Optimal Deployment of Cloud-native Applications with Fault-Tolerance and Time-Critical End-to-End Constraints

Remo Andreoli
Scuola Superiore Sant'Anna
Pisa, Italy
remo.andreoli@santannapisa.it

Harald Gustafsson
Ericsson Research
Lund, Sweden
harald.gustafsson@ericsson.com

Luca Abeni
Scuola Superiore Sant'Anna
Pisa, Italy
luca.abeni@santannapisa.it

Raquel Mini
Ericsson Research
Lund, Sweden
raquel.mini@ericsson.com

Tommaso Cucinotta
Scuola Superiore Sant'Anna
Pisa, Italy
tommaso.cucinotta@santannapisa.it

ABSTRACT

Cloud environments are becoming increasingly interesting to host *time-critical* use cases with far more stringent latency requirements than conventional cloud-native applications, such as smart industrial control systems or cloud-enabled autonomous vehicles. In these emerging domains, fault tolerance mechanisms play a critical role, due to the catastrophic consequences a fault might lead to, in the real world. This work presents a formal model for designing and deploying time-critical, cloud-native applications under fault conditions. Our model considers the interactions and interferences among service components and the possible occurrence of faults. We present an optimization framework to solve the deployment problem of minimizing the resources needed to achieve fault-tolerance under precise end-to-end deadline constraints. The ability of the optimizer to deliver precise temporal and fault-tolerance guarantees is validated through extensive simulations.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

Time-Critical Cloud, Fault Tolerance, Capacity Planning, Resource Management, Optimization

ACM Reference Format:
Remo Andreoli, Harald Gustafsson, Luca Abeni, Raquel Mini, and Tommaso Cucinotta. 2023. Optimal Deployment of Cloud-native Applications with Fault-Tolerance and Time-Critical End-to-End Constraints. In *Proceedings of Conference on Utility and Cloud Computing (UCC '23).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
UCC '23, December 04– 07, 2023, Taormina, Italy
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Cloud computing has emerged as a key technology for the growth and development of modern web applications, thanks to the tremendous benefits it offers to a wide spectrum of clients, ranging from large businesses to private customers and public administrations. These are relieved from the burden of managing physical resources of a traditional on-premise infrastructure [6], exploiting computational, networking, data management, security, auditing, and other services with a pay-for-use cost model. Moreover, with cloud-native [17] services and the recent Serverless Computing paradigm [32], we are seeing a shift of the complete DevOps toolchain to the cloud, easing the tasks of continuous deployment, integration, proper resource provisioning, and adaptation. Not only this paradigm shift reduced drastically the complexity of application life-cycle management, but it also led to the emergence of fully-managed services with performance guarantees. For instance, DynamoDB [14] or BigTable [15] promise "single-digit millisecond latency" when accessed from within the provider's infrastructure, attracting even more time-sensitive use-cases.

However, providing a reliable cloud service through the network is not easy: services must be replicated and spread across geographical locations with independent powering systems, but must also be "close enough" to quickly scale in case of temporary traffic outbursts or migrate virtual resources in case of faults. The use of traditional QoS-aware networking techniques [22, 34], as well as of novel technologies such as 5G and Time-Sensitive Networking [16], are crucial to facilitate deterministic and low-latency communications. The physical hosts themselves are another source of unpredictability, due to the contention of shared resources, which is typical of multi-tenant architectures: caches, memory controllers, and buses are contended in unpredictable ways. There have been multiple studies [10, 25, 33] proposing mechanisms to provide temporal isolation for time-critical applications on shared servers, thus reducing the unpredictability among co-located virtualized components.

These latest developments towards performance predictability, coupled with the recent advancements in 5G technologies, have exponentially increased the interest in hosting innovative time-critical applications on cloud-based infrastructures, with a strong focus on private, dedicated and geo-distributed cloud/edge infrastructures [26, 31]. Contrary to conventional web and time-sensitive applications, time-critical ones require a high level of availability and reliability, as well as precise latency guarantees, sometimes

in the order of milliseconds [4]. Such strict requirements must be met regardless of the other workloads deployed within the cloud system, or the possible occurrence of hardware/software/network failures [9]. Typical interactive multimedia cloud services, such as video-call platforms, are not seriously impacted, in the event of temporary slowdowns. On the other hand, an emerging use case like a machine vision system for intersection safety in smart cities requires cloud/edge-based services to be constantly up and running. The inability to respect temporal constraints may have critical, if not catastrophic, consequences. A cloud provider may decide to dedicate a single-tenant infrastructure for a time-critical system or offer dedicated hardware-software interfaces [12, 28]. However, such approaches most likely result in a huge waste of resources and money for both the provider and the customer, if not fully utilized, especially considering the recent efforts towards reducing energy consumption in cloud datacenters [24]. Hence, the need to take full advantage of multi-tenancy and shared resources to minimize costs and resource waste, while ensuring performance guarantees.

Contributions. This paper focuses on the management and optimization of time-critical, fault-tolerant applications in cloud infrastructures. A formal model is introduced to represent a cloud-native application as a composition of microservices in a Directed Acyclic Graphs (DAG) topology, with associated precise end-to-end temporal constraints. The individual microservices may be shared among a multitude of applications, causing resource contentions at runtime. Fault tolerance is implemented through replication and resubmission. At the heart of our proposal, there is an analysis tool able to evaluate if the to-be-deployed, time-critical applications can be admitted to the cloud infrastructure without violating the temporal constraints of the co-hosted workloads. The main goal is to configure the distributed applications and services to use the minimum amount of resources to achieve reliability (i.e., use replication as little as possible) while ensuring compliance with the temporal and fault-tolerance constraints. The deployment problem is solved in two ways: *standard* and *capacity planning* modes. The first one adopts an optimization approach based on Mixed-Integer Linear Programming (MILP), and it is used to decide at design-time the best fault tolerance technique to detect a fault in a timely manner and react to it within the temporal constraints, for a pre-defined configuration of the microservices. The second mode is based on (non-convex) Mixed-Integer Quadratic Constraint Programming (MIQCP), and it serves to suggest the best microservice configuration to comply with the requirements above.

This paper tackles the problem of fault-tolerance in real-time Cloud Computing introduced in [1], where no solution was proposed, and is an extension of [5], where a preliminary evaluation of the MILP optimizer was conducted. Here we present for the first time: the complete mathematical formulation, the complete optimization-based methodology needed to tackle it using either a MILP or a MIQCP approach (see Section 3), and an extensive evaluation of the approach based on the optimization and simulation of randomly generated scenarios (see Section 4).

2 PROBLEM FORMALIZATION

This section formally presents the application, the reference cloud infrastructure, and the fault model for time-criticality.

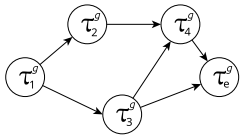

Figure 1: Parallel real-time application $\mathcal{A}^g$ modeled as a DAG.

