lower failure cost, its running cost was much greater than the running cost of our proposed algorithm, thereby accounting for the difference in total cost.

This is perhaps the strongest argument for the utility of our proposed algorithm. If the cost of a failure can be suitably quantified, our proposed algorithm can achieve a lower total cost in comparison with an algorithm based on a static reliability requirement through its market-following behavior.

7. Conclusions

In this paper, we proposed a novel cloud orchestration algorithm that seeks to achieve an optimal solution for reliability relative to the spot market price and cost of failure.

Using a Task-Based Redundancy (TBR) model, we developed an algorithm for the orchestrator which, in comparison with a static reliability control algorithm, achieved improvements of between 37% and 46% in total cost for clouds where failures were immediately identified and remedied, and between 68% and 72% for clouds where there was latency in detecting and remedying the failure.

Thus, in cloud environments where the financial cost of a failure can be approximated with reasonable accuracy, the results indicate that our algorithm can be used to improve the cloud redundancy configuration, giving a lower overall total cost in comparison with an algorithm based on a static reliability requirement.

Future research may seek to extend our model to include more complex spot market behavior through the creation of new *Orchestrator* subclasses (Figure 5), and to investigate the effects of different latency periods on the magnitude of the improvement. Another

avenue of further inquiry may be to create more complex probabilistic failure models by overriding the failure functions of the microservices. Further studies may also consider the possibility of combining PAYG, OD, and SM models within the same cloud. In this study, we compared our proposed approach with a control algorithm that used a static reliability level, typical of scenarios where a service level is specific in an SLA. However, future research may explore comparing our proposed algorithm with other non-TBR algorithms, including consideration of their comparative performance across such factors as efficiency, scalability, adaptability to varying workloads, and other factors.

Author Contributions: V.O. and B.S. contributed equally to the manuscript. All authors have read and agreed to the published version of the manuscript.

Funding: V.O. was supported in this research by an Australian Government Research Training Program Scholarship.

Data Availability Statement: The source code for Cloud Reliability Simulator is publicly available at [34].

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

References

1. Ardagna, D.; Casale, G.; Ciavotta, M.; Perez, J.; Wang, W. Quality-of-Service in Cloud Computing: Modelling Techniques and Their Applications. *J. Internet Serv. Appl.* **2014**, *5*, 11. [CrossRef]
2. Bhardwaj, A.; Rama Krishna, C. Virtualization in Cloud Computing: Moving from Hypervisor to Containerization—A Survey. *Arab. J. Sci. Eng.* **2021**, *46*, 8585–8601. [CrossRef]
3. Gan, Y.; Delimitrou, C. The Architectural Implications of Cloud Microservices. *IEEE Comput. Archit. Lett.* **2018**, *17*, 155–158. [CrossRef]
4. Yang, H.; Kim, Y. Design and Implementation of High-Availability Architecture for IoT-Cloud Services. *Sensors* **2019**, *19*, 3276. [CrossRef] [PubMed]
5. Pahl, C.; Brogi, A.; Soldani, J.; Jamshidi, P. Cloud Container Technologies: A State-of-the-Art-Review. *IEEE Trans. Cloud Comput.* **2019**, *7*, 677–692. [CrossRef]
6. O'Neill, V.; Soh, B. Orchestrating the Resilience of Cloud Microservices Using Task-Based Reliability and Dynamic Costing. In Proceedings of the IEEE Conference on Computer Science and Data Engineering, Gold Coast, Australia, 18–20 December 2022. [CrossRef]
7. Dimitri, N. Pricing Cloud IaaS Computing Services. *J. Cloud Comput.* **2020**, *9*, 14. [CrossRef]
8. Tao, Z.; Xia, Q.; Hao, Z.; Li, C.; Ma, L.; Yi, S.; Li, Q. A Survey of Virtual Machine Management in Edge Computing. *Proc. IEEE* **2019**, *107*, 1482–1499. [CrossRef]
9. da Silva, V.; Kirikova, M.; Alksnis, G. Containers for Virtualization: An Overview. *Appl. Comput. Syst.* **2018**, *23*, 21–27. [CrossRef]
10. Marquez, J.; Castillo, M. Performance Comparison: Virtual Machines and Containers Running Artificial Intelligence Applications. In *Information Technology and Systems*; Rocha, A., Ferras, C., Lopez-Lopez, P., Eds.; Springer: Cham, Switzerland, 2021. [CrossRef]
11. Zhang, Q.; Liu, L.; Pu, C.; Dou, Q.; Wu, L.; Zhou, W. A Comparative Study of Containers and Virtual Machines in Big Data Environment. In Proceedings of the IEEE 11th International Conference on Cloud Computing, San Francisco, CA, USA, 2–7 July 2018. [CrossRef]
12. Gillani, K.; Lee, J. Comparison of Linux Virtual Machines and Containers for a Service Migration in 5G Multi-Access Edge Computing. *ICT Express* **2020**, *6*, 1–2. [CrossRef]
13. Shirinbab, S.; Lunberg, L.; Casalicchio, E. Performance Comparison Between Scaling of Virtual Machines and Containers Using Cassandra NoSQL Database. In Proceedings of the 10th International Conference on Cloud Computing, GRIDs, and Virtualization, Venice, Italy, 5–9 May 2019.
14. Balalaie, A.; Heydarnoori, A.; Jamshidi, P. Microservices Architecture Enables DevOps: Migration to Cloud-Native Architecture. *IEEE Softw.* **2016**, *33*, 42–52. [CrossRef]
15. Megargel, A.; Shankararaman, V.; Walker, D. Migrating from Monoliths to Cloud-Based Microservices: A Banking Industry Example. In *Software Engineering in the Era of Cloud Computing*; Ramachandran, M., Mahmood, Z., Eds.; Springer: Cham, Switzerland, 2020. [CrossRef]
16. Singleton, A. The Economics of Microservices. *IEEE Cloud Comput.* **2016**, *3*, 16–20. [CrossRef]
17. Mohamed, M.; Engel, R.; Warke, A.; Berman, S.; Ludwig, H. Extensible Persistence as a Service for Containers. *Future Gener. Comput. Syst.* **2019**, *87*, 10–20. [CrossRef]
18. Colman-Meixner, C.; Develder, C.; Tornatore, M.; Mukherjee, B. A Survey on Resiliency Techniques in Cloud Computing Infrastructure and Applications. *IEEE Commun. Surv. Tutor.* **2016**, *18*, 2244–2281. [CrossRef]
19. Johnson, B. *Design and Analysis of Fault-Tolerant Digital Systems*; Addison-Wesley: Reading, MA, USA, 1989.

