IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 26 October 2023, accepted 4 December 2023, date of publication 7 December 2023, date of current version 18 December 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3340195

RESEARCH ARTICLE

Model-Driven Dependability and Power Consumption Quantification of Kubernetes-Based Cloud-Fog Continuum

IURE FÉ[1], TUAN ANH NGUYEN[2], (Member, IEEE), ANDRÉC. B. SOARES[3], SEOKHO SON[4], EUNMI CHOI[5], DUGKI MIN[6], JAE-WOO LEE[7], AND FRANCISCO AIRTON SILVA[1]

[1]Laboratory of Applied Research to Distributed Systems (PASID), Federal University of Piauí (UFPI), Picos, Piauí 64606-000, Brazil
[2]Konkuk Aerospace Design-Airworthiness Research Institute (KADA), Konkuk University, Seoul 05029, South Korea
[3]Distributed Systems and Network Computer Laboratory (DisNeL), Federal University of Piauí (UFPI), Teresina, Piauí 64049-550, Brazil
[4]Electronics and Telecommunications Research Institute (ETRI), Daejeon 34129, South Korea
[5]School of Software, College of Computer Science, Kookmin University, Seoul 02707, South Korea
[6]Department of Computer Science and Engineering, College of Engineering, Konkuk University, Seoul 05029, South Korea
[7]Department of Aerospace Information Engineering, Konkuk University, Seoul 05029, South Korea

Corresponding authors: Tuan Anh Nguyen (anhnt2407@konkuk.ac.kr), Dugki Min (dkmin@konkuk.ac.kr), and Jae-Woo Lee (jwlee@konkuk.ac.kr)

This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education under Grant 2020R1A6A1A03046811 and Grant 2021R1A2C2094943.

ABSTRACT System dependability is pivotal for the reliable execution of designated computing functions. With the emergence of cloud-fog computing and microservices architectures, new challenges and opportunities arise in evaluating system dependability. Enhancing dependability in microservices often involves component replication, potentially increasing energy costs. Thus, discerning optimal redundancy strategies and understanding their energy implications is crucial for both cost efficiency and ecological sustainability. This paper presents a model-driven approach to evaluate the dependability and energy consumption of cloud-fog systems, utilizing Kubernetes, a container application orchestration platform. The developed model considers various determinants affecting system dependability, including hardware and software reliability, resource accessibility, and support personnel availability. Empirical studies validate the model's effectiveness, demonstrating a 22.33% increase in system availability with only a 1.33% rise in energy consumption. Moreover, this methodology provides a structured framework for understanding cloud-fog system dependability, serves as a reference for comparing dependability across different systems, and aids in resource allocation optimization. This research significantly contributes to the efforts to enhance cloud-fog system dependability.

INDEX TERMS Cloud-fog continuum, dependability, Kubernetes, stochastic modeling.

I. INTRODUCTION

The cloud-fog continuum represents a theoretical model delineating the various strata of computing resources within distributed environments [1], [2], [3]. Within this continuum, the cloud provides centralized management services that offer high scalability and availability to the systems [4]. Interposed between the cloud and the edge, the fog layer facilitates edge computing, signifying intermediate computing resources in proximity to the edge. These resources are customarily deployed in applications demanding low latency, encompassing domains such as the Internet of Things (IoT), augmented reality, virtual reality, and real-time analytics. The integration of the cloud-fog continuum has been demonstrated to enhance the overall system's performance, reliability, and security, leveraging the virtues of both cloud and fog computing [1], [5], [6].

Kubernetes, an open-source platform, facilitates the automated management of containerized applications across diverse computing environments [7]. It supports various

The associate editor coordinating the review of this manuscript and approving it for publication was Usama Mir.

deployment models, including on-premises, cloud, and hybrid setups, and is instrumental in building and overseeing large-scale microservices architectures. When integrated with extensions like KubeEdge, Kubernetes becomes highly suitable for the cloud-fog spectrum [8]. Microservices-based systems exhibit growing complexity due to numerous components and intricate interplay between applications and the infrastructure [9]. Deployment in Kubernetes requires detailed configuration, including specific allocations, infrastructure checks, storage protocols, and more. Additionally, defining support teams and diverse hardware configurations becomes crucial, particularly in cloud-fog integration. Traditional methods mainly enhance reliability and availability via redundant resources [10]. However, the potential energy implications of redundancy cannot be overlooked, especially considering operational costs and environmental considerations. Information and communication technology accounts for about 2% of global energy use, expected to soar to 14% by 2040 without intervention [11], [12], [13]. Therefore, dependability-enhancing techniques should be energy-conscious, aiming to judiciously determine necessary replications with environmental impact in mind. Achieving desired dependability levels in such complex systems is challenging [1]. A mere high-level architecture assessment may yield inaccurate results for diverse applications. A comprehensive evaluation should consider underlying infrastructure impacts on reliability and energy consumption, including support capabilities, repair duration, and the dynamic relationship between microservices and infrastructure. The primary impetus for this research lies in addressing the planning challenges of creating reliable, eco-friendly Kubernetes microservices systems, factoring in intricate interactions with the foundational cloud and fog infrastructure.

Improvements through empirical measurements in complex environments can be challenging due to potential economic limitations or the demand for results that generalize across diverse scenarios. Model-based methods have shown their effectiveness in strategizing microservices and cloud systems [14], [15], [16], facilitating quantitative validation and adaptability across different situations. Nonetheless, a standardized approach to dependability modeling, especially when assessing the energy consumption impacts of various strategies in cloud-fog systems, is urgently needed. Quantitative models, particularly those using stochastic methods, are widely employed to examine the dependability of Kubernetes-based cloud-fog systems [14], [15], [16], [17], [18], [19]. These mathematical frameworks, representing systems' random behaviors, are instrumental in predicting impacts on established infrastructures or in designing new applications. A pivotal aspect of dependability in Kubernetes systems is managing failures, which requires understanding Kubernetes' inherent recovery mechanisms and associated response times. Given the intricacy and multifaceted components of these systems, continued research is vital. This research should aim to create innovative approaches and tools to bolster cloud-fog system dependability, while emphasizing energy conservation and environmental responsibility.

The field of improving dependability and reducing energy consumption in microservices is extensive and complex [20], [21], [22]. Major research efforts have been directed towards implementing fault tolerance techniques. This includes adopting a hybrid methodology combining both active and passive approaches to enhance availability [23], and creating a fault tolerance framework based on replicated Pods [24]. There has also been a significant focus on dependability planning using various models [15], [16], [25], [26] and investigating strategies for dependability enhancement through experiments [17], algorithmic methods [27], and modeling [28]. Simultaneously, several studies have ventured into utilizing models and frameworks to strategize and optimize energy consumption [22], [29], [30].

The seminal contribution of the present work resides in the formulation of comprehensive analytical models elucidating the infrastructure of systems predicated on Kubernetes. The adopted hierarchical modeling strategy encapsulates high-level elements and the underlying low-level infrastructure, inclusive of hardware configurations and application and container specifications within Pods. The high-level constituents depict nodes, processes, Pods, support structures, and behaviors. This hierarchical paradigm facilitates the facile generalization of the model across various infrastructures, enabling the identification of potential bottlenecks within the sub-components of the systems. The model is devised to compute availability, reliability, support utilization, the probability of support scarcity, and energy consumption, thereby empowering system administrators to discern the configuration requisite for redundant resources to attain desired levels of dependability concomitant with a reduction in electrical consumption.

The specific contributions of this endeavor are delineated as follows:

- An examination of the impact on energy consumption in strategies deployed to enhance Kubernetes dependability. This encompasses the replication of compute nodes or Pods within systems comprising numerous microservices and compute nodes, with a concomitant consideration of energy and environmental costs in delineating dependability-oriented architectural improvements.
- The introduction of models to facilitate the management of system support teams, acknowledging that the potential number of simultaneous repairs will influence availability and must be congruent with the system's scale.
- An analytical evaluation of the behavior of these metrics under diverse redundancy strategies, enabling the assessment of cost-efficacy in augmenting availability relative to the escalation in energy consumption, and assisting in the proper dimensioning of support teams.
- The application of a sensitivity analysis technique in an incremental fashion to attain predefined levels

of availability, guiding system managers to identify pivotal components necessitating improvement, while concurrently monitoring other metrics.

The implications and findings emanating from this study are profound:

- The availability and reliability of Kubernetes-based systems within the cloud/fog continuum are profoundly influenced by the allocation of Pods, necessitating demand-proportional Pod redundancy strategies in high-demand systems.
- The strategy of redundant Pod addition must be meticulously dimensioned with demand to ensure that availability benefits are commensurate with the elevation in energy consumption.
- Concurrent support in larger systems with insufficient support staff may culminate in extended repair time, mandating proper sizing of support teams in accordance with Pod demand.
- The synergistic utilization of models and sensitivity analysis may facilitate the planning of cloud and fog Kubernetes systems, delineating the elements that warrant prioritization to achieve predefined availability levels.

To our best understanding, this paper introduces, for the first time, a holistic and exhaustive modeling and evaluation of dependability and energy consumption within a Kubernetes-based cloud-fog continuum. This encompasses a consideration of multifarious fine-grained operational policies at an incipient stage of contemporary research on operational planning and efficiency evaluation of cloud-fog systems.

The structure of this paper is organized as follows: Section II delineates related works. Section III explicates the Kubernetes architecture. Section IV presents and elucidates the generated models and corresponding metrics. Section V practically illustrates use cases pertinent to system administration. Finally, Section VI articulates the conclusions and contemplates potential future work.

II. RELATED WORK

This section delves into the existing literature relevant to the core subject of this research. The primary focus of the reviewed literature centers on the orchestration and optimization of metrics within microservices' architecture. Initially, attention was given to studies that strengthen the dependability planning of these systems, especially those predicting metric behavior in theoretical frameworks. Subsequently, we analyzed works aimed at improving system dependability using different methodologies or model-based strategies. Finally, we examined research highlighting metrics related to the energy consumption attributes of specific architectural designs or strategies.

Within the realm of Dependability Planning, several notable contributions stand out for their innovative methodologies in orchestrating availability and reliability in microservice ecosystems. Jagadeesan et al. [14] introduce a microservice modeling framework emphasizing the impact of individual service failures on the overall system. This is achieved using Continuous-Time Markov Chains (CTMCs) and the Probabilistic Symbolic Model Checker (PRISM) to model inter-service communication, thus addressing potential cascade failures. Khazaei et al. [15] present an assessment of capacity provisioning for microservice platforms, utilizing a model to guide capacity planning. Liu et al. [16] offer a model focused on enhancing service quality by identifying and addressing critical microservices, assessing request and invocation relationship frequency. Pietrantuono et al. [25] develop a reliability estimation method for microservice architectures, considering frequent updates, service interaction variations, and usage fluctuations, using monitoring data and an analytical model to determine failure probabilities. Notably, Nguyen et al. [18] put forth a model-based approach for planning availability, performance, and energy consumption in software-defined networks, delivering key metrics in these domains.

Efforts to enhance dependability have been manifold. Liu et al. [31] introduce a reliability model using Predicate Petri net (PrT net) for hospital information systems, assessing various redundancy and circuit breaker strategies to improve system reliability. Vayghan et al. [17] outline potential availability-centric architectures for deploying stateful microservices on Kubernetes, underscoring the insufficiency of mere repair actions for high availability. They propose an integrated state controller on Kubernetes for concurrent replication and repair. Alwis et al. [27] address challenges in achieving scalability, availability, and efficiency during the transition from monolithic to microservice systems, suggesting a method based on queue theory and business object relationship analysis. Khatami et al. [28] demonstrate a high-availability system using Kubernetes and distributed storage, emphasizing the system's resilience even during node failures. In another study, Liu et al. [26] describe a method for modeling and optimizing cloud-hosted microservice application reliability using a hierarchical model in Predicate Petri Net (PPN), aiming for maximal reliability and cost-effectiveness. Lyu et al. [32] compare a monolithic power generation system with a Kubernetes-based microservices architecture, presenting an algorithm using the Mixed Integer Linear Programming (MILP) model for resource management, resulting in enhanced reliability and reduced costs. Lastly, Jeffery et al. [24] advocate for a new Kubernetes etcd architecture prioritizing performance and availability, especially for performance-sensitive edge applications.

In the realm of Electric Power Management, several pivotal works stand out. Jazayeri et al. [33] introduce a Markov-based analytical model to determine the best module execution location, balancing performance with energy consumption. Menouer et al. [22] detail a Kubernetes container scheduling optimization strategy, leveraging CloudSim Plus for evaluations under diverse container loads, emphasizing both performance and energy efficiency. Xu et al. [30] focus on reducing carbon footprints by maximizing renewable

TABLE 1. Related work comparison.

Paper	Context	Strategy	Metric	Hierarchical Modeling	Power and Dependability Planning	Metric Maximization	Cloud/Fog Infrastructure
[14]	Dependability Planning	CTMC model	Availability, Downtime	✗	✗	✗	✓
[15]	Dependability Planning	CTMC model, PMSM model	Response Time, Availability, Utilization, Expected Number of Containers	✓	✗	✗	✓
[18]	Dependability Planning	Petri Net model	Availability, Response Time, Throughput, and Power Consumption	✓	✓	✗	✗
[16]	Dependability Planning	Petri Net model	Reliability	✗	✗	✓	✓
[25]	Dependability Planning	Analytical Modeling	Reliability	✗	✗	✗	✓
[31]	Dependability Improvement	Predicate Petri nets model	Reliability	✓	✗	✗	✗
[26]	Dependability Improvement	Predicate Petri nets model	Reliability	✓	✗	✗	✓
[17]	Dependability Improvement	New State Controller	Availability, Outage Time	✗	✗	✗	✓
[27]	Dependability Improvement	Queueing Model	Execution Time, Availability, Resource Usage	✓	✗	✗	✓
[28]	Dependability Improvement	Architecture Change	MTBF, MTTR, Availability	✗	✗	✓	✓
[32]	Dependability Improvement	MILP	Availability, Cost, Resource Usage	✗	✗	✓	✓
[24]	Dependability Improvement	Measurement	Latency, Requests per second	✗	✗	✓	✓
[33]	Electric Consumption Planning	CTMC model, Analytical Modeling	Delay, Electric Consumption, Network Usage	✗	✗	✓	✓
[22]	Electric Consumption Planning	Analytical Modeling, CloudSim	Electric Consumption, Average Number of Containers	✗	✗	✓	✓
[30]	Electric Consumption Planning	Analytical Modeling	Electric Consumption, Average Number of Containers	✗	✗	✓	✓
[29]	Electric Consumption Planning	Test Framework	Descartes Modeling Language (DML), Energy Efficiency	✗	✗	✓	✓
[19]	Electric Consumption Planning	CloudSim	Energy Consumption, Quality of Service	✗	✗	✓	✓
[20]	Electric Consumption Planning	Measurement	Energy Consumption, Latency	✗	✗	✓	✗
[21]	Electric Consumption Planning	Simulation	Energy Consumption, Utilization	✗	✗	✓	✓
This Work	Dependability Planning, Dependability Improvement, Electric Consumption Planning	DRBD, CTMC, Petri Net, Sensitivity Analysis	Availability, Reliability, Electric Consumption, Resource Usage, Expected Number of Containers	✓	✓	✓	✓

energy usage, using electric consumption models based on VM resource utilization to achieve performance goals while minimizing environmental impact. Kistowski et al. [29] present a tool for rigorous microservice application testing, facilitating performance modeling, cloud resource management, and energy efficiency analyses. Murtaza et al. [19] propose an optimized fog task allocation method, analyzing both electric consumption and service quality. Hou et al. [20] offer a comprehensive method to enhance both latency and energy metrics in microservices-based systems. Lastly, Valera et al. [21] develop a simulator designed to reduce energy consumption in microservices systems without sacrificing quality, providing tools for system planning with these attributes in mind.

Table 1 contrasts the key features of this research with related works. While there's a thematic overlap in the contexts and metrics between this study and others, the combined examination of dependability and energy consumption remains unique to this work. Our analysis reveals an inherent interdependence between these metrics, emphasizing their significance for system administrators. Distinctively, our research provides comprehensive hierarchical models of the entire infrastructure, allowing for the pinpointing of potential bottlenecks and offering multiple optimization pathways. Although we, like other studies, suggest a method for enhancing dependability, our approach is multidimensional, considering secondary metrics. Additionally, we introduce metrics geared towards support team scaling, a crucial aspect in managing component-rich systems, as highlighted in [17], but with a distinct focus on metric generation.

III. SYSTEM ARCHITECTURE

Microservice architectures, characterized by an intricate web of interlinked containers and applications, can dynamically replicate high-demand components to meet increased system demand. Figure 1 illustrates a typical microservice architecture using a Kubernetes cluster. This setup includes both physical elements and primary processes, collectively hosting a range of applications. Within Kubernetes, the cluster divides into the Control Plane and Worker nodes, with the latter responsible for running containerized applications [34]. Containers are grouped into Pods, cohesive units containing one or more containers, structured as directed by the cluster manager. As shown in Figure 1, App 01 has two sub-applications within two containers, yet together they form a single Pod. In contrast, App 02 resides entirely within a container in a Pod.

In a Kubernetes cluster framework, Worker Nodes execute Pods that encapsulate various applications. Each cluster requires at least one Worker and one Control Plane node, with Worker Nodes typically outnumbering their counterparts and showcasing diverse configurations [34]. For instance, in a cloud environment, Worker components might possess superior capacity and reliability than those in a fog layer, as depicted in Figure 1. Another configuration could involve partial system provisioning by an external provider, such as renting virtual machines. In this setup, the cluster manager solely configures the Kubernetes components within the public cloud.