2.1 Application Model

A cloud platform hosts a collection of $n_{\mathcal{A}}$ cloud-native applications $\mathcal{A} = \{\mathcal{A}^g\}_{g=1}^{n_{\mathcal{A}}}$, where $\mathcal{A}^g$ is built by composing a set of n^g independent and loosely coupled tasks $\Gamma^g = \{\tau_i^g\}_{i=1}^{n^g}$. A task $\tau_i^g \in \Gamma^g$ is a sequential activity that receives some input data, processes it, and then generates output data. An application is constructed as a Directed Acyclic Graph (DAG) whose topology is represented as a set of directed edges $\mathcal{E}^g \subseteq \Gamma^g \times \Gamma^g$. A directed edge $(\tau_\mu^g, \tau_i^g) \in \mathcal{E}^g$ is a logical communication link between two tasks, such that τ_μ^g sends its output data to τ_i^g. More specifically, τ_μ^g is a predecessor of task τ_i^g, and τ_i^g is a successor of task τ_μ^g. To ease readability, directed edges are also represented with the notation $\tau_\mu^g \rightarrow \tau_i^g \iff^{\text{def}} (\tau_\mu^g, \tau_i^g) \in \mathcal{E}^g$. Figure 1 depicts a possible application. Tasks are synchronously "activated", meaning that each task τ_i^g waits for all its predecessors $Prev_i^g = \{\tau_\mu^g \in \Gamma^g : (\tau_\mu^g, \tau_i^g) \in \mathcal{E}^g\}$ to generate their outputs, before processing them as input. Once the computations are finished, the resulting output is then propagated to all the successors $Next_i^g = \{\tau_\lambda^g \in \Gamma^g : (\tau_i^g, \tau_\lambda^g) \in \mathcal{E}^g\}$, concluding the task activation. A single activation of an application $\mathcal{A}^g$ may cause multiple tasks in Γ^g to be concurrently activated, according to the topology in $\mathcal{E}^g$. The latter defines *explicit dependencies* between contiguous tasks, as well as *implicit dependencies*: two non-contiguous tasks $\tau_i^g, \tau_j^g \in \Gamma^g$ have an implicit dependency if there is at least a directed path connecting the two. More generally, any two tasks $\tau_i^g, \tau_j^g \in \Gamma^g$ *have a dependency* (either explicit or implicit), represented as $\tau_i^g \sim \tau_j^g$ (or equivalently $\tau_j^g \sim \tau_i^g$), if there exists at least a directed path connecting them in either direction:

$$\tau_i^g \sim \tau_j^g \iff^{\text{def}} \exists\{\tau_{i_1}^g, \tau_{i_2}^g \ldots, \tau_{i_k}^g\} : \tau_{i_1}^g \rightarrow \tau_{i_2}^g \wedge \ldots \wedge \tau_{i_{k-1}}^g \rightarrow \tau_{i_k}^g \wedge \wedge((i_1 = i \wedge i_k = j) \vee (i_k = j \wedge i_1 = i)).$$

For each activation of an application $\mathcal{A}^g$, only tasks without a dependency (i.e., $\tau_i^g \not\sim \tau_j^g$ or $\tau_j^g \not\sim \tau_i^g$) may activate in parallel. Each application $\mathcal{A}^g$ has exactly one task with no predecessors (i.e., $Prev_1^g = \emptyset$), called the *input task*, which is assumed to be $\tau_1^g \in \Gamma^g$, without loss of generality. Analogously, there is exactly one task with no successors (i.e., $Next_{n^g}^g = \emptyset$), called the *output task*, which coincides with $\tau_{n^g}^g \in \Gamma^g$. In what follows, the special notation $\tau_e^g \equiv \tau_{n^g}^g$ will be used for ease of reading. An application activates when τ_1^g receives input data from an external source (i.e., a user request) and concludes after τ_e^g generates an output. Every application corresponds to a completely connected DAG so that for every node $\tau_i^g \in \Gamma^g$ there is always at least a directed path from the input to the output task which includes τ_i^g.

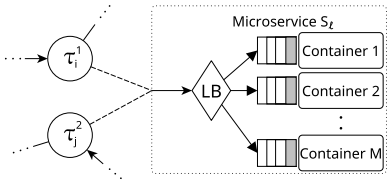

Figure 2: Two tasks from two different applications $\mathcal{A}^1$ and $\mathcal{A}^2$, mapped to the same microservice $S_l = \varphi^1(\tau_i^1) = \varphi^2(\tau_j^2)$.

Each time-critical application $\mathcal{A}^g$ is characterized by a minimum inter-arrival period P^g, which specifies the minimum time interval between two consecutive activations of the input task τ_1^g, and an end-to-end deadline D^g (*relative deadline*), assumed to be $D^g \leq P^g$. An application activation must be completed within D^g time units to be considered successful. More formally: the k^{th} activation of application $\mathcal{A}^g$, requested at time t_k^g, is considered successful if it concludes before its *absolute deadline* $d_k^g = t_k^g + D^g$.

2.2 Cloud Model

A task activation is realized by a service component hosted on the cloud infrastructure. To this end, the cloud platform offers a set of m_S microservices $\mathcal{S} = \{S_l\}_{l=1}^{m_S}$, each dedicated to the execution of a certain activity. A microservice S_l is implemented by a pool of M_l workers, normally cloud instances such as containers or virtual machines, and a load balancer to distribute the workload among them. The load balancer routes the input data coming from the preceding microservice(s) to a worker, which processes it (thus realizing the task invocation). Then, the produced output data is sent to subsequent microservice(s), as instructed by the application topology calling the execution. A set $\varphi = \{\varphi^g\}_{g=1}^{n_\mathcal{A}}$ of *task-to-microservice* mapping functions $\varphi^g : \Gamma^g \rightarrow \mathcal{S}$ is required to distinguish the tasks of different applications that share the same microservice. Such mapping functions are important to assess the worst-case response time of a microservice according to the number of tasks that can be concurrently active on the same microservice. For instance, Figure 2 depicts an example of two tasks belonging to different applications but being implemented by the same microservice. However, there may not be a one-to-one correspondence between tasks and microservices. Multiple tasks from the same application may invoke the same microservice, therefore φ^g may not be bijective. In general, two tasks $\tau_i^g \in \mathcal{A}^g, \tau_j^h \in \mathcal{A}^h$ mapped on the same microservice $S_l = \varphi^g(\tau_i^g) = \varphi^h(\tau_j^h)$, may be concurrently active only if they belong to different applications ($g \neq h$), or if they do not have a dependency relationship ($\tau_i^g \nsucc \tau_i^h \wedge g = h$). More formally: given $\Gamma = \bigcup_{\mathcal{A}^g \in \mathcal{A}} \Gamma^g$ the collection of *all* tasks of *all* applications $\mathcal{A}^g \in \mathcal{A}$, the set Φ_l of subsets of tasks that share S_l and that may be activated simultaneously, is defined as:

$$\Phi_l = \{B \subseteq \Gamma \mid \forall \tau_i^g \in B, \varphi^g(\tau_i^g) = S_l \wedge \wedge (\forall \tau_j^h \in B, h \neq g \vee \tau_i^g \nsucc \tau_j^h)\}. \quad (1)$$

Notice that set Φ_l contains every possible subset of tasks mapped to microservice S_l whose activations might interfere with each other, regardless of the application they belong to. The worst-case maximum number of tasks sharing microservice S_l that activate at the same time is the cardinality N_l of the largest subset in Φ_l:

$$N_l = \max_{B \in \Phi_l} |B|. \quad (2)$$

The execution of a given task τ_i^g by a worker in S_l takes at most a worst-case execution time (WCET) c_l. More specifically, c_l is assumed to account for the maximum amount of time required to process the input data and generate an output, regardless of the type of request, the application submitting it, and the data size. In other words, we associate a single WCET c_l to all the tasks executed by S_l. But since a microservice may simultaneously receive N_l requests from the same or different applications, the execution of a task invocation may experience a *queuing delay*, if the number of concurrent task activations exceeds the number of workers (i.e., $N_l > M_l$). This delay adds up to the execution time of the request, affecting the response time of the microservice. Therefore, a provider interested in providing temporal guarantees for co-located time-critical applications should plan the capacity of its infrastructure taking into account the overall worst-case response-time (WCRT) $C_l = c_l + queue\_delay$ for a microservice invocation.

In relation to what is described about the network in the introduction (Section 1), we make the assumption that packet transmissions among microservices and end-users are carried out within a maximum known time. This assumption is reasonably verified in scenarios where the provider has physical control over the networking infrastructure, such as private clouds, Cloud-Edge continuum environments [13], horizontal communications within a datacenter, or inter-datacenter communications over dedicated or leased lines. Therefore, we assume a maximum communication time of $\delta^{(NET)}$.

2.3 Fault Model

In a conventional cloud infrastructure, a worker may fail due to a software issue, a hardware fault, or a network disconnection. However, deadline misses and timeouts due to resource contention, or a lack of resource availability, are also considered faults in the context of time-sensitive applications.