20. Louati, T.; Abbes, H.; Cerin, C. LXCloudFT: Towards High Availability, Fault-Tolerant Cloud System Based Linux Containers. *J. Parallel Distrib. Comput.* **2018**, *122*, 51–69. [CrossRef]
21. Florin, R.; Ghazizadeh, A.; Ghazizadeh, P.; Olariu, S.; Marinescu, D. Enhancing Reliability and Availability Through Redundancy in Vehicular Clouds. *IEEE Trans. Comput.* **2019**, *9*, 1061–1074. [CrossRef]
22. Ahmed, F.; Abdul Majid, M. Towards Agent-Based Petri Net Decision Making Modelling for Cloud Service Composition: A Literature Survey. *J. Netw. Comput. Appl.* **2019**, *130*, 14–28. [CrossRef]
23. Liu, Z.; Fan, G.; Yu, H.; Chen, L. An Approach to Modelling and Analyzing Reliability for Microservice-Oriented Cloud Applications. *Wirel. Commun. Mob. Comput.* **2021**, *2021*, 5750646. [CrossRef]
24. Ha, W. Reliability Prediction for Web Service Composition. In Proceedings of the 13th International Conference on Computational Intelligence and Security, Hong Kong, China, 15–18 December 2017. [CrossRef]
25. Li, C.; Song, M.; Zhang, M.; Luo, Y. Effective Replica Management for Improving Reliability and Availability in Edge-Cloud Computing Environment. *J. Parallel Distrib. Comput.* **2020**, *143*, 107–128. [CrossRef]
26. Pham, T.-P.; Ristov, S.; Fahringer, T. Performance and Behavior Characterization of Amazon EC2 Spot Instances. In Proceedings of the 11th IEEE International Conference on Cloud Computing, San Francisco, CA, USA, 2–7 July 2018. [CrossRef]
27. Baughman, M.; Hass, C.; Wolski, R.; Foster, I.; Chard, K. Predicting Amazon Spot Prices with LSTM Networks. In Proceedings of the 9th Workshop on Scientific Cloud Computing, Tempe, AZ, USA, 11 June 2018. [CrossRef]
28. Spot VMs. Available online: https://cloud.google.com/spot-vms (accessed on 22 August 2023).
29. Azure Spot Virtual Machines. Available online: https://azure.microsoft.com/en-us/products/virtual-machines/spot (accessed on 22 August 2023).
30. Kumar, D.; Baranwal, G.; Raza, Z.; Vidyarthi, D.P. A Survey on Spot Pricing in Cloud Computing. *J. Netw. Syst. Manag.* **2017**, *26*, 809–856. [CrossRef]
31. O'Neill, V.; Soh, B. Improving Fault Tolerance and Reliability of Heterogeneous Multi-Agent Systems Using Intelligence Transfer. *Electronics* **2022**, *11*, 2724. [CrossRef]
32. Li, Q.; Li, B.; Mercati, P.; Illikkal, R.; Tai, C.; Kishinevsky, M.; Kozyrakis, C. RAMBO: Resource Allocation for Microservices Using Bayesian Optimization. *IEEE Comput. Archit. Lett.* **2021**, *10*, 46–49. [CrossRef]
33. Li, S.; Zhang, H.; Jia, Z.; Li, Z.; Zhang, C.; Li, J.; Gao, Q.; Ge, J.; Shan, Z. A Dataflow-Driven Approach to Identifying Microservices From Monolithic Applications. *J. Syst. Softw.* **2019**, *157*, 110380. [CrossRef]
34. Cloud Reliability Simulator Source Code. Available online: https://github.com/vyas-oneill/cloud_reliability_sim (accessed on 30 July 2023).