Within a Kubernetes cluster, Worker Nodes, as depicted in Figure 1, include components such as the kubelet, kube-proxy, and container runtime. The kubelet, aligned with the cluster manager's directives, ensures proper container operation within the Node. The kube-proxy manages inter-node network regulations, while the container runtime, with Docker as an example, is responsible for running containers [34]. Pods assigned to Nodes are also highlighted in Figure 1. The Control Plane node, pivotal for orchestrating processing nodes and Pods, interprets system blueprints, allocates Pods, and manages system events, such as Pod or Worker failures. Central components of the Control Plane are the kube-apiserver, etcd, kube-scheduler, and kube-controller-manager. The kube-apiserver provides access to the Kubernetes API, facilitating state queries and modifications [34]. The etcd, essential for storing cluster configuration, often operates in a distributed manner for enhanced availability. The kube-scheduler assigns Pods to nodes based on constraints and affinities, while the kube-controller-manager oversees controllers ensuring desired cluster states. Stateful Pods, relying on persistent external data, need to maintain data persistence even amid failures. Data for such Pods is usually stored externally, separate from their hosting Worker Node, ensuring data recovery during disruptions [17]. Storage servers, potentially holding data for stateful Pods and etcd, might use replication to bolster data availability. For Pod allocation in Kubernetes, it's vital to consider resource availability and Pod-node affinities. Proper resource allocation planning is essential for optimal Pod placement. To enhance cluster availability, redundancy for components like Worker nodes, the Control Plane, and Pods is common, but this can increase energy consumption. Strategic planning is crucial, balancing component criticality, reliability, and energy implications. Through thorough assessment, an infrastructure can be designed that achieves desired availability while also emphasizing energy efficiency and environmental sustainability.

IV. PROPOSED STOCHASTIC MODELS

In the pursuit of ascertaining the dependability metrics and electrical consumption characteristics of the system delineated in Section III, a methodical construction of a hierarchical model was undertaken. This construction encompasses a high-level Generalized Stochastic Petri Net (GSPN) model, imbued with the intricacies of Kubernetes components, in conjunction with a Dynamic Reliability Block Diagram (DRBD) model (as illustrated in Figure 8) designated for the computation of the mean time to failure (MTTF) of the Control Plane and Worker nodes. Further, an additional DRBD model was engaged for storage considerations (as portrayed in Figure 9), and a Continuous-Time Markov Chain (CTMC) model was deployed for the analysis of Pods (as exhibited in Figure 9).

Within this intricate modeling framework, the GSPN model serves as a panoramic representation of the entire system, encapsulating both the overarching structure and the nuanced components, whilst the complementary DRBD and CTMC models are strategically utilized to compute the MTTF of the Control Plane nodes, Worker nodes, and Pods, respectively. The values gleaned from these DRBD and CTMC models are subsequently integrated into the GSPN model, thereby enabling a holistic analysis of the entire system's complex behavior.

The adoption of this hierarchical modeling approach was not merely serendipitous but a deliberate strategy to engender a facilitation of comprehension and interpretation of the model. Moreover, it grants the flexibility required for the facile modification of underlying elements of the system, an essential attribute in the dynamic environment of microservices architectures. Analyzing these interconnected models helps identify how configuration and component choices affect dependability and power consumption. This offers valuable insights for creating optimized architectures.

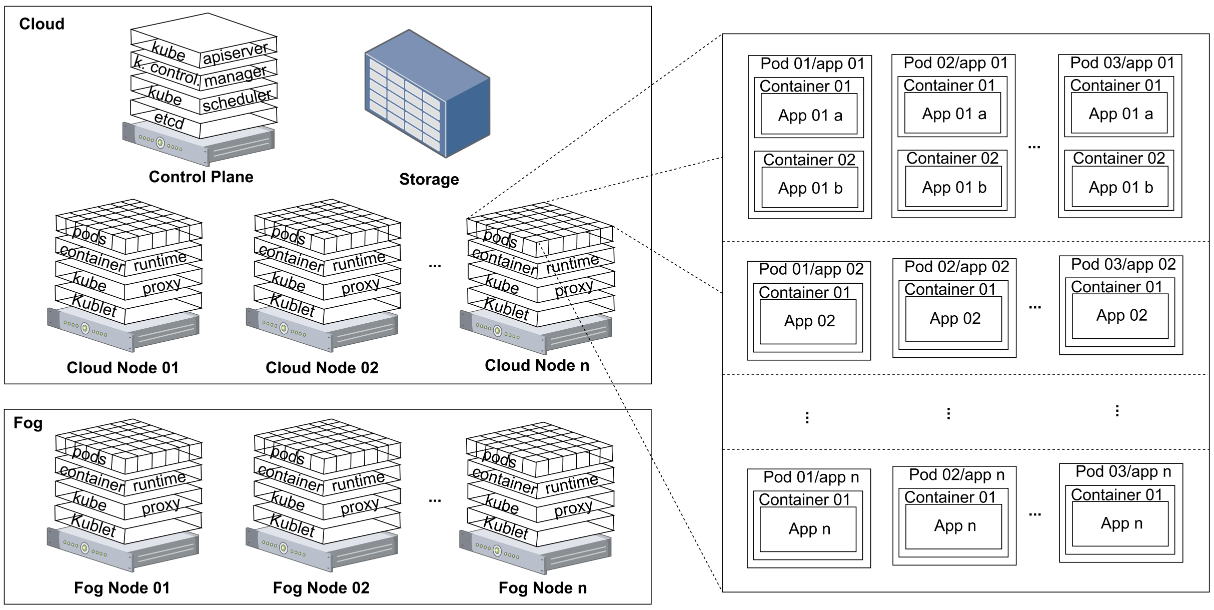


FIGURE 1. Kubernetes' architecture.

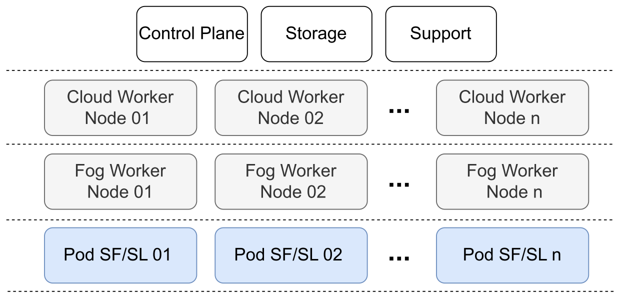


FIGURE 2. GSPN Block model overview.

Figure 2 delineates an encompassing overview of the system, inclusive of its multifaceted high-level components. Each constituent block encapsulates a high-level Generalized Stochastic Petri Net (GSPN) model of the Kubernetes structure within the cloud-fog paradigm. The nuanced details of the GSPN models will be explicated in the subsequent section.

The components labeled as Control Plane, Worker nodes, PodSF/SL (Stateful/Stateless), Support, and Storage are emblematic of the models portrayed in Figures 3, 4, 5, 6, and 7, respectively. It is imperative to note that both Worker Nodes and Pods necessitate replication congruent with the precise specifications of each individual node within the intricate system structure.

Within the architecture of the Stochastic Petri Net (SPN) models, timed transitions are employed, adhering to an exponential distribution (indicated by white rectangles), a selection congruent with conventional practices in modeling within the domain of availability and dependability [35], [36], [37]. This modeling option was deliberately chosen to facilitate the numerical analysis essential for the extraction of pertinent metrics. Nevertheless, flexibility exists for the substitution of this distribution with an alternative more apposite to a specific component. Such a transition may be supplanted by a phase-type distribution,as used in [38], or potentially through simulation, thereby permitting the direct insertion of distribution parameters within the transition itself. Moreover, simulation may serve as a viable approach if the model's application culminates in a large model with an extensive state space, thereby precluding conventional stationary analysis.

The modeling approach embodied in Figure 2 was meticulously conceived to accommodate the extensive spectrum of configurations conceivable within a microservice architecture. Manifesting a modular design, the system model inherently possesses the adaptability requisite for alignment with the diverse scenarios and exigencies encountered within a real-world context.

In pursuit of a representation that encapsulates the complexity and variability inherent to such systems, Worker nodes were imbued with additional configurable options. These include, but are not limited to, the provision for variable capacities, the reliability of constituent physical components, and the precision calibration of the Pods executable on each respective node. Such granularity within the model serves to mirror significantly heterogeneous physical architectures, exemplified by the one depicted in Figure 1, encompassing nodes distributed across both cloud and fog computing layers.

Furthermore, the model incorporates intricate configuration options pertinent to the state persistence of Pods and introduces a dedicated block to symbolize the available support teams, a critical element in a system characterized by a plethora of components that may concurrently vie for repair

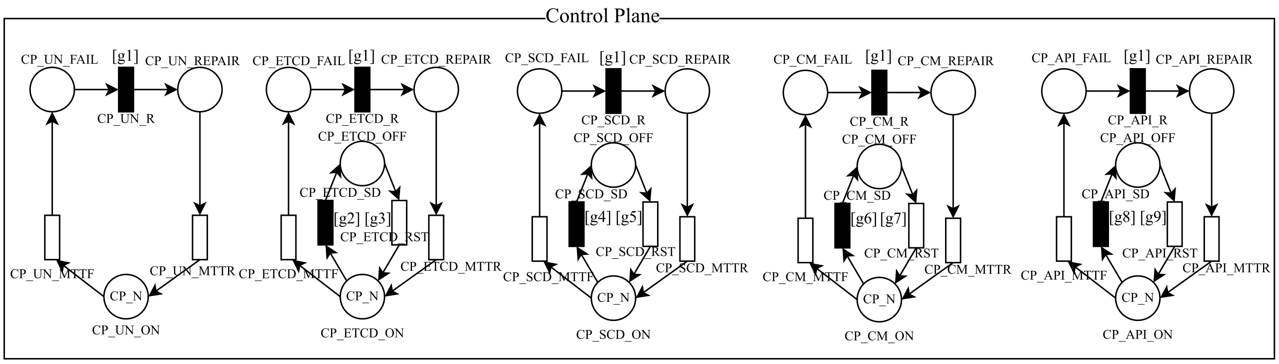


FIGURE 3. GSPN Control Plane Block.

and maintenance. This amalgamation of details within the model manifests a sophisticated and nuanced representation, reflective of the intricate and multifaceted nature of modern microservice architectures, thereby providing a robust foundation for analytical exploration and optimization.

A. CONTROL PLANE

The Control Plane is modelled at a high level in the GSPN shown in the Control Plane block of Figure 3. The model considers the state of the node's processes and its underlying system, which includes the hardware and software necessary to keep the node active. The meanings and firing semantics of the transitions are available in Table 2, while the guard expressions are outlined in Table 3.

The number of Control Plane nodes in the infrastructure can be adjusted by changing the *CP_N* variable. For the system to be active, at least one token must be present in each place ending with *_ON*. The failure of the underlying component occurs when the *CP_UN_MTTF* transition is fired. This transition receives its value from the DRBD analysis in Figure 8, in which the designer can carefully define the components of this node and identify the resulting MTTF. Alternatively, when leveraging a public cloud environment, the MTTF of the Virtual Machine (VM) in such a setting can be directly employed, obviating the necessity of relying on values derived from the DRBD.

The firing of *CP_UN_MTTF* results in the removal of a token from *CP_UN_ON* and the insertion of a token into *CP_UN_FAIL*, which represents the failure of one of the Control Plane nodes. The underlying system will remain failed until at least one support team is available to start repairing the system, that is, if there is a token in *TEAM_AVAILABLE* in the support team model Figure 6, as defined by the expression [$g1$]. If a support team is available, the immediate transition *CP_UN_R* is fired, and the underlying system will enter maintenance for the time defined by timed transition distribution *CP_UN_MTTR*.

In addition to the failure of the component itself, the firing of *CP_UN_MTTF* also enables the transitions *CP_etcd_SD*, *CP_SCD_SD*, *CP_CM_SD* and *CP_API_SD*, as can be seen from their respective guard expressions ([g2] in Table 3). Firing these transitions also puts the Control Plane processes on the failed node in a failed state. The failure in these processes is different because it does not require support, only the Kubernetes processes restart time, which is given by the distributions inserted in the transitions *CP_ETCD_RST*, *CP_SCD_RST*, *CP_CM_RST* and *CP_API_RST*. The restart only occurs when the underlying system is recovered and with the subsequent enablement given by the guard conditions of these transitions.

The designer can add storage for etcd data. In the model, this is done using the full expression of [$g2$] and [$g3$], that is, with the underlined part of the expression, if they are not added, etcd can only fail in case of failure of the underlying system like the other processes. Failure of the storage environment results in the satisfaction of expression [$g2$] and the firing of *CP_etcd_SD*. Repairing the storage environment leads to restoring this process, that is, by enabling the expression [$g3$] and transition *CP_etcd_RST*.

The phenomena of failures and restorations pertaining to Control Plane processes transpire in a manner analogous to those described for the Underlying System. A discernible deviation lies in the omission of explicit modeling of the interdependence of the control manager processes. The initialization time of dependent processes must be integrated into the repair dynamics of the failed process. Furthermore, it is observable that the restoration of simultaneous component failures is governed by the expression [$g1$], a circumstance solely transpiring in the presence of available teams. An additional consideration pertains to the augmentation in both availability and electrical consumption concomitant with the incorporation of new Control Plane nodes, reflecting the intricate balance and considerations involved in optimizing microservice architectures.

B. WORKER NODES

Worker nodes exhibit structural characteristics analogous to those found within the control plane, encompassing models of the underlying system, as well as the requisite processes for their operation, namely, kubelet, kube-proxy, and container

TABLE 2. Control plane transitions meaning.

Transition	Description	Firing Semantic	Weight	Priority
CP_UN_MTTF	CP Underlying System MTTF	IS	-	1
CP_UN_R	CP Underlying System Initiate Repair	-	1.0	1
CP_UN_MTTR	CP Underlying System MTTR	IS	-	1
CP_etcd_MTTF	CP etcd MTTF	IS	-	1
CP_etcd_R	CP etcd Initiate Repair	-	1.0	1
CP_etcd_MTTR	CP etcd MTTR	IS	-	1
CP_etcd_SD	CP etcd Shutdown	-	1.0	2
CP_etcd_RST	CP etcd Restart	IS	-	1
CP_SCD_MTTF	CP Kube Scheduler MTTF	IS	-	1
CP_SCD_R	CP Kube Scheduler Initiate Repair	-	1.0	1
CP_SCD_MTTR	CP Kube Scheduler MTTR	IS	-	1
CP_SCD_SD	CP Kube Scheduler Shutdown	-	1.0	2
CP_SCD_RST	CP Kube Scheduler Restart	IS	-	1
CP_CM_MTTF	CP Kube Controller manager MTTF	IS	-	1
CP_CM_R	CP Kube Controller manager Initiate Repair	-	1.0	1
CP_CM_MTTR	CP Kube Controller manager MTTR	IS	-	1
CP_CM_SD	CP Kube Controller manager Shutdown	-	1.0	2
CP_CM_RST	CP Kube Controller manager Restart	IS	-	1
CP_API_MTTF	CP API Server MTTF	IS	-	1
CP_API_R	CP API Server Initiate Repair	-	1.0	1
CP_API_MTTR	CP API Server MTTR	IS	-	1
CP_API_SD	CP API Server Shutdown	-	1.0	2
CP_API_RST	CP API Server Restart	IS	-	1

TABLE 3. Control plane guard expression.

Index	Guard Expression	Description
[g1]	#TEAM_AVAILABLE>0	Support team available to support
[g2]	((#STORAGE_ON=0)OR ((#CP_UN_ON)<(#etcd_ON)))	Storage failure or number of Underlying System less than the number of `etcd` processes
[g3]	((#STORAGE_ON>0)AND (#CP_UN_ON>(#etcd_ON)))	Storage Recovery or Underlying System number greater than the number of `etcd` processes
[g4]	(#CP_UN_ON)<(#CP_SCD_ON))	Number of Underlying Systems less than the number of SCHEDULER processes
[g5]	(#CP_UN_ON)>(#CP_SCD_ON)	Number of Underlying System greater than the number of SCHEDULER processes
[g6]	(#CP_UN_ON)<(#CP_CM_ON)	Number of Underlying System less than the number of Controller Manager processes
[g7]	(#CP_UN_ON)>(#CP_CM_ON)	Number of Underlying System greater than the number of Controller Manager processes
[g8]	(#CP_UN_ON)<(#CP_API_ON)	Number of Underlying System less than the number of API Server processes
[g9]	(#CP_UN_ON)>(#CP_API_ON)	Number of Underlying System greater than the number of API Server processes

runtime. Extending beyond these foundational components, the model also portrays the total capacity designated for the allocation of Pods within the node, in conjunction with both the allocated and allocatable Pods. This configuration can be discerned in Figure 4, where the upper section, labeled as the infrastructure sub-block, delineates the amalgamation of physical and logical components requisite for the efficacious operation of Kubernetes. Conversely, the lower section, identified as the Pods sub-block, illustrates the available capacity in tandem with the allocated Pods.

The intricate composition of the worker nodes can be further elucidated through an examination of Table 4, which offers a detailed exposition of the transitions pertaining to the worker nodes. This tabulation elucidates not only the functional transitions but also the underlying characteristics that govern their operation. Complementing this, Table 5 presents an enumeration of the guard expressions associated with these transitions, thereby providing a comprehensive insight into the conditions that dictate the activation or inhibition of specific transitions within the model. The formulation of worker nodes in this manner serves to encapsulate the complex interplay between various components and processes within the node, reflecting the inherent sophistication of microservice architectures. By delineating both the infrastructure and Pod-related aspects within a unified framework, the model facilitates an intricate understanding of the worker nodes' dynamics.