We model two possible ways to handle a fault: i) *static replication*: submit the task invocation request in parallel to 2 workers; ii) *task invocation re-submission*: the failed invocation is re-submitted, allowing the load balancer to choose another, healthy worker. In both cases, the 2 workers are assumed to reside in different physical machines or availability zones, so that it is safe to assume that the simultaneous failure of 2 workers is an event with negligible probability (more on this later). Clearly, we assume that the software is capable of handling multiple invocations of the same task in parallel, or sequentially, keeping functional correctness. How exactly this is done, is out of scope for the present paper.

Figure 3 shows how a fault is handled by each method for a given task τ_i^g. Static replication is a well-known fault tolerance methodology, but it may lead to a waste of resources. Task re-submission does not require redundant cloud resources, but it may extend the aggregated duration of all task activations beyond the end-to-end deadline. Each method incurs a different WCRT, namely

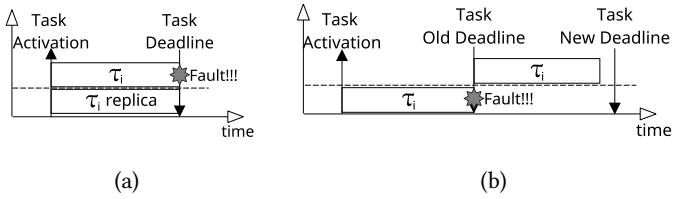

Figure 3: Fault handling in the case of task τ_i: (a) activated with static replication; (b) re-submitted after failure. The dotted lines highlight the fact that each task activation is realized by a different container of the same microservice (the g superscripts have been removed to simplify the figure).

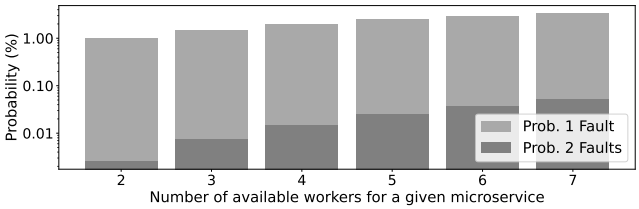

Figure 4: Probability of fault(s) in a microservice, using AWS EC2 uptime percentage values. Note the logarithmic Y scale.

C^{repl} for static replication, and $C_l + C'_l$ for re-submission. The latter is due to the fact that the *total* WCRT for a task activation must take into account both the *first* execution (i.e., C_l) and the re-submission (i.e., C'_l), as highlighted in Figure 3b.

In our model, an invocation of a task τ_i^g completes successfully if it finishes within a *relative partial deadline* D_i^g since the application activation at time t_k^g, despite possible faults occurring during its execution (i.e., this time should account for possible re-execution if needed in case of fault). An activation of an application is successful only if: all task activations throughout the whole topology are successful; the partial deadlines are assigned so that their sum does not exceed the application deadline D^g on every end-to-end path. A naïve fault model might need to assume that *all* task activations of the application may experience a fault, but this is highly unlikely to happen in practice, as some of these failures may actually be deadline misses due to transient problems like occasional spikes in execution times or transmission times due to unforeseeable interferences. Therefore, in our approach, we aim at guaranteeing successful completion of an application activation, *assuming that at most F tasks may experience a fault throughout the whole execution path* for each activation of each application, with F being a tunable parameter. Formally, it is convenient to introduce the *absolute partial deadline* within which a task activation needs to complete to be successful: $d_{k,i}^g = t_k^g + D_i^g$. The partial deadline D_i^g directly depends on the partial deadlines on the predecessors of τ_i^g in the topology $\mathcal{E}^g$, but also on the fault tolerance method chosen for each task, and the associated WCRT. More precisely, the k^{th} activation of task τ_i^g is considered successful if it is concluded by time

$$d_{k,i}^g = t_k^g + D_i^g = \begin{cases} t_k^g + \max\{D_\mu^g \mid \tau_\mu^g \in Prev_i^g\} + \mathcal{X}_i^g \ (i \neq 1) \\ t_k^g + \mathcal{X}_i^g \ (i = 1) \end{cases} \quad (3)$$

where $\mathcal{X}_i^g \in \{C_l^{repl}, C_l + C'_l, C_l\}$ is the WCRT of the underlying microservice, such that $\varphi^g(\tau_i^g) = S_l$. If the task activation is configured with static replication, then $\mathcal{X}_i^g = C_l^{repl}$, and therefore it can tolerate a worker fault without incurring extra delays; if a possible fault is handled by re-submission, then $\mathcal{X}_i^g = C_l + C'_l$. Finally, we may have the case in which F faults have already been considered in the tasks preceding τ_i^g, resulting in $\mathcal{X}_i^g = C_l$.

When focusing on a specific microservice, the fault handling mechanism by replication or re-submission discussed above assumes that a second worker is very unlikely to experience a fault at the same time of another worker of the same microservice, or in the period of time prior to the recovery of the first faulty worker. This is often the case with workers of the same microservice deployed on different availability zones. Indeed, let p be the probability of a worker being faulty; such value is the overall availability percentage of a worker. For instance, each individual AWS EC2 instance has a guaranteed instance-level uptime percentage of at least 99.5% ($p = 0.005$)[1]. Under the assumption of fault independence, the probability of event $2\mathcal{F}$="*exactly* 2 faulty workers" is $\mathrm{P}(2\mathcal{F}) = \mathrm{P}(\mathcal{F}) \cdot \mathrm{P}(\mathcal{F}) = p^2$. Consequently, the probability of event $1\mathcal{M}$="*exactly* 1 out of M workers are faulty," must consider all possible combinations of the i^{th} worker being faulty while the other $M-1$ are healthy:

$$\mathrm{P}(1\mathcal{M}) = \binom{M}{1} \cdot \mathrm{P}(\mathcal{F}) \cdot \mathrm{P}(\neg\mathcal{F})^{M-1} = M \cdot p \cdot (1-p)^{M-1}. \quad (4)$$

The same reasoning applies to the probability of event $2\mathcal{M}$="*exactly* 2 out of M workers are faulty". All possible combinations of the i^{th} and j^{th} workers being faulty, while the other $M-2$ are healthy, must be considered:

$$\mathrm{P}(2\mathcal{M}) = \binom{M}{2} \cdot \mathrm{P}(\mathcal{F})^2 \cdot \mathrm{P}(\neg\mathcal{F})^{M-2} = \frac{M(M-1)}{2} \cdot p^2 \cdot (1-p)^{M-2}. \quad (5)$$

The industrial standards for availability make the probability of event $1\mathcal{M}$ non-negligible, and that of event $2\mathcal{M}$ small enough to be neglected. Using $p = 0.005$ (the availability probability of a AWS EC2 instance) and $M = 3$ fault-independent workers, the probability of event $1\mathcal{M}$ is ~ 0.01485, the probability of $2\mathcal{M}$ is ~ 0.0000746. Figure 4 reports some more mathematical computations. Note that the probability of $2\mathcal{M}$ increases significantly for $M > 3$, but from the point of view of availability, more containers are a great advantage even if the probability of failure is higher.

3 APPROACH

In this work, we make sure that a set of cloud-native applications $\mathcal{A}$ respect their end-to-end relative deadline D^g, regardless of interference or failures suffered during an activation. For each application $\mathcal{A}^g \in \mathcal{A}$: i) we introduce a relative partial deadline D_i^g for each task $\tau_i^g \in \Gamma^g$; ii) we ensure that the individual relative partial deadlines D_i^g cannot be missed for each task τ_i, regardless of possible

[1]Accessed on February 2023: https://aws.amazon.com/compute/sla/

fault conditions we may incur at run-time; iii) we ensure that the sum of partial deadlines throughout all the end-to-end paths in the application topology Γ^g does not exceed D^g; iv) we optionally suggest the optimal capacity (number of workers) to be used for each microservice, to avoid deadline misses.

