


Università degli Studi di Napoli Federico II

Ph.D. Program in

Information Technology and Electrical Engineering

XXXVII Cycle

THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Mixed-criticality Orchestration of Real-time Containerized Systems

by

MARCO BARLETTA

Advisor: Prof. Marcello Cinque

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE

A Fabio,
ed a tutto ciò che di lui
rimarrà per sempre in noi

Mixed-criticality Orchestration of Real-time Containerized Systems

Ph.D. Thesis presented
for the fulfillment of the Degree of Doctor of Philosophy
in Information Technology and Electrical Engineering
by

Marco Barletta

October 2024

Approved as to style and content by

_____

Prof. Marcello Cinque, Advisor

Università degli Studi di Napoli Federico II
Ph.D. Program in Information Technology and Electrical Engineering
XXXVII cycle - Chairman: Prof. Stefano Russo

http://itee.dieti.unina.it

Candidate's declaration

I hereby declare that this thesis submitted to obtain the academic degree of Philosophiæ Doctor (Ph.D.) in Information Technology and Electrical Engineering is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.
Parts of this dissertation have been published in international journals and/or conference articles (see list of the author's publications at the end of the thesis).

Napoli, December 12, 2024

_____

Marco Barletta

Abstract

As critical computing systems employed in industry verticals grow increasingly complex, hardware components are replaced with software ones that demand greater computational resources. Hence, embedded systems are evolving into more powerful mixed-criticality platforms.

Recent research has explored edge computing, virtualization, and cloud-native technologies to automate component management and increase flexibility, interoperability, and scalability. However, cloud-native technologies like container orchestration systems were not designed to meet the heterogeneity of hardware and non-functional requirements of industrial settings.

This dissertation investigates whether current container orchestration systems meet the requirements of critical systems, and introduces mixed-criticality orchestration to address their limitations. The dissertation makes three key contributions: i) it introduces a model for mixed-criticality orchestration, ii) it performs a failure and timing analysis to assess the behavior of container orchestration systems in non-nominal conditions, and iii) it proposes a set of solutions based on the model to improve the resilience, timeliness, and isolation from interference for both container orchestration and containers.

The analysis reveals that even a single error can cause overloads and disrupt an entire cluster, whereas high orchestration loads can cause delays of tens of seconds in scaling services or handling failures, threatening service level objectives (SLOs). The proposed solutions include designs and methods for mixed-criticality orchestration, which builds upon the concepts of node *assurance* and service and pod *criticality* to differentiate the management of critical services when orchestrating, placing, and running them. The implemented prototypes demonstrate that mixed-criticality orchestration improves the resilience of critical services by providing stable orchestration times, and improved timing and failure isolation for critical containers.

Keywords: containers, orchestration, real-time, mixed-criticality, resilience, cloud.

Sintesi in lingua italiana

La crescente complessità dei sistemi di elaborazione dati in ambienti critici industriali ha portato ad una progressiva sostituzione di componenti *hardware* con componenti *software* che richiedono maggiori risorse computazionali. Quindi, i sistemi integrati stanno evolvendo in più potenti piattaforme a criticità mista, le quali ospitano componenti di diversi livelli di criticità.

Ricerche recenti hanno esplorato l'uso dell'*edge computing*, della virtualizzazione e delle tecnologie *cloud* per semplificare la gestione dei componenti e aumentare flessibilità, interoperabilità e scalabilità. Tuttavia, le tecnologie *cloud* come i sistemi di orchestrazione di container, non sono state progettate per soddisfare l'eterogeneità dei requisiti non funzionali e dell'*hardware*, tipica dei contesti industriali.

Questa tesi analizza se gli attuali sistemi di orchestrazione di container soddisfano i requisiti dei sistemi critici e introduce l'orchestrazione a criticità mista per superare i loro limiti. Essa fornisce tre contributi principali: i) introduce un modello per l'orchestrazione a criticità mista, ii) esegue un'analisi dei fallimenti e dei tempi per valutare il comportamento dei sistemi di orchestrazione in condizioni non nominali, e iii) propone delle soluzioni basate sul modello per migliorare la resilienza, la tempestività e l'isolamento dalle interferenze, sia per l'orchestrazione dei container che per i container stessi.

L'analisi mostra che anche un singolo errore può causare sovraccarichi e far fallire un intero *cluster* e che, sotto carico elevato, un orchestratore può impiegare decine di secondi per ampliare le risorse di un servizio o gestire un fallimento, mettendo a rischio gli obiettivi di livello di servizio. Le soluzioni proposte includono architetture e metodi per un'orchestrazione guidata dalla criticità, basata sui concetti di *garanzia* del nodo e *criticità* di servizio per differenziarne la gestione. I prototipi implementati mostrano un miglioramento della resilienza dei servizi critici attraverso tempi di orchestrazione stabili e isolamento dai fallimenti e dalle interferenze temporali per i container critici.

Parole chiave: container, orchestrazione, tempo reale, criticità mista, resilienza, cloud.

Acknowledgements

I want to express my sincere gratitude to my advisor Prof. Marcello Cinque. He knew since the first day of my Ph.D. that he should have been a mental coach more than a scientific guide, and he did it perfectly. He has always pushed me to focus on the vision rather than thinking of publishing, and I will keep this precious lesson with me forever. Nonetheless, I want to thank the rest of my team: Dr. Luigi De Simone, one of the most enthusiastic people in his job; Dr. Raffaele Della Corte, who tried to calm me down several times; and, of course, my lab mates Daniele and Giorgio.

I am grateful to the DEPEND research group of the University of Illinois at Urbana-Champaign and the network automation department of Nokia Bell Labs for hosting me during my research abroad. A warm acknowledgment goes to Prof. Ravi Iyer, a humble legend and one of the most inspiring people I have ever met, Prof. Zbigniew Kalbarczyk, one of the most precise and determined people out there, and the indefatigable Lelio Di Martino (don't worry Lelio, I will find my *why*).

Last but not least, I want to thank the entire "DEpendable and Secure Software Engineering and Real-Time systems" (DESSERT) research group for hosting me during the past three years. The thoughtful confrontation with everyone in the lab and the fun times together were essential to my growth, and I will carry with me every lesson from Prof. Cotroneo and Prof. Russo.

The research work contained in this dissertation has been funded by a Ph.D. scholarship from Federico II University of Naples, and partially supported by the University of Illinois at Urbana-Champaign and Nokia Bell Labs.

Dedication

This dissertation is dedicated to Fabio Barletta.
Through his battle against cancer, he taught me how to smile through the storm and how to climb a peak after a fall.

I hope one day I will become what you thought I could have become.
With all the love of this world,
forever your brother,
Marco

Contents

List of Acronyms

The following acronyms are used throughout the thesis.

AI	artificial intelligence
API	application programming interface
APU	application processing unit
CAN	controller area network
CBS	constant bandwidth server
CNI	container networking interface
COTS	commercial-off-the-shelf
CPU	central processing unit
CRI	container runtime interface
DAG	direct acyclic graph
DNS	domain name system
DS	deferrable server
ECU	electronic control unit

EDF earliest deadline first

FaaS function as a service

FFDA field failure data analysis

FPGA field gate programmable array

GPU graphic processing unit

HA high availability

I/O input/output

ILP integer linear programming

IT information technology

ISA instruction set architecture

MAE mean absolute error

MTBF mean time between failures

MTTF mean time to failure

MTTR mean time to repair

MMU memory management unit

MPU memory protection unit

NFV network function virtualization

OCI Open Container Initiative

OS operating system

OT operational technology

PLC programmable logic controller

pWCET probabilistic worst case execution time

pWCRT probabilistic worst case response time

QoS quality of service

RPU real-time processing unit

RTOS real-time operating system

SDN software defined network

SLO service level objective

SOA service oriented architecture

SWaP-C size, weight, power, and cost

TBS total bandwidth server

TCM tightly coupled memory

TDMA time division multiple access

TMR triple modular redundancy

TSN time sensitive networking

UPF user plane function

VM virtual machine

VMM virtual machine monitor

VNF virtual network function

WCET worst case execution time

WCRT worst case response time

List of Figures

List of Tables

Chapter 1

Introduction

Take risks. Ask big questions. Don't be afraid to make mistakes; if you don't make mistakes, you're not reaching far enough.

David Packard

1.1 The Cloudification of Mixed-criticality Systems

Modern industry verticals such as factories, transports, banks, and networks rely on critical computing systems, including mission-critical and safety-critical system components. A failure or disruption of a critical component can result in unacceptable losses such as injuries and environmental catastrophes for safety-critical systems, or economic losses, social turmoil, and serious impacts on business operations for mission-critical systems. In this perspective, "criticality is a designation of the level of assurance against failure required for a system component" [1].

The exchange, elaboration, and use of information in such critical systems have traditionally been managed through simple logic, implemented by digital or analogic custom hardware or embedded software running on simple *ad hoc* microcontrollers. This design guaranteed meeting the non-functional requirements such as strict timing constraints, very low failure probability (e.g., $< 10^{-7}$), low power consumption, etc.

However, critical systems are growing increasingly complex, as they must be intelligent, through the use of artificial intelligence (AI) algorithms, inter-

connected with the surrounding ever-changing environment, and broad, due to the multitude of auxiliary electronic systems. What is more, those systems need to be more flexible and quickly reconfigurable than before to keep up with the evolving and fluctuating needs, while maintaining high standards of non-functional requirements (e.g., availability, reliability, fault-tolerance, safety, and real-time constraints).

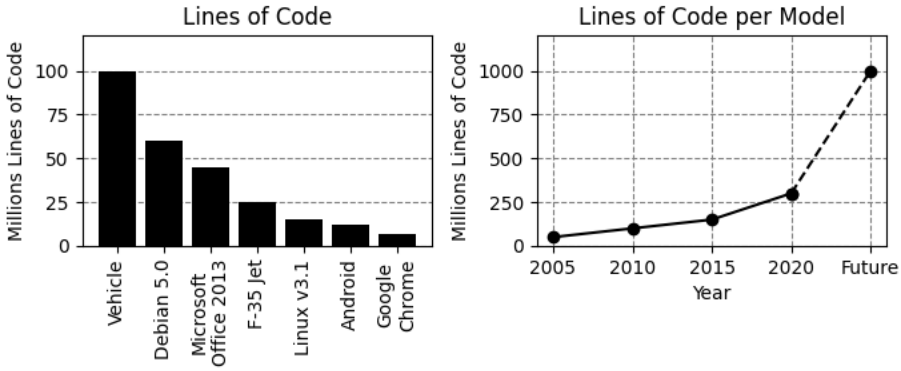


Figure 1.1. Complexity of automotive systems over the years. Data from [2, 3]

Examples of critical industry verticals following the described trends are: (i) reconfigurable manufacturing, [4, 5, 6], which foresees connected factories populated with reconfigurable production lines that can be rented hourly for small-scale production, (ii) transports, including aircraft [7], railways [8], and cars (see Figure 1.1) [9], which have increasingly complex features, like assisted driving and environmental sensing to support interconnected use cases like car platooning, and (iii) 5G networks [10, 11], which serve applications with different needs (e.g., large bandwidth or low latencies) on the same infrastructure and devices that continuously join/leave the network.

The required levels of flexibility, reconfigurability, and feature complexity call for a shift from embedded or hardware-implemented components to software system components that rely on fully-fledged operating systems, runtimes, and libraries. Indeed, hardware/embedded components are costly to scale, develop, and maintain and difficult to update. Conversely, software components running on operating systems and general-purpose hardware are cheaper, more portable, and can implement more complex functionalities.

Accordingly, the hardware computing infrastructure is evolving from old-styled embedded systems to mixed-criticality systems characterized by more powerful and heterogeneous platforms hosting multiple components with different levels of criticality [1]. Indeed, mixed-criticality systems are a viable solution to cope with the increasing complexity while meeting the strict size, weight, power, and cost (SWaP-C) requirements of onboard electronics.

In the landscape of the progressive component *softwarization*, integrating recent information technology paradigms, methods, and technologies into the industry verticals is an opportunity to further improve maintainability and flexibility and support unprecedented use cases. The research literature often refers to this integration as information technology (IT)-operational technology (OT) convergence. The IT-OT convergence can take advantage of the evolution that software system architectures, development methods, and operational processes underwent over the past two decades to address the challenges raised by web services. In that context, cloud computing, microservice paradigms, and virtualization technologies like *containers* (i.e., virtualization units containing applications and all their dependencies) have become a *de facto* standard. They have provided maintainability, scalability, elasticity, and resilience to an increasing number of software units deployed across increasingly large clusters of computing nodes.

On a similar note, cloud models and their properties are a possible solution to tame the unmanageable complexity of increasingly dynamic industrial systems. Turning *softwarized* components into cloud-native components (e.g., containers) allows them to be easily upgraded, migrated, and deployed across the nodes of the computing infrastructure. Some computing nodes, like onboard electronic control units, are turned into increasingly powerful mixed-criticality systems. Conversely, when SWaP-C constraints are stricter (e.g., robots), onboard devices remain simple embedded controllers that offload heavy computational tasks to micro-datacenters and servers placed at the *edge* (e.g., inside the factories, on the roadsides). Offloading allows leveraging the edge computational resources with lower communication latencies than the cloud, while meeting SWaP-C constraints of onboard electronics.

The joint use of cloud-native components, service-based paradigms, edge computing, and cloud tools and technologies gives birth to a real-time and mixed-criticality edge-cloud environment. In this environment, industrial components with different criticality levels and timing constraints are seam-

lessly integrated into DevOps (Development and Operational) disciplines as any other cloud component. This integration fosters the above-mentioned properties of maintainability, scalability, elasticity, and resilience.

In this perspective, container orchestration systems constitute one of the essential building blocks of the real-time mixed-criticality edge-cloud to tame the proliferation of software components and the increased computational resources. Container orchestration systems (hereon, *orchestrators*) are distributed systems that manage the life cycle of containers, including placement, deployment, scaling, and self-healing, across a cluster of nodes (ranging from a few tenths to thousands of nodes) [12, 13, 14, 15]. Examples of commercially spread out orchestrators are *Docker Swarm*, *Kubernetes*, *Apache Mesos*, and by extension *OpenStack*. Hence, they provide automated management of the computing infrastructure and software units, acting as cloud operating systems [16, 12, 17].

This dissertation focuses on container orchestration systems and containers, due to their importance in mixed-criticality real-time edge-cloud systems.