The node is considered functional when there is a token in each place $N_n\_UN\_ON$, $N_n\_CR\_ON$, $N_n\_KP\_ON$, and Nn_KL_ON, that is, the underlying system, the container runtime, the kube-proxy, and the kubelet must be operational. The MTTF (mean time to failure) of the underlying system depends on the characteristics of the hardware and software components of the node, and its value can be obtained for each worker node based on the input of the hardware data and software components of the DRBD model in Figure 8. Another possibility is to directly insert the resulting MTTF of the underlying system being used, whether from a third party or a component with a known final MTTF.

The failure of the underlying system occurs when the $N_n\_UN\_MTTF$ is fired, removing a token from $N_n\_UN\_ON$ and creating one in $N_n\_UN\_FAIL$. Consequently, all of the

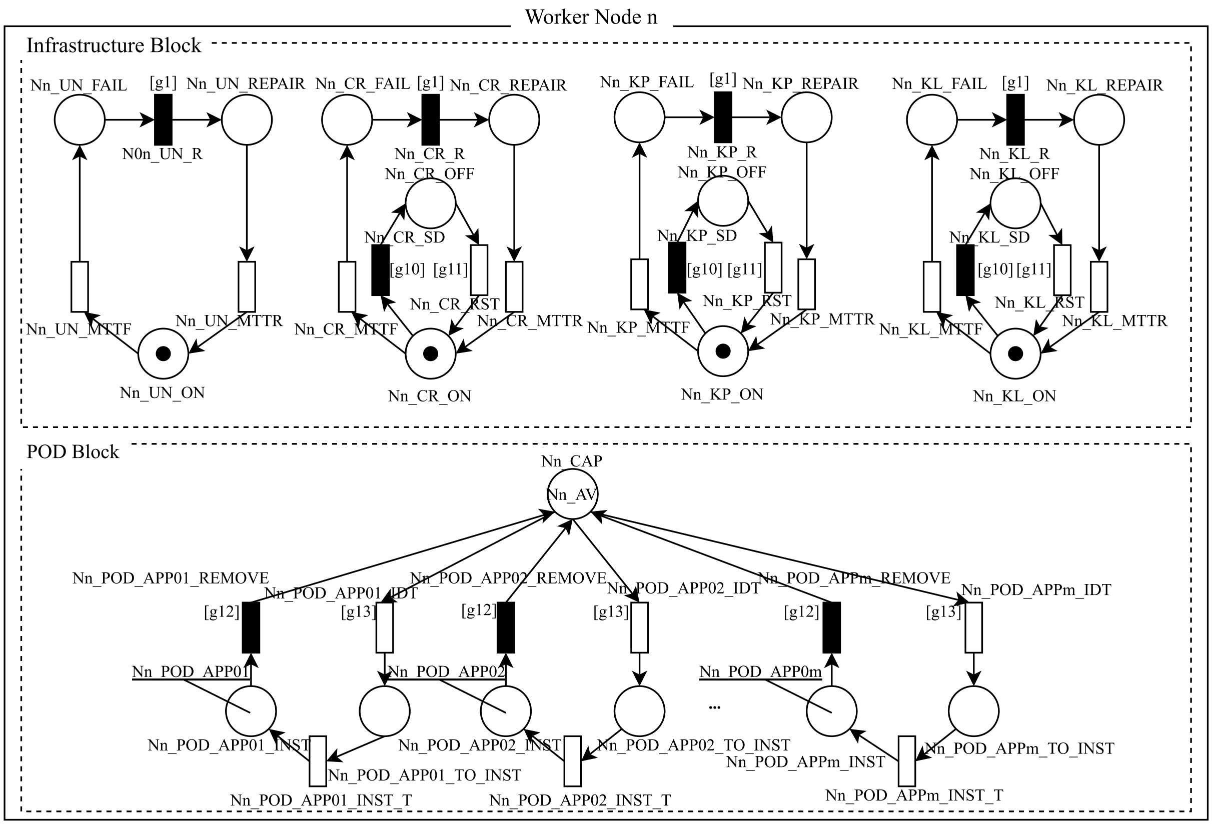


FIGURE 4. GSPN Worker Nodes Block.

TABLE 4. Worker node transitions meaning.

Transition	Description	Firing Semantic	Weight	Priority
$N_n\_UN\_MTTF$	WN_n Underlying System MTTF	SS	-	1
$N_n\_UN\_R$	WN_n Underlying System Initiate Repair	-	1.0	1
$N_n\_UN\_MTTR$	WN_n Underlying System MTTR	SS	-	1
$N_n\_CR\_MTTF$	WN_n Container Runtime MTTF	SS	-	1
$N_n\_CR\_R$	WN_n Container Runtime Initiate Repair	-	1.0	1
$N_n\_CR\_MTTR$	WN_n Container Runtime MTTR	SS	-	1
$N_n\_CR\_SD$	WN_n Container Runtime Shutdown	-	1.0	2
$N_n\_CR\_RST$	WN_n Container Runtime Restart	SS	-	1
$N_n\_KP\_MTTF$	WN_n Kube Proxy MTTF	SS	-	1
$N_n\_KP\_R$	WN_n Kube Proxy Initiate Repair	-	1.0	1
$N_n\_KP\_MTTR$	WN_n Kube Proxy MTTR	SS	-	1
$N_n\_KP\_SD$	WN_n Kube Proxy Shutdown	-	1.0	2
$N_n\_KP\_RST$	WN_n Kube Proxy Restart	SS	-	1
$N_n\_KL\_MTTF$	WN_n kubelet MTTF	SS	-	1
$N_n\_KL\_R$	WN_n kubelet Initiate Repair	-	1.0	1
$N_n\_KL\_MTTR$	WN_n kubelet MTTR	SS	-	1
$N_n\_KL\_SD$	WN_n kubelet Shutdown	-	1.0	2
$N_n\_KL\_RST$	WN_n kubelet Restart	SS	-	1
$N_n\_POD\_APP_m\_IDT$	WN_n POD_m Identify Event	IS	-	1
$N_n\_POD\_APP_m\_INST\_T$	WN_n POD_m Instantiation Time	IS	-	1
$N_n\_POD\_APP_m\_REMOVE$	WN_n POD_m Remove	-	1.0	2

Kubernetes processes on the node also become unavailable, which is represented by the subsequent evaluation of the expression $[g10]$ of the model as ''true'', which results in the firing of $N_n\_CR\_SD$, $N_n\_KP\_SD$, and $N_n\_KL\_SD$, causing these processes to become non-operational. As mentioned earlier, for the control plane, the recovery from a failure can only start when there is available support, which is ensured by the expression $[g1]$ in $N_n\_UN\_R$. $[g1]$ also is used in the control node block (Figure 6), so if simultaneous failures occur in both nodes, they will compete for the available support teams.

When support team is available, the token is removed from $N_n\_UN\_FAIL$ and deposited in $N_n\_UN\_REPAIR$, effectively initiating the repair. The repair time is defined by the transition time distribution of $N_n\_UN\_MTTR$. At the end of the repair, the token is removed from $N_n\_UN\_REPAIR$, and a new one is created in $N_n\_UN\_ON$. The return of the operation of the Underlying system leads to the

TABLE 5. Worker node guard expressions.

Index	Guard Expression	Description
[g10]	#Nn_UN_ON=0	WN_n failure
[g11]	#Nn_UN_ON>0	WN_n recovery
[g12]	$((\#N_n\_UN\_ON == 0) OR (\#N_n\_CR\_ON == 0) OR (\#N_n\_KP\_ON == 0) OR (\#N_n\_KL\_ON == 0))$	When the node fails all instanced Pods must be removed from Kubernetes capacity.
[g13]	Equation 2	If there are failed Pods, and instantiation capability exists, the node will attempt to recreate the failed Pod.

restart of the Kubernetes processes that are initiated as a result of the evaluation of expression [g11], which becomes "true", enabling transitions $N_n\_CR\_RST$, $N_n\_KP\_RST$, and $N_n\_KL\_RST$. The Firing of transitions with expression [g11] represents the restart of Kubernetes processes on the node, making it active and able to allocate Pods.

The failure of the other processes, Container Runtime, Kube-proxy, and kubelet, follows a similar flow to that presented in the Underlying System, reaching a failure state and subsequently competing for some support team to initiate the repair and then returning to normal activity without causing the failure of any other Kubernetes process. However, the failure of any of the components in the Infrastructure sub-block of the node leads to the failure of the Pods hosted on that node.

The Pods sub-block of the node in Figure 4 represents the total allocation capacity and the allocated amount of Pods on the node. The available capacity of the node is given by the number of tokens in place N_n_CAP, and the initially available capacity is given by the variable N_n_AV. The capacity must be computed as the total amount of the node reduced by the amount of Pods initially allocated on the node (Equation 1). The # character in the Equation represents the number of tokens in the place. Each possible pod allocated on the node should be represented in this block. A Pod m on node n is represented by the places $N_n\_POD\_APP_m\_TO\_INST$, $N_n\_POD\_APP_m\_INST$ and the associated transitions $N_n\_POD\_APP_m\_REMOVE$, $N_n\_POD\_APP_m\_IDT$, and $N_n\_POD\_APP_m\_INST\_T$.

T = $\{y \mid y$ are the indexes of Pods enabled to run in node $n\}$

$$\begin{aligned} N_n\_CAP = N_n\_CAP\_TOTAL \\ - \sum_{y \in T} (\#N_n\_POD\_APP_y\_TO\_INST \\ + \#N_n\_POD\_APP_y\_INST) \end{aligned} \tag{1}$$

Often, the micro-services infrastructure has many different APPs with different hardware, software, or network requirements, which should only be allocated on certain nodes. Therefore, if a particular Pod should not be included in the node, the designer simply does not include the set of places and transitions on that node, and in any event, that node will not be eligible to start that Pod.

The number of Pods of type m on node n at a given time is given by the number of tokens in $N_n\_POD\_APP_m\_INST$. The initial amount is configured in the variable $N_n\_POD\_APP_m$. As previously mentioned, if the node or Kubernetes processes fail, all Pods allocated on the node also fail. In the model, this failure leads to the enabling of transitions $N_n\_POD\_APP_m\_REMOVE$ (guard expression [g12] for the failed node), with m representing the entire set of Pods eligible for execution on that node. The immediate enabling of this transition leads to the removal of all the instantiated capacity for the available capacity of the node.

On the other hand, if another node in the infrastructure with allocated Pods fails, Kubernetes will try to bring the system to the state with the minimum number of Pods configured by the designer. To do this, Kubernetes will try to allocate the failed Pods from the failed node to other nodes. In the model, the identification of the failure of some nodes is given by the guard expression [g13]. The expression [g13] considers the presence of failed Pods in the models presented in the blocks in Figure 5 and all the eligible existing nodes for allocating the failed Pods.

The expression [g13] will be "true" on nodes capable of instantiating the failed Pod if the infrastructure sub-block of the node is operational and if the total number of failed Pods is not yet instantiated on all nodes enabled for the type of Pod. In addition, the node must have available capacity for Pod allocation, a precondition for firing the transition $N_n\_POD\_APP_m\_IDT$.

If expression [g13] is evaluated as "true" and the time that Kubernetes takes to identify the failure and propagate it in its system, represented by the time assigned to transition $N_n\_POD\_APP_m\_IDT$, the instantiation of the Pod on the node is initiated. A token is removed from the total capacity of the node (N_n_CAP), and one is created in $N_n\_POD\_APP_m\_TO\_INST$ representing the start of the instantiation of the Pod. The expression [g13] after this change may change its evaluation to "false" if all the failed nodes have started their instantiation. The time of the instantiation of the Pod is given by the transition $N_n\_POD\_APP_m\_INST\_T$, with the firing of this transition, the token is removed from $N_n\_POD\_APP_m\_TO\_INST$ and deposited in $N_n\_POD\_APP_m\_INST$. As will be seen in the next sections, this change also affects the allocated Pods presented in the Pods model in Figure 5.

S = $\{x \mid x$ are the worker nodes indexes enabled to run the Pod $m\}$

$$\begin{aligned} &(\#UN\_POD\_APP_m\_FAIL > \sum_{x \in S} \#N_x\_POD\_APP_m\_TO\_INST) \\ &and\ (\#N_n\_UN\_ON == 1)\ and\ (\#N_n\_CR\_ON == 1) \\ &and\ (\#N_n\_KP\_ON == 1)\ and\ (\#N_n\_KL\_ON == 1) \end{aligned} \tag{2}$$

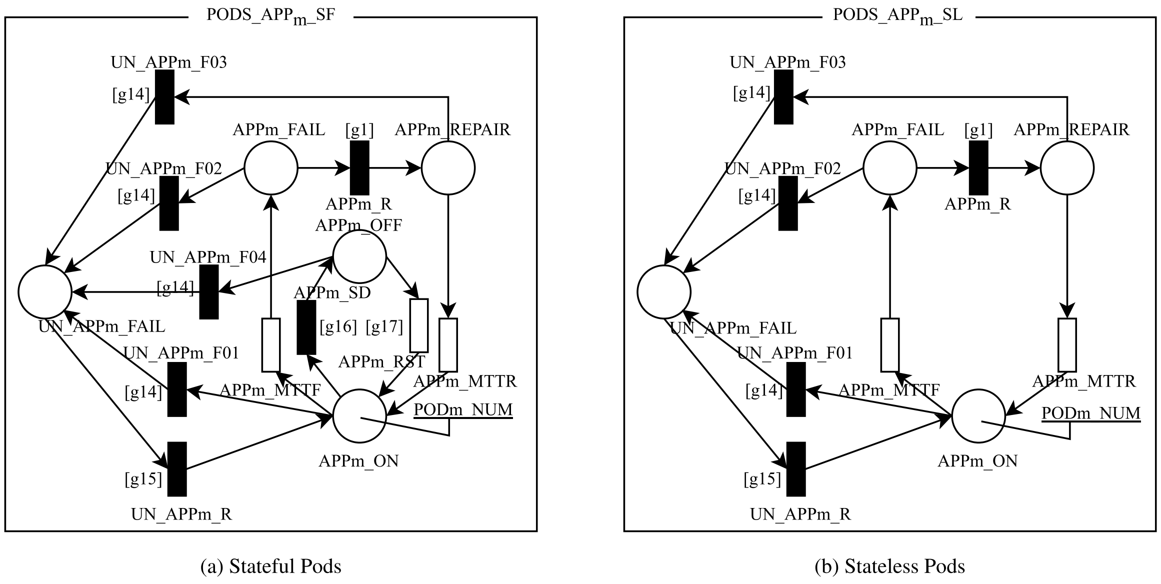


FIGURE 5. GSPN High level Pods Block.

Each Worker Node block represents a specific node in the system. Another block must be created to add new nodes (see Figure 2). The node worker individualization allows for a higher level of infrastructure detail, such as nodes with higher capacity, nodes with Pods allocation restrictions, or initial allocation state with more or fewer Pods of each type.

C. PODS

Figure 5 presents the blocks of the model that represent the Pods of the system. The block of subfigure 5a models the stateful Pods and those of subfigure 5b the stateless ones, as many of these blocks should be added as the types of Pods present in the system. Each added block should have its allocation using the resources of one or more Worker Nodes from Figure 4 as commented. The descriptions of the transitions of this block can be found in Table 6 and the guard expressions in Table 7.

S = $\{x \mid x$ are the worker nodes indexes enabled to run the Pod $m\}$

$$\#APP_m\_ON > \sum_{x\in S}(\#N_x\_POD\_APP_m\_TO\_INST + \#N_x\_POD\_APP_m\_INST) \quad (3)$$

$$\#APP_m\_ON < \sum_{x\in S}(\#N_x\_POD\_APP_m\_TO\_INST + \#N_x\_POD\_APP_m\_INST) \quad (4)$$

The total number of active Pods at a given time is represented by the number of tokens in place APP_m_ON, m representing the type of Pod. The variable POD_m_NUM defines the initial number of active Pods the initial value of this variable is equal to the sum of all Pods m distributed throughout the Worker Nodes infrastructure (Equation 5), that is, in all existing blocks in Figure 2. The number of Pods instantiated on the Worker nodes may be less than the number of Pods in the different states of the models in Figure 5, which may occur due to the failure of a Worker Node. Whenever a worker node fail, the model will seek to maintain the initial amount equal to the Pods instantiated on the Worker nodes.
U = $\{x \mid x$ are the nodes indexes enabled to run the Pod $m\}$

$$POD_m\_NUM = \sum_{x\in U}(\#N_x\_POD\_APP_m\_TO\_INST + \#N_x\_POD\_APP_m\_INST) \quad (5)$$

Active Pods have the failure time given by the MTTF time distribution of the APP_m_MTTF transition. The value of this MTTF is obtained by the model that represents the set of containers of the Pod represented by the CTMC model in Figure 10. After the fire of APP_m_MTTF, a token is removed from APP_m_ON and inserted into APP_m_FAIL. Similarly to the previous blocks, the fire of the immediate transition responsible for initiating the repair depends on the guard expression [$g1$], which checks for the availability of the support team (Model in Figure 6). Therefore, Pods also compete for support along with other system elements.

After the fire of APP_m_R, the repair of the Pod is initiated with removing the token from APP_m_FAIL and creating a token in APP_m_REPAIR. The repair takes the time defined by the time distribution of the APP_m_MTTR transition. After the repair, the token is removed from APP_m_REPAIR and returns to its operational state in place APP_m_ON.

$$n\_pods_m = \#APP_m\_ON + \#APP_m\_FAIL + \#APP_m\_REPAIR + \#APP_m\_OFF$$

$$W\_UN\_APP_m\_F01 = \frac{\#APP_m\_ON}{n\_pods_m}$$

TABLE 6. Pods transitions meaning.