3.1 WCRT Estimation

Firstly, it is crucial to derive possible formulas for the WCRTs presented in Section 2. In the worst-case, a task execution on microservice S_l may have to wait for $N_l - 1$ invocation requests already queued up, over the number of available workers M_l. Considering also the additional load balancer delay $\delta^{(LB)}$ to select a worker, the network delay $\delta^{(NET)}$ to receive input and send output data, respectively, and assuming a Round-Robin load balancing strategy, we obtain the overall WCRT C_l for the microservice invocation:

$$C_l = c_l + \left\lfloor \frac{N_l - 1}{M_l} \right\rfloor c_l + \delta^{(LB)} + 2\delta^{(NET)} \tag{6}$$

For static replication, the WCRT of a task execution on microservice S_l depends on all the replicas of the interfering task invocations, as well as on its own replica. The total delay amounts to $2(N_l-1)+1 = 2N_l-1$ worst-case invocation requests queued up, over the number of available workers M_l:

$$C_l^{repl} = c_l + \left\lfloor \frac{2N_l - 1}{M_l} \right\rfloor c_l + \delta^{(LB)} + 2\delta^{(NET)} \tag{7}$$

Finally, for re-submission, the WCRT of a re-executed task activation takes into account the fact that the number of available, healthy workers is decreased by one:

$$C_l' = c_l + \left\lfloor \frac{N_l - 1}{M_l - 1} \right\rfloor c_l + \delta^{(LB)} + 2\delta^{(NET)} \tag{8}$$

Notice that Equation (7) and (8) work under the "no double fault" assumption introduced in Section 2.3.

3.2 Optimal Partial Deadline Assignment

The cloud platform model presented in the previous section can react to a fault in two ways: static replication and task re-submission. A naïve approach to achieve fault tolerance is to use static replication for every task activation, effectively nullifying the effect of a fault (under the assumptions of Section 2.3), but this would require excessive resources. On the other hand, using re-submission for every task activation may be unfeasible in case of a tight end-to-end deadline requirement that does not leave enough spare time.

In this subsection, we introduce an offline analysis and optimization tool to configure the optimum fault tolerance method for every task of a given application so that each application meets its given end-to-end deadline constraint, despite the occurrence of a number of faults F, as well as the interferences from other applications possibly sharing the same microservices. The idea is to use static replication only for a few tasks whose failure would otherwise necessarily violate the end-to-end deadline D^g. By minimizing the number of statically replicated tasks, the usage of redundant resources is in turn minimized. In practice, our optimizer computes the optimal absolute deadline $d_{k,i}^g$ within which every τ_i^g invocation must finish to guarantee a successful activation of the task (recall Section 2.3). For the sake of readability, the k subscript used to identify the absolute partial deadlines $d_{k,i}^g$ of the k^{th} activation of application $\mathcal{A}^g$ will be dropped from now. In fact, since we analyze the task activation patterns under worst-case conditions, the optimizer does not need to take into account the series of individual application activations over time (however, should the application topology change, the optimizer would have to be run again). The role of the optimizer is to choose the best value for each X_i^g in Equation (3), regardless of *when/where* the F faults occurred, so that the application activation is successful (i.e., end-to-end deadline D^g is not violated). In short, this means that the optimizer has to figure out the best possible fault tolerance method for each task activation, with the main goal of minimizing static replication. For each task τ_i^g, we use a binary variable z_i^g to indicate whether it is statically replicated or not, and a set of possible partial deadline variables $d_i^{g,(f)}$, one for each possible number of faults $f \leq F$ throughout the partial execution of the application (i.e., from τ_1^g to τ_i^g).

$$z_i^g = \begin{cases} 1 \text{ if } \tau_i^g \text{ is statically replicated} \\ 0 \text{ if } \tau_i^g \text{ is } \textit{not} \text{ statically replicated} \end{cases} \tag{9}$$

$$d_i^{g,(f)} = \text{partial deadline in case of at most } f \text{ faults} \tag{10}$$

The analysis that follows focuses on a single application $\mathcal{A}^g$, so the g superscripts are dropped whenever they're implicitly referred to, without risks of ambiguity. We assume that the application activation starts at time 0 (i.e., $t_k^g = t = 0$), simplifying Equation (10) for the partial deadlines $d_i^{(f)}$. Based on what has been said so far, the following conditional recurrence relations describe how the $d_i^{(f)}$ are computed based on task dependencies and previous task activation WCRTs, as previously shown in Equation (10).

i) *Base Case*: input task τ_1 is implemented by a dedicated microservice S_y ($\varphi(\tau_1) = S_y$):

$$z_1 = 1 \implies \begin{matrix} d_1^{(0)} = C_y^{repl} \\ d_1^{(1)} = C_y^{repl}; \end{matrix} \quad z_1 = 0 \implies \begin{matrix} d_1^{(0)} = C_y \\ d_1^{(1)} = C_y + C_y' \end{matrix} \tag{11}$$

Equation (11) (left) defines the possible partial deadlines for input task τ_1 if it is statically replicated. Equation (11) (right) defines the partial deadline if τ_1 is not statically replicated: in this case, if a fault happens ($f = 1$), task τ_1 is certainly affected by it and therefore the tentative partial deadline $d_1^{(1)}$ must consider the re-submission execution time.

ii) *Recursive Case*: intermediate / output task τ_i, each implemented by a dedicated microservice S_l ($\varphi(\tau_i) = S_l$):

$$z_i = 1 \implies \begin{matrix} \forall \tau_\mu \in Prev_i : \\ d_i^{(0)} \geq d_\mu^{(0)} + C_l^{repl} \\ d_i^{(f)} \geq d_\mu^{(f)} + C_l^{repl} \quad \forall f \in K_i \\ d_i^{(f)} \geq d_\mu^{(f-1)} + C_l^{repl} \quad \forall f \in K_i \setminus \{1\} \end{matrix} \tag{12}$$

$$z_i = 0 \implies \begin{array}{l} \forall \tau_\mu \in Prev_i : \\ \quad d_i^{(0)} \geq d_\mu^{(0)} + C_l \\ \quad d_i^{(f)} \geq d_\mu^{(f)} + C_l \qquad \forall f \in K_i \\ \quad d_i^{(f)} \geq d_\mu^{(f-1)} + C_l + C'_l \;\; \forall f \in K_i \setminus \{1\} \end{array} \tag{13}$$

where $Prev_i = \{\tau_\mu \in \Gamma : (\tau_\mu, \tau_i) \in \mathcal{E}\}$, as defined in Section 2.1, and $K_i = \{1, ..., \min\{F, max\_dist(\tau_1, \tau_i)\}\}$ contains the range of possible faults between the activation of input task τ_1 and τ_i. Procedure $max\_dist()$ returns the max number of tasks among the possible paths between τ_1 and τ_i, and it is implemented as a modified version of topological sorting, therefore it is solvable in linear time. Equations (12) and (13) take into account the partial deadlines previously assigned to the predecessors. If there are no faults so far (i.e., $f = 0$), the first fault of the application activation has yet to happen (i.e., to the invocations of subsequent tasks). Conversely, if a number of faults $f > 0$ have happened, they could *all* have happened during the invocations of the preceding tasks. In this case, there is no need to consider re-submission in the partial deadline. Otherwise, $f - 1$ faults happened to the task activation sequences leading to task τ_i, and since $f \leq F$, it may just so happen that the next fault will affect task τ_i invocation. If static replication is not in use, the partial deadline must take into account the additional re-submission WCRT, as shown in the third inequality (13).