1.2 Challenges

The adoption of cloud technologies in industrial mixed-criticality systems introduces significant challenges, demanding substantial adaptation. While recent research has thoroughly investigated real-time containers, much less attention has been devoted to understanding orchestration systems in order to support real-time and mixed-criticality requirements.

The main challenge is that, although cloud technologies are well-suited to manage large-scale computing infrastructures encompassing many nodes and software units, they assume limited heterogeneity in both the computing infrastructure and requirements of deployed applications. While cloud environments may include constraints like accelerators or specific central processing unit (CPU) architectures, nodes generally have similar characteristics. As a result, orchestration decisions are primarily guided by available resources (e.g., CPU and memory) and topology constraints.

Conversely, in industrial settings, the heterogeneity of the computing infrastructure and application requirements is the key. On the one hand, the computing infrastructure features nodes ranging from entry-level boards to virtual machines on edge servers, different network types (e.g., real-time,

best-effort, and monitoring networks), peculiar devices (e.g., sensors, actuators, real-time processing units), and different hypervisors and operating system (OS) (e.g., commodity, certified, real-time, lib-OSes). On the other hand, mixed-criticality systems have timing and reliability/availability constraints stricter than cloud services, requiring a deeper understanding of orchestrators' non-functional characteristics like resilience, timeliness, and isolation.

While recent research extends orchestrators to support real-time containers and networks, it still relies on functionalities provided by the vanilla orchestrator core, which is not designed from the ground up to support critical applications. In this perspective, such a research trend aims to provide one-size-fits-all technologies like Linux containers and cloud orchestrators with real-time capabilities, overcoming heterogeneity.

However, heterogeneity is necessary to cope with criticality, strict non-functional requirements, and backward compatibility for legacy applications. For example, some applications may be required to run on certified hardware/software to guarantee safety despite faults/errors and transient load conditions. Such requirements must be considered in every aspect of the system design, and critical applications must be treated differently from real-time yet not critical ones.

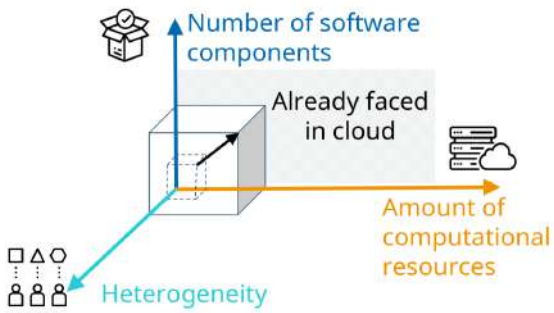


Figure 1.2. Challenges coped by this dissertation.

1.3 Contributions

This dissertation investigates whether it is possible to use container orchestration systems in mixed-criticality clusters and proposes models and ar-

chitectures to improve the non-functional properties of critical services. In particular, the dissertation analyzes both functional and non-functional properties of current orchestrators and their impact on managed applications. Among the functional properties, the dissertation focuses on placement policies that are aware of the criticality and real-time constraints of applications. As for the non-functional properties of orchestrators and containers, the dissertation focuses on both dependability and real-time capabilities.

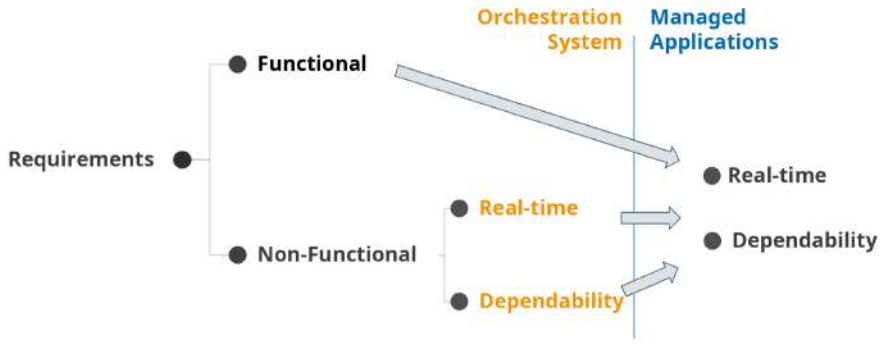


Figure 1.3. Aspects analyzed in the dissertation.

In detail, the contributions of the dissertation are the following:

- it introduces a model for mixed-criticality orchestration. The model introduces the concepts of node assurance and service criticality. Based on the model, the orchestration commands of diversified replication, seamless migration, and diversified rolling update are introduced. The failure probability of services relying on those orchestration commands is analyzed with both non-state-space and state-space models.
- it performs a failure and timing analysis to assess the behavior of containers and container orchestration systems in non-nominal conditions. The analysis shows that even a single error can disrupt an entire cluster and that, under heavy orchestration loads, orchestration times can grow up to tens of seconds, compromising applications' service level objective (SLO). Furthermore, Linux real-time containers present unacceptable timing variability for critical tasks when running on nodes that do not have *ad hoc* software/hardware features to mitigate interferences, regardless of possible kernel isolation configurations.

- it proposes solutions to improve the resilience, timeliness, and isolation from interference for both container orchestration systems and containers. The proposed solutions, based on the blueprint of the introduced model, include architectural designs for mixed-criticality orchestrators and partitioned containers. The former are orchestrators that differentiate the service management based on their SLOs, the latter are applications running in an isolated partition of a mixed-criticality platform but seen and managed as any other container by orchestrators.

Throughout the chapters, the dissertation shows that current orchestrators are not ready for mixed-critical environments. However, the implemented prototypes demonstrate that the proposed architectural designs built upon the proposed model improve the resilience and timing isolation of critical services, and provide them with orchestration times independent of concurrent orchestration load.

1.4 Thesis Outline

The rest of the dissertation is structured as follows.

Chapter 2 provides some background regarding orchestration systems, dependability assessment, and real-time systems, which are necessary to understand the rest of the dissertation.

Chapter 3 introduces the mixed-criticality orchestration model and commands, which abstract the entities of a mixed-criticality and real-time containerized system. The chapter analyzes the service failure probability and shows the impact of the orchestrator on the services SLOs.

Chapter 4 analyzes orchestrators' failure modes through real-world failures, assesses their fault tolerance through fault injection, and presents a fine-grained timing analysis of orchestration times and real-time containers under concurrent increasing stress load. The chapter introduces fault/error injection methodologies to systematically test the fault tolerance of orchestration systems and investigates the sources of delay that can affect the orchestration of critical services.

Chapter 5 introduces architectural designs and policies for mixed-criticality and real-time container orchestration and partitioned containers. The chapter provides details of the implementation of some of the proposed concepts on Kubernetes, i.e., the most widely used orchestrator.

Chapter 6 evaluates the implemented prototypes. The chapter evaluates boot times, isolation from timing interferences, and isolation from failures of partitioned containers, comparing them with other real-time containers. Moreover, the chapter evaluates the criticality-aware placement policy and repeats the timing analysis of orchestration times present in chapter 4 on the implemented prototype.

Chapter 7 presents related work in the research literature on both the dependability and timing aspects of container orchestration systems and containers. The related work is organized into three parts, based on the subsystem involved: orchestration logic, placement policies, and container runtimes and technologies.

Chapter 8 summarizes the contribution of the dissertation and provides possible future research directions.

Chapter 2

Background

To break the rules you must first master them.

Audemars Piguet

This chapter provides definitions and other background concepts widely used in the rest of the dissertation. In particular, the chapter contains information regarding container orchestration systems, dependability assessment, and real-time systems.

Container orchestration systems are the main focus of this dissertation. Dependability assessment and real-time capabilities are of interest in mixed-criticality systems because of the requirements of failure probability and impact prescribed by industrial standards, and the temporal constraints due to, for example, the surrounding physical environment.

Section 2.1 gives a primer on containers and container orchestration systems, with details regarding Kubernetes, the standard *de facto* among container orchestration systems.

Section 2.2 provides the primer of dependability assessment, with basic dependability definitions and a focus on fault injection techniques.

Section 2.3 provides the primer of real-time systems, with basic definitions and a focus on real-time scheduling algorithms and hierarchical frameworks useful for virtualized environments.

Finally, section 2.4 illustrates key example scenarios addressed in this dissertation which help define the system requirements.

2.1 Container Orchestration Systems

2.1.1 Basic definitions

A Container is a "standard unit of software that packages applications and all their dependencies" [18], which can be easily distributed and ported as *container image* to multiple computing environments. Commonly adopted container solutions leverage OS kernel features to isolate processes running on a shared kernel [19]. For this reason, containers are often referred to as OS-level (or lightweight) virtualization techniques. The Open Container Initiative (OCI) defined standards to describe container images [20] and container runtimes [21] to guarantee interoperability between tools that pack and run containers. Container images are divided into layers to reuse and share the layers containing libraries.

Container orchestration systems (from now on, also shortened as *orchestrators*) are distributed systems that manage the life cycle of containers across a cluster of nodes (ranging from a few tens to thousands of nodes). The management includes placement, deployment, scaling, health monitoring, self-healing, load balancing, and networking of containers [12, 13]. The functionalities are similar to the ones provided by cloud management platforms, like OpenStack. A pod is the minimum deployable unit managed by an orchestrator, and it consists of a set of related containers. Some orchestrators do not support the abstraction of a group of containers, and the pod coincides with a single container.

In this dissertation, I use the word *service* to indicate a set of one or multiple running pods responding to clients' requests to provide a discrete unit of functionality. Each service is characterized by an SLO, which is an agreed-upon performance target for a particular service over a period of time and specifies non-functional requirements (e.g., availability, latency).

An orchestrator can be divided without loss of generality into a *i) control plane* and a *ii) compute cluster* [12].

The control plane receives user requests, monitors the *current state* of the compute cluster, and compares it with a *desired state*. If the two states differ, the orchestrator performs *orchestration commands* to harmonize the two states [17]. I refer to this harmonization of states as *reconciliation*, and I define as *orchestration time* the time required for an orchestration command to be completed. For example, if a cluster user defines a desired number of

load-balanced pods for a service and the failure of one of them is detected, the control plane reacts accordingly to restore the desired state. The po is restarted on the same node if available; otherwise, it is rescheduled on a different node.

The compute cluster is in charge of running the pods belonging to the services, and it is composed of a set of *worker nodes*, to which the pods are assigned to run. The control plane issues orders to agents deployed on each worker node, while the agents periodically report the state of nodes to the control plane.

I define *orchestration workload* the set of orchestration commands that the orchestrator must perform to reconcile the current state and the desired state, upon either user request or cluster state change (e.g., a failure). Hence, the orchestration workload is composed of the set of events that cause an intervention of the orchestrator rather than the set of deployed services, since at steady state the services trigger almost no orchestration activity.

An orchestrator handles several *resource types* (e.g., pods, services, nodes, etc.), which have multiple *resource instances*. Each resource instance has a desired and an observed state, reconciled by orchestration commands.

2.1.2 Orchestration commands

An orchestration command can be decomposed into a flow of simpler actions. This flow represents the sequence of actions that a human operator would manually do when managing the computing infrastructure. Each action can modify the state of one or multiple resource instances, and thus the state of the cluster c_i, which includes the state of all resource instances.

A state change, in turn, may require to be handled with other cascading actions. For example, dependent resource instances may be created, deleted, or updated. Hence, an orchestration command can be represented as a linearizable sequence of state changes, i.e., $[c_i, c_k, ..., c_j]$, and modeled as a direct acyclic graph (DAG) of actions (see Figure 2.1). The fan-out of a graph node is the set of cascading actions triggered.

Practically, each resource type implements a control loop to perform the required actions. The control loop performs necessary reconciliations (including different actions) for the controlled resource instances.

One of the most important actions in an orchestration command is the *scheduling* process, through which the control plane decides the placement of

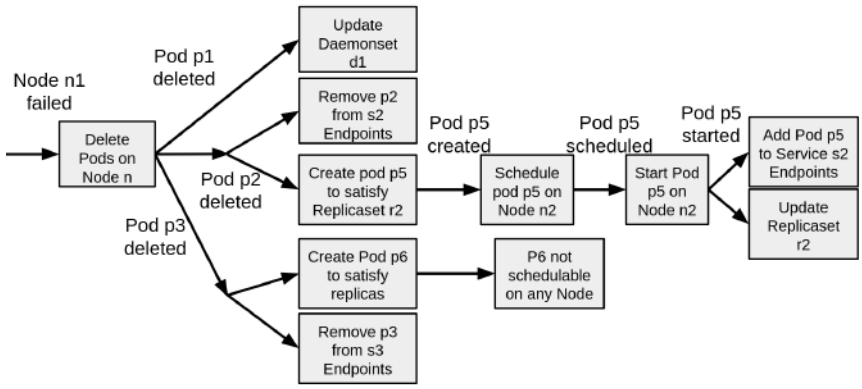


Figure 2.1. Simplified example graph of the orchestration command to handle a node failure in Kubernetes. Specific Kubernetes terminology is introduced later in the chapter.

a pod on the worker nodes. The scheduling can be triggered by a deployment request, a failure requiring a respawn, resource reclaiming, or a worker node that no longer respects pods' placement constraints. The decision is based on both placement constraints defined by the user and optimization of the cluster usage metrics.

Orchestrators generally provide some methods to support custom orchestration commands, with *ad hoc* custom workflows. For example, Apache Mesos supports custom modules and K8s allows writing custom controllers, which can operate on custom resources. For example, KubeEdge[1] extends Kubernetes by introducing controllers designed to provide MQTT integration, support device management, and manage edge nodes with intermittent connectivity and limited resources.

I define the *"deploy"*, *"scale"*, and *"failover"* orchestration commands, which are widely used by orchestrators [16]. The "deploy" and "scale" commands create new pods or increase the pod replicas, respectively, including scheduling and starting such pods. The "failover" command restores the desired number of pod replicas upon a worker node failure.

[1]Additional information available at `https://kubeedge.io/`

2.1.3 Architecture of an orchestrator

Simplifying the architecture and terminology introduced in [12], the main logical components of an orchestrator are described as follows.

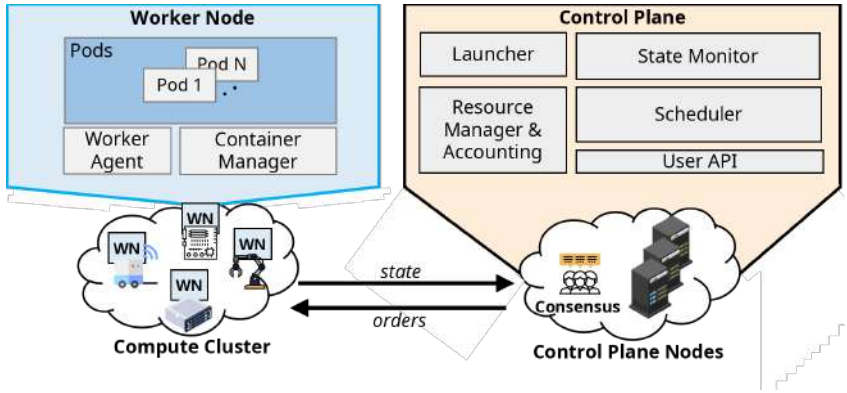


Figure 2.2. Logical view of the architecture of a container orchestration system.