Transition	Description	Firing Semantic	Weight	Priority
APPm_MTTF	Pod with App m MTTF	IS	-	1
APPm_R	Pod with App m start repair	-	1	1
APPm_MTTR	Pod with App m MTTR	IS	-	1
UN_APPm_F01	Pod with App m UN started Pods removal	-	$W\_UN\_APP_m\_F01$ in Eq 6	1
UN_APPm_F02	Pod with App m UN failed Pods removal	-	$W\_UN\_APP_m\_F02$ in Eq 6	1
UN_APPm_F03	Pod with App m UN repairing Pods removal	-	$W\_UN\_APP_m\_F03$ in Eq 6	1
UN_APPm_F04	Pod with App m UN halted Pods removal	-	$W\_UN\_APP_m\_F04$ in Eq 6	1
APPm_SD	Pod with App m shutdown	-	1	3
APPm_RST	Pod with App m restart	IS	-	1

TABLE 7. PODs guard expressions.

Index	Guard Expression	Description
[g14]	Equation 3	The number of Pods of an APPs m must be equal to the amount of the sum of Pods instantiated on all nodes eligible for instantiation. If it is fewer Pods beyond that number must be removed.
[g15]	Equation 4	The amount of Pods of an APPs m must be equal to the amount of the sum of Pods instantiated on all nodes eligible for instantiation. If it is greater, Pods below that number must be added.
[g16]	#STORAGE_ON=0	If there is no working storage, stateful Pods with external storage fail.
[g17]	#STORAGE_ON>0	If storage is working, stateful Pods with failed external storage are recovered.

$$W\_UN\_APP_m\_F02 = \frac{\#APP_m\_FAIL}{n\_pods_m}$$
$$W\_UN\_APP_m\_F03 = \frac{\#APP_m\_REPAIR}{n\_pods_m}$$
$$W\_UN\_APP_m\_F04 = \frac{\#APP_m\_OFF}{n\_pods_m} \tag{6}$$

In addition to the flaws inherent in the internal components of the Pods, these can also fail due to the failures of the worker nodes that maintain the Pods (as presented in the subsection IV-B). If a Worker node fails, all allocated Pods must also become inoperable. The failure of the Pods is modelled by the guard expression $[g14]$ and the transitions $UN\_APP_m\_F01$, $UN\_APP_m\_F02$, $UN\_APP_m\_F03$, and $UN\_APP_m\_F04$ (the last only in stateful Pods), which removes the Pods allocated by the node in all states of the block.

The probability of failed Pods being removed from a given location depends on the number of Pods at each location in the model. The failure probability of each location is modelled using transition weights $UN\_APP_m\_F01$, $UN\_APP_m\_F02$, $UN\_APP_m\_F03$, and $UN\_APP_m\_F04$ seen in Equation 6. If there are more Pods in the place APP_m_ON, it is expected that with the failure of a node, the Pods of that place are more likely to be in the failed node and consequently be removed by firing the failure transitions of Worker Nodes.

After the failure of a worker node, if there is available capacity on the other servers or if the failed node is recovered, the number of Pods that failed due to the failure of the underlying system can be recreated on another active node, or the old node after recovery, as discussed in the behaviour of transitions with the expression $[g14]$ in Subsection IV-B. In the general identification of Pods of the model, the recovery is made by the transition $UN\_APP_m\_R$ and guard expression $[g15]$ that removes the failed tokens from $UN\_APP_m\_FAIL$ and makes them active again.

In addition to the failure methods mentioned previously, stateful Pods can also become inoperable due to the failure of the respective storage system. Failure due to storage is represented in the model by the evaluation of guard expression $[g16]$ and the immediate firing of transition APP_m_SD. On the other hand, the recovery of the storage environment of stateful Pods leads to the evaluation of guard expression $[g17]$ to be true, which restarts the stateful Pods.

The number of Pods the system needs to be functional is configurable through a variable MIN_POD_m, so the system manager can add redundant Pods by adding more initial Pods than this value variable. However, the energy consumption that adding additional Pods will have on the system should also be considered.

The number of Pods the system needs to be functional is configurable through a variable MIN_POD_m, so the system manager can add redundant Pods by adding more initial Pods than this value variable. However, the energy consumption that adding additional Pods will have on the system should also be considered.

D. SUPPORT

In complex system architectures, a specialized support team with technical expertise is essential for effective diagnosis and rectification of malfunctioning components. Such teams are tasked with identifying and addressing system anomalies. If a single support team is employed, a sequential intervention approach is adopted, ensuring each issue receives dedicated attention, thus optimizing the repair process. Figure 6 provides a schematic of the support element, while Table 8 offers detailed transition specifications. Additionally, Table 9

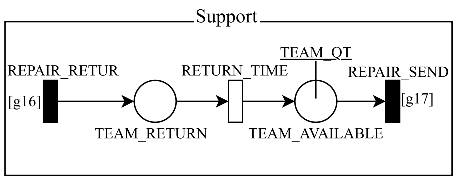


FIGURE 6. GSPN Support Block.

TABLE 8. Transitions for the support model.

Transition	Description	Firing semantic	Weight	Priority
REPAIR_RETURN	Start of support team return	-	1	1
RETURN_TIME	Time for support to return availability	IS	-	1
REPAIR_SEND	Start of repair of a component	-	-	5

TABLE 9. Guard expressions for the support model.

Index	Guard Expression	Description
[g18]	Equation 9	Enable repair completion when finished on component.
[g19]	Equation 10	Enable start of repair when a component is failed.

outlines the guard conditions governing the support mechanism's operations. Together, these representations offer a holistic understanding of the support subsystem, aiding system architects in enhancing system dependability.

The *TEAM_QT* variable defines the initial number of available support teams. The start of a component repair occurs after the *REPAIR_SEND* transition is fired, which is enabled by the [g19] condition, which compares the number of repairs being carried out in the system (Equation 7) with the number of absent support teams in the support environment (Equation 8). If the number of repairs is higher (Equation 10), the transition should fire, removing the token from place *TEAM_AVAILABLE*, which represents the removal of a support team from availability because it is repairing the failed component. We must note that the *REPAIR_SEND* transition is immediate and occurs as a result of the immediate transitions of the components related to the place representing a failure in one of the model's components.

Once the repair on the component is completed, the team carrying out the support must return to availability. This behaviour is fired by the *REPAIR_RETURN* transition, which the g[18] condition enables. This condition compares the number of repairs on all system components (Equation 7) with the number of absent teams (Equation 8). When the number of absent teams is greater than the repairs (Equation 9), a team must return and wait for their return time to be able to attend to the new support. The return time is defined by the time distribution defined by the *RETURN_TIME* transition. The use of the guard conditions [g18] and [g19] is due to the option of modelling the removal of arcs that could be used between the *REPAIR_SEND* and *REPAIR_RETURN* transitions and the start and end of support for each component. Removing these arcs made the model more easily adaptable and extensible for large systems, as expected in systems with multiple nodes and different types of Pods. Furthermore, also facilitates understanding of the model in very large systems.

U = {x | x are the worker nodes},

T = {y | y are the indexes of Pods}

$$\begin{aligned} away\_teams &= TEAM\_QT - (\#TEAM\_RETURN \\ &\quad + \#TEAM\_AVAILABLE) \qquad (7) \\ repair\_comp &= \sum_{x \in U} (\#Nn\_UN\_REPAIR \\ &\quad + \#Nn\_CR\_REPAIR \\ &\quad + \#Nn\_KP\_REPAIR \\ &\quad + \#Nn\_KL\_REPAIR) \\ &\quad + \sum_{y \in T} (\#APP_y\_REPAIR) \\ &\quad + \#CP\_UN\_REPAIR \\ &\quad + \#CP\_etcd\_REPAIR \\ &\quad + \#CP\_SCD\_REPAIR \\ &\quad + \#CP\_CM\_REPAIR \\ &\quad + \#CP\_API\_REPAIR \qquad (8) \end{aligned}$$

$$repair\_comp < away\_teams \qquad (9)$$

$$repair\_comp > away\_teams \qquad (10)$$

E. STORAGE

The storage block represents the high-level storage system in the SPN model. This component can store the Kubernetes Control Plane's `etcd` and the data of stateful Pods. The Storage model is shown in Figure 7. The description of the model's transitions can be seen in Table 10.

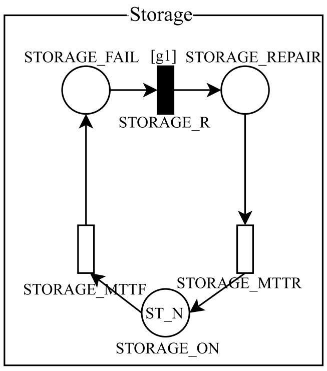


FIGURE 7. GSPN Storage Block.

The number of replicas of this component is given by the value of the variable *ST_N*, the higher the component is less likely to be unavailable, but there will also be a higher energy cost. The failure of a Storage happens by firing the transition *STORAGE_MTTF*. The time value of this MTTF can be obtained by the DRBD model shown in Figure 9 in cases of private infrastructure, as well as it can also be obtained by the contractor in some cloud storage services.

TABLE 10. Storage transitions.

Transition	Description	Firing Semantic	Weight	Priority
STORAGE_MTTF	Storage MTTF	IS	-	1
STORAGE_R	Storage start repair	-	1	1
STORAGE_MTTR	Storage MTTR	IS	-	1

After the failure, a token is removed from $STORAGE\_ON$ and deposited in $STORAGE\_FAIL$. If there are no tokens in $STORAGE\_ON$, the component will be unavailable, making all dependent components unavailable. With a failed storage, if there is any support team available (guard expression $[g1]$), the $STORAGE\_R$ transition fires and the token from $STORAGE\_FAIL$ are removed and deposited in $STORAGE\_REPAIR$, representing the start of the repair, which will take the time defined by the $STORAGE\_MTTR$ transition.

F. DRBD - PHYSICAL NODE

The Mean Time to Failure (MTTF) for both the Control Plane and Worker nodes is derived from the Distributed Replicated Block Device (DRBD) model, as shown in Figure 8. This model facilitates comprehensive configuration of the system's numerous sub-components. The adoption of DRBD stems from its ability to depict intricate interdependencies among RDB blocks. As a result, hardware issues might lead to cascading failures in associated software components, represented by the gray *SDEP* block. For instance, a CPU malfunction can cause subsequent failures in the Hypervisor, leading to Virtual Machine (VM) and systemic failures. Similar cascades are observable in other core components like memory, disk, and power supply. Thus, the DRBD model provides a detailed insight into the potential failure trajectories and intricate dependencies, enhancing our understanding of the system's dependability.

Furthermore, this model also allows for the inclusion of redundant hardware, as commonly found in actual servers, such as RAID disks or backup components, such as additional power supplies and network interfaces. The behaviour of hardware redundancy is represented by defining the minimum number of functional components (K) out of the total number of components (N). In Figure 8, the minimum and the total number of disks are represented by the variables DK and NK, the number of network interfaces by NK and NN, and the power supply is represented by PK and PN.

For a node to function properly in this model, each block must be functional and at least K out of the total N blocks must work (known as KooN in RBDs [39]). The user of this model must input the MTTF values of each block based on the values provided by the hardware manufacturers used in their computational park. Once input, it will be possible to obtain the MTTFs of the nodes used in the model as depicted in Figure 2. It's worth noting that if only one type of hardware is used, this calculation can only be done once and applied to all nodes in the model. On the other hand, if mixed configurations are used, each variation must be added to the corresponding node. For example, if Fog Worker nodes have different hardware from cloud nodes, this may result in different MTTF values. Additionally, hierarchical modelling makes it possible to evaluate the impact of changing a hardware component on the entire system.

G. DRBD - STORAGE NODE

Similarly to the nodes, the underlying storage was also modelled with DRBD. The model can be seen in Figure 9. It has as subcomponents the CPU, Memory, Disk, Network Interfaces, Power Supply, and System. The System block is the difference between the underlying component models, the software component responsible for managing the protocols and storage offered by the server. The dependency between the blocks was also modelled so that the failure of the CPU, Memory, Disk, NIC, or Power supply leads to the system's failure. The model user must provide the MTTFs and the KooN configuration of the subcomponents to compute the MTTF of the system that will be used in the model in Figure 7.

H. CTMC - PODS

The system's Pods are represented by the CTMC model shown in Figure 10. The model aims to generate the MTTF used in the respective Pod's SPN model found in Figure 5. The model considers that a Pod can contain one or more containers, and each container comprises the application, libraries, and elements necessary for its execution. In the model, we have the initial state $UU..UU$ containing all containers and applications functioning properly. From this state, the failure of an application or the failure of the container can occur, and the failure of the applications occurs through the rates of the arcs λ_APP_n, such that n can be any of the applications in the Pod. On the other hand, the failure of the container occurs due to the rate of the arcs with the pattern λ_CONT_n.

The failure of one of these arcs leads the Pod to the state containing the respective failed container. In the case of Figure 10a, assuming the failure of λ_APP_A, we would have $DU..UU$. From this state, the Pod can recover this application through the arc μ_APP_A, which leads to the state in which there are no failures again. These failures can occur successively until the state is reached in which all are failed $DD..DDD$. Another possibility is that successive failures occur in other containers or applications through the rate of any arc started by λ.

In many applications, not all Pod containers are essential for the proper functioning of the application; some may only be used to capture metrics, and the designer of the microservice system may not wish for the failure of these containers to be counted as a failure of the Pod. In this

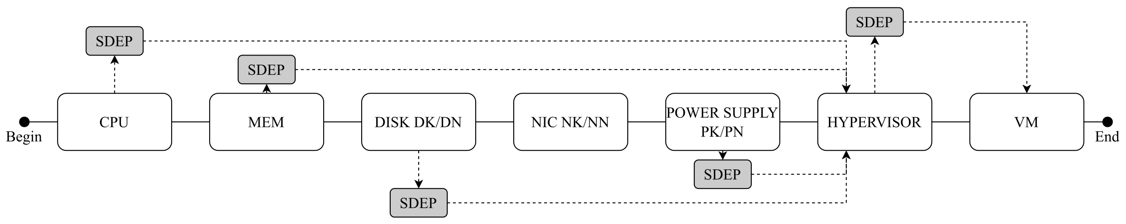


FIGURE 8. Node underlying system.

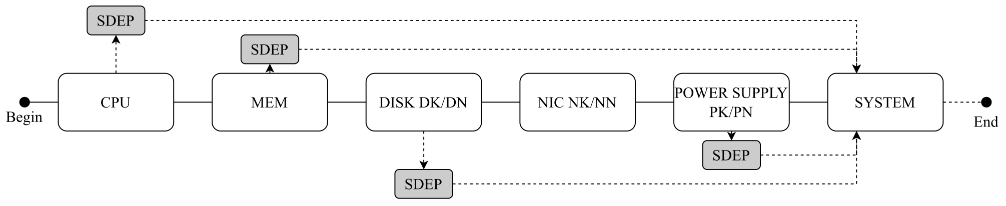


FIGURE 9. Storage underlying system.

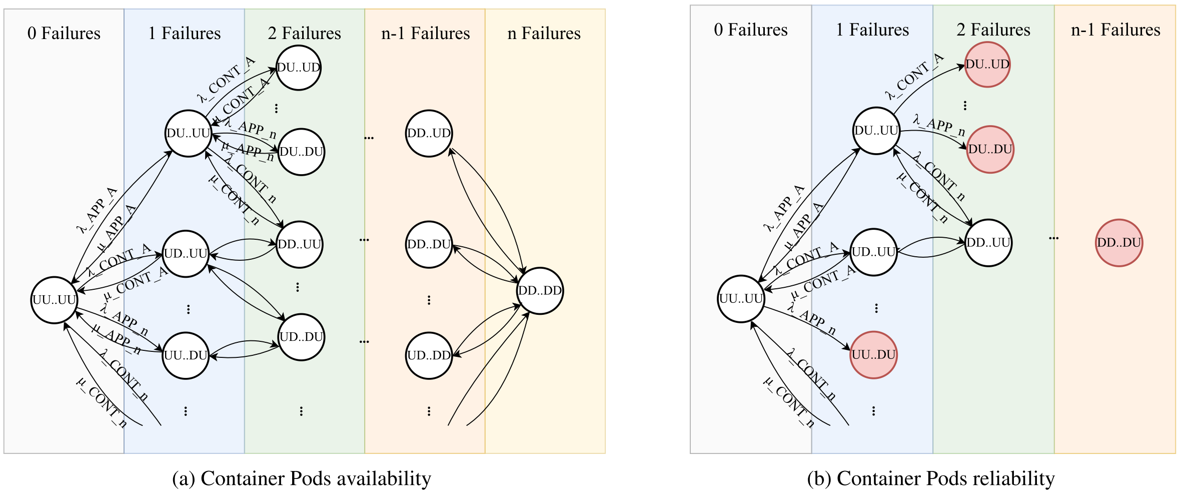


FIGURE 10. Underlying Pods.

case, the failure of this container or APP should also not be counted as a failure for calculating the Pod's MTTF. This behaviour is modelled in the absorbing state model shown in Figure 10b used for calculating the MTTA, which also represents the MTTF for the Pod. The reliability model is obtained by modifying the model in Figure 10a to contain absorbing states as shown in Figure 10b. Red states represent the absorbing states; from them, there are no recovery rate arcs (starting with the letter μ).