The above relations are transformed into constraints for a Mixed-Integer Linear Programming (MILP) problem using Pyomo[2]. The objective is to minimize the number of statically replicated task invocations (due to the resource waste). The optimizer ensures that any possible partial deadline assignment does not violate the end-to-end deadline D, regardless of the number of transient faults F, or of the interferences. In other words, the optimizer evaluates every possible path between τ_1 and τ_e, using z_i as decision variables to pick a proper WCRT for each task activation. Ultimately, this is done for all the applications to be deployed:

$$\text{Minimize} \sum_{\mathcal{A}^g \in \mathcal{A}} \sum_{i=1}^{n^g} z_i^g, \;\; \text{subject to:} \tag{14}$$

$$\text{constraints (11), (12), (13)} \; \forall \mathcal{A}^g \in \mathcal{A}$$

$$d_e^{g,(F)} \leq D^g \;\; \forall \mathcal{A}^g \in \mathcal{A}. \tag{15}$$

In our proposed approach, this optimization problem constitutes a capacity test to be run every time there is an attempt to change the hosted workload characteristics, to check whether the request can be admitted. This is needed on: (i) new, additional applications to be deployed; or (ii) a change in the maximum number of concurrent tasks (i.e., N_l) or number of workers for a given microservice (i.e., M_l). For example, this may be due to a horizontal scaling action of an already deployed application or a microservice, respectively.

Under the assumption that a given end-to-end deadline D^g is not ill-posed (i.e., application $\mathcal{A}^g$ should be able to fulfill D^g under a no-interference, no-failure condition), the optimizer cannot find a feasible solution only if a subset of microservices is unable to accommodate all the to-be-deployed applications. To this end, our optimizer allows turning some, or all, $\{M_l\}$ into decision variables to be optimized. In practice, the WCRT Equations (6), (7) and (8) are converted into quadratic constraints. This allows for using the optimization objective of minimizing for example the amount of total workers, i.e., $\sum_l M_l$. This capacity planning feature transforms the partial deadline assignment problem into a MIQCP with a non-convex feasible region, which considerably increases the computational complexity. This is viable when the availability of underlying physical resources is sufficiently high to deploy all the suggested M_l workers for each microservice $\mathcal{S}_l$.

Regarding the complexity of the proposed optimization approach, the solving time grows with the problem size, which is related to the number of variables and constraints instantiated in Pyomo. For estimating the WCRTs, the number of variables is $\#wcrtVar = 8\, m_S$ and the number of constraints is $\#wcrtConstr = 11\, m_S$. For assigning the partial deadlines we need $\#pdVar = \sum_{g=1}^{n_\mathcal{A}} 2\, n^g + F\, n^g$ and $\#pdConstr = \sum_{g=1}^{n_\mathcal{A}} 3 + 2\,(F+1)\, n^g$. An empirical evaluation of the solving times is discussed in Section 4.3.

4 EXPERIMENTAL RESULTS

In this section, we evaluate the optimization strategy presented in Section 3.2, under the assumption of an underlying infrastructure capable of accommodating all the microservices. We show how the optimizer computes partial deadlines and decides a fault tolerance method for each task. For our experiments, we used the Gurobi solver v9.5.1 on a 112-core system (x86-64 server with 2 Xeon Gold CPUs and 125 GB of RAM). The quality of the obtained offline decisions is evaluated with simulations: first, we focus on the simple scenario depicted in Figure 1 and Table 1, then we perform a campaign of experiments with randomly generated scenarios.

4.1 Analysis of Scenario in Table 1

For the sake of simplicity, every task activation is realized by a dedicated microservice, thus tasks and microservices can be referred to using the same indexes: task τ_i^g activations are executed by microservice S_l, where $i = l$. This means that every $\varphi^g \in \varphi$ is a bijective mapping. Temporal interferences are modeled by duplicating the very same application topology 4 times, i.e., $n_\mathcal{A} = 4$ and $\forall S_i \in \mathcal{S}, N_i = 4$ (simply referred to as N in what follows). This simplifies the description of the example significantly. Since the application topologies and requirements are all the same, we can focus the analysis on one of the 4 application deployments, for which the other 3 ones are considered as interfering workloads. Henceforth the g superscripts and the k subscripts are omitted. Table 1 describes the given characteristics of each microservice, together with the WCRTs computed using Equations (6) to (8). The total number of workers is $M_1 + M_2 + M_3 + M_4 + M_e = 14$. For simplicity, we assume the load balancer delay $\delta^{(LB)}$ and communication time $\delta^{(NET)}$ to be negligible w.r.t. the processing times, in the WCRTs.

We run the optimizer with a variety of end-to-end deadlines, to compute the number of critical task executions for an application invocation, and their corresponding partial deadlines. The ability for a DAG application to meet a given end-to-end deadline generally depends on the *critical path*. However, in our context, the critical path length, which corresponds to the total application end-to-end worst-case response time (or *makespan*), is affected by the choices to be made by the optimizer for the variables $\mathcal{X}_i \in \{C_i^{repl}, C_i, C_i + C'_i\}$ for each task τ_i. Consider a failure-free ($F = 0$) and interference-free

[2]https://www.pyomo.org/

	Given Parameters		Inferred Parameters			
	c_i	M_i	C_i^{repl}	C_i	C'_i	$C_i + C'_i$
S_1	10	3	30	20	20	40
S_2	15	2	60	30	60	90
S_3	33	4	66	33	66	99
S_4	10	2	40	20	40	60
S_e	20	2	80	40	80	120

Table 1: Characteristics of the microservices implementing the application depicted in Figure 1. Assuming a one-to-one task-to-microservice mapping, tasks and microservices can share the same indexes. The inferred parameters are computed using $N = 4$ tasks sharing each microservice.

($N = 1$) scenario with the characteristics in Table 1: the critical path for the application under analysis is $\{\tau_1, \tau_3, \tau_4, \tau_e\}$ and its makespan is $c_1 + c_3 + c_4 + c_e = 73$ time units. Removing the interference-free assumption, the makespan becomes $C_1 + C_3 + C_4 + C_e = 113$ time units (depicted in Figure 5). Ideally, $N = 4$ workers per microservice (20 total workers) are needed to ensure fully parallel executions with no queuing delay, but this may be too expensive.

If the application activates at time $t = 0$ and $D = 113$, the earliest absolute partial deadlines assigned for a successful completion are: $d_1 = 20, d_2 = 50, d_3 = 53, d_4 = 73, d_e = 113$, using Equation (3) with $\mathcal{X}_i = C_i \ \forall S_i \in \mathcal{S}$. The *partial makespan* for task τ_i, defined as the cumulative worst-case response time up to τ_i, must be lower than d_i to ensure a reliable application activation at run-time. With the given $\{M_i\}$ values, $D = 113$ is clearly the minimum possible end-to-end deadline that can be met with the microservices in Table 1 in any possible interference scenario, under a failures-free assumption. However, it is not possible to meet such a stringent deadline if $F > 0$. This is because there is no spare time for re-submissions, due to the partial deadline d_e of output task τ_e being equal to the end-to-end deadline D. On the other hand, the deadline would be met with *all* tasks statically replicated. Indeed, using $2 * N = 8$ workers per microservice (40 total workers), would ensure the concurrent executions of all replicas, leading to an end-to-end critical path meeting the tight deadline of $D = 113$. However, this may be too expensive. The solution is to settle for longer deadlines, since a lax deadline requires fewer workers, or to let the optimizer plan the optimal number of workers for each microservice. For instance, the optimizer suggests a 84% increase in virtual resources (from 13 to 24 total workers) if the activation has to tolerate at most 3 transient faults ($F = 3$) with deadline $D = 113$, allocated as: $M_1 = 4, M_2 = 4, M_3 = 8, M_4 = 4, M_e = 4$. Naturally, this invalidates the inferred characteristics in Table 1 as well as the critical path in Figure 5.

Figure 6 presents the results from a series of optimization runs with different end-to-end deadlines (x-axis) higher than 113. The number of possible faults during an application activation is $F = 3$. The blue line depicts the minimum amount of statically replicated task executions required to safely deploy the application in Figure 1, alongside 3 other interfering instances of the same application, using the predefined characteristics in Table 1. The number of statically replicated tasks increases as the end-to-end deadline shrinks,

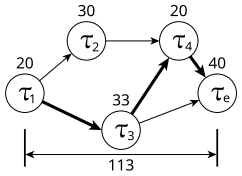

Figure 5: Critical path and critical path length in a failure-free scenario (path in bold), using the microservice parameters in Table 1. Since there may be interferences, the length depends on the WCRTs C_i.