Control plane The control plane includes the following functional modules: *i)* a *launcher* that issues orders to the worker nodes (i.e., push-based updates) or informs the worker nodes about changes upon query (i.e., pull-based updates); *ii)* a *state monitor* that stores the global state of the worker nodes and pods, and issues orders when the state deviates from the desired one; *iii)* a *resource manager & accounting* that tracks the resource availability within the compute cluster and their usage for both scheduling and accounting purposes; *iv)* a *scheduler* that decides on which worker node to place a pod, between the nodes having enough available resources; *v)* a cluster *user application programming interface (API)*, which receives deployment requests, validates them and triggers the scheduler if needed. Each of these modules communicates with the others to carry out its tasks. For example, the scheduler decides the worker node based on the available resources tracked by the resource manager and the state of the worker nodes tracked by the state monitor, and then delegates to the launcher the task to issue the order. The functional modules can implemented by one or multiple deployment components, i.e. components deployed independently that have a defined set of responsibilities. The components are deployed onto one or multiple *control*

plane nodes, that may run a consensus protocol.

Compute cluster The compute cluster is made up of several worker nodes, each of which runs: *i)* a *worker node agent*, which collects and reports to the control plane the state and health information regarding the worker node itself and the pods scheduled on the node; *ii)* a *container manager*, which spawns the containers; and *iii)* the pods deployed. The health information regarding the node is sent to the control plane, while the one regarding pods is used to restart them when necessary.

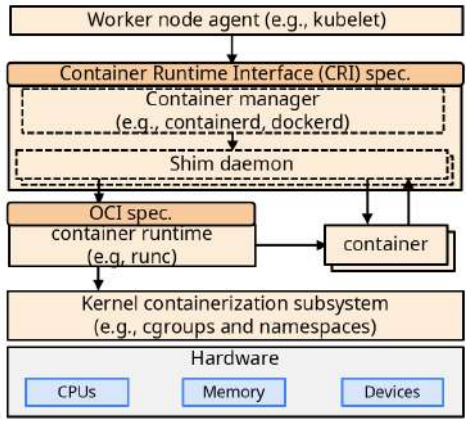


Figure 2.3. Worker node software stack required by containers

In some orchestrators, the orchestrator depends on a specific container manager (e.g., Docker for Docker Swarm), while for other orchestrators (e.g., Kubernetes) the container manager is interchangeable. Over the years, the software stack required to manage containers has been divided into layers (see Figure 2.3) with specific responsibilities and standardized interfaces to foster modularity and customization. In the following, the duties of each layer are summarized.

The worker node agent interacts with the container manager through the container runtime interface (CRI) API, introduced by Kubernetes to allow the use of multiple container managers (e.g., containerd, docker, CRI-O).

The container manager tracks the state and lifecycle of containers, downloads or updates the container images, and manages virtual networks for the containers. Specifically, the container manager downloads (if available) a ver-

sion of a container image that is compatible with the worker node on which it runs. I define a *platform* as the hardware/software determining the container image to download. Usually, the platform is composed of the OS and the instruction set architecture (ISA). Thus, different platforms require different versions of the same container image, for example one for Linux x86-64 and one for ARM64 platforms. Internally, some container managers (like containerd) may use a shim daemon (typically one per container) to interact with the container runtime and the container itself.

Container runtimes (e.g., runc, runv, youki) expose an API defined by the OCI standard, which mandates exposing at least the functions to create, start, kill, delete, and retrieve the status of a container. The OCI runtime performs the system calls to configure the required resources for the container. Once a process runs in a container, its visibility and access to hardware resources are regulated by the kernel. For example, runtimes for Linux containers perform system calls to configure the *cgroup* and *namespaces* and move the processes into the container. The namespaces define the visibility of a container over the system, while cgroups regulate the access to hardware resources, including CPU time, memory, etc.

2.1.4 Kuberentes

Kubernetes [22](shortened as *K8s* hereon) is currently the most widely used orchestrator, with 97% of companies surveyed in [23] that use or evaluate to use it. Multiple solutions inherit its codebase (e.g., OpenShift, K0s, K3s, microK8s) [24], stripping down features and re-packing the core K8s components. The rest of this dissertation mainly refers to K8s for the design and implementation of the prototypes. Nonetheless, whenever needed, it is specified how the concepts can be generalized to the entire class of orchestrators.

K8s resources [2] K8s handles several *resource kinds* (e.g., *Pod*, *ReplicaSet*, *Deployment*, *Node*), which are representations of the system entities used to implement orchestration functionalities[3]. For each resource kind, there are

[2]This paragraph is taken from "Mutiny! How Does Kubernetes Fail, and What Can We Do About It?" ©IEEE 2024 (see author's publications section), with adaptations for this thesis.

[3]More information available at `https://kubernetes.io/docs/reference/using-API/API-concepts/`

multiple *resource instances*, i.e., objects. The following paragraph provides some details about the resource kinds relevant to this dissertation, referring to the official K8s documentation for a complete understanding [22].

A *Pod* is a set of containers deployed in an isolated sandbox with reserved hardware resources. The Pods are stateless or store their states externally (e.g., in *Volumes*). I write "Pod" to indicate a K8s pod, and "pod" to refer to the general concept as defined above. A *ReplicaSet* ensures that the desired number of *Pod replicas* (i.e., Pod instances running the same application) is running at any given time. A *Deployment* manages rolling updates of the container images and the replica number of a ReplicaSet. A *DaemonSet* is similar to a *Deployment*, but it spawns Pods on every *Node* (defined later) that satisfies the constraints. A *Service* exposes a single network address to contact an application that is running as one or more Pods matching the Service *Label*, which are called *Endpoints*. The Service load balances incoming requests between the Endpoints. I write "Service" to indicate the K8s resource kind, and "service" to indicate the general concept introduced in §2.1. A Node is either a control plane node or a worker node of the cluster, characterized by a state and available resources. All resource instances carry some metadata, like *Annotations* and *Labels*, which provide a flexible mechanism to group resource instances, create relationships between them, and store custom information. *Taints* are a type of label that can prevent the placement of new Pods on a Node (*NoSchedule* taint) or force the termination of Pods currently running on a Node (*NoExecution* taint).

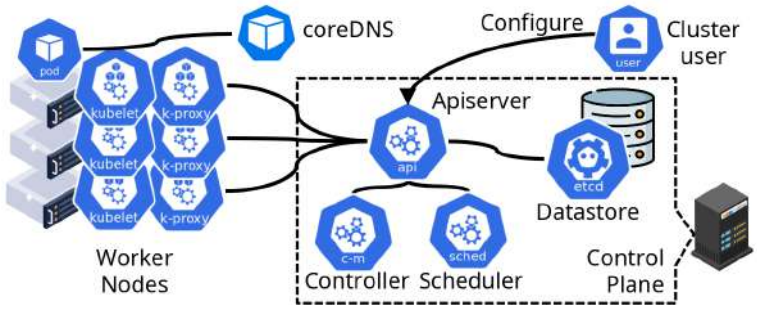


Figure 2.4. K8s architecture.

K8s components [4] K8s components run in Pods and include (see Figure 2.4): i) *Etcd*, a key-value store guaranteeing sequential consistency to the data of the state of the cluster; ii) *kube-apiserver* (hereafter, *Apiserver*), which exposes the API, allowing users to interact with the cluster and interconnecting the control plane components; iii) *kube-scheduler* (hereafter *Scheduler*), which assigns the Pods to Nodes based on resource requests, availability, and constraints; and iv) *kube-controller-manager* (hereafter, *Controller*), which reconciles the current cluster state with the desired one.

The Apiserver communicates with two components deployed on each Node: i) the *kube-proxy*, which maintains the virtual networks, connecting Pods and Services; and ii) the *kubelet*, which sends Node heartbeats and manages the lifecycle of the assigned Pods, restarting them if they are unhealthy. By extension, I consider as part of K8s also *coreDNS* and the network manager. They are, respectively, a service providing name translation to the Pods, and a DaemonSet managing networking between Nodes.

The components can be seen as stateless services that indirectly communicate by storing data on Etcd. They can be restarted at any time, fetching necessary data from Etcd. The Apiserver is the only component communicating with Etcd; the others send requests to Apiserver and observe state changes.

With regard to Figure 2.2, the Apiserver covers the functionalities of both the launcher and the user API, the Controller fulfills the duties of the state monitor, and the Scheduler implements the functionalities of the scheduler and of the resource manager. External monitoring frameworks are usually used to implement the accounting component.

The Controller is composed of multiple independent *controllers*, each in charge of managing the instances of one resource type. Each controller executes multiple concurrent *reconciliation loops* through *control loop threads* (CTs), and keeps resource instances to be processed in queues. When a CT is not busy, it immediately starts to process a resource instance upon arrival.

The Scheduler executes a workflow composed of sequential phases (*filtering*, *scoring*, *reserve*, *permit*, and *bind*). At each phase, the enabled *plugins* can execute some logic to contribute to the placement decision. Custom plugins can be integrated, and a custom scheduler can be used instead of the default one. During the filtering phase, the Scheduler can filter out the worker nodes

[4] This paragraph is taken from "Mutiny! How Does Kubernetes Fail, and What Can We Do About It?" ©IEEE 2024 (see author's publications section), with adaptations for this thesis.

that do not match with user-defined constraints (including taints and labels).

K8s example workflow [5]

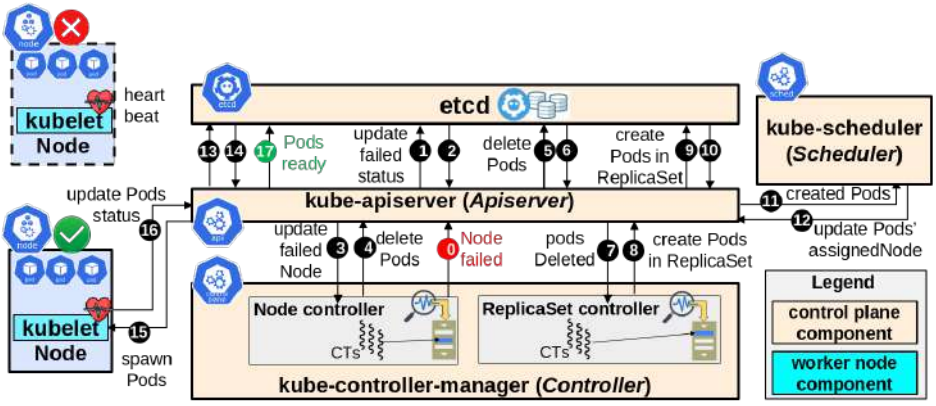


Figure 2.5. Kubernetes' operations workflow during a failover command. ©IEEE 2024.

This paragraph describes the flow of actions (represented in Figure 2.5) that happen upon a worker node failure in K8s. The failover workflow is chosen because it involves multiple K8s subsystems. An abstract version of the flow was represented in Figure 2.1.

Firstly, the user *declares* a desired steady state, i.e., the users specify *what* they want, not *how* it should be obtained. Once the steady state is reached, the Pods are running on the Nodes, which may fail. After multiple missing heartbeats from a worker node, the Controller issues a state change to the Apiserver, which dispatches it to Etcd to modify the persisted cluster state ①. The Controller is hence notified of the failed worker node (② and ③). At this point, the Controller handles the update regarding the failed worker node and orders to delete the Pods running on it (④,⑤). Later, the Controller handles the Pod deletion event (⑥, ⑦), e.g., the ReplicaSet controller reacts by creating the necessary number of Pods for the ReplicaSet involved (⑧, ⑨, and ⑩). The creation response ⑩ is forwarded to the Scheduler ⑪ that assigns the Pod to a worker node (⑫, ⑬, and ⑭). The required informa-

[5] This paragraph is taken from "Failover Timing Analysis in Orchestrating Container-based Critical Applications" ©IEEE 2024 (see author's publications section), with adaptations.

tion is forwarded to the Kubelet to spawn the Pod (15), and update its status ((16)(17)). Once the Pod is ready, the endpoint controller will include the Pod in the Service load balancing.

Similarly to the failover workflow, when a user creates a new Deployment, it is stored on Etcd. The Controller consequently creates a ReplicaSet for the Deployment. Afterward, the ReplicaSet controller notices the newly created ReplicaSet and creates the requested number of Pods. Next, the Scheduler notices the new Pods and schedules them.

K8s resiliency techniques [6] In cloud systems failures are expected. Therefore, orchestrators' design includes resiliency techniques to withstand errors and failures. Here, I provide a non-exhaustive list of techniques that K8s uses to increase its resiliency.

• Support for optional redundant control plane components on different nodes for availability: Etcd, Controller, and the Scheduler can work in a leader-follower scheme. Etcd uses the Raft consensus algorithm [25] and quorum reads among the replicas. The Controller and the Scheduler use leader election so that there is only one active replica at a time.

• Level-triggered reconciliation and stateless components: decisions are based on the current and desired states[7], and the messages exchanged between components are states, not commands. This guarantees easy recovery if restarts occur.

• Independent components to foster failure isolation: the control plane components are deployed as independent Pods. A failed component can be restarted without affecting the other components.

• Circuit breakers that prevent a repeatedly failing operation from overloading the system with a cascading effect. For example, when a Pod fails several consecutive times, it is restarted with increasing back-off delays.

• Timeouts in the communications between components to release resources in a timely manner if failures occur.

• *MaxUnavailability* to guarantee a minimum number of available replicas during rolling updates of Deployments, limiting the impact of incorrect

[6] This paragraph is taken from "Mutiny! How Does Kubernetes Fail, and What Can We Do About It?" ©IEEE 2024 (see author's publications section), with adaptations for this thesis.

[7] For further details, see `https://hackernoon.com/level-triggering-and-reconciliation-in-kubernetes-1f17fe30333d`

updates.

• *Server Side Apply* prevents unauthorized entities from modifying fields of data structures not owned by them.

• Full disruption mode that stops Pod evictions when all Nodes are reported as unhealthy, since the issue could be in the heartbeat reporting mechanism itself.