The model in Figure 10 must be generated for each Pod in the model and inserted into the corresponding transitions in the APP_MTTF_m transition of the Pod SPN models. The presented model represents a generic version of a Pod. In the case of having only one container and application per Pod, we have a simpler version, which can be seen in Figure 11. In this model, it is enough to insert the application and container failure rates. If a Pod only has one container and one APP, any failure of that container or APP will cause the entire Pod to fail, in CTMC, reaching states UD or DU.

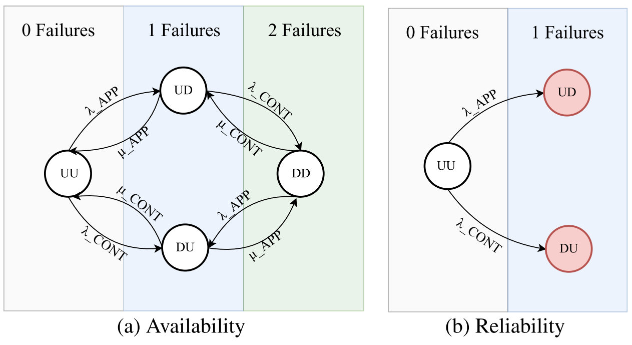


FIGURE 11. One container per Pod underlying model.

I. SYSTEM METRICS

The model is designed to evaluate dependability, repairability, and energy costs of a given configuration by generating metrics such as availability, number of accessible containers, Pod instantiation rate, support team utilization, and energy expenditure. The system's availability is ascertained through Equation 11. It takes into account the likelihood of having at least one operational control plane and storage (when relevant), and maintaining a minimum required number of functional Pods. While the availability of Worker Nodes is crucial, their specific location is not considered for determining the minimum expected quantity of each Pod type due to the flexibility of Pod allocation. Hence, the minimum requirement for each Pod type, represented by *min_num_app_m* (where *m* signifies different Pod sets), should be at least the system's initial count. In the metrics notation, $P\{(\#CP\_UN\_ON > 0)\}$ evaluates the probability of the enclosed expression, where the symbol # denotes the token count in the place *CP_UN_ON*.

$$\begin{aligned} A = P\{&(\#CP\_UN\_ON > 0) \wedge (\#CP\_etcd\_ON) \wedge \\ &\#CP\_SCD\_ON > 0) \wedge (\#CP\_CM\_ON) \wedge \\ &(\#CP\_API\_ON > 0) \wedge (\#CP\_STORAGE\_ON) \wedge \\ &(\#APP\_01\_ON \geq min\_num\_app\_01) \wedge \ldots \\ &(\#APP\_m\_ON \geq min\_num\_app\_m)\} \end{aligned} \tag{11}$$

A metric was also included to identify the average number of Pods in the system. The *N_PODS* metric is important to verify situations of under-provisioning or over-provisioning. The metric can be observed in Equation 12. This equation adds the expected value of each type of Pod in the system. The notation $E\{(\#APP\_m\_ON)\}$ represents the statistical expectation of the number of tokens in place *#APP_m_ON*, that is, $E\{(\#APP\_m\_ON)\} = (\sum_{i=1}^{n} P(\#APP\_m\_ONi) \times i)$, where *i* will vary from 1 to the maximum possible quantity of tokens for the evaluated place *n*.

$$\begin{aligned} \mathrm{C} &= \{m | m \text{ are Pods indexes}\} \\ N\_PODS &= \sum_{m \in C} E\{(\#APP\_m\_ON)\} \end{aligned} \tag{12}$$

To gauge the necessity for system support and maintenance, metrics pertaining to availability have been formulated. In the complex environment of microservice architectures, replete with diverse components, worker nodes, and Pods, these metrics are vital for performance assessment. A system devoid of adequate support might grapple with disruptions from unrepaired components, while an excessively staffed support team might squander resources. To address this, two metrics have been introduced: Support Utilization (*SU*) and Lack of Support (*LS*), aiming to appraise the efficiency of the support team. The *SU* metric, depicted in Equation 13, quantifies the fraction of the support team that is actively engaged, calculated by the ratio of the utilized support teams to the total system support. Conversely, the *LS* metric, as defined in Equation 14, computes the likelihood of a component being in need of support when none is accessible. Elevated values of this metric flag potential inadequacies in support team allocation, which can precipitate prolonged component repair durations.

$$\begin{aligned} SU &= \frac{team\_number - E\{\#TEAM\_AVAILABLE\}}{team\_number} \\ \mathrm{U} &= \{n | n \text{are the worker nodes}\}, \\ \mathrm{T} &= \{m | m \text{are the indexes of Pods}\}, \end{aligned} \tag{13}$$

$$\begin{aligned} LS = P\{&(\#TEAM\_AVAILABLE = 0) \wedge ((\#CP\_etcd\_FAIL) \vee \\ &(\#CP\_UN\_FAIL > 0) \vee (\#CP\_SCD\_FAIL > 0) \vee \\ &(\#CP\_CM\_FAIL) \vee (\#CP\_API\_FAIL > 0) \vee \\ &(\#CP\_STORAGE\_FAIL) \vee (\#N_n\_UN\_FAIL > 0) \vee \\ &(\#N_n\_CR\_FAIL) \vee (\#N_n\_KP\_FAIL > 0) \vee \\ &(\#N_n\_KL\_FAIL) \vee (\#APP\_m\_ON))\} \end{aligned} \tag{14}$$

The Electrical Power (*P*) metric, as represented in Equation 15, allows determining the energy consumption of the system in Watts (W). The equation considers the different configurations, the dependability metrics, and their corresponding energy consumption. To calculate this metric, it is necessary to provide the average power consumption of various system components, such as the power consumed by the Control Planes (*pw_cp*), Storage (*pw_storage*), Worker Nodes (pw_n_n), Pods on each Node ($pw\_n_n\_app_m$), as well as the power used to instantiate a Pod on a Node ($pw\_up\_n_n\_app_m$). The Electrical Power metric is important as it allows for evaluating the system's energy efficiency and identifying areas for optimization.
F=(m, n) | *m* are the indices of the Pods, *n* are the indexes of Nodes eligible for Pod *m* allocation,

$$\begin{aligned} P = {} & E\{(\#CP\_UN\_ON)\} \times pw\_cp \\ & + E\{\#STORAGE\_ON\} \times pw\_storage \\ & + \sum_{(m,n) \in F} (E\{\#N_n\_UN\_ON\} \times pw\_n_n \\ & + E\{\#N_n\_INST\_APP_m\} \times pw\_n_n\_app_m \\ & + E\{\#N_n\_TO\_INST\_APP_m\} \times pw\_up\_n_n\_app_m) \end{aligned} \tag{15}$$

Accurately quantifying energy output from physical nodes requires precise instrumentation, while assessing energy consumption of Pods demands a rigorous computational methodology. This involves leveraging mathematical models tailored to the unique characteristics and operational requirements of each container. Such models often find their roots in empirically validated energy studies from scholarly sources, such as those presented by Fieni et al. [40] and Zhang et al. [41]. Energy consumption, as consolidated from these comprehensive analyses, is calculated using Equation (16). This equation is structured to holistically capture temporal dynamics, ensuring a precise representation of energy usage over time.

$$E = P \times Time \tag{16}$$

Ultimately, we have the metric outlined in Equation (17), which enables the identification of the rate of Pod instantiation within the system, denoted as *POD_IR*. This metric facilitates the identification of potential issues with the Worker nodes, which may result in the constant restarting of Pods within the system. High values of this metric also imply increased energy expenditure associated with initiating Pods without actual execution of requests. The equation correlates the expected value of tokens representing instantiated Pods with the average transition time of their instantiation, denoted as ttu_app_m.

F=(m, n) | m are the indices of the Pods, n are the indexes of Nodes eligible for Pod m allocation,

$$POD\_IR = \sum_{(m,n)\in F} E\{\#N_n\_TO\_INST\_APP_m\} \times ttu\_app_m \tag{17}$$

V. CASE STUDIES

This section presents three case studies to critically examine the dynamics of key metrics: availability, reliability, and power consumption across various scenarios. These studies illustrate the model's responsiveness to parameter variations, highlighting scenarios that might be challenging or costly to implement in real systems. Such challenges might emerge when the system operates in a production environment where experimental changes could negatively impact end-users, or when the system is still in its planning phase. The models were executed using the Mercury tool [39]. Across the case studies, the architectural components remain consistent, differentiated only by the number of Nodes, Pods, or support teams. Both control plane and worker node components within the cloud are based on the same hardware configuration, adopting the MTTF values listed in Table (11). These values were derived from the works of Rosendo et al. [42], Torquato et al. [43], and Melo et al. [44], combined with data from hardware manufacturers. A separate set of values was designated for edge nodes, with cloud-based values detailed in Table (11). As previously described, these values were used to populate the DRBD model shown in Figure (8), which determined the MTTF of the base system and informed the SPN model for the nodes. Additionally, the storage components, as illustrated in Figure (9), align with the Underlying System's attributes, with specific components detailed in Table (11).

TABLE 11. Underlying system parameters.

Submodel	Parameter	Value
Cloud	CPU MTTF	292000.0 h
	MEM MTTF	480000.0 h
	HD MTTF	719424.46 h
	HD KooN	2/3
	HD MTTR	1.0 h
	NIC MTTF	120000.0 h
	NIC KooN	1/2
	NIC MTTR	1.0 h
	VM MTTF	2880 h
	POWER SUP. MTTF	610000.0 h
Fog	HD MTTF	431654.6763 h
	VM MTTF	1440.0 h
	POWER SUP. MTTF	200000.0 h
Storage	HD MTTF	431654.6763 h
	HD KooN	10/12 h
	SYSTEM MTTF	2893.0 h
	POWER SUP. MTTF	200000.0 h

To provide input to the CTMCs that will produce the MTTF values of the Pods, the values from Table (12) were utilized. These values represent application failure rates, with Pod 01 containing a MongoDB database, Pod 02 containing a NodeJs application, and Pod 03 containing a frontend application. The values of these rates were obtained from the studies of Melo et al. ([44]).

TABLE 12. Microservices parameters.

Submodel	Parameter	Value
App and Containers	APP 01 failure rate	0.00069 (h^{-1})
	APP 02 failure rate	0.00126 (h^{-1})
	APP 03 failure rate	0.00245 (h^{-1})
	Container failure rate	0.00079 (h^{-1})
PODs	APP_1_MTTR	1 h
	APP_1_RST	0.03 h
	APP_2_MTTR	1 h
	APP_3_MTTR	1 h

To finalize the foundation models, one must ascertain the Mean Time to Failure (MTTF) values, and subsequently integrate these values into the timed transitions of the Stochastic Petri Net (SPN) models. For timed transitions not addressed by the foundational models, values should be sourced from either the academic literature or the historical records of the organization in question. For the Control Plane and Storage components, we adopt values from Table 13. The selection of these MTTFs and Mean Time to Repair (MTTRs) aims to mirror software configurations, as delineated in studies by Melo et al. and Kharchenko [44], [45]. Rather than being arbitrary, these choices stem from a careful examination of comparable systems, ensuring the values are emblematic of the nuances and behaviors typical of similar microservice architectures. This precision underscores our dedication to devising a model that is both academically rigorous and pragmatically relevant to contemporary computing landscapes.

TABLE 13. Node parameters.

Submodel	Parameter	Value
Control Plane	etcd MTTF	788.4 h
	etcd MTTR	1 h
	etcd RESTART	0.0166 h
	SCHEDULER MTTF	360 h
	SCHEDULER MTTR	1 h
	SCHEDULER RESTART	0.0166666 h
	CONTROLLER MANAGER MTTF	360 h
	CONTROLLER MANAGER MTTR	1 h
	CONTROLLER MANAGER RESTART	0.016666 h
	API SERVER MTTF	480 h
	API SERVER MTTR	1 h
	API SERVER RESTART	0.016666 h
	STORAGE MTTR	1 h
Worker node	CONTAINER RUNTIME MTTF	2516 h
	CONTAINER RUNTIME MTTR	1 h
	CONTAINER RUNTIME RESTART	0.0166666 h
	KUBE PROXY MTTF	576 h
	KUBE PROXY MTTR	1 h
	KUBE PROXY RESTART	0.01666 h
	KUBELET MTTF	788.4 h
	KUBELET MTTR	1 h
	KUBELET RESTART	0.01666 h
	Kubernetes NODE STATUS UPDATE	0.0833333 h
	TIME TO UP Pod 01 (cloud 1,2)	0.03 h
	TIME TO UP Pod 02 (cloud 1,2)	0.02 h
	TIME TO UP Pod 03 (cloud 1,2; fog 1)	0.05 h
	CLOUD NODE 01 CAP	64
	CLOUD NODE 02 CAP	64
	FOG NODE 01 CAP	32

Utilizing Dynamic Reliability Block Diagrams (DRBDs) for both cloud and fog frameworks enabled the determination of the Worker nodes' Mean Time to Failure (MTTF). Concurrently, the allocation of values for other time-bound transitions is vital, as detailed in Table (12) for Kubernetes processes and utilized Pods. The node status update value aligns with Kubernetes' default settings, though adjustments can provide insights into the repercussions of such changes. Pod initiation times can vary based on hardware and the Pods, especially in environments with resource constraints and lightweight Kubernetes iterations like KubeEdge, k3s, or MicroK8s ([8], [46], [47]). Additionally, the model recognizes potential variations in Pod hosting capacity due to hardware differences between cloud and fog computing. Thus, the model offers flexibility, capturing the nuanced dynamics of microservice architectures across diverse configurations.

Pods were hierarchically modeled, incorporating values derived from the CTMC's absorbing state. Alongside these, values for other transitions and the number of each Pod type are essential, as outlined in Table (12). Of note, only the App 01 Pod possesses the APP_1_RST timed transition, being the unique stateful Pod, specifically a database. The remaining Pods are storage-independent, exempt from failure due to condition [g16]. The initial distribution of Pod types among worker nodes, as determined by Equation (5), is another pivotal consideration. In conclusion, the $RETURN\_TIME$ timed transition for the support block depicted in Figure (6) is designated as 0.1 hours. This timeframe, however, can vary based on organizational and system specifics. All time distributions are presumed exponential. Post configuration, power values of system components, detailed in Table (14), are integrated to determine overall power consumption across diverse system setups.

TABLE 14. Power consumption parameters.

Parameter	Value (W)
pw_cp	80
pw_storage	90
pw_n01	50
pw_n01_app_01	5
pw_n01_app_02	7
pw_n01_app_03	5
pw_n02	50
pw_n02_app_01	5
pw_n02_app_02	7
pw_n02_app_03	5
pw_fog_01	40
pw_fog_n01_app_03	11
pw_up_app_01_n01	0.00001
pw_up_app_02_n01	0.00001
pw_up_app_03_n01	0.00001
pw_up_app_01_n02	0.00001
pw_up_app_02_n02	0.00001
pw_up_app_03_n02	0.00001
pw_up_app_03_fog_n01	0.00003

A. SCENARIO ONE - CHANGING THE FAILURE REQUIREMENT

The primary focus of the initial case study is to analyze the system's dynamic behavior as the number of Required Pods increases, shedding light on the system's capacity to handle more demanding Service Level Agreements (SLAs) related to Pod count. An expansion in system size might exacerbate the negative impacts of failures. This model aids strategic planning by offering a comparison of key metrics across different SLAs and approaches. The configuration presented in Figures 12 and 13 was adopted for this purpose. The system comprises a Control Plane, dedicated Storage, two cloud-based Worker nodes, and a fog-based Worker node. Designed to support various Pods, each falls into one of three application categories. Each Pod contains a single container with its respective application, as shown in Figure 13. Database and back-end Pods are confined to the cloud, whereas frontend Pods can operate in both cloud and fog layers. The system also incorporates a singular support team, depicted in Figure 13. This configuration represents a typical microservices system, offering insights into its dependability and performance. Beyond theoretical insights, this model provides actionable guidance for optimizing real-world systems, considering performance, dependability, and architectural nuances.

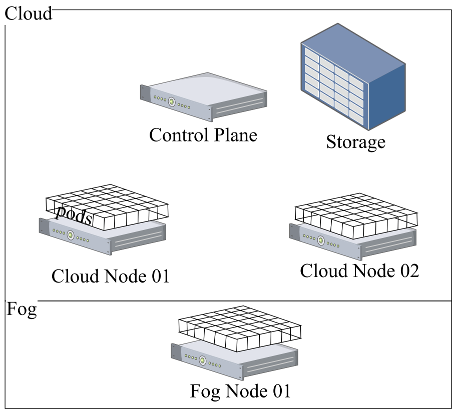


FIGURE 12. Use Case 01 - Physical Infrastructure.

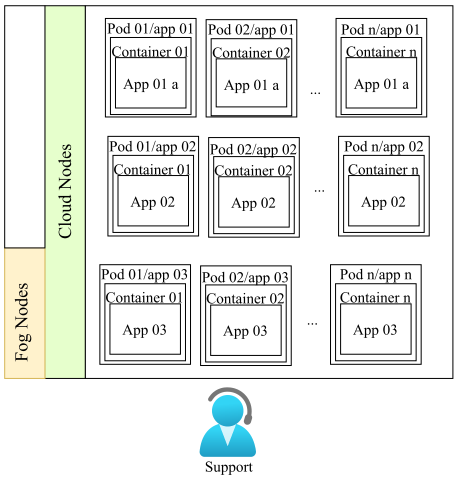


FIGURE 13. Use Case 01 - Configuration.