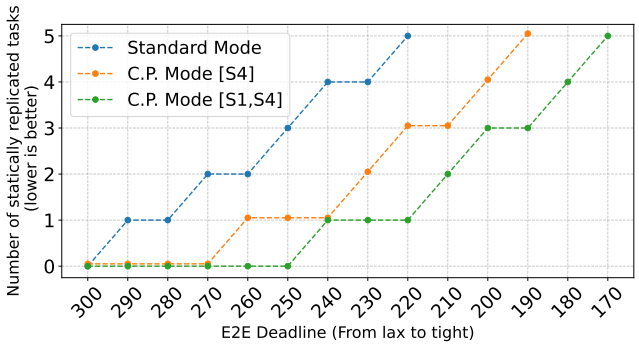

Figure 6: Number of statically replicated tasks as the end-to-end deadline shrinks. The number of tolerable transient faults is $F = 3$, and the number of deployed applications is $N = 4$. Standard mode uses the characteristics in Table 1. Capacity Planning mode lets the optimizers decide the optimal number of workers M_i for microservice S_4 (orange line), and microservices S_1 and S_4 (green line).

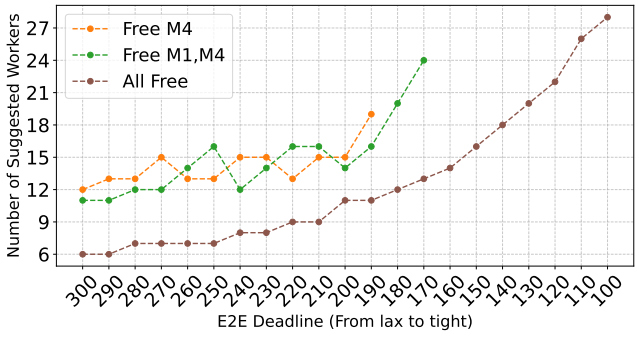

Figure 7: Total number of suggested workers in capacity planning mode, as the end-to-end deadline shrinks.

and the assignment problem becomes unfeasible for deadlines earlier than 220. The orange line depicts how the capacity planning capabilities of the optimizer allow to push back the deadline boundary by using M_4 as a "free" parameter (i.e., the solver picks the optimal amount of workers for microservice S_4). As the end-to-end deadline shrinks, the number of statically replicated tasks rises

again, leading to a bottleneck on a different microservice. Finally, the green line shows the same trend, but with two free parameters, M_1 and M_4. The optimizer avoids static replications more often if it is allowed to set the capacity of some microservices.

Figure 7 shows the number of total workers, regardless of which microservice they belong to, as suggested by the optimizer in capacity mode. For instance, the green line demonstrates that it is possible to deploy $n_{\mathcal{A}} = 4$ applications and successfully complete an application activation within $D = 300$ units of time with half the number of workers in Table 1, even in the case of $F = 3$ tolerable faults. The workers are allocated as: $M_1 = 1, M_2 = 1, M_3 = 2, M_4 = 1, M_e = 1$. Similarly, the plot shows that it is not possible to successfully complete an activation within $D = 170$ with less than 24 workers with only two free parameters (orange line). If all $\{M_i\}$ are free parameters, the optimizer suggests an overall better deployment configuration than Table 1 (brown line). With the given simple scenario, the solving time for the partial deadline assignment ranges from 200 to 500 milliseconds.

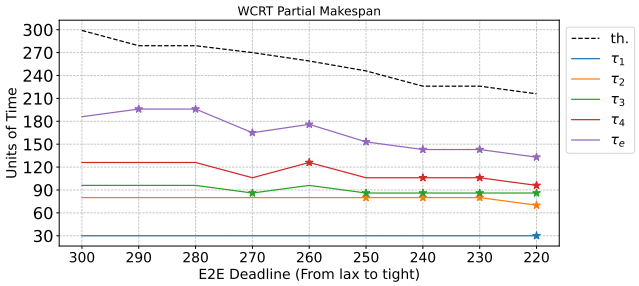

Figure 8: 100^{th} percentile partial makespan (from simulations) for each task, compared to the theoretical partial deadline for the output task τ_e (dashed line). The star marker indicates a statically replicated task execution, as decided by the optimizer. The parameters used are $F = 3$, $N = 4$, 100% failure rate and the microservice configuration in Table 1.

4.2 Evaluation by Simulation

The previous Subsection 4.1 assessed through offline analysis validation the concepts presented in the model, such as the WCRTs and the idea of a maximum amount of tolerable faults per activation. This section is dedicated to providing a practical evaluation of the analysis results. The goal is to show that the partial deadlines computed by the optimizer are not violated through simulations. To this end, we developed a realistic simulator in Python using the discrete event simulator SimPy, which simulates the interferences and transient faults experienced by a given application topology. The essential parameters for the simulator are the application topology, the number of deployments N (with the same topology, like in the previous subsection) that will be interfering with each other, a list of microservice characteristics, and the number of F faults that may occur. The simulator sets up the defined microservices with the required number of workers and a load balancer. A simulation performs a number of application activations (for each application instance) by periodically generating execution requests to the microservice dedicated to the realization of the input task. Each worker of microservice S_i is simulated in a separate simulated thread characterized by a certain processing duration c_i. A worker either performs a task execution, picks a new task invocation request from a dedicated wait queue (that is fed by a Round-Robin load balancer), or waits for one to arrive. Consequently, a microservice S_i can complete M_i task executions in c_i units of time (assuming no failures). The remaining $N - M_i$ pending task activations, if any, will be delayed until a worker is available. A task completion generates an activation request for the successors, i.e., is queued on a corresponding worker, even if it finished earlier than the partial deadline. This means that parallel task activations at worst-case may not be synchronized during the simulation, but "spread out" due to not being delayed to the same partial deadline (because that depends on the "actual" queuing delay at simulation time).

The simulator offers a set of optional parameters to further personalize the simulation behaviors, such as several execution time distributions, application activation patterns and fault rate. For this evaluation, we have selected a configuration that is at the border of the model guarantees (i.e., the worst-case). The applications are activated with equal periodicity and issued simultaneously. Next, all task executions last as long as their WCET c_i. The fault rate is set to 100%, i.e., F faults will occur for every activation of an application. Recall that due to the "no double fault" assumption in Section 2.3, the replica of a statically replicated task execution always succeeds, as well as the re-submission of a previously failed task activation. The failures are modeled as a WCET or partial deadline "overrun", which is the latest time a fault detector could have noticed the failure. The simulation ends after a predefined simulation time.

Figure 8 shows some simulation results for the simple scenario in Figure 1 and the microservice characteristics listed in Table 1. The simulator performs roughly 13000 application activations, and none of them exceeds the end-to-end deadline, as expected from the offline analysis. The optimizer overestimates the partial deadlines because it solves the assignment problem in the worst-case scenario, where every task invocation for a given application activation is delayed by all other $N - 1$ *same* task invocations. In reality, task activations would be more spread out. Nonetheless, the infrastructure must be ready to handle the worst-case scenario to guarantee the highest possible availability and reliability, thus leaving room for re-submission of the F longest tasks execution on the critical path, at the highest peak of interference. Indeed, the optimizer provides configurations that are robust regardless of where the faults occur, as long as they affect no more than F tasks (see Section 3).