• Deletion of undecryptable resources (i.e., resources that cause errors when deserialized) to prevent failures when getting lists that contain them.

2.2 Dependability Assessment

2.2.1 Basic definitions

In this dissertation, I refer to the widely accepted definitions of dependability introduced in [26]. In particular, "a system failure is an event that occurs when the delivered service deviates from the correct service. [...] A failure is a transition from correct service to incorrect service [...]. A transition from incorrect service to correct service is service restoration. The time interval during which incorrect service is delivered is a service outage. An error is that part of the system state that may cause a subsequent failure: a failure occurs when an error reaches the service interface and alters the service. A fault is the adjudged or hypothesized cause of an error. A fault is active when it produces an error, otherwise it is dormant." [26]

The failures of a system can be analyzed to assess and improve the system's dependability, defined as "the ability to deliver a service that can justifiably be trusted". A field failure data analysis (FFDA) is performed when data related to failures from real operational systems are used to assess and evaluate the failures of a system or system category. An FFDA can range from a simple analysis of reports to a well-structured process consisting in *log tupling* and curve fitting following a spatial or temporal coalescence of logs.

In the latter case, quantitative dependability metrics can be derived for the system, including mean time to failure (MTTF), mean time to repair (MTTR), and mean time between failures (MTBF). Those metrics can be used to get, for example, an estimation of the availability (i.e. the "readiness for correct service" [26]), reliability (i.e, the "continuity of correct service" [26]) of the system. I refer to [26] for other basic dependability definitions. An impactful

example of FFDA can be found in [27], where authors analyze the failures of the Bluewaters supercomputer. In order to derive those metrics, the numeric values obtained from the FFDA can be used to populate system models with or without state space, like Markov models and Petri nets (state-space models), or fault trees and reliability block diagrams (non-state-space models). Those analytical models have been widely used in research to assess the dependability of cloud and edge computing environments [28, 29, 30, 31].

2.2.2 Fault injection

Some errors or failures may seldom manifest but represent a huge threat to a system. In these cases, an FFDA may lack essential information about the system's *resilience*, i.e., the ability to provide a correct service despite faults and errors. In this perspective, "fault injection is a practical approach for achieving this confidence, by deliberately introducing faults in a system. This approach can assess fault tolerance properties, such as robustness, in the presence of faults." [32]. Fault injection was first introduced for hardware testing, simulating short circuits and similar hardware faults to understand the system behavior [33]. Later on, it was introduced for software systems.

The results of a *fault injection campaign* (i.e., the set of fault injection experiments) can be used for both a qualitative or quantitative assessment of the system's dependability.

Quantitative results are meaningful if and only if the characteristics of the injected faults are representative of the faults that the system experiences during operation. These characteristics include the *fault model* along with the probabilistic distribution. The fault model describes the set of faults that are injected to mimic the fault experienced in operation. The fault model requires the definition of *what* to inject (i.e., which kind of fault), *when* to inject (i.e., the timing of the injection), and *where* to inject (i.e., the system part targeted by the injection).

A fault injection setup includes a *target system*, i.e., a system under analysis, a *load generator*, which triggers system functionalities during a fault injection experiment, a *monitor*, which performs a check of the system status and correctness, and an *injector*, which introduces a fault in the system. The load generated by the load generator is referred to as workload, while the faults injected are referred to as faultload. A fault is injected by tampering with the structure or the system state or with the environment in which it executes.

The authors of [32] give an overview of software fault-injection techniques, mainly dividing injections into data errors, interface errors, and code changes. In this sense, the injected faults can both mimic the faults in the software (e.g., bug), or the effect of another fault (e.g., an error caused by the activation and propagation of a hardware, software, or environment fault). It is worth reporting that the authors of [34] show that injections at the component's interface are not equivalent to injections in the code.

In the survey ([32]), the authors identify three applications of software fault injection: improvement of fault tolerance algorithms and mechanisms, forecasting of dependability measures, and dependability benchmarking.

The improvement of fault tolerance algorithms and mechanisms qualitatively evaluates the adequacy of fault tolerance to its specifications. Issues related to fault handling paths can be corrected or improved to prevent error propagation and failure. In particular, fault injection aims to trigger assertions, redundant logic, exception handlers, and checkpoint/rollback mechanisms, which detect and correct erroneous states. A particular application of fault injection targeting the improvement of fault tolerance is Chaos engineering [35], introduced by Netflix. Chaos engineering automatically, randomly, and deliberately introduces common faults/errors through injections in production systems to find and improve dependability bottlenecks. The key idea of Chaos engineering is that injections must be done in production because testing workloads are never fully representative of production workloads.

The forecasting of dependability measures quantitatively evaluates the fault tolerance properties of a system. System metrics obtained through fault injection can be fed to an analytical model to get probabilistic system dependability measures. These metrics include the coverage factors of fault tolerance mechanisms and their latency, the probability of specific failure modes, fail-stop behavior, and catastrophic failures.

A dependability benchmark compares the dependability of different computer components or systems. The precise procedures and rules aim to give meaningful, comparable, and reproducible results, taking into account the stakeholder interests, including product manufacturers and users.

While this primer aim is to introduce basic definitions and required knowledge for the reader of this dissertation, I refer to the survey [32] for a detailed description of the concepts introduced here.

2.3 Real-time Systems

2.3.1 Basic definitions

Real-time systems are systems in which "the correctness of the system depends not only on the logical results of computation, but also on the time at which the results are produced" [36]. The importance of timely producing correct results is typically tied to the external environment that a real-time system interacts with. Therefore, in real-time systems, predictability is usually prioritized over average performance: a very fast system that occasionally produces results too late is worse than a slower system that is always on time. When a real-time system does not produce the results within the expected and acceptable time, it misses its *deadline*.

Real-time systems can be divided into *hard real-time systems* and *soft real-time systems*, based on the severity of the consequences of a deadline miss. In hard real-time systems, even a single deadline miss is considered a system failure and can lead to catastrophic consequences (e.g., flight control systems, chemical plants, automotive applications). In soft real-time systems, deadline misses degrade the system performance and quality of service, but they do not have catastrophic consequences (e.g., video streaming, online gaming).

However, this categorization only grasps the extremes of a more shaded spectrum of real-time systems. Indeed, a deadline miss can have troublesome consequences, which are anyway less severe than catastrophic environmental consequences or deaths, like economic losses. Moreover, a system can be designed to tolerate some deadlines, e.g. *k out of n* systems, in which at least *k* out of *n* consecutive deadlines must be guaranteed.

Real-time tasks A real-time system is composed of real-time tasks. A real-time task is a computation executed in a sequential fashion. Real-time tasks are commonly characterized by the following parameters (I refer to [37] for other parameters):

• *arrival time (a)*: the time at which a task becomes ready for execution.

• *computation time (C)*: the amount of time required by a task to complete its execution without interruption. This is often referred to as the worst case execution time (WCET), which is the maximum time a task might take to execute in worst-case conditions. Since the complexity of modern hardware platforms often prevents a meaningful WCET analysis, in literature the notion

of probabilistic worst case execution time (pWCET) has emerged [38].
• *deadline (D)*: the time by which a task must complete its execution. The task is said to be *late* if it completes its execution after its deadline.
• *finish time (f)*: the time at which the task completes its execution.
• *response time (R)*: the difference between the finish time and arrival time.

Figure 2.6. Parameters of a real-time task.

A *task model* defines the assumptions about the behavior of the tasks and defines the complete set of task parameters. For example, tasks can be characterized by a period T (i.e., the time between two consecutive task releases), or a minimum inter-arrival time, if tasks are aperiodic.

Scheduling algorithms At any time, tasks are selected to execute based on the rules defined by a scheduler, which takes into account the task parameters. For example, a static priority scheduler requires the tasks to be characterized by a scheduling priority P. Conversely, dynamic priority schedulers define the priority of a task dynamically. The most spread out dynamic priority scheduler is earliest deadline first (EDF), which guarantees that at any time the unfinished task with the closest future deadline is running.

A scheduling algorithm can preempt an already running task and schedule it at a later moment. Hence, the scheduling algorithm determines the tasks' worst case response time (WCRT) based on all the possible tasks interleaving. The WCRT can be computed through a *response time analysis*. When a probabilistic WCET is used, probabilistic analysis have been proposed to compute the probabilistic worst case response time (pWCRT). The probabilistic schedulability analyses are surveyed in [39].

Each scheduling algorithm is characterized by a *schedulability test* that can be used to validate the feasibility of a task set, i.e., to determine *a priori* if all tasks are guaranteed to finish within their deadlines. A particular class of schedulability tests is the *utilization-based* test, which determines the feasibility of a task set by comparing the total CPU utilization with an algorithm-

dependent threshold. The total CPU utilization is defined as $U = \sum_i U_i = \sum_i C_i/T_i$, where U_i is the task i utilization, and C_i and T_i are respectively its computation time and period. EDF allows reaching unitary (i.e., 100%) processor utilization for a periodic task set in uniprocessor systems. This means that the schedulability test for EDF in such conditions is $U = \sum_i U_i < 1$.

2.3.2 Mixed-criticality task model

A real-time system might include tasks with different *criticality* levels that run on the same hardware platform. The criticality is defined as the designation of the level of assurance against failure [1].

For example, the mission-critical systems targeted in this dissertation usually have some hard real-time applications (e.g., industrial and vehicle controllers) and some soft real-time applications (e.g., accounting, infotainment).

A model for mixed-criticality systems was first introduced by Vestal in [40]. This model extends the real-time task model, characterizing each task with a vector of WCETs $C(L)$, one for each criticality level L, with $C(L_1) > C(L_2)$ if $L_1 > L_2$. The system starts in the lowest criticality mode. When a job exceeds its WCET (or activates more frequently than planned), a mode switch to a higher-criticality mode occurs. In higher-criticality modes, only the timely execution of high-critical tasks is guaranteed, while less critical tasks may be discarded or delayed.

The mixed-criticality model allows enhanced resource utilization under normal conditions while guaranteeing that the critical tasks remain correct despite timing anomalies or unexpected system stress.

I refer to [1] for a complete review of mixed-criticality systems.

2.3.3 Scheduling for virtualized environments

Virtualization allows co-locating independent components/applications (with different criticality levels) on the same hardware platform to consolidate resources, reduce costs, and improve scalability.

In this perspective, a mixed-criticality system is not necessarily a *closed* system (i.e., "a system in which detailed attributes of all real-time applications are known" [41]) with tasks at different criticality levels. Conversely, it can also be an *open* system, i.e. a system in which applications with different criticality levels may be developed and validated independently [41].

In [41], the authors show how two-level hierarchical schedulers can be used to design an open system architecture. At the OS-level, the OS allocates processor time to the servers, sets their deadlines, and schedules the servers according to the EDF algorithm. At the application level, each application has its own scheduler to schedule the application tasks, which anyway must respect determined assumptions.

In particular, the servers used were the total bandwidth server (TBS) or the more robust constant bandwidth server (CBS) [37]. These algorithms are *reservation-based* algorithms, i.e., algorithms in which each task (or set of tasks) has a reserved CPU bandwidth (i.e. utilization). Both algorithms define specific rules to assign a deadline to tasks, which are then scheduled with EDF. The CBS guarantees to a task or a set of tasks the same (theoretical) behavior as if they were alone running on a processor of speed $1/U$, where U is the server bandwidth.

Besides dynamic priority servers, there are also static priority servers, which are simpler to implement but do not benefit from the theoretical properties of dynamic priority schedulers. A static priority server that I will use in this dissertation is the deferrable server (DS) [37], which preserves its unused budget until the end of its period, making the schedulability analysis more complex. The schedulability test for hierarchical fixed priority preemptive scheduling for the DS can be found in [42].

Subsequently, building upon the concepts of open systems, in [43] the compositional schedulability framework was proposed to address the problem of the component demand abstraction. It provided a more flexible schedulability analysis for open systems, enabling more than two scheduling levels and overcoming the problem that the "parent model's scheduler was limited to the EDF scheduler that needs to interact with the child model's scheduler for the knowledge of the task-level deadline information" [44]. In other words, a component or application can provide a much simpler *scheduling interface* that guarantees the schedulability of its tasks, while hiding their modeling details. For example, a set of partitioned (i.e., tasks cannot migrate between cores) tasks scheduled with EDF can be transformed into a single task model or bounded-delay model. Those models can be used at OS-level scheduling.

Although the described scheduling algorithms enable the sharing of a CPU by different applications, some difficulties emerge when it comes to the actual implementation of systems relying on them. In fact, some hardware resources

are inherently complex to isolate in a predictable manner, like caches, memory controllers, I/O devices, etc. Therefore, when applications have strict non-functional requirements, complex scheduling policies are set aside in favor of simpler yet more predictable ones.

In this perspective, the simplest scheduling policy that can be used in virtualized environments is the *static partitioning*: different applications have reserved resources that are not shared. For example, each application can own a dedicated CPU core. The static partitioning policy is fostered by the diffusion of multi-core platforms, in which more efficient resource usage is possible but also very challenging from both a theoretical and practical perspective. The static partitioning scheduling simplifies the analysis and implementation of the system, providing strict non-functional guarantees.

2.3.4 Real-time networking

In real-time systems, timing properties must be guaranteed not only when tasks perform computations on a node, but also when they exchange messages over a network.

To this aim, a plethora of networking protocols were proposed over the years to support deterministic networking. Those protocols have had a wide application, especially in industrial environments. This section focuses only on the ones mentioned in this dissertation.

The simplest example are time division multiple access (TDMA) networks [45], in which nodes access a shared medium (wired or wireless) in predetermined time slots within periodic time frames to prevent conflicts. A TDMA network requires the transmitters to have a common time reference, which can be achieved through clock synchronization or electing a TDMA regulator that sends *sync* packets at the beginning of each time frame. RTnet is an example networking stack that supports TDMA over standard ethernet hardware [46], and it is shipped with Xenomai[8].

A popular protocol used in vehicles is controller area network (CAN): a non-preemptive shared bus with fixed priority packets. It has low bit rates, but it is simple and resilient to noise, representing a good fit for low-end

[8]Xenomai is a popular real-time co-kernel that runs besides of Linux. More information available at `https://xenomai.org/`. In a co-kernel system, Linux can always be preempted and runs only when no process of the co-kernel is ready to run. The co-kernel intercepts interrupts directed to Linux and injects them only when no real-time process is running.

microcontrollers. The prioritization is determined by bitwise arbitration in the access to the bus: when multiple devices transmit at the same time, the value of the bus is determined by the highest priority message.