The analytical examination commences with the deployment of four Pods of type 1, four Pods of type 2, and two Pods of type 3. In each successive iteration, the count of type 1 and type 2 Pods is augmented by 2, while the quantity of type 3 Pods is incremented by 1. This progressive escalation in the number of Pods perpetuates until the configuration encompasses 58 Pods each of types 1 and 2, concomitant with 29 Pods of type 3, culminating in an aggregate of 145 Pods. This model captures the growth in demand and the necessary increase in the minimum Pod count required to meet this demand. Figure 14 plots system availability against the growing number of Pods. Starting at 96.73% availability with 10 Pods, it decreases to 72.23% with 145 Pods. This drop is expected since adding more components raises the likelihood of system failures. These results correspond to a system with only the essential number of Pods without redundancy. However, the model allows for redundancy beyond the basic demand. Table 15 outlines various redundancy strategies. Strategy A shows a notable rise in system availability: from 98.08% with 10 Pods to 88.37% with 145 Pods—an improvement of 1.4% and 22.3% over the non-redundant system. Yet, the benefits of adding more Pods diminish with each increase. For instance, Strategy B achieves 98.46% availability with 10 Pods and 91.67% with 145 Pods—only a slight enhancement compared to Strategy A. Figure 14b portrays the power consumption against the Pod count across different redundancy strategies. Unsurprisingly, the non-redundant configuration is the most energy-efficient. Strategy A consumes 3.1% more power with 10 Pods and 1.3% more with 145 Pods than the non-redundant setup. Increased redundancy levels, as seen in strategies B through F, correspondingly raise power consumption.

There's a notable disparity between the increases in system availability and the associated energy costs from infrastructural changes. This dynamic is tied to redundancy strategies and the number of Pods in the system. To delve deeper into the relationship between these metrics, we recommend analyzing the percentage increase in both availability and energy consumption across different redundancy strategies and Pod counts. These percentage increases can be compared using Figure 15. In this figure, the x-axis lists the redundancy strategies, while the left y-axis highlights the increase in availability, and the right y-axis indicates the rise in energy consumption.

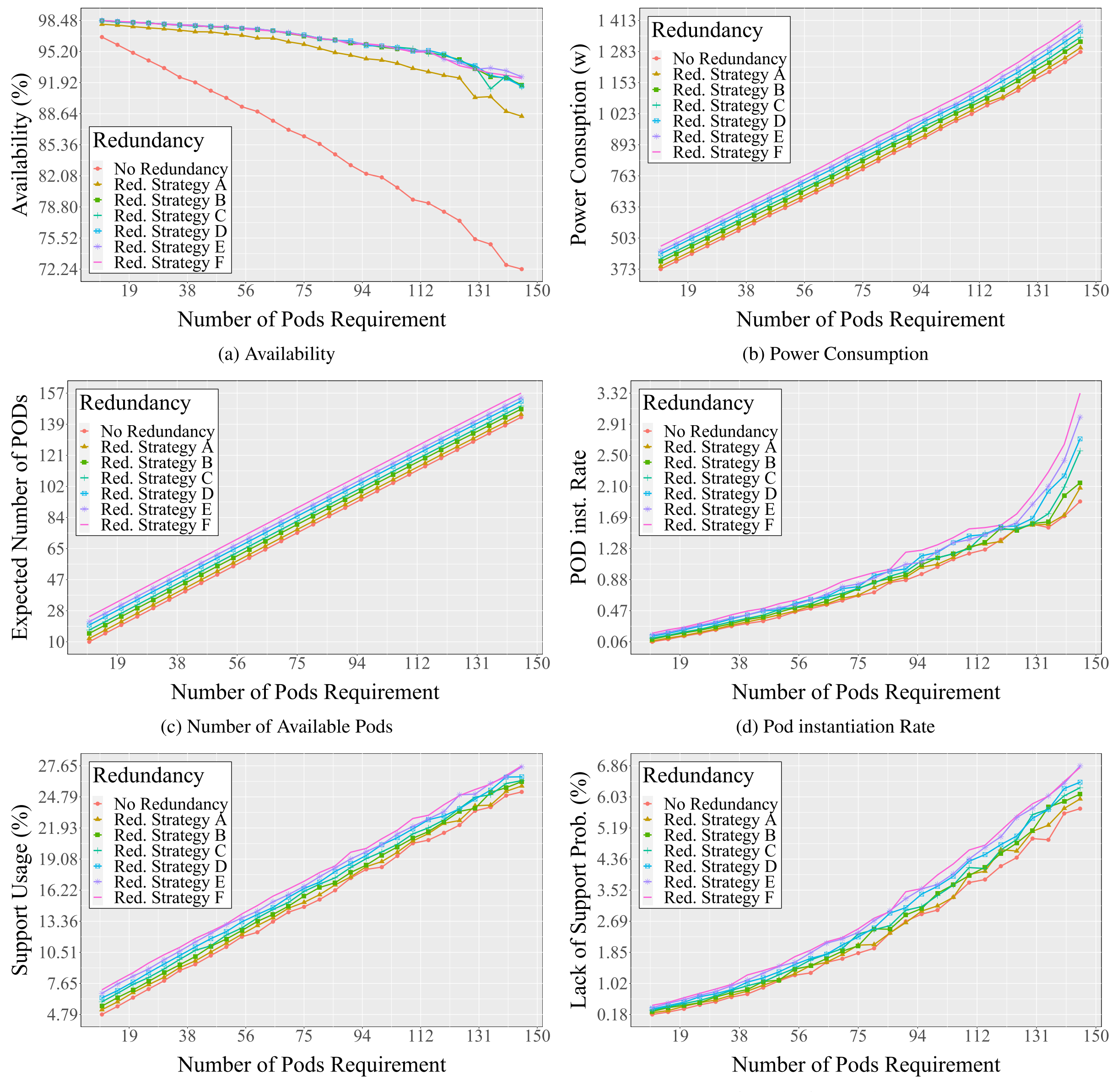
TABLE 15. Pods added in each redundancy strategy.

Redundancy Strategy	Pod type 1	Pod type 2	Pod type 3
No Redundancy	0	0	0
Red. Strategy A	1	1	0
Red. Strategy B	2	2	1
Red. Strategy C	3	3	1
Red. Strategy D	4	4	2
Red. Strategy E	5	5	2
Red. Strategy F	6	6	3

In Figure 15a, analyzing a system with 40 Pods, the availability percentage increase from no redundancy to Strategy A (denoted $NR - A$) is around 5.81%. However, the shift from Strategy E to F only registers a minor 0.07% growth. Contrarily, the energy consumption across strategies fluctuates between 1.99% and 3.51%, attributable to varying Pod increments per strategy, as detailed in Table 15. Notably, Strategy A yields a favorable availability-to-power consumption ratio, unlike subsequent strategies where availability gains diminish relative to power costs. For a system with 145 Pods, Figure 15b depicts a significant 22% availability hike versus a modest 1.3% power increase from no redundancy to Strategy A. Transitioning from Strategy A to B yields a 3.74% growth in availability contrasted with a 1.9% rise in power. Later strategies exhibit diminishing availability gains in relation to power surges. These insights can guide stakeholders in assessing the merits of redundancy, balancing availability benefits against power consumption costs.

A supplementary metric pivotal to system assessment is the anticipated number of Pods supported by the system, illustrated in Figure (14c). Typically, the expected number of Pods is less than the total set for system execution, resulting from Pod and related system component failures. This metric determines if the redundancy in Pods meets the demand. Without redundancy, a 10-Pod requirement yields about 9.97 Pods, while a 145-Pod requirement results in roughly 143.11 Pods. In the latter case, the system faces potential performance challenges due to the anticipated shortfall of 1.89 Pods, alongside existing availability issues. Implementing Redundancy Strategy A reveals a notable rise in the projected Pod count: 11.96 Pods for a 10-Pod requirement and 144.89 Pods for a 145-Pod requirement. For the 10-Pod scenario, Strategy A alone meets the desired Pod count. However, for the 145-Pod demand, the expected number falls short. Meeting the 145-Pod demand necessitates combining Strategies A and B, yielding an estimated 147.91 Pods. This highlights that even with orchestrator configurations to maintain a certain Pod count, system imperfections can lead to lower realized values. Hence, to fulfill a Service Level Agreement (SLA), more Pods than stipulated by the SLA might be required. Additionally, the choice of redundancy strategy for Pods depends on system size, requiring a tailored, nuanced evaluation.

Figure 14d portrays the Pod instantiation rate, a metric capturing the frequency of Pod reinitialization due to perturbations in infrastructure. Specifically, Worker node failures cause the removal of associated Pods, denoted by guard condition [$g12$]. Should resources on other nodes be available, affected Pods are reallocated. This metric assists administrators in detecting reduced node capacity, which might result from excessive instantiations and energy inefficiencies. For instance, without redundancy and a 10-Pod requirement, the rate is 0.062 (Pods/h), but it rises to 1.89 (Pods/h) for 145 Pods. Redundancy Strategy A slightly elevates these rates, while Strategy F, which maximally utilizes infrastructure, significantly increases them. A surge in Pod numbers naturally elevates the instantiation rate,



(a) Availability

(b) Power Consumption

(c) Number of Available Pods

(d) Pod instantiation Rate

(e) Support Usage

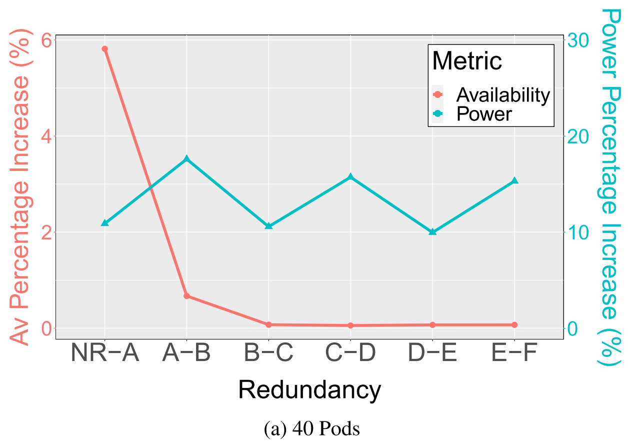
(f) Lack of Needed Support

FIGURE 14. Use Case 01 - Results.

TABLE 16. Transitions to CP and storage reliability models.

Transition	Index	Guard Expression	Description
CP_UN_R	[g18]	(#TEAM_AVAILABLE>0)AND (#CP_UN_ON>0)	Support team available to support and $\#CP\_UN\_ON$ available
CP_etcd_R	[g19]	(#TEAM_AVAILABLE>0)AND (#CP_etcd_ON>0)	Support team available to support and $\#CP\_etcd\_ON$ available
CP_SCD_R	[g20]	(#TEAM_AVAILABLE>0)AND (#CP_SCD_ON>0)	Support team available to support and $\#CP\_SCD\_ON$ available
CP_CM_R	[g21]	(#TEAM_AVAILABLE>0)AND (#CP_CM_ON>0)	Support team available to support and $\#CP\_CM\_ON$ available
CP_API_R	[g22]	(#TEAM_AVAILABLE>0)AND (#CP_API_ON>0)	Support team available to support and $\#CP\_API\_ON$ available
STORAGE_R	[g23]	(#TEAM_AVAILABLE>0)AND (#STORAGE_ON>0)	Support team available to support and $\#STORAGE\_ON$ available

due to increased Worker node capacities and more frequent instantiations following node failures. Figure 14e then delineates the correlation between support utilization and requisite Pod numbers. Increasing Pod numbers enhances support utilization across all strategies. For instance, without redundancy and with 10 Pods, support utilization probability is 4.79%, but this rises to 25.25% for 145 Pods. Even with maximal redundancy (Strategy F), the increase in support utilization remains modest. Figure 14f delves into the probability of support personnel shortages, providing insights for optimizing staffing. A heightened shortage probability suggests potential delays in system recovery post-failure,



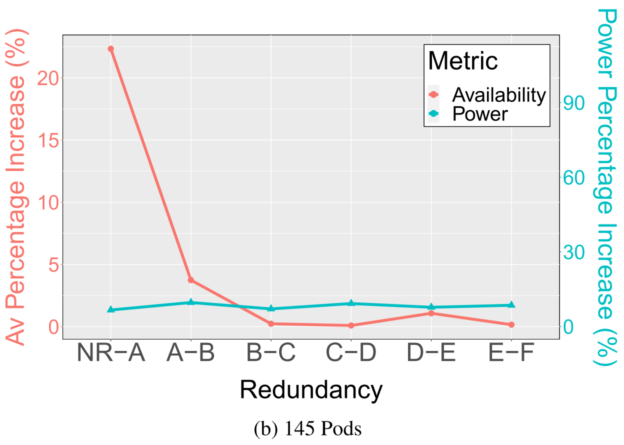
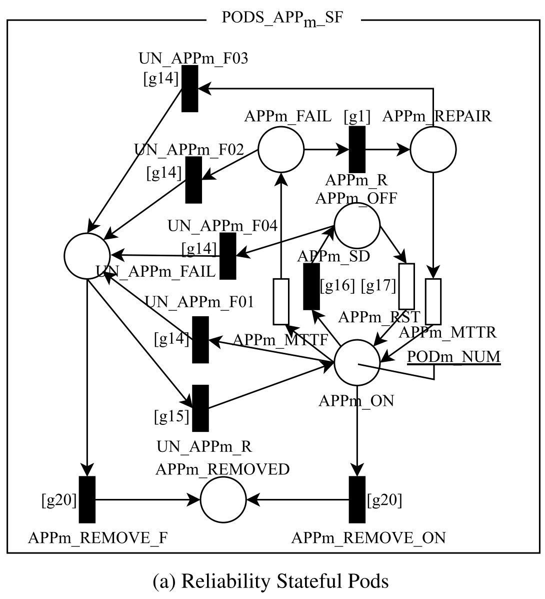


FIGURE 15. Use case 01 - Percentage increase.



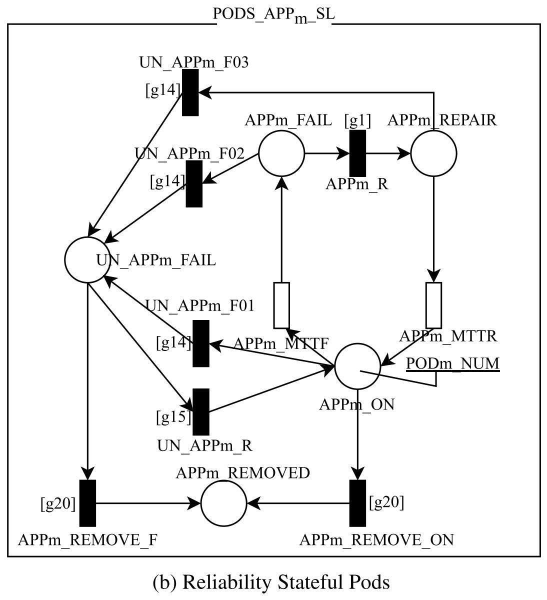


FIGURE 16. Reliability high-level Pods model.

emphasizing the necessity for strategic resource planning in microservice architectures.

In evaluating system dependability, it's paramount to consider the likelihood of support unavailability. For a configuration with 10 non-redundant Pods, this probability is a mere 0.1%, but it surges to 5.71% for a system with 145 Pods. Implementing Strategy F, the highest redundancy level, sees this probability reduce to 0.4% for 10 Pods but increase to 6.81% for 145 Pods. While redundancy overhead is modest for this metric, potential delays in repair due to accumulated maintenance requests might arise. In cases where repair time is contractually set, adding more support teams could mitigate repair delays. Enhancing this dependability analysis is a deep dive into system reliability, which assesses the system's consistent functionality over a set period. Specific model components, including the Control Plane models (Figure 3), SPN Pods (Figure 5), and the Storage subsystem (Figure 7), underwent targeted modifications to enable transient simulations, capturing system unavailability probabilities at specific times, especially when recovery isn't possible. The nuances of these changes will be discussed subsequently.