4.3 Randomly Generated Scenarios

In order to validate more generally our optimization framework, and highlight better how it behaves also in terms of solving time overheads, we carried out an extensive campaign of experiments with 100 randomly generated scenarios, with an overall 38588 execution runs. These include random DAG topologies and chains of tasks composed of $n^g \in [4, 7]$ tasks, replication factor of $n_{\mathcal{A}} \in [1, 3]$, number of workers $M_i \in [2, 8]$ and number of faults $F \in [0, 3]$. The WCETs are randomly generated in the range of $50 - 100$ time units, and the end-to-end deadlines are randomly generated to compute feasible solutions. The topologies aim to mimic the typical interaction in a NFV context, i.e., handover of a mobile device between base

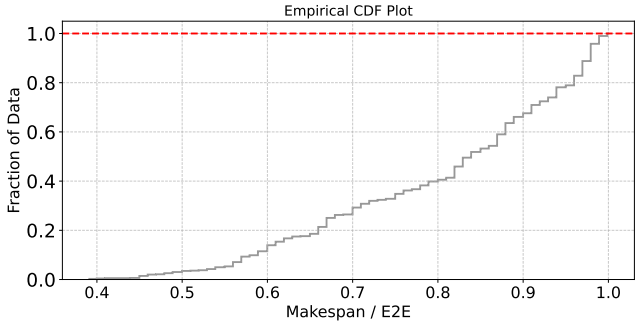

Figure 9: Empirical CDF of the simulated makespan divided by the end-to-end deadline for the considered scenarios.

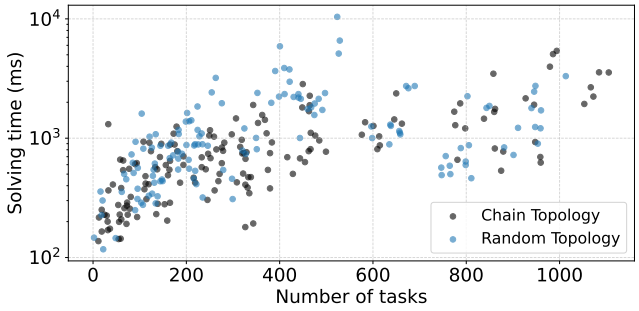

Figure 10: Total solving time over problem size (note the logarithmic scale on the Y-axis).

stations or baseband packet processing in 5G [2, 27]. We assume that every task activation is realized by a dedicated microservice.

Figure 9 shows the empirical CDF of the obtained ratio between the makespan of an application activation (from simulations) and the end-to-end deadline, for each randomly generated scenario. We can see that all application activations terminate within their end-to-end deadline (corresponding to 1.0 on the X axis), with the majority of tasks completing safely below it. Figure 10 depicts the increase in solving time as the problem size grows in terms of number of tasks n^g and interfering applications $n_{\mathcal{A}}$. In this case, we increased the complexity of the generated problems: $n^g \in [5, 20]$ and $n_{\mathcal{A}} \in [1, 60]$. Each data point corresponds to a randomly generated application. The optimizer runs in capacity planning mode (with higher complexity), meaning that it also computes the optimal number of workers per microservice. As expected, the solving time ramps up quickly as the problem size grows. We faced solving times within $1 - 2$ seconds for relatively small problems with up to 200 tasks, and higher solving times for larger problems, up to ~10 seconds in the performed runs.

5 RELATED WORK

Resource management in cloud-based infrastructures is a broad field of study, and there are many works focusing on fault-tolerance techniques [19, 20]. There are three types of fault tolerance policies: 1) reactive, which reduces the effect of a failure after it occurs; 2) proactive, which avoids fault recovery by predicting the fault; 3) adaptive, a hybrid approach that depends on the current state of the system or application. The most common reactive approaches are check-pointing, which reproduces the task executions in another instance after a fault occurred, and replication. Examples of proactive approaches are preemptive migration and rejuvenation, which reduce the likelihood of a fault occurring by removing the task executions from the potentially faulty system, in advance. However, there is a lack of research in fault detection and recovery for time-critical cloud-native applications.

Guo et al. [18] focus on the problem of reducing energy consumption when introducing redundancy to ensure fault tolerance for real-time tasks in cloud-based 5G networks. This is done by scheduling primary and backup copies of real-time tasks to different virtual machines, which are then dynamically rearranged to fully utilize the idle time slots. Siyadatzadeh et al. [29] propose a learning-based primary-backup placement technique to improve the reliability of fog-based Internet-of-Things (IoT) real-time systems. The proposed model tolerates failure on the communication links and processing units by establishing a balance between communication delay and workload on each fog device. In this way, the backup tasks are only sent to fog nodes where they will have enough time to complete within the deadline. Yao et al. [35] presents a scheduling approach for deadline-constraint independent tasks that selects the proper fault-tolerant strategy, either resubmission or replication, based on the characteristics of both task and cloud resources. The proposal then reserves a suitable amount of resources for task execution and dynamically adjusts them to improve resource utilization. Ahmad et al. [3] describes a workflow management system with Quality-of-Service (QoS) and fault-tolerant capabilities for real-time scientific applications. A scheduler converts an admitted workflow into jobs and assigns them the required resources, based on QoS requirements. A workflow engine executes the jobs and migrates those affected by a hardware or software failure to the nearest available node with the same configuration. Zenpeng et al. [21] presents a greedy fault-tolerant scheduling algorithm for microservice-based, deadline-constrained applications. Their heuristic strategy assigns task replicas to suitable virtual resources so that the task deadline and reliability requirements are met. Moreover, they propose a resource adjustment strategy to improve resource utilization, and therefore to reduce the cost of the initial task scheduling solution. Malink and Huet [23] present a fault tolerance scheme for virtual machine (VM) placement in a real-time cloud scenario. Each real-time task is replicated and placed on the VMs with the highest reliability score. The latter is dynamically computed according to the number of outputs produced within the task deadline: if the VM produces a correct result within the deadline, its reliability score is increased. If no VM achieves the pre-defined minimum level of reliability, the systems will perform backward recovery or other safety measures, such as stopping it.

Most of the existing approaches to fault tolerance are reactive, and, specifically, they are based on replication, due to the relatively low recovery time. In these works, fault tolerance is mainly investigated and implemented without worst-case latency guarantees. Furthermore, to be best of our knowledge, no other work considers interferences between microservice-based applications deployed on the same cloud infrastructure, as proposed in this paper.

6 CONCLUSIONS & FUTURE WORK

In this paper, we formalized the problem of design and deployment of cloud-native, time-critical applications. We introduced a novel model for describing interferences between time-critical applications, represented as compositions of software tasks provided by the cloud platform in the form of microservices. Furthermore, we presented an optimization approach to tackle the problem of optimum deployment of time-critical applications with formal fault-tolerance guarantees, based on the number of expected transient faults and on the interferences among application instances sharing the same microservices. The optimizer includes two modes of operation: the first one evaluates the feasibility of a certain application deployment, based on the number of workers per microservice; the second mode allows to plan the capacity of each microservice, in relation to the available computing resources in the underlying infrastructure, to guarantee successful activations. The quality of the decisions taken by the solver is shown through a series of simulations.

In the future, we plan to conduct a number of further investigations: more realistic scenarios that take into account network latency requirements; additional experimental evaluations using well-known simulators in the literature [8]; a more advanced CPU utilization model based on reservations [30]; integration with the problem of placement over possibly large physical infrastructures. The latter will cause the optimization problem to quickly become intractable, so a heuristic-based approach will be investigated to reduce the optimization complexity, while still finding good enough solutions. A starting point is the heuristic algorithms proposed in [7, 11] to find suboptimal partitions of real-time, graph-based applications. Last but not least, we are planning to evaluate the obtained theoretical results on a real-world Kubernetes deployment.

REFERENCES

[1] Luca Abeni, Remo Andreoli, Harald Gustafsson, Raquel Mini, and Tommaso Cucinotta. 2023. Fault Tolerance in Real-Time Cloud Computing. In *2023 IEEE 26th International Symposium on Real-Time Distributed Computing*. IEEE, 170–175.

[2] Luca Abeni, Tommaso Cucinotta, Balázs Pinczel, Péter Mátray, Murali Krishna Srinivasan, and Tobias Lindquist. 2022. On the Use of Linux Real-Time Features for RAN Packet Processing in Cloud Environments. In *High Performance Computing. ISC High Performance 2022 International Workshops*, Hartwig Anzt, Amanda Bienz, Piotr Luszczek, and Marc Baboulin (Eds.). Springer International Publishing, Cham, 371–382.