A modern solution that is spreading in industrial environments is time sensitive networking (TSN), which is a set of protocols for low-latency networking. TSN is designed to support soft real-time industrial traffic, allowing the transmission of real-time and best-effort traffic on Ethernet infrastructures through deterministic packet scheduling, synchronization, and other techniques. TSN relies on clock synchronization protocols to have a common time reference and supports multiple classes of communication flows. Each traffic class has its time slot, which repeats cyclically, like in TDMA protocols. TSN also supports frame preemption: high-priority traffic can preempt lower-priority traffic.

2.4 Use Cases and Requirements

This section provides a summary of use cases and requirements gathered from research work that relies on containers, orchestrators, and virtualization in general for the scenarios introduced in §1.1.

Industry 4.0 Industry 4.0 foresees the integration between information technologies and operational technologies to realize visions like reconfigurable manufacturing. Reconfigurable manufacturing "supports the rapid addition, removal, or modification of process controls, functions, and/or operations, through reconfigurable hardware and software, to scale production capability and capacity." [4]. The vision of reconfigurable manufacturing was first introduced by Koren in 1995 [47].

In this perspective, recent efforts were directed to transform traditional hardware industrial equipment into virtualized and cloud-native components. For example, a traditional hardware-based programmable logic controller (PLC), which is an essential building block for modern industries, falls short in flexibility, interoperability, scaling, and computational resources required by, for example, complex AI algorithms [48]. Therefore, several works have dealt with the transformation of hardware-based PLCs into virtualized [49] and/or cloud-native components [48, 50].

Cloud-native PLCs allow efficient communications with popular IoT wire-

less protocols, offloading [51], flexible deployment, and live migration of controllers through the integration into cloud orchestrators [48]. In this context, the timing requirements are strict. In some cases, response times, signal switching times [49], and live migrations [50] must be in the order of tens of milliseconds to respect the cycle times. In other cases, redundant distributed control systems must present failover times around 500*ms* [52].

In [49], the authors showed that a soft PLC deployed on a real-time server (using also a hypervisor) connected to a real-time fieldbus can achieve the target determinism and response times. In [50], the authors showed that using ad hoc K8s components, the migration times of controllers deployed as Pods can respect the requirements.

The use of cloud-native components naturally enables the possibility of offloading tasks from resource-constrained devices to leverage computing resources of edge servers, which are more abundant [51]. However, to accomplish this vision, cloud-native components must rely on deterministic networking to communicate with the field. This problem was faced in [53, 54, 55], where TSN was integrated in K8s Pods.

When offloading to a remote node is not possible, more complex hardware boards designed for mixed-criticality systems can be used. For example, in [56], the authors used a platform equipped with both microcontrollers and microprocessors to boost the control algorithm of a motor servomechanism. The authors relied on Jailhouse as a partitioning hypervisor (see §7.3 for additional details on partitioning hypervisors).

Cloud-native components provide further advantages during the system design phase, allowing interoperability and easy composition of modular components, as shown in [57]. The authors divided services into four categories: control, mediators, data management, and interoperability.

The composition of real-time services was already explored by the past research trend of real-time service oriented architecture (SOA) [58, 59, 60]. For example, in [60] an architecture for real-time SOA in industrial environments was presented, highlighting the importance of real-time communication channels and resource reservation to meet the specified quality of service (QoS). Similarly, in [59] the importance of having a reserved communication bandwidth to perform real-time service discovery, composition, and deployment was highlighted.

5G networking In [61] authors analyze the scaling problem of Open5GS[9], showing that it is not designed to be cloud native and run in containers. Hence, they propose a more salable design of the 5G control plane, with stateless workers implementing all the required virtual network function (VNF) and communicating with a central database. In this way, the authors solve the problem of stateful connection management, which undermines scalability. Nephio[10] is also a promising product in the first development stages. Nehpio delivers carrier-grade, simple, open, Kubernetes-based cloud native intent automation of the network management.

The work in [62] focuses on standards, providing an overview of the evolution of ETSI network function virtualization (NFV) Management and Orchestration standards to integrate new technologies that became a standard *de facto*, including containers and orchestrators. Other works, like the one in [63], rely on orchestrators to manage a scalable SDN control plane.

Multiple works used networking and VNF as use cases. For example, the work in [64] (and another close work from the same authors [65]) considers virtual baseband units, while the authors of [66] consider generic VNFs.

In the networking context, network functions may be frequently scaled to cope with highly variable request rates. Even in those cases, the functions must be correctly scaled and the SLOs including packet transmission and processing times must be met. Indeed, a time-sensitive and critical application may on the network.

Transports In [9] the authors provide a survey of technologies, visions, and architectures of software-defined vehicle paradigm. Indeed, the authors report that car manufacturers rely increasingly more on software components to cope with an increased amount of information to elaborate and transmit. Therefore, microcontroller-based electronic control unit (ECU) are being replaced by complex computing platforms hosting fully-fledged operating systems and different networking interfaces. The use cases range from autonomous driving to advanced infotainment services, vehicle-to-vehicle communication, and vehicle-to-infrastructure communications.

Managing all the software components is becoming increasingly complex, and new solutions that take advantage of cloud-native architectures are

[9] Available at `https://github.com/open5gs/open5gs`.
[10] More information available at `https://nephio.org/about/`

emerging. For example, SOAFEE [11] is an industry-led collaboration that introduces an architecture that includes cloud-native and mixed-criticality applications for automotive. Mixed-criticality systems are leveraged in cases like [67], where the authors use virtualization to guarantee fault tolerance of a vehicle without redundant hardware. They divide the ECUs in SOA ECU (i.e. microprocessor-based with support for virtual memory) and signal ECU (i.e. microcontroller-based with a lot of peripherals), and show that in case of failure, a task running on a signal ECU can be effectively migrated to a SOA ECU managed by a hypervisor (e.g., Jailhouse).

In other cases, research works leveraged container orchestration systems. For example, in [68] a fleet of vehicles together with road-side units is seen as a Kubernetes cluster in which services are placed while keeping into account the expected permanence of the vehicle in the cluster. In [69], the authors use K8s to manage non-safety critical services running on the ECUs. The authors introduce AXIL, i.e., a metric to prioritize services and optimize the user experience by dynamically selecting the services' functioning modes based on resource availability. Finally, in [70], the authors state that "limitations in real-time scheduling and functional safety describe the most urgent challenges towards an adoption" of orchestration systems for automotive.

In these settings, hard real-time components coexist with soft or firm real-time ones, and old-styled microcontroller-based ECUs coexist with more powerful hardware platforms supporting virtualization. Hardware resources and applications must be managed with a global vision, with jobs that may be triggered at any time by external events (e.g., emergency braking), other vehicles, users onboard, or failure handling.

[11] `https://www.soafee.io/`

Chapter 3

A Model for Mixed-criticality Container Orchestration

We have to focus on our vision, and we have to realize which is the first step to take from where we are to reach that vision.

Lelio Di Martino

This chapter introduces a system model for mixed-criticality container orchestration. The model aims to catch the characteristics of the target industrial fog/edge computing infrastructure and applications with diverse non-functional requirements. The system model is the basis for the design of the mixed-criticality orchestrator, which is presented in the next chapter.

Section 3.1 introduces the model, focusing on the novel concepts of criticality and assurance that characterize the services and the worker nodes.

Next, section3.2 introduces novel orchestration commands enabled by the model and based upon the concept of *diversity*.

Then, section 3.3 models through non-state-space techniques the failure probability of services relying on the introduced concept of diverse replication. It shows how an orchestrator can use those concepts to decide the number of replicas, their criticality, and the replication schemes to meet a desired failure probability.

Finally, section 3.4 explains through state-space models the role that the orchestrator plays concerning services' failures and scaling, highlighting why it is paramount to study the failures and the timeliness of orchestrators.

3.1 System Model

This section first delves into the model of services and applications (in §3.1.1), and then presents the computing infrastructure model, including heterogeneous worker nodes, networks, and industrial devices (in §3.1.2). In this regard, I recall that in the background chapter (§2.1), a *service* was defined as "a set of one or multiple deployed pods responding to clients' requests", where a pod was defined as "the minimum deployable unit managed by an orchestrator, and it consists of a set of related containers."

The main novelties of the model include: i) the concept of criticality introduced for *applications*, *services*, and *pods*; ii) the concept of assurance provided by the worker nodes for each of the shared resources; iii) topology-awareness with industrial network requirements; and iv) a fine-grained real-time interface for pods that includes a per-task WCET that is a function of both the worker node and the criticality level.

In the following paragraphs, I formalize the abstractions for each resource type of the orchestration system. Table 3.1 summarizes all the symbols I used in describing the abstractions.

Part of this section is taken from "Criticality-aware monitoring and orchestration for containerized industry 4.0 environments" ©ACM 2024 (see author's publications section), with possible adaptations for this thesis.

3.1.1 Applications and their criticality

Application A cloud application A_i, $i \in [0, A[$, consists of a DAG of services, more formally represented with a tuple $A_i = (\Lambda_i, E_i, D_i, P_i, C_i, AP_i)$, where $\Lambda_i \subseteq \Lambda$ is a subset of all the services Λ in the environment, $E_i \subseteq \Lambda_i x \Lambda_i$ is a set of directed edges that represent the service-to-service communications, D_i is the end-to-end deadline that the application must respect, and P_i is the period or minimum inter-arrival time of the application, i.e. the time between two consecutive application triggers.

I characterize each application with an *application criticality level*, i.e., $C_i \in \{NO, LOW, HI\}$, which is the required designation of the level of assurance against failures for the application. The more severe the consequences of an application failure for the system are, the more critical the application is. The application criticality level is related to service criticality levels through an *application aggregation policy* AP_i. The application aggregation policy spec-

ifies the criticality level of the services belonging to the application to meet the application criticality level. For example, an application can specify that the totality or only a part of the services belonging to the application must be of the same or higher criticality level of the application.

Service A service $\sigma_j \in \Lambda$ with $j \in [0, S[$ is a self-contained, reusable, and loosely coupled software component that performs a specific business function or task. A service receives some data in input, processes it, and then generates some output, which is sent to the user or to another service, as specified by the edges E_i. More formally, a service can be described by the tuple $\sigma_j = (f_j^{I/O}(), SLO_j, \vec{J}_j, f_j^D(\vec{J}_j), C_j, AP_j)$. $f_j^{I/O}()$ is the data elaboration function that relates input and output, and the service SLO_j is an agreed-upon performance target over a period of time. The service SLO can include an implicit deadline imposed by the deadline of the applications to which the service belongs. The service σ_j includes a vector $\vec{J}_j$ of pods that actually implement its functionalities, and a distribution function $f_j^D(\vec{J}_j)$, which manages the load balancing and/or redundancy of the inputs to the pods. A service σ_j can belong to multiple applications at the same time.

In order to have a fine-grained and flexible system model, I characterize each service with a *service criticality level*, C_j which can be defined as the designation of the level of assurance against a service SLO violation. Violations of critical SLOs must be avoided despite errors and/or overloads.

Similarly to the application criticality level, the service criticality level is related to the pod criticality level through a *service aggregation policy* AP_j. The service aggregation policy specifies the criticality level of the pods belonging to the service to meet the service criticality level. For example, a service can specify that the totality or only a part of the pods belonging to it must be of the same or higher criticality level of the application.

Pod A pod J_l, with $l \in [0, J[$, is a set of related containers, each of which includes one or multiple running tasks (defined later). A pod is the minimum scheduling unit for the orchestrator, and it has a set of allotted resources. Since resources are allotted to the whole pod, I do not care about how tasks and resources are split among containers in the pod.

More formally, $J_l = (TK_l, \vec{BR}_l, AR_l, NW_l, RT_l, C_l, AP_l)$ is the tuple that describes the pod J_l. TK_l is the set of tasks belonging to the pod, $\vec{BR}_l$ is the

vector of the basic resources required to carry out the pod activities (i.e., CPU, memory, and disk request). AR_l are the additional resources the pod may optionally require, like accelerators, sensors, actuators, field gate programmable array (FPGA) tiles, or guaranteed fractions of inherently shared resources (e.g., cache colors or memory bandwidth). $\vec{BR}$ and AR have the same structure as the worker node resources (described later). A pod can also have a set of communication requirements $NW_l = \{NW_{l,0}, ..., NW_{l,Q-1}\}$ on one or more networks. Expressing the network requirements is strictly related to the network type. For example, a pod could ask for a maximum round-trip time on a best-effort network, or it could require to be placed on a worker node attached to the same CAN bus of a specific sensor, or it could require a TDMA time slot.

A pod can optionally have real-time constraints ($RT_l \in \{0, 1\}$). A real-time pod can only be placed on worker nodes configured for real-time and that support real-time scheduling.

I further introduce the notion of *pod criticality* C_l, which is the designation of the level of assurance against hardware errors and hardware interference caused by the tasks co-located on the same worker node. The real-time requirements of the pod are independent of its criticality: a pod can be real-time but with low criticality (i.e., soft real-time), or it can have high criticality but with no real-time requirements (i.e., only the correctness of value matters, requiring certified hardware to tolerate faults).

As specified by the service aggregation policy, different replicas of a pod can be deployed, each of them having a different criticality level. For example the service distribution function can implement a triple modular redundancy (TMR) between a leader replica with a higher criticality level, and other backup replicas with lower criticality.

The pod criticality level must be backed by suitable guarantees of the worker node where the pod is placed: a pod with high criticality must be deployed on a worker node with a high *assurance level*. Conversely, less critical pods can take advantage of dynamic environments, increasing resource utilization at the cost of fewer guarantees.

The assurance level of a worker node can be different for different resources requested by a pod, as the assurance level mainly depends on algorithms and techniques implemented in hardware or the virtualization layer. For example, hypervisor-based containers may benefit from cache coloring,

etc. How the assurance of the different resources must match the pod criticality is specified in a *pod aggregation policy* AP_l: a pod can require a minimum level of assurance on each of the used resources, or it can require a minimum aggregated assurance score. For example, a pod can specify a minimum score computed as the weighted sum of the assurance of each resource and some parameters specified in the pod AP ($A_l = w_0 * A_{l,1} + ... + w_r * A_{l,r} \geq Threshold$).

Task A task is defined as TK_m^l with $l \in [0, J[$, $m \in [0, M^j[$, and M^j being the number of tasks composing the pod J_l. In the case of a pod with real-time requirements, its real-time tasks are described by a *real-time scheduling interface* $TK_m^l = (WCET_m^l(n, C_l), T_m^l, P_m^l)$, where T is the period (or minimum inter-arrival time for sporadic tasks) and P the (optional) task priority. The priority is only required by static priority scheduling algorithms.