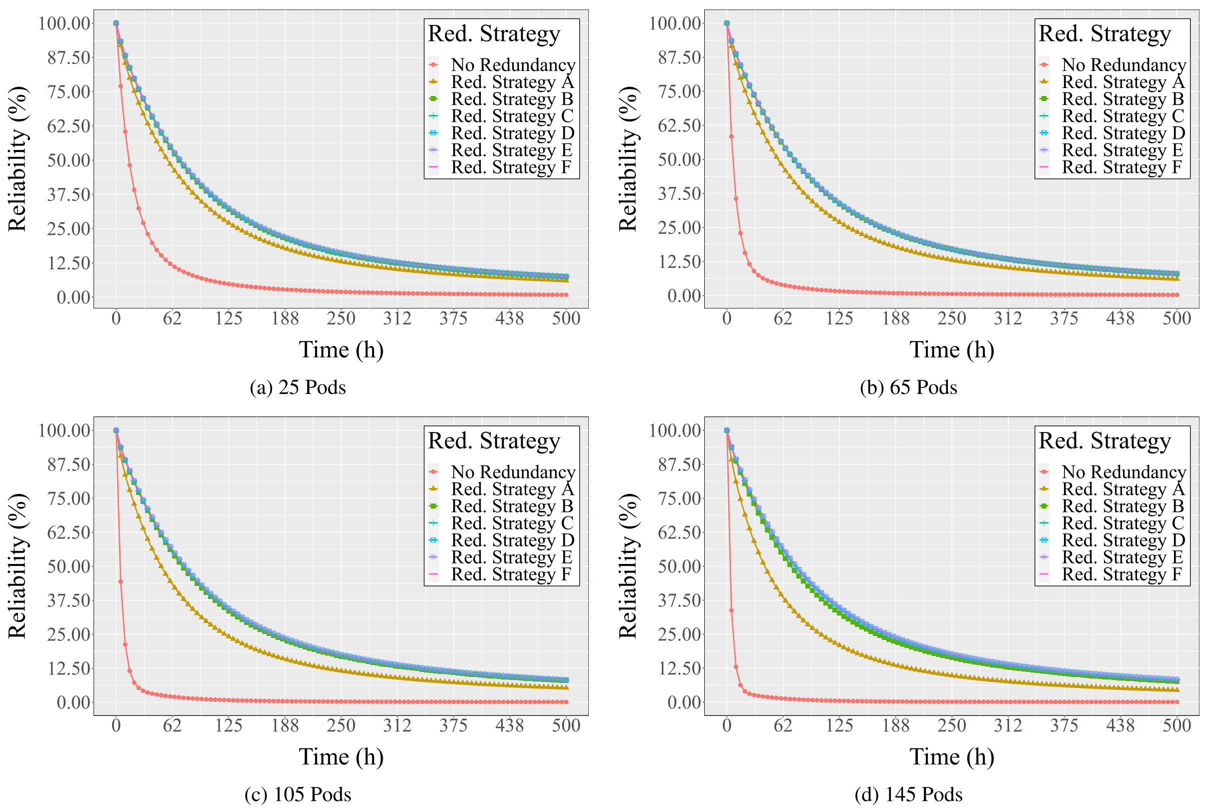
Both the Control Plane and Storage models had alterations in their immediate transition guard expressions (specifically, index expressions [$g1$]) vital for repairs. With their critical role, their downtime equates to system failure. The modified guard expressions ensure the presence of at least one operational token for each component. These updated expressions can be found in Table 16. Besides the support team's availability, components must be active for maintenance; otherwise, repairs don't proceed, affecting reliability determination. Moreover, to be considered operational, the

TABLE 17. **Reliability pods transitions.**

Transition	Description	Firing Semantic	Weight	Priority
APPm_REMOVE_F	Remove failed Pods	-	1	5
APPm_REMOVE_ON	Remove available Pods	-	1	5

TABLE 18. **Reliability pods guard expressions.**

Index	Guard Expression	Description
[g20]	$\#APP_m\_ON < MIN\_POD_m$	If the number of Pods m is less than the minimum value, they will be removed.



(a) 25 Pods

(b) 65 Pods

(c) 105 Pods

(d) 145 Pods

FIGURE 17. **Use Case 01 - Reliability results.**

system must sustain a minimum number of each Pod type. Given the Pod's mobility across Worker Nodes, only their overall operational count verifies their system status. The SPN Pods model was adapted to discard Pods if their count drops below the requisite number. New transitions were introduced to the SPN Pods model for removing failed or operational Pods and an added place for discarded tokens, depicted in Figure 16. The descriptions for these additions are in Tables 17 and 18. Aimed at discarding Pods when the system or its components fail, these changes apply to all system Pods.

The reliability analysis for the given configuration is visually represented in Figure 17, which showcases the system under varying Pod requirements: specifically, 25, 65, 105, and 145 Pods, reflecting counts without redundancy. Each sub-figure in 17 offers insights into the system's reliability across different redundancy strategies, starting from an initial operational probability of 100% at time zero. Taking Figure 17a as a case in point, a system without redundancy exhibits the lowest reliability values over the studied duration. After 100 hours of operation, its reliability dwindles to 6.44%, plummeting further to 0.84% after 500 hours. In contrast, under Redundancy Strategy A, the system displays slightly improved reliability. After 100 hours, it stands at 33.27%, decreasing to 6.03% by the 500-hour mark. This underscores the incremental benefits of redundancy but also highlights the need for more robust strategies to sustain reliability over extended periods.

Figure 17a reveals that Strategy B yields reliabilities of 38.59% and 7.34% at the 100 and 500-hour junctures, respectively. Subsequent strategies offer marginal improvements over Strategy B, with the transition from no

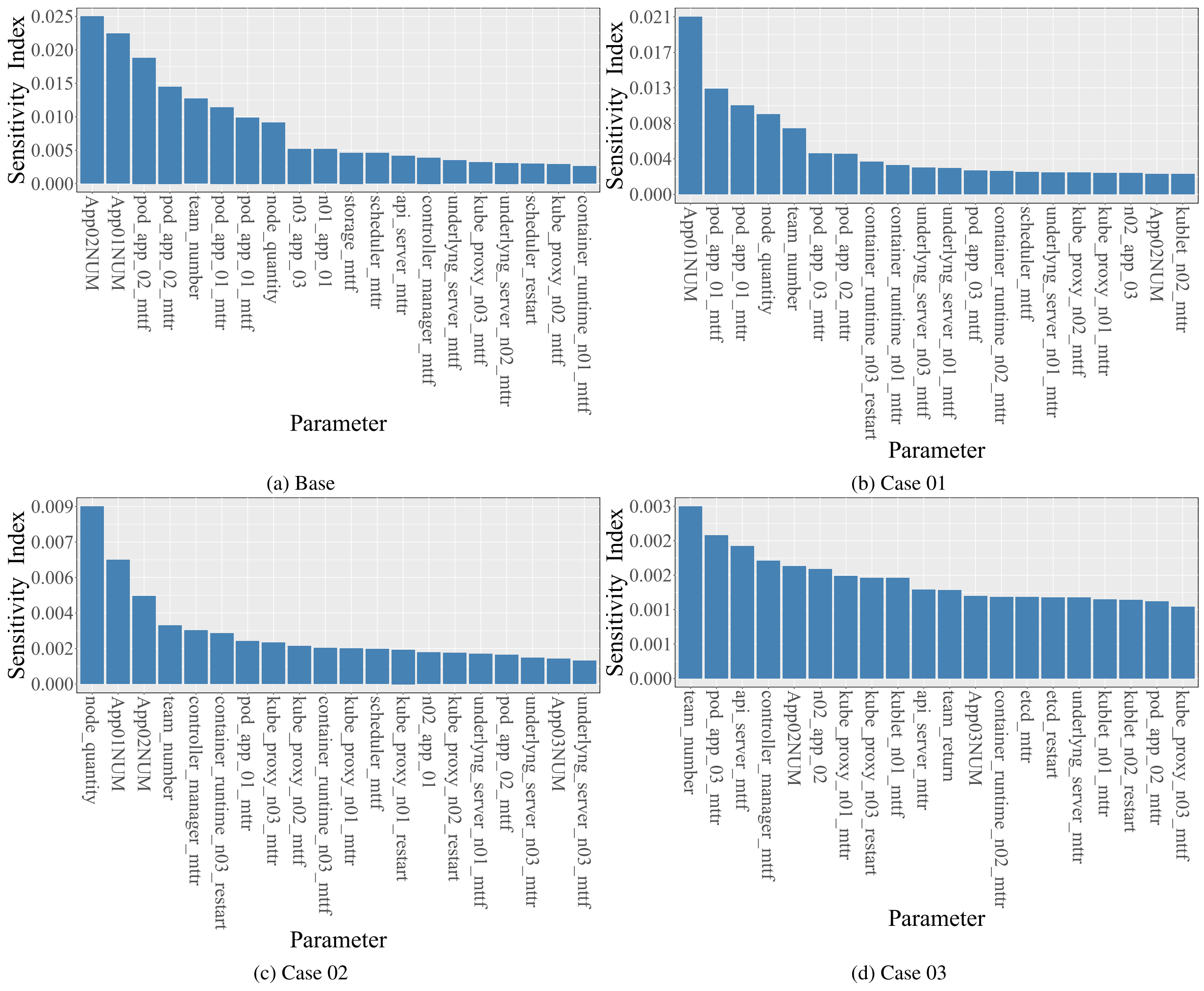


FIGURE 18. **Use Case 02 - Sensitivity analysis.**

redundancy to Strategy A marking the most pronounced reliability enhancement: 416% at 100 hours and 610% at 500 hours. A subsequent transition from Strategy A to B results in reliability gains of 16% and 22% at these respective timeframes. This figure underscores a diminishing return on reliability as redundancy levels increase. Yet, as the availability analysis indicated, redundancy accrual has concomitant power implications. Hence, decision-makers must judiciously evaluate the marginal benefits of each redundancy tier against its costs. A comparative analysis across Figures 17a, 17b, 17c, and 17d offers insights into reliability shifts with escalating system demand. As Pod requirements rise, there's a notable reliability degradation. For a non-redundant system at 100 hours, reliabilities are 6.44%, 2.15%, 1.10%, and 0.06% for Pod requirements of 25, 65, 105, and 145 respectively. This reliability decay is visually evident, with steeper declines for each increment in Pod demand. Conversely, under Strategy A across the same period, the reliability figures remain relatively stable: 33.27%, 33.00%, 29.79%, and 25.94% for Pod requirements of 25, 65, 105, and 145 respectively. These subtler reductions, juxtaposed with the non-redundant scenario, underscore the resilience imparted by redundancy. The data suggests that optimal deployment strategies hinge on both the desired Pod count and the chosen redundancy level.

B. SCENARIO TWO - CHANGES GUIDED BY SENSITIVITY ANALYSIS

Designing microservices in hybrid cloud-fog environments demands meticulous selection and configuration of myriad components, often complicating the achievement of specific Service Level Agreements (SLAs). The intricacy of infrastructure planning stems from not only the abundance of potential configurations but also the unpredictable impact of these configurations on key metrics. This use case demonstrates how incremental sensitivity analysis in the model can guide the identification of pivotal changes, optimizing system availability.

$$S_\theta\{Y\} = \frac{max\{Y(\theta)\} - min\{Y(\theta)\}}{max\{Y(\theta)\}} \tag{18}$$

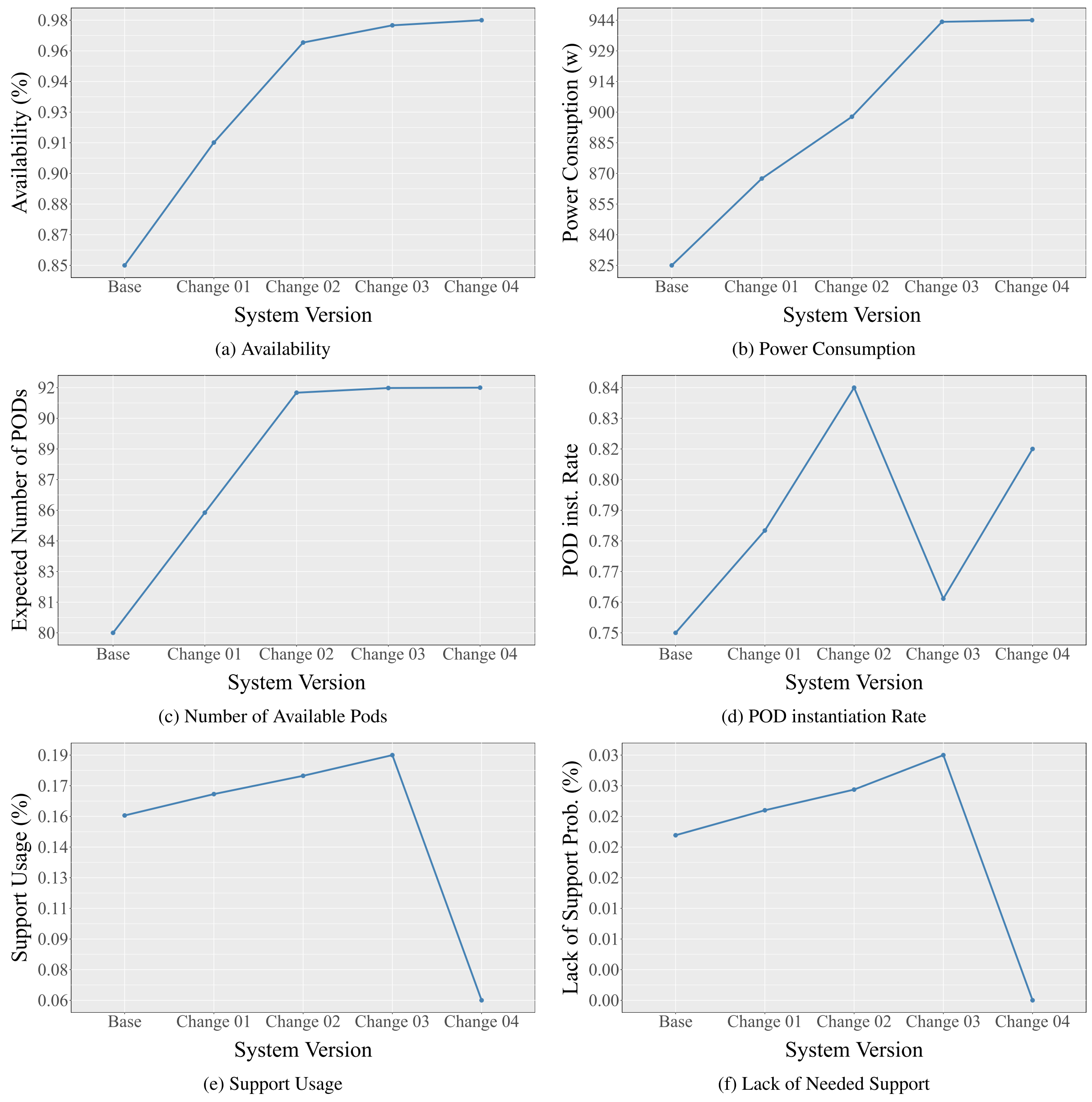


FIGURE 19. Use Case 02 - results.

The sensitivity analysis technique employed was the percentage difference [39], which involves varying a parameter within a specific range and then identifying its highest and lowest values, as outlined in Equation 18. The variables max $Y(\theta)$ and min $Y(\theta)$ represent the highest and lowest values found in the metric of interest for the parameter within the range. This technique was applied to the parameters of the model components. As a result, the values of MTTF and MTTR of the components and processes, the number of worker nodes, the number of support teams, and the number of Pods of each type, were among the varied model parameters. A variation range 10% was used, with a minimum of 1 variation when the parameter is an integer. With each application of this technique, a modification was made to the component with the greatest impact until a 97.5% availability rate was achieved. The nature of the alteration to be made to the element with the highest impact is contingent upon its type. Configuration variables can have their value modified in the direction of the increasing availability, while elements that are not amenable to change, such as the MTTF of the container, had their number of components increased to achieve a redundancy effect.

In evaluating the system's dependability, understanding support unavailability is pivotal. For a base configuration comprising 32 Pods each for types 1 and 2, and 16 for type 3, with a single control plane, two cloud-based Worker nodes, one Fog-based Worker node, and one support team, the initial system availability is 85.11%. Sensitivity analysis, shown in Figure 18a, highlighted APP_{02}_NUM as the most impactful element. Adding six redundant Pods of this type improved availability to 91.32%. Subsequent analyses, as detailed in Figures 18b, 18c, and 18d, led to changes resulting in availabilities of 96.37%, 97.24%, and finally 97.50%, meeting desired thresholds. In Figure 17a, Strategy B yielded reliabilities of 38.59% and 7.34% at 100 and 500 hours. Comparatively, Strategy B's improvements were marginal to other strategies, but significant relative to non-redundant systems. The reliability increased by 416% and 610% at 100 and 500 hours, respectively, when transitioning from no redundancy to Strategy A, and by 16% and 22% when comparing Strategies A and B.

As observed in Figure 17a, increasing redundancy layers marginally improves reliability but incurs additional energy costs. Hence, system managers must judiciously assess the cost-benefit dynamics of each enhancement. Analyzing redundancy strategies' reliability against system demand, as depicted across Figures 17a to 17d, reveals decreasing reliability with increased Pod requirements. For a non-redundant system, 100-hour reliabilities for 25, 65, 105, and 145 Pods are 06.44%, 02.15%, 01.10%, and 00.06%, respectively. In contrast, with Redundancy Strategy A, these values are 33.27%, 33.00%, 29.79%, and 25.94%. The study underscores that reliability assessments are contingent upon the Pod count and chosen redundancy strategy.

VI. CONCLUSION

This study presents a sophisticated methodology for evaluating the reliability and energy characteristics of cloud-fog systems based on the Kubernetes framework. The developed model covers essential determinants affecting system reliability and provides a detailed analysis of how parameter changes impact key metrics, such as availability, reliability, Pod quantities, instantiation speed, and system failure risks. Two case studies demonstrated the model's efficacy. The first revealed that increasing the minimal Pod requirement reduces system availability and reliability. The second utilized the model to guide improvements toward achieving a specific availability target. Adopting this model-driven approach offers multiple benefits, including a structured method for understanding cloud-fog system reliability, a benchmark for comparing dependability metrics across systems, and optimizing computational resource allocation. Future research avenues include examining system performance issues and resource variability with auto-scaling, assessing both reliability and energy consumption concerning demand variability, and incorporating advanced optimization techniques for resource allocation in cloud-fog systems. In essence, this research significantly advances our understanding of the reliability and energy dynamics in cloud-fog computing and establishes a foundation for future studies on system optimization and performance evaluation.

REFERENCES

[1] X. Merino, C. Otero, D. Nieves-Acaron, and B. Luchterhand, "Towards orchestration in the cloud-fog continuum," in *Proc. SoutheastCon*, Mar. 2021, pp. 1–8.

[2] Z. Bakhshi, G. Rodriguez-Navas, and H. Hansson, "Dependable fog computing: A systematic literature review," in *Proc. 45th Euromicro Conf. Softw. Eng. Adv. Appl. (SEAA)*, Thessaloniki, Greece, Aug. 2019, pp. 395–403. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:mdh:diva-45152

[3] A. A. Alli and M. M. Alam, "The fog cloud of things: A survey on concepts, architecture, standards, tools, and applications," *Internet Things*, vol. 9, Mar. 2020, Art. no. 100177.