[3] Zulfiqar Ahmad, Babar Nazir, and Asif Umer. 2021. A fault-tolerant workflow management system with QoS-aware scheduling for scientific workflows in cloud computing. *International Journal of Communication Systems* 34, 1 (2021), e4649.

[4] Fredrik Alriksson, Lisa Boström, Joachim Sachs, Y-P Eric Wang, and Ali Zaidi. 2020. Critical IoT connectivity Ideal for Time-Critical Communications. *Ericsson technology review* 2020, 6 (2020), 2–13.

[5] Remo Andreoli, Harald Gustafsson, Luca Abeni, Raquel Mini, and Tommaso Cucinotta. 2023. Design-time Analysis of Time-Critical and Fault-Tolerance Constraints in Cloud Services. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE, 415–417.

[6] Stamatia Bibi, Dimitrios Katsaros, and Panayiotis Bozanis. 2010. Application Development: Fly to the clouds or stay in-house?. In *19th Intern. Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises*. IEEE, 60–65.

[7] Giorgio Buttazzo, Enrico Bini, and Yifan Wu. 2011. Partitioning Real-Time Applications Over Multicore Reservations. *IEEE Transactions on Industrial Informatics* 7, 2 (2011), 302–315.

[8] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. 2011. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience* 41, 1 (2011), 23–50.

[9] Mehdi Nazari Cheraghlou, Ahmad Khadem-Zadeh, and Majid Haghparast. 2016. A survey of fault tolerance architecture in cloud computing. *Journal of Network and Computer Applications* 61 (2016), 81–92.

[10] Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Riccardo Mancini, and Carlo Vitucci. 2021. Strong Temporal Isolation among Containers in OpenStack for NFV Services. *IEEE Transactions on Cloud Computing* (2021), 1–1.

[11] Tommaso Cucinotta, Luigi Pannocchi, Filippo Galli, Silvia Fichera, Sourav Lahiri, and Antonino Artale. 2022. Optimum VM Placement for NFV Infrastructures. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE.

[12] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. 2012. High performance network virtualization with SR-IOV. *J. Parallel and Distrib. Comput.* 72, 11 (2012), 1471–1480.

[13] Qiang Duan, Shangguang Wang, and Nirwan Ansari. 2020. Convergence of Networking and Cloud/Edge Computing: Status, Challenges, and Opportunities. *IEEE Network* 34, 6 (2020), 148–155.

[14] Mostafa Elhemali, Niall Gallagher, Bin Tang, Nick Gordon, Hao Huang, Haibo Chen, Joseph Idziorek, Mengtian Wang, Richard Krog, Zongpeng Zhu, et al. 2022. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. In *USENIX Annual Technical Conference*. 1037–1048.

[15] Fay Chang et al. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions Computing Systems* 26, 2, Article 4 (jun 2008), 26 pages.

[16] Norman Finn. 2018. Introduction to time-sensitive networking. *IEEE Communications Standards Magazine* 2, 2 (2018), 22–28.

[17] Dennis Gannon, Roger Barga, and Neel Sundaresan. 2017. Cloud-native applications. *IEEE Cloud Computing* 4, 5 (2017), 16–21.

[18] Pengze Guo, Ming Liu, Jun Wu, Zhi Xue, and Xiangjian He. 2018. Energy-efficient fault-tolerant scheduling algorithm for real-time tasks in cloud-based 5G networks. *IEEE Access* 6 (2018), 53671–53683.

[19] Moin Hasan and Major Singh Goraya. 2018. Fault tolerance in cloud computing environment: A systematic survey. *Computers in Industry* 99 (2018), 156–172.

[20] Chesta Kathpal and Ritu Garg. 2019. Survey on fault-tolerance-aware scheduling in cloud computing. In *Information and Communication Technology for Competitive Strategies: Proc. of Third International Conf. on ICTCS*. Springer, 275–283.

[21] Zengpeng Li, Huiqun Yu, Guisheng Fan, and Jiayin Zhang. 2023. Cost-efficient Fault-tolerant Workflow Scheduling for Deadline-constrained Microservice-based Applications in Clouds. *IEEE Trans. on Network and Service Management* (2023).

[22] I. Mahadevan and K.M. Sivalingam. 1999. Quality of Service architectures for wireless networks: IntServ and DiffServ models. In *Proceedings Fourth International Symposium on Parallel Architectures, Algorithms, and Networks*. 420–425.

[23] Sheheryar Malik and Fabrice Huet. 2011. Adaptive Fault Tolerance in Real Time Cloud Computing. In *2011 IEEE World Congress on Services*. 280–287.

[24] Toni Mastelic, Ariel Oleksiak, Holger Claussen, Ivona Brandic, Jean-Marc Pierson, and Athanasios V. Vasilakos. 2014. Cloud Computing: Survey on Energy Efficiency. *ACM Comput. Surv.* 47, 2, Article 33 (dec 2014), 36 pages.

[25] Bruno Ordozgoiti, Alberto Mozo, Sandra Gómez Canaval, Udi Margolin, Elisha Rosensweig, and Itai Segall. 2017. Deep convolutional neural networks for detecting noisy neighbours in cloud infrastructure. *COSTAC 2017* (2017), 59.

[26] Peter O'Donovan, Colm Gallagher, Kevin Leahy, and Dominic T.J. O'Sullivan. 2019. A comparison of fog and cloud computing cyber-physical interfaces for Industry 4.0 real-time embedded machine learning engineering applications. *Computers in Industry* 110 (2019), 12–35.

[27] Aleksi Peltonen, Ralf Sasse, and David Basin. 2021. A Comprehensive Formal Analysis of 5G Handover. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks* (Abu Dhabi, United Arab Emirates) *(WiSec '21)*. Association for Computing Machinery, New York, NY, USA, 1–12.

[28] Vicent Selfa, Julio Sahuquillo, Lieven Eeckhout, Salvador Petit, and María E. Gómez. 2017. Application Clustering Policies to Address System Fairness with Intel's Cache Allocation Technology. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 194–205.

[29] Roozbeh Siyadatzadeh, Fatemeh Mehrafrooz, Mohsen Ansari, Bardia Safaei, Muhammad Shafique, Jörg Henkel, and Alireza Ejlali. 2023. ReLIEF: A Reinforcement Learning-Based Real-Time Task Assignment Strategy in Emerging Fault-Tolerant Fog Computing. *IEEE Internet of Things Journal* (2023), 1–1.

[30] Václav Struhár, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, and Alessandro V. Papadopoulos. 2021. REACT: Enabling Real-Time Container Orchestration. In *26th IEEE Int. Conf. on Emerging Techn. and Factory Automation*.

[31] Márk Szalay, Péter Mátray, and László Toka. 2021. Real-time task scheduling in a FaaS cloud. In *IEEE 14th International Conference on Cloud Computing*. 497–507.

[32] Erwin van Eyk, Lucian Toader, Sacheendra Talluri, Laurens Versluis, Alexandru Uță, and Alexandru Iosup. 2018. Serverless is More: From PaaS to Present Cloud Computing. *IEEE Internet Computing* 22, 5 (2018), 8–17.

[33] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. 2011. RT-Xen: Towards real-time hypervisor scheduling in Xen. In *Proceedings of the ninth ACM international conference on Embedded software*. 39–48.

[34] Xipeng Xiao, Alan Hannan, Brook Bailey, and Lionel M Ni. 2000. Traffic Engineering with MPLS in the Internet. *IEEE network* 14, 2 (2000), 28–33.

[35] Guangshun Yao, Qian Ren, Xiaoping Li, Shenghui Zhao, and Rubén Ruiz. 2022. A Hybrid Fault-Tolerant Scheduling for Deadline-Constrained Tasks in Cloud Systems. *IEEE Trans. on Services Computing* 15, 3 (2022), 1371–1384.