I argue that the pod real-time scheduling interface $(budget, period)$ used in previous studies [71, 72, 73] only works for soft real-time tasks. Although compositional schedulability analysis can be used to expose a resource model as a scheduling interface of a set of tasks, the parameters of the resource model heavily depend on the tasks' WCET. In turn, the WCET drastically changes on different platforms, especially in the heterogeneous environment considered.

Moreover, hierarchical scheduling and compositional scheduling require some assumptions for the tasks running inside a container, which may not hold if assuming a plethora of different OSes, hypervisors, and applications. For example, fixed-priority scheduling must be used by the guest OS (or container) to use the compositional schedulability framework. In addition, partitioned scheduling must be used when the schedulers at the host and guest level cannot communicate [74].

Therefore, to be as generic as possible, in my model, each task has its scheduling parameters that depend on the hardware. For example, assuming a periodic task model, each task TK is described by a period T and a WCET that is a function of the node and criticality level of the pod (i.e., $WCET_m^l(n, C_l)$). Similarly to the approach of the Vestal model [40], the execution times depend on the criticality level: for high-criticality pods, the WCET should be precisely computed for a set of specific hardware platforms, and only these platforms should be taken into account during the placement phase. Since precisely estimating the WCET on every worker node of the

environment is not feasible for every application, low or non-critical pods could leverage cheaper methods to estimate WCET, at the cost of reduced confidence. For example, the WCET can be expressed as a reference WCET multiplied by a characteristic factor of the platform, or the task parameters can be tuned at runtime, like in [75]. Defining the precise WCET computation methods is beyond the scope of this work, and I generally assume a WCET that is a function of both the criticality level and the worker node considered.

3.1.2 Computing infrastructure and its assurance

Worker Nodes I assume a cluster composed of N heterogeneous worker nodes. Each worker node is described by the tuple $WN_n = (\vec{BR}_n, AR_n, RT_n, \vec{A}_n)$ $n \in [0, N[$, where $\vec{BR}$ is a vector of *basic hardware resources*, AR_n is a set of *additional heterogeneous resources*, RT is a boolean representing the real-time capability of the worker node, and $\vec{A}_n$ is a vector of *assurance levels* for the worker node resources. $\vec{BR}_n = [CPU_n, Disk_n, Mem_n]$ is the vector of basic resources, i.e. CPU, disk, and memory. $AR_n = \{\vec{S}_n, \vec{ACT}_n, \vec{Acc}_n, \vec{FPGA}_n, CRB_n\}$ is the set of additional resources. The elements of the set are not fixed and depend on the worker nodes. For example, the set may include vectors of sensors and actuators that can be accessed from the node (i.e., $\vec{S}_i$ and $\vec{ACT}_i$), accelerators and/or hardware features (e.g., FPGA tiles), and common resources' fractions (i.e., CRB_n) like in [76] (e.g., cache colors, memory bandwidth).

$\vec{BR}$ values represent the absolute available quantity of the resource, for example, the available CPU utilization, defined as $U = C/T$ (recall §2.3). Depending on the scheduling algorithms used, the CPU utilization can be a global property or a property of each CPU core. The values in AR_i can either contain integer values that represent how many resources are available on the node, or fraction amounts like for $\vec{BR}$.

$\vec{A}_n$ is the *assurance level* of WN_n. It is conceived as a vector of assurance levels of the individual resources: e.g., $\vec{A}_n = [A_{n,CPU}, A_{n,Disk}, A_{n,Mem}]$ is the assurance level of basic resources. The vector $\vec{A}_n$ also includes the additional resources, depending on their availability. Each element $A_{n,r}$ of the vector represents the degree of isolation and dependability that WN_n can provide for the resource r, based on its hardware/software characteristics. For example, a memory management system supporting cache coloring, bandwidth allocation, and error correction codes has a greater assurance compared to a memory system that does not have those features.

Each $A_{n,r}$ can be modeled as $A_{n,r} = f(\alpha_{n,r}, \beta_{n,r}, \gamma_{n,r}(t))$ where $\alpha_{n,r}$ and $\beta_{n,r}$ are the assurance level scores associated with the presence of hardware and software assurance mechanisms, respectively. $\gamma_{n,r}(t)$ is a time-dependent function that represents the assurance level determined by the system state at time t, i.e., how current co-located loads affect the isolation of the real-time critical task, as seen in the previous section. Hence, $\gamma_{n,r}(t)$ enables a temporary reduction in the assurance of a resource when co-located workloads threaten the guarantees of a more pod. For example, component aging and anomalous or malicious applications can increase the bit-flip probability [77].

Network topology The worker nodes are connected by one or multiple networks NW_q, $q \in [0, Q[$, with different purposes, requirements, and features. For example, NW_1 can be a *best-effort network* connected to the Internet, NW_2 a separate monitoring network, and NW_3 another network with real-time guarantees, such as TSN, CAN bus, or a *slice* of a wireless network.

NW_q is a graph described by (V_q, E_q). $V_q = \{WN_x, ..., WN_y\}$ is the set of nodes belonging to the network, while E_q represents the links between nodes. The links are characterized by a communication delay E_q^d and a capacity E_q^c, defining, in general, a matrix $M_q : V_q^2 \rightarrow (E_q^d, E_q^c)$.

Each network has its current status (e.g., allotted time slots) and (optionally) schedulability algorithm to guarantee real-time traffic. However, those two aspects are strictly related to the type of network, hence I leave them undefined.

Table 3.1. Summary of symbols for the system model.

Symbol	Meaning
$i \in [0, A[$	Index of cloud application
$A_i = (\Lambda_i, E_i, D_i, P_i, C_i, AP_i)$	Cloud application i
Λ	All the services in the environment
$\Lambda_i \subseteq \Lambda$	Subset of services that compose application i
$E_i \subseteq \Lambda_i x \Lambda_i$	Communication matrix of the services of application i
D_i	End-to-end deadline of application i
P_i	Period or minimum inter-arrival time of application i
$C_i \in \{NO, LOW, HI\}$	Application i criticality level
AP_i	Application aggregation policy
$j \in [0, S[$	Index of service
$\sigma_j = (f_j^{I/O}(), SLO_j, \vec{J}_j, f_j^D(\vec{J}_j), C_j, AP_j)$	Service j
$f_j^{I/O}()$	Input/Output elaboration function of service j
SLO_j	Performance target over a period of time of service j
$\vec{J}_j = [J_{j0}, J_{j1}, ..., J_{jrep}]$	Pods of service j implementing its functionalities
$f_j^D(\vec{J}_j)$	Distribution function of service j
$C_j \in \{NO, LOW, HI\}$	Service criticality level
AP_j	Service aggregation policy
$l \in [0, J[$	Index of pod
$J_l = (TK_l, \vec{BR}_l, AR_l, NW_l, RT_l, C_l, AP_l)$	Pod l
$\vec{BR}_l, AR_l$	Basic and additional resources required by pod l
$NW_l = \{NW_{l,0}, ..., NW_{l,Q-1}\}$	Network requirements of pod l
$RT_l \in \{0, 1\}$	Real-time requirement of pod l
$C_l \in \{NO, LOW, HI\}$	Criticality of pod l
AP_l	Pod l aggregation policy
$m \in [0, M^j[$	Index of tasks of pod j
$TK_m^l = (WCET_m^l(n, C_l), T_m^l, P_m^l)$	Task m of pod l
$WCET_m^l(n, C_l)$	WCET of task m, pod l on node n, at criticality C_l
T_m^l, P_m^l	Period and priority of task m of pod l
$n \in [0, N[$	Index of node
$WN_n = (\vec{BR}_n, AR_n, RT_n, \vec{A}_n)$	Worker node n
$\vec{BR}_n = [CPU_n, Disk_n, Mem_n]$	Basic resources of node n
$AR_n = \{\vec{S}_n, \vec{ACT}_n, \vec{Acc}_n, \vec{FPGA}_n, CRB_n\}$	Additional resources of node n
$\vec{S}_n, \vec{ACT}_n$	Sensors and actuators accessible from node n
$\vec{Acc}_n, \vec{FPGA}_n$	Available accelerators and FPGA slices on node n
$RT_n \in \{0, 1\}$	Real-time capable node n
$A_{n,r} = f(\alpha_{n,r}, \beta_{n,r}, \gamma_{n,r}(t))$	Assurance level of node n for the resource r
$q \in [0, Q[$	Index of network
$NW_q = (V_q, E_q)$	Network q described as graph
$M_q : V_q^2 \rightarrow (E_q^d, E_q^c)$	Delay and capacity matrix of network q

3.2 Mixed-criticality Orchestration

3.2.1 Diversified containers

In my model, pods belonging to the same service can have different criticality levels. With such abstraction, I want to model and leverage the *diversity* of pods' implementations. In this subsection, I clarify this concept.

I defined in §2.1.3 a platform as "the hardware/software determining the container image to download". Hence, each platform requires a different (binary) implementation of the same application. For example, even recompiling the same source code for Arm64 and x86-64 architectures, generates two different binaries. Besides different architectures, industrial environments also feature heterogeneous processing units (real-time or tensor processing units), platforms (e.g., different vendors), and software stacks (e.g., OSes, hypervisors, diversified application versions).

Currently, such heterogeneity is seen as an obstacle for cloud technologies, since the same application must be recompiled, built, or rewritten for a different OS, ISA, or containerization technology.

In my vision, an orchestrator can leverage such heterogeneity as a precious resource to enable automated replication schemes (e.g., spare redundancy) between diversified containers and improve dependability. The inherent diversity (provided by OSes, architectures, virtualization technologies, etc.) provides a cheap way to reduce common cause failures [78].

Container runtimes that take advantage of diversified software/hardware stacks (e.g., sandboxed containers[1]) can be used for this purpose. In this perspective, each container version is characterized by a distinct criticality level, which is influenced by its implementation, the virtualization technology, and the hardware on which it is intended to be hosted. For example, an application running in a sandboxed container provides improved isolation guarantees compared to a Linux container.

This approach requires overcoming some inherent difficulties. For example, the application source code dependency from the platform must be reduced or zeroed not to be as expensive as the original N-version programming, and the building system must be automated, seamless, and easy to use.

[1] A sandboxed container is a container that offers greater isolation compared to OS-level containers thanks to, for example, the hardware virtualization provided by hypervisors. In §7.3 I survey sandboxed container runtimes besides other container technologies.

A compelling use case of diverse containers: AI-powered controllers

Diversified containers can model an increasingly recurring use case: AI algorithms coupled with simpler algorithms used for spare redundancy. Indeed, system complexity increases as AI-powered algorithms spread out in mission-critical environments, such as reinforcement learning for control algorithms. This complexity represents a problem from multiple perspectives: an AI algorithm could not have the resources to run locally to a moving device (e.g., drone or robot) but it must run on an edge server, the connection may fail, the algorithm could give wrong outputs, etc. Hence, a solution that is becoming increasingly popular [79] is to couple an AI-powered algorithm with a classical controller running locally. The classical controller can provide basic functionalities for the system, still guaranteeing the system's survival in case of faults and errors of the AI-powered controller. In this perspective, the classical and AI-powered controllers provide diversified implementations of the same functionality, with the classical controller being a traditional real-time application compiled against a library real-time operating system (RTOS).

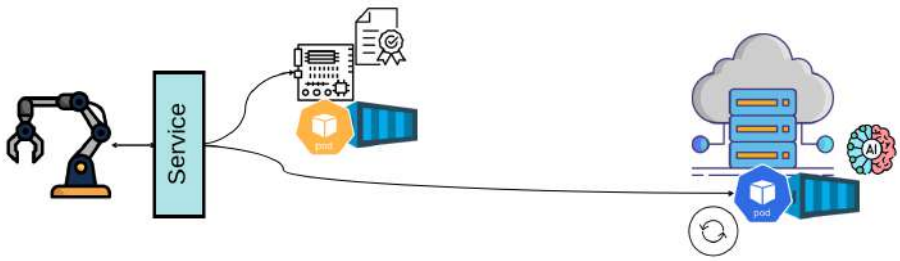


Figure 3.1. Diversified containers for mixed-criticality industrial controllers. One version is safety-critical and deployed on certified hardware close to the controlled object. The other version runs on an edge server to use more computing resources.

3.2.2 Diversified orchestration

The availability of the same service pod for multiple platforms with different assurance and criticality enables unseen orchestration primitives, particularly relevant for industrial settings. Such primitives are defined as follows:

- **Diversified replication:** when an application must run multiple in-

stances for either redundancy or load-balancing reasons, this primitive selects the computing nodes where to spawn instances to meet additional constraints prescribed by the service distribution function $f_j^D(\vec{J}_j)$ and aggregation policy. The two main considered constraints are (i) *criticality*, indicating the criticality of instances, and (ii) *redundancy scheme*, of two or more instances, e.g., in TMR or spare redundancy. For example, for a service that requires three pod instances, an orchestrator can spawn two pods with low criticality (e.g., Linux containers) and one critical pod (e.g., based on a minimal RTOS running on certified hardware) across two different platforms. Incoming requests can be load-balanced between the Linux containers, while the critical pod is used for spare redundancy. A common orchestrator resource instance has to manage the desired distribution function and coordinate the related resource instances (e.g., the diversified pods).

• **Seamless migration:** this primitive allows the seamless migration of pods between potentially heterogeneous platforms while accounting for the isolation guarantees provided by the worker nodes. When a migration is required (e.g., upon a node failure), the primitive allows respawning a pod onto a (potentially different) platform with the same assurance level as the previous one. If no computing node with comparable assurance is available, the primitive can select a worker node with lower guarantees (e.g., a Linux node) to provide a possibly degraded service or command for a fail-stop. The opposite can also hold: a complex controller running on a powerful Linux-based edge server can be migrated to a closer platform to run a simpler, *diversified* implementation. This implementation can only provide minimal functionality or take the system to a safe state. For instance, a complex AI-powered robot controller runs on a powerful Linux-based edge server to leverage large computing resources. Upon failure, the controller can be migrated locally to the robot (hence on a different platform) to run a simpler library RTOS-based implementation, which can only provide minimal functionality or bring the system to a safe state.