[4] M. Kumar, S. C. Sharma, A. Goel, and S. P. Singh, "A comprehensive survey for scheduling techniques in cloud computing," *J. Netw. Comput. Appl.*, vol. 143, pp. 1–33, Oct. 2019.

[5] T. A. Nguyen, D. Min, E. Choi, and J.-W. Lee, "Dependability and security quantification of an Internet of Medical Things infrastructure based on cloud-fog-edge continuum for healthcare monitoring using hierarchical models," *IEEE Internet Things J.*, vol. 8, no. 21, pp. 15704–15748, Nov. 2021.

[6] L. Peng, A. R. Dhaini, and P.-H. Ho, "Toward integrated cloud-fog networks for efficient IoT provisioning: Key challenges and solutions," *Future Gener. Comput. Syst.*, vol. 88, pp. 606–613, Nov. 2018.

[7] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Comput. Surveys*, vol. 55, no. 7, pp. 1–37, Jul. 2023.

[8] *KubeEdge Kubeedge Kubernetes Native Edge Computing Framework*. Accessed: Aug. 2, 2023. [Online]. Available: https://kubeedge.io/

[9] A. Luntovskyy and B. Shubyn, "Highly-distributed systems based on micro-services and their construction paradigms," in *Proc. IEEE 15th Int. Conf. Adv. Trends Radioelectronics, Telecommun. Comput. Eng. (TCSET)*, Feb. 2020, pp. 7–14.

[10] K. Camboim, J. Ferreira, C. Melo, J. Araujo, F. Alencar, and P. Maciel, "Dependability and sustainability evaluation of data center electrical architectures," in *Proc. IEEE Int. Syst. Conf. (SysCon)*, Apr. 2021, pp. 1–8.

[11] L. Charfeddine and M. Umlai, "ICT sector, digitization and environmental sustainability: A systematic review of the literature from 2000 to 2022," *Renew. Sustain. Energy Rev.*, vol. 184, Sep. 2023, Art. no. 113482.

[12] J. C. Wang, "Understanding the energy consumption of information and communications equipment: A case study of schools in Taiwan," *Energy*, vol. 249, Jun. 2022, Art. no. 123701.

[13] L. Belkhir and A. Elmeligi, "Assessing ICT global emissions footprint: Trends to 2040 & recommendations," *J. Cleaner Prod.*, vol. 177, pp. 448–463, Mar. 2018.

[14] L. J. Jagadeesan and V. B. Mendiratta, "When failure is (Not) an option: Reliability models for microservices architectures," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Oct. 2020, pp. 19–24.

[15] H. Khazaei, N. Mahmoudi, C. Barna, and M. Litoiu, "Performance modeling of microservice platforms," *IEEE Trans. Cloud Comput.*, vol. 10, no. 4, pp. 2848–2862, Oct. 2022.

[16] Z. Liu, G. Fan, H. Yu, and L. Chen, "An approach to modeling and analyzing reliability for microservice-oriented cloud applications," *Wireless Commun. Mobile Comput.*, vol. 2021, pp. 1–17, Aug. 2021.

[17] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "A Kubernetes controller for managing the availability of elastic microservice based stateful applications," *J. Syst. Softw.*, vol. 175, May 2021, Art. no. 110924.

[18] T. A. Nguyen, M. Kim, J. Lee, D. Min, J.-W. Lee, and D. Kim, "Performability evaluation of switch-over moving target defence mechanisms in a software defined networking using stochastic reward nets," *J. Netw. Comput. Appl.*, vol. 199, Mar. 2022, Art. no. 103267.

[19] F. Murtaza, A. Akhunzada, S. U. Islam, J. Boudjadar, and R. Buyya, "QoS-aware service provisioning in fog computing," *J. Netw. Comput. Appl.*, vol. 165, Sep. 2020, Art. no. 102674.

[20] X. Hou, C. Li, J. Liu, L. Zhang, Y. Hu, and M. Guo, "ANT-man: Towards agile power management in the microservice era," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2020, pp. 1–14.

[21] H. H. A. Valera, M. Dalmau, P. Roose, J. Larracoechea, and C. Herzog, "DRACeo: A smart simulator to deploy energy saving methods in microservices based networks," in *Proc. IEEE 29th Int. Conf. Enabling Technol., Infrastructure Collaborative Enterprises (WETICE)*, Sep. 2020, pp. 94–99.

[22] T. Menouer, "KCSS: Kubernetes container scheduling strategy," *J. Supercomput.*, vol. 77, no. 5, pp. 4267–4293, May 2021.

[23] M.-N. Tran, X. T. Vu, and Y. Kim, "Proactive stateful fault-tolerant system for Kubernetes containerized services," *IEEE Access*, vol. 10, pp. 102181–102194, 2022.

[24] A. Jeffery, H. Howard, and R. Mortier, "Rearchitecting Kubernetes for the edge," in *Proc. 4th Int. Workshop Edge Syst., Analytics Netw.* New York, NY, USA: ACM, Apr. 2021, pp. 7–12. [Online]. Available: https://dl.acm.org/doi/10.1145/3434770.3459730

[25] R. Pietrantuono, S. Russo, and A. Guerriero, "Testing microservice architectures for operational reliability," *Softw. Test., Verification Rel.*, vol. 30, no. 2, p. e1725, Mar. 2020.

[26] Z. Liu, H. Yu, G. Fan, and L. Chen, "Reliability modelling and optimization for microservice-based cloud application using multi-agent system," *IET Commun.*, vol. 16, no. 10, pp. 1182–1199, Jun. 2022.

[27] A. A. C. De Alwis, A. Barros, C. Fidge, and A. Polyvyanyy, "Availability and scalability optimized microservice discovery from enterprise systems," in *Proc. OTM Confederated Int. Conf. Move Meaningful Internet Syst.* Cham, Switzerland: Springer, 2019, pp. 496–514.

[28] A. A. Khatami, Y. Purwanto, and M. F. Ruriawan, "High availability storage server with Kubernetes," in *Proc. Int. Conf. Inf. Technol. Syst. Innov. (ICITSI)*, Oct. 2020, pp. 74–78.

[29] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A micro-service reference application for benchmarking, modeling and resource management research," in *Proc. IEEE 26th Int. Symp. Model., Anal., Simul. Comput. Telecommun. Syst. (MASCOTS)*, Sep. 2018, pp. 223–236.

[30] M. Xu, A. N. Toosi, and R. Buyya, "A self-adaptive approach for managing applications and harnessing renewable energy for sustainable cloud computing," *IEEE Trans. Sustain. Comput.*, vol. 6, no. 4, pp. 544–558, Oct. 2021.

[31] Z. Liu, H. Yu, G. Fan, and L. Chen, "Reliability modeling and analysis of hospital information system based on microservices," in *Proc. IEEE Int. Conf. Prog. Informat. Comput. (PIC)*, Dec. 2021, pp. 313–318.

[32] Z. Lyu, H. Wei, X. Bai, and C. Lian, "Microservice-based architecture for an energy management system," *IEEE Syst. J.*, vol. 14, no. 4, pp. 5061–5072, Dec. 2020.

[33] F. Jazayeri, A. Shahidinejad, and M. Ghobaei-Arani, "A latency-aware and energy-efficient computation offloading in mobile fog computing: A hidden Markov model-based approach," *J. Supercomput.*, vol. 77, no. 5, pp. 4887–4916, May 2021.

[34] *Kubernetes Production-Grade Container Orchestration*. Accessed: Aug. 2, 2023. [Online]. Available: https://kubernetes.io/

[35] F. A. Silva, C. Brito, G. Araújo, I. Fé, M. Tyan, J.-W. Lee, T. A. Nguyen, and P. R. M. Maciel, "Model-driven impact quantification of energy resource redundancy and server rejuvenation on the dependability of medical sensor networks in smart hospitals," *Sensors*, vol. 22, no. 4, p. 1595, Feb. 2022.

[36] C. Melo, J. Araujo, J. Dantas, P. Pereira, and P. Maciel, "A model-based approach for planning blockchain service provisioning," *Computing*, vol. 104, no. 2, pp. 315–337, Feb. 2022.

[37] E. Araujo, P. Pereira, J. Dantas, and P. Maciel, "Dependability impact in the smart solar power systems: An analysis of smart buildings," *Energies*, vol. 14, no. 1, p. 124, Dec. 2020.

[38] P. A. Lima, A. S. B. Neto, and P. Maciel, "Data centers' services restoration based on the decision-making of distributed agents," *Telecommun. Syst.*, vol. 74, no. 3, pp. 367–378, Jul. 2020.

[39] T. Pinheiro, D. Oliveira, R. Matos, B. Silva, P. Pereira, C. Melo, F. Oliveira, E. Tavares, J. Dantas, and P. Maciel, "The mercury environment: A modeling tool for performance and dependability evaluation," in *Proc. 17th Int. Conf. Intell. Environments Workshop*, vol. 29. Amsterdam, The Netherlands: IOS Press, 2021, p. 16.

[40] G. Fieni, R. Rouvoy, and L. Seinturier, "SmartWatts: Self-calibrating software-defined power meter for containers," in *Proc. 20th IEEE/ACM Int. Symp. Cluster, Cloud Internet Comput. (CCGRID)*, May 2020, pp. 479–488.

[41] R. Zhang, Y. Chen, B. Dong, F. Tian, and Q. Zheng, "A genetic algorithm-based energy-efficient container placement strategy in CaaS," *IEEE Access*, vol. 7, pp. 121360–121373, 2019.

[42] D. Rosendo, D. Gomes, G. L. Santos, L. Silva, A. Moreira, J. Kelner, D. Sadok, G. Gonçalves, A. Mehta, M. Wildeman, and P. T. Endo, "Availability analysis of design configurations to compose virtual performance-optimized data center systems in next-generation cloud data centers," *Softw., Pract. Exper.*, vol. 50, no. 6, pp. 805–826, Jun. 2020.

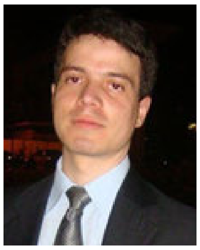
[43] M. Torquato, I. M. Umesh, and P. Maciel, "Models for availability and power consumption evaluation of a private cloud with VMM rejuvenation enabled by VM live migration," *J. Supercomput.*, vol. 74, no. 9, pp. 4817–4841, Sep. 2018.

[44] C. Melo, J. Dantas, D. Oliveira, I. Fé, R. Matos, R. Dantas, R. Maciel, and P. Maciel, "Dependability evaluation of a blockchain-as-a-service environment," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jun. 2018, pp. 909–914.

[45] V. Kharchenko, Y. Ponochovnyi, O. Ivanchenko, H. Fesenko, and O. Illiashenko, "Combining Markov and semi-Markov modelling for assessing availability and cybersecurity of cloud and IoT systems," *Cryptography*, vol. 6, no. 3, p. 44, Aug. 2022.

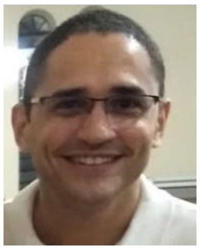
[46] *K3s Lightweight Kubernetes*. Accessed: Jan. 14, 2023. [Online]. Available: https://microk8s.io/

[47] *MicroK8s the Lightweight Kubernetes*. Accessed: Jan. 14, 2023. [Online]. Available: https://microk8s.io/



IURE FÉ received the degree in computer science from the Federal University of Piauí and the master's degree in computer science from the Federal University of Pernambuco. He is currently a Systems Analyst with the Brazilian Army. His research interests include distributed systems and performance evaluation of systems.

TUAN ANH NGUYEN (Member, IEEE) received the B.Eng. and M.Sc. degrees in mechatronics from the Hanoi University of Science and Technology (HUST), in 2008 and 2010, respectively, and the Ph.D. degree in computer science and systems engineering from the Department of Computer Engineering, Korea Aerospace University, in 2015. He was a Research Engineer with the TRON Laboratory and F-Space Laboratory, FPT Software and FPT Technology Research Institute (FTRI), Hanoi, Vietnam, in 2008 and from 2009 to 2010, respectively. From 2011 to 2015, he was a Ph.D. Research Associate with the Network Security and Systems Laboratory (NS Lab), Korea Aerospace University. From August 2015 to February 2016, he was a Postdoctoral Research Associate with the Distributed Multimedia Systems Laboratory (DMS Lab), Konkuk University. From March 2016 to February 2020, he was a Research (Assistant) Professor with the Office of Research, University-Industry Cooperation Foundation, Konkuk University. He is currently an Academic Research (Assistant) Professor of aerospace design-airworthiness research with the Konkuk Aerospace Design-Airworthiness Research Institute (KADA), Konkuk University, Seoul, South Korea. He has been a technical project leader of the project "Artificial Intelligence Digital Twin Research for Urban Air Mobility (UAM)" funded by the Korea National Research Foundation (NRF), from 2020 to 2029. His current research interests include dynamics and control theory and systems, AI-based digital twin systems and methods, computer science, and software engineering, with a specialization in dependable, autonomous, and intelligent systems, dependable computing and fault-tolerance of systems and networks, and mechatronics and aerospace robotic systems. He is a Professional Member of AIAA, IEEE Computer, Robotics and Automation (IEEE RAS), Aerospace and Electronic Systems (IEEE AESS), and Reliability Societies.



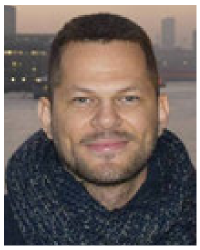
ANDRÉ C. B. SOARES was born in Teresina, Brazil. He received the B.Sc. degree in computer science from the Federal University of Piauí (UFPI), Teresina, in 2001, the M.Sc. degree in computer networks from Universidade Salvador, Salvador, Brazil, in 2004, and the Ph.D. degree in computer science from Universidade Federal de Pernambuco, Recife, Brazil, in 2009. He is currently a Professor with the Departamento de Computação, UFPI. He coordinates the Distributed Systems and Network Computer Laboratory (DisNeL), UFPI. His research interests include optical networks, the IoT, and eye tracking.

DUGKI MIN received the B.S. degree in industrial engineering from Korea University, in 1986, and the M.S. and Ph.D. degrees in computer science from Michigan State University, in 1991 and 1995, respectively. He is currently a Professor with the Department of Computer Science and Engineering, Konkuk University. His research interests include cloud computing, distributed and parallel processing, big data processing, intelligent processing, software architecture, and modeling and simulation.

SEOKHO SON received the B.S. degree in computer engineering from Pukyong National University, South Korea, and the M.S. and Ph.D. degrees in information and communications engineering from the Department of Information and Communications, Gwangju Institute of Science and Technology (GIST), South Korea, in 2013. He is currently a Senior Researcher/Special Fellow with the Electronics and Telecommunications Research Institute (ETRI), which is a national research institute in South Korea. He designs system architectures and devises algorithms to accomplish national research projects in the cloud computing domain. His research has been presented at several conferences, journals, and organizations. He is carrying out various activities, including international standardization (ITU-T international standards), reviewer of research papers, and contributor to open source projects. He is a founding member of the open-source project Cloud-Barista provides various functionalities to accomplish multi-cloud and is the leader and maintainer of the CB-Tumblebug project. Also, he is maintaining the CNCF Cloud Native Glossary project as well as the Kubernetes website project as an active contributor. He received several honors in the activities, including the Kubernetes Contributor Awards, in 2022.

JAE-WOO LEE received the B.S. and M.S. degrees in aerospace engineering from Seoul National University, Seoul, South Korea, and the Ph.D. degree from the Department of Aerospace Engineering, Virginia Tech, Blacksburg, VA, USA, in 1991. He is currently a Professor and the Director of the Konkuk Aerospace Design-Airworthiness Research Institute (KADA), Konkuk University. He is the corresponding author or coauthor of more than 570 publications, including 13 patents and 74 international journal articles. His research interests include multidisciplinary design and optimization, MDO, aerodynamic design and optimization, aerospace vehicle design for aircraft, space launchers, and UAV/Drones. He received several honors and awards in the academic career. He has been serving as the Conference Chair, the Technical Program Chair, and the Symposium Chair for various international conferences, including APISAT, KSAS, and KSAA. He served as a Specialized Member of the Defense Acquisition Committee, Ministry of National Defense, South Korea, and the Policy Planning Committee/Business Committee, DAPA, MND, South Korea. He served as the President of the Korean Society of Aeronautics and Space Sciences, KSAS, and the Korean Society of Design Optimization, KSDO.

EUNMI CHOI received the B.S. degree (Hons.) in computer science from Korea University, in 1988, and the M.S. and Ph.D. degrees in computer science from Michigan State University, East Lansing, MI, USA, in 1991 and 1997, respectively. Since March 1998, she has been an Assistant Professor with Handong University, South Korea, before joining Kookmin University, South Korea, in 2004. She is currently the Head of the Distributed Information System and Cloud Computing Laboratory, Kookmin University. She is also a Professor with Kookmin University. Her current research interests include big data infra system and analysis, cloud computing, intelligent systems, information security, parallel and distributed systems, and SW architecture and modeling. She received the Top Student Award for the B.S. degree.



FRANCISCO AIRTON SILVA received the B.S. degree in information systems from Instituto Federal do Piauí (IFPI), in 2008, and the M.S. and Ph.D. degrees in computer science from Universidade Federal de Pernambuco (UFPE), in 2013 and 2017, respectively. In 2015, he studied with Sapienza Università di Roma. He was a Collaborative Researcher in multinational technology companies, such as EMC and OKI. He is currently a Professor in information systems with the Federal University of Piauí (UFPI). His research interests include distributed systems with an emphasis on cloud-fog integration, mobile computing, and systems performance evaluation techniques.