• **Diversified Rolling Update:** Rolling updates gradually replace the running application pods with updated ones, guaranteeing a minimum number of running pods to avoid downtime. In this regard, the primitive manages a diversified rolling update, where the pods to update for each round are selected according to the platform and criticality. The idea is to guarantee always a minimum pod criticality level and keep diverse containers running on

different platforms to prevent common mode failures due to regression faults affecting the same platform. Thus, when the primitive selects a set of containers to stop, it ensures that diverse containers continue running on different platforms while updated containers are not running yet. When the spawned containers are ready, a new update round starts.

3.3 Failure Model of Mixed-criticality Clusters

Based on the model introduced in this chapter, I here derive the failure probabilities equations for a service with *diversified replication*. The equations aim to show how diversity influences the failure probability for common redundancy schemes.

3.3.1 Pod failure

A pod may include a set of multiple tasks. Without loss of generality, I consider the entire task set as a single fake task with a response time equal to the pod response time, defined as the time from when the input is received, to when the output is produced. This is possible assuming a finite and known sequence of task activations to produce the pod output. Different activation sequences are allowed as they can be modeled as different functioning modes of the fake task. In the following, I analyze a pod failure.

$$
\begin{aligned}
&P(F_{J_l}) = P(pWCRT_l > D_l) = \\
&= P(pWCRT_l > D_l | J_l \text{ correct val.}) * P(J_l \text{ correct val.}) + P(\neg J_l \text{ correct val.}) \\
&= P(DM) * (1 - P(VF)) + P(VF) \\
&= P(DM) + P(VF) - P(DM) * P(VF) < P(DM) + P(VF)
\end{aligned}
\tag{3.1}
$$

Where D is the service deadline (see later), VF is the value failure event, and DM is the deadline miss event, defined as a task exceeding the deadline given that its output value is correct. In the last step, I bound the failure probability to consider the deadline misses and value failures as disjoint.

The DM event accounts for: recovered software/hardware transient faults (e.g., re-execution, local recovery mechanisms), scheduling algorithm (e.g.,

overload management, task interference, blocking times), hardware speed and assurance (i.e., isolation), and optionally service queuing. To simplify, DM accounts also for network delays.

The VF event accounts for: bugs that cause value errors, unrecovered transient fault (e.g., error burst with failed re-executions), task and OS omission, task and OS crash, and hardware crash. To simplify, VF accounts also for network omissions.

Summary: By definition of a real-time system, the pod failure probability is bounded by the probability of having a deadline miss OR a value failure.

3.3.2 Service failure

I analyze the failures of a service deployed with diversified replication in the target edge-cloud environment. I consider the failure probability for a request arriving at any time, with a non-state-space model (i.e., neglecting the system dynamics of components that fail and are repaired). The analysis was inspired by [80]. To simplify the analysis, without lack of generality, I assume that the deadline D belongs to the service, i.e., an application is made of only one service. If the application is a graph of services, scheduling algorithms with precedence constraints [37] can be used to derive the services' deadlines (like in [81]) based on the application deadline.

When dealing with multiple related pods, some considerations are needed about failure correlations. Since the VF accounts for several failure causes, I split them into two disjoint groups: a group of failures correlated between multiple instances, and a group of failures independent of other VF.

$$P(VF_0) = P(VF_{u0} \cup VF_{r0}) \tag{3.2}$$

Correlated failures mean that experiencing a VF for one instance increases the probability of experiencing a VF in another instance, because of some common-cause failure (e.g., common hardware, common bug activation pattern, common OS failure due to environment conditions). By definition, VF_{u0} and VF_{r0} are disjoint events (i.e., $P(VF_{u1}, VF_{r1}) = 0$). Conversely, common-cause failures increase the conditional probability of VF.

To simplify the analysis, I introduce the following assumptions:

Assumption 1 The decider/voter of the distribution function implemented

by the orchestrator is correct.
Assumption 2 It is perfectly detectable a DM for an instance.
Assumption 3 It is perfectly detectable a VF for an instance.
Assumption 4 Any DM is independent of any VF.
Assumption 5 Any DM is independent of any other DM because of the definition of pWCET [38][2]. This assumption implies that at the moment no queuing is considered, as the queuing time would contribute to the response time, and the deadline misses would not be independent. I plan to model this in the same way I do for coupled value failures.

In the next subsections, I model two common redundancy schemes: hot-standby redundancy and TMR [82].

Hot standby redundancy

This redundancy scheme consists of a parallel redundant system, with an active instance and one or more stand-by instances (ordered by decreasing priority). Each request is sent in parallel to all instances. The response is selected in order: when the active instance provides a value on time, it is used. Otherwise, the available from the highest priority stand-by instance is used. For example, two instances of a complex AI controller and a simpler but certified controller are used in decreasing priority order, similarly to the example in [79]. Therefore, a service failure happens when all the pod instances fail.

$$P(F_{\sigma_j}) = P(\bigcap_{l \in [1,|\vec{J}_j|]} F_{J_l}) =$$

Assuming 3 instances:

$$\begin{aligned}
&= P(\bigcap_{l \in [1,3]} (DM_l \cup VF_l)) = \\
&= P(DM_0, DM_1, DM_2) + P(VF_0, DM_1, DM_2) + P(DM_0, VF_1, DM_2) + \\
&\quad + P(DM_0, DM_1, VF_2) + P(VF_0, VF_1, DM_2) + P(VF_0, DM_1, VF_2) + \\
&\quad + P(DM_0, VF_1, VF_2) + P(VF_0, VF_1, VF_2)
\end{aligned} \tag{3.3}$$

[2]Usually, in literature an independent and identically distributed pWCET is assumed. I extend the assumption to different tasks of the same service.

Let α be the coupling factor of the VF of different instances, then:

$$\begin{aligned}
P(VF_1, VF_2) =& P(VF_{u1}, VF_{u2}) + P(VF_{u1}, VF_{r2}) + \\
&+ P(VF_{r1}, VF_{u2}) + P(VF_{r1}, VF_{r2}) = \\
=& P(VF_{u1}, VF_{u2}) + P(VF_{u1}, VF_{r2}) + \\
&+ P(VF_{r1}, VF_{u2}) + P(VF_{r1}|VF_{r2})P(VF_{r2}) = \\
=& P(VF_{u1}, VF_{u2}) + P(VF_{u1}, VF_{r2}) + \\
&+ P(VF_{r1}, VF_{u2}) + \alpha_{12} P(VF_{r1})P(VF_{r2}) \quad \alpha_{12} > 1
\end{aligned} \tag{3.4}$$

rewriting:

$$P(VF_1, VF_2) = P(VF_1)P(VF_2) + (\alpha_{12} - 1)P(VF_{r1})P(VF_{r2})$$

A similar expression holds for a three-wide joint event.

$$\begin{aligned}
&P(VF_1, VF_2, VF_3) = \\
=& P(VF_{u1}, VF_{u2}, VF_{u3}) + P(VF_{u1}, VF_{r2}, VF_{u3}) + P(VF_{r1}, VF_{u2}, VF_{u3}) + \\
&+ P(VF_{r1}, VF_{r2}, VF_{u3}) + P(VF_{u1}, VF_{u2}, VF_{r3}) + P(VF_{u1}, VF_{r2}, VF_{r3}) + \\
&+ P(VF_{r1}, VF_{u2}, VF_{r3}) + P(VF_{r1}, VF_{r2}, , VF_{r3}) = \\
=& P(VF_1)P(VF_2)P(VF_3) + (\alpha_{12} - 1)P(VF_{r1})P(VF_{r2})P(VF_{u3}) + \\
&+ (\alpha_{23} - 1)P(VF_{u1})P(VF_{r2})P(VF_{r3}) + (\alpha_{13} - 1)P(VF_{r1})P(VF_{u2})P(VF_{r3}) + \\
&+ (\alpha\beta - 1)P(VF_{r1})P(VF_{r2})P(VF_{r3})
\end{aligned} \tag{3.5}$$

The product $\alpha\beta$ is the correlation among failures of all instances.
Bringing Equation 3.5 into Equation 3.3 and rewriting:

$$\begin{aligned}
P(F_{\sigma_j}) = & \prod_{j \in [1,3]} P(F_{J_l}) + \\
&+ (P(DM_1) + P(VF_1))(\alpha_{23} - 1)P(VF_{r2})P(VF_{r3}) + \\
&+ (P(DM_2) + P(VF_2))(\alpha_{13} - 1)P(VF_{r1})P(VF_{r3}) + \\
&+ (P(DM_3) + P(VF_3))(\alpha_{12} - 1)P(VF_{r1})P(VF_{r2}) + \\
&+ (\alpha\beta - 1)P(VF_{r1})P(VF_{r2})P(VF_{r3})
\end{aligned} \tag{3.6}$$

The second part of the equation represents the increase in the failure probability when failures are related compared to fully independent failures case,

which only contains the product of the pod failures. When failures are completely independent, $\alpha = \beta = 1$, while they are $1/P(F_x)$ for completely coupled and deterministic failures. For 2 instances, the Equation 3.7 holds.

$$P(F_{\sigma_j}) = \prod_{j \in [1,2]} P(F_{J_l}) + (\alpha_{12} - 1) P(VF_{r1}) P(VF_{r2}) \tag{3.7}$$

Triple Modular Redundancy

In this redundancy scheme, the response is decided by majority voting between 3 instances. Thus, a service fails when 2 out of 3 pod instances fail.

$$\begin{aligned} P(F_{\sigma_j}) = {} & P(F_{J_1}, F_{J_2}, \neg F_{J_3}) + P(F_{J_1}, \neg F_{J_2}, F_{J_3}) + P(\neg F_{J_1}, F_{J_2}, F_{J_3}) + \\ & + P(F_{J_1}, F_{J_2}, F_{J_3}) \end{aligned} \tag{3.8}$$

The last term takes the structure of Equation 3.6. The other terms have a different structure since they contain the negated event of a failure. In the following, I examine the structure of one of those terms.

$$\begin{aligned} P(F_{J_1}, F_{J_2}, \neg F_{J_3}) = {} & P((DM_1 \cup VF_1) \cap (DM_2 \cup VF_2) \cap (\neg DM_3 \cap \neg VF_3)) \\ & \text{omitting the } \cap \text{ symbol for brevity} \\ = {} & P((DM_1, DM_2, \neg DM_3, \neg VF_3) \cup (DM_1, VF_2, \neg DM_3, \neg VF_3) \cup \\ & \cup (VF_1, DM_2, \neg DM_3, \neg VF_3) \cup (VF_1, VF_2, \neg DM_3, \neg VF_3)) \\ = {} & P(DM_1, DM_2, \neg DM_3, \neg VF_3) + P(DM_1, VF_2, \neg DM_3, \neg VF_3) + \\ & + P(VF_1, DM_2, \neg DM_3, \neg VF_3) + P(VF_1, VF_2, \neg DM_3, \neg VF_3) \end{aligned} \tag{3.9}$$

The DM terms are independent and can be factored out because of the **Assumption 4**. Conversely, the VF terms cannot be factored. For example:

$$P(DM_1, VF_2, \neg DM_3, \neg VF_3) = P(DM_1) P(DM_3) P(VF_2, \neg VF_3) \tag{3.10}$$

Similarly to Equation 3.4, let $\gamma < 1$ be an anti-correlation factor.

$$\begin{aligned}
P(VF_2, \neg VF_3) &= P(VF_{u2}, \neg VF_{u3}, \neg VF_{r3}) + P(VF_{r2}, \neg VF_{u3}, \neg VF_{r3}) = \\
&= P(VF_{u3})(1-P(VF_{u3}))(1-P(VF_{r3})) + P(VF_{r2}, \neg VF_{r3})(1-P(VF_{u3})) = \\
&= P(VF_{u3})(1-P(VF_{u3}))(1-P(VF_{r3})) + \\
&\quad + P(\neg VF_{r3}|VF_{r2})P(VF_{r2})(1-P(VF_{u3})) = \\
&= P(VF_{u3})(1-P(VF_{u3}))(1-P(VF_{r3})) + \\
&\quad + \gamma_{2,3}(1-P(VF_{r3}))P(VF_{r2})(1-P(VF_{u3})) < P(VF_2)P(\neg VF_3)
\end{aligned} \tag{3.11}$$

3.4 The Role of the Orchestrator

The probability equations considered the service failure for a request arriving at any time without accounting for the system state, i.e., the failure probabilities for two consecutive requests are equal. In addition, those equations assume a correct orchestrator, since the only failures considered are the ones of the pods. However, the reality is more complex. For example, if a service experiences a pod failure due to a crash, a closely upcoming request will likely experience it as well, as it takes some time to restore an instance after the crash. In the following, I analyze how the orchestrator affects some model parameters, including the case of a non-correct orchestrator.

The orchestrator creates and manages applications and services, maintaining the relationships among entities. When the pods are created, the orchestrator defines rules to place them onto the worker nodes such that $\vec{BR}_n \vdash \vec{BR}_l$, $AR_n \vdash AR_l$, $RT_n \vdash RT_l$, $\vec{A}_n \vdash AP_l$ (where $\vdash$ means that the node can satisfy the pod requests). In other words, the orchestrator must select a worker node with sufficient available resources to host the pod and an assurance level that aligns with the pod's criticality. If the worker nodes support real-time scheduling, the orchestrator must perform schedulability tests to guarantee the timely completion of the applications. Further, the orchestrator must be aware of each network topology, meet pods' networking requests, and enforce network schedulability tests when needed.

Once the pods are running, the orchestrator is in charge of implementing the distribution function $f_j^D(\vec{J}_j)$, and guaranteeing the set of pods $\vec{J}_j$ to be running and healthy. Upon a pod or worker node failure, the orchestrator

must restore the cluster to the desired state, taking a certain recovery time. The time to failure is described by a stochastic variable X, and its expectation $E[X]$ is called MTTF. Homologous definition holds for the MTTR.

The example below clarifies the concept (represented in Figure 3.2).

Example of mixed-criticality service

A service is composed of 2 low-criticality and 1 critical pod. The distribution function sends every request to the critical pod, while it load balances the requests between the two low-criticality pods (because their throughput is lower). Hence, each request has one response from a low-criticality pod and one from a high-criticality pod, with the pods working in a hot standby replication scheme. The response from the low-criticality pod is preferred when available, as it is more precise.

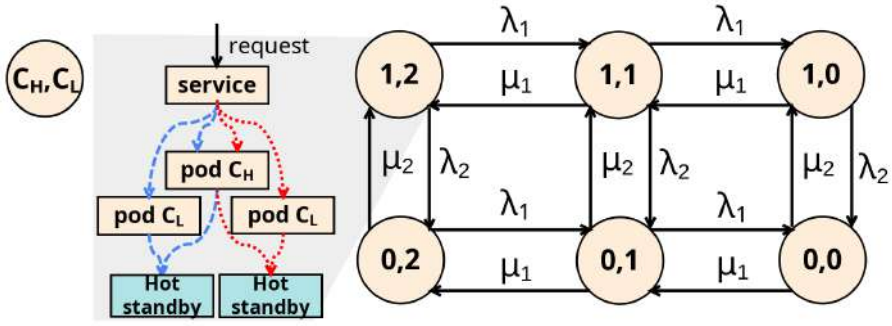


Figure 3.2. Markov model of the mixed-criticality service. The service has two instances of criticality LO and one of criticality HI. $\lambda_1 = 1/MTTF_1$ and $\mu_1 = MTTR_1$ are the transition rates of a pod instance with lower criticality, while $\lambda_2 = 1/MTTF_2$ and $\mu_2 = 1/MTTR_2$ are related to the critical instance. The model assumes constant rates and memoryless distributions of the stochastic variables.

In this perspective, the service failure probability previously modeled only holds in the (1,2) state. When the system moves out of the (1,2) state, it provides a degraded service, affecting the service SLOs. When the system is in one of the states of the lower row, it may be switched to a fail-safe state due to the lack of a critical spare pod. Conversely, in the upper-row states,

the low-criticality pods may provide a late service or fail both.

In this perspective, the MTTR determined by the orchestrator has a direct impact on the service SLO. Indeed, the MTTR includes the time for the orchestrator to detect a failure and the time for the orchestrator to restore the system state. If the orchestrator suffers a *timing failure*, i.e., the time to recover is significantly longer than expected, the system has a higher probability of lying in a degraded state. For example, an increased time to recover implies a higher probability of having simultaneous failures. In the case of particular constraints, e.g., the service can only provide minimal functionality when there is no critical spare pod, the time to restore the critical pod directly affects the service availability.

Even worse, the orchestrator may misbehave. For example, it may fail to enforce the distribution function (e.g., traffic lost, wrong replication or voting), it may not correctly handle a worker node failure (e.g., fewer pods are running than expected), or it may place a pod on a worker node that does not satisfy the pod requirements. In any case, the service deployed suffers (temporarily or permanently) from a degraded SLO.

Such behaviors are difficult to model. Referring to Figure 3.2, a failure may introduce a new edge between two system states, cut an MTTR edge, or affect the specific behavior of the service.

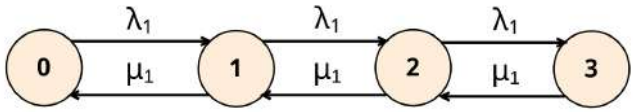


Figure 3.3. Markov model of an autoscaled service with maximum 3 instances. $\lambda_1 = 1/MTTSU$ and $\mu_1 = MTTSD$ are the transition rates defined by the autoscaler, where $MTTSU$ is the mean time to scale up, and $MTTSD$ is the mean time to scale down. The model assumes constant rates and memoryless distributions of the stochastic variables.

Similar concepts hold when the orchestrator is in charge of automatically scaling a service based on the current service workload. In Figure 3.3 a simple example is modeled with a Markov chain. I assume a memoryless distribution of stochastic variables in order to use a Markov model. To respect this assumption, I analyze the system on a shorter timescale (e.g., minutes), during which I can approximate the variables with memoryless distribution,

neglecting long-term trends (e.g., daily workload periodicity).

The time to scale up the service is given by the time that the workload takes to vary enough to require a scale-up, the time to monitor the workload and decide to scale up, plus the time to scale up and create the pod. Homologous concepts hold for the time to scale down.

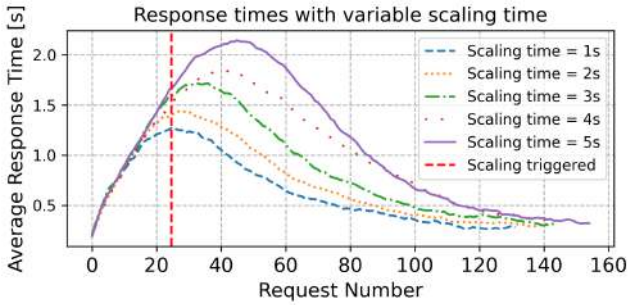


Figure 3.4. Simulated effect of the scaling time on response times. The simulation assumes decreasing exponential distributions for serving times (avg. $\mu = 0.2s$) and inter-arrival times (avg. $\sigma = 0.16s$). A scaling is triggered when the queued requests become 10. The plots are averaged over 300 runs.

In contexts like function as a service (FaaS), where pods are frequently spawned to execute a set of functions on demand, the time to scale up and down the services determines how quickly the cloud application can respond to requests (i.e., the SLOs). In particular, two situations are possible: i) a function needs to be spawned and there are no instances available, ii) the request rate for a function increases, and it must be scaled. In the first case, the spawning time is on the critical path for the response times of the first requests. In the second case, the authors of [83] showed that, although the time to spawn a new pod is not in the critical path, it determines the end-to-end service response times during the scaling process. A simulation (depicted in Figure 3.4) shows that as the scaling times vary from 1s to 5s, the response times increase by up to 90%. What is more, the authors point out that the orchestration times can be a bottleneck in a context where the start time of instances is reduced to a few milliseconds.

3.4.1 Discussion

Although several assumptions were made to simplify the analysis, the mathematical analysis showed how failure probability equations can be rewritten assuming independent failures and adding factors that summarize the coupling level of the replicas. The more the replicas differ, the more additional terms that increase the failure probability become negligible.

In cloud systems, the analysis of failures is simplified, and often only worker node failures are accounted for. For example, as long as two replicas run on two different worker nodes or availability zones, they are considered independent. The same simplifications cannot hold when dealing with critical applications, since low probability events must be carefully considered.

From this perspective, the failure equations illustrate how an orchestrator can account for pod diversity and criticality to automatically determine pod placement, ensuring compliance with criticality constraints and redundancy schemes required by services.

Furthermore, state-space Markovian models highlighted the importance of the orchestrator also in terms of timeliness and functional correctness. Indeed, a wrong or late orchestrator decision (i.e., an error or long orchestration times) directly impacts the services SLOs. For example, cloud-native 5G applications require even six nines of availability (i.e., $\approx 30s/year$ of downtime) or probability of meeting the response times: a few seconds of failover time or increased response times can violate the SLOs and must be considered a failure [84]. Hence, orchestration times must be subject to dedicated *orchestrator* SLOs [85] that must be accounted for when deploying critical services. For this reason, in the next chapters, I assess the resilience and timing properties of K8s, generalizing the claims to other orchestrators when needed.

Chapter 4

A Failure and Timing Analysis of Orchestrators

If you believe this will never happen, you are the one who must prove to me that this will never happen.

Zbigniew Kalbarczyk

As orchestrators host increasingly critical applications, it is imperative to have a clear taxonomy of their failure modes, analyze errors that cause severe disruptions, and understand the factors determining the orchestration times.

However, on the one hand, there are no available studies to assess the resiliency of K8s in a systematic way or that categorize its failures. On the other hand, fine-grained timing analysis of orchestration times is neglected by recent research[1], which focused on reducing scaling and startup times by carefully managing replicas and pre-warmed sandboxes.

This chapter analyzes K8s failure modes through real-world failures, assesses its resilience through fault injection, and presents a fine-grained timing analysis, generalizing when needed concepts to other orchestrators.

Both experiments and real-world K8s failures will show that one incorrect data value can propagate and cause system-wide failures despite the resiliency strategies in place. In particular, overloads and capacity-related failures will prove to be harsh because of the scale of the system, which is an inherent

[1] A detailed review of the related work can be found in §7.1.3

threat. Such high load conditions can induce orchestration times in the order of tens of seconds with high variability, possibly overshadowing downstream boot times optimizations in the hundreds of milliseconds.

Based on the insights of this chapter, I claim that *ad hoc* resilience mechanisms and systematic tests through fault injection must be incorporated into the orchestrators and that the orchestration times of high-priority services must be constant and unaffected by concurrent requests.

Section 4.1 analyzes the real-world failures' reports collected from online sources. Those reports are mainly blogs, talks, and forums [86, 87, 88, 89, 90, 91] regarding real-world failures of enterprise production clusters in the order of a thousand nodes [92].

Next, section 4.2, based on the takeaways of the failure analysis, presents a K8s injection framework called *Mutiny* that is used to perform a fault/error injection campaign targeting the Etcd datastore. A comparison between the results of the FFDA and the fault/error injection experiments shows that the Mutiny triggers error propagation patterns observed in real-world failures.

Then, section 4.3 analyzes how orchestration times vary under an increasing orchestration load in order to understand i) whether commonly used orchestrators can fully prioritize services by their criticality, and ii) the key parameters and sources of delay determining the orchestration times.

After that, section 4.4 analyzes the timing behavior of real-time containers based on Linux when intense stress is co-located on the same node to determine whether Linux containers can host real-time critical applications.

Finally, section 4.5 provides some discussion about how to improve testing and cluster configurations to cope with highlighted issues.

4.1 Field Failure Data Analysis

This section analyzes data on real-world K8s failures collected in [86]. Large part of this section is taken from "Mutiny! How Does Kubernetes Fail, and What Can We Do About It?" ©IEEE 2024 (see author's publications section), with possible adaptations for this thesis.

A quantitative analysis of the available failure data is not feasible because: i) the failures reported (e.g., in online blogs) by companies are a subset (e.g., the most impactful) of all failures that happened in different systems; ii) the lack of a systematic approach to data collection may mean failures go unre-

ported; and iii) the available failure descriptions frequently do not provide relevant failure details and data, hindering a traditional FFDA.

Nonetheless, although those human-generated reports can be vague and incomplete, a qualitative FFDA is useful to i) create a taxonomy of failure modes, ii) get insights about the most relevant and impactful error/failure patterns in the wild, iii) understand dependability bottlenecks of the system, and inform the design of an effective fault/error injection campaign.

From the analysis, it emerges that errors or misconfigurations in subsystems such as networking or replication control can cause cluster-wide failures.

4.1.1 Methodology

I categorized the faults, errors, and failures by relying on the system knowledge and the information provided by the online sources. The final categorization is an agreement between multiple experts.

The categorization was performed by subsequent refinements, grouping faults, errors, and failures with common characteristics together. I did not rely on established mythologies like the Orthogonal Defect Classification (ODC) proposed in the '90s due to a lack of information. The categories of faults and errors do not want to be exhaustive, but rather a hint of common issues.
In order to categorize the faults, the following questions were used:

Q1 Is the system behaving as it is supposed to? (E.g., cluster failures may be caused by misuse of the orchestrator due to lack of understanding of its specification, or there might be a bug in the orchestrator)

Q2 Where is the fault located? (E.g., inside K8s, outside K8s but in underlying software layers, in the workload or scaling configurations provided by the user, in the hosted applications)

Q3 Is the fault causing erroneous states related to the size of the workload? (E.g., a user may provide a bad workload that is excessive compared to the system size)

Q4 Is the fault permanent? (E.g., the fault may be a fault that has always been in K8s, a fault introduced by an update that introduces bugs or changes in system specification)
In order to categorize the erroneous states, the following questions were used:

Q1 Which architectural subsystem is erroneous?

Q2 Is the subsystem still responsive?
In order to categorize the failures, the following questions were used:

Q1 Are all the hosted services available?

Q2 Does the failure affect the entire cluster or only a small fraction?

Q3 Did the system get to the desired state in the expected amount of time?

Q4 Is the amount of resources allotted less or more than expected?

Q5 Does the cluster behave as expected upon changes? (E.g. new nodes joining, new services created)

I divided the failures into orchestrator-level failures and client-level failures. An orchestrator-level failure (OF) is a misbehavior of the orchestration system that may or may not impact deployed services. A client-level failure (CF) is the perception of an orchestrator's misbehavior from the point of view of a service client. In this sense, a client-level failure is an orchestrator-level failure that has enough impact to be perceived by the client (e.g., a majority or all pod replicas unavailable). Conversely, if an orchestrator-level failure affects a few resource instances, the clients might not perceive any statistically significant effect on the service metrics (e.g., response time and availability). In this section, I only analyze orchestrator-level failures due to the lack of data that prevents any analysis of client-level failures.

Table 4.1 provides a *Fault-Error-Failure* categorization derived from 81 failure instances analyzed.

4.1.2 Orchestrator-level failures

Based on my analysis of the failure data, I classified the K8s failures into the following categories: *Timing Failure* (`Tim`), *Fewer Resources* (`FeR`), *More Resources* (`MoR`), *Service Network* (`Net`), *Stall* (`Sta`), and *Cluster Outage* (`Out`) (see Table 4.1 (c)). Despite being relative to the K8s failure dataset, the failure categories do not depend on any specific feature of K8s and can be used with other orchestrators as well.

In my categorization, I explicitly differentiate between Cluster Outage (`Out`) and Stall (`Sta`) failure types: a Cluster Outage implies that a majority of services are down, while a Stall implies that currently running services are still up, but that the cluster's ability to react to changes is limited. For example, already running services are working but newly deployed services are not correctly configured, Nodes joining the cluster are not networked, or new services cannot be deployed due to lack of resources. In an environment with limited evolution, services across the cluster could remain healthy. In an environment where the user asks for frequent reconfigurations and redeployments, or Nodes frequently leave and join the cluster, the cluster might suffer a degradation that finally leads to a system outage.

Importantly, error patterns that lead to `Out` and `Sta` failures can be similar. For example, spawning an infinite number of Pods can lead to a `Sta` or `Out` depending on the Pod priority: preemptive Pods evict all the lower-priority Pods, leading to an `Out` failure. Conversely, non-preemptive Pods saturate cluster hardware resources, possibly preventing new services from being deployed, but do not kill currently running instances.

I consider the `MoR` failure type to be more severe than `FeR` because, although `FeR` impacts the application SLOs, allotting more resources carries higher costs and risks related to computing resources exhaustion or system overload.

The differences between `FeR`, `Net`, and `Sta` are mainly in the scale of the failure impact. `FeR` and `Net` impact a limited number of services, while `Out` compromises one of the vital cluster functionalities, impacting almost every running service. For example, a stuck Node might impact a few services, depending on its size, but it does not lead to a system outage. In this sense, `Sta` and `Out` are system-wide failures, while the others are *local* (e.g., to a service or node) failures.

Table 4.1. Fault-Error-Failure chain of real-world Kubernetes failures. Failures are listed in order of increasing severity. ©IEEE 2024

Fault	Fault Description
Wrong Autoscaling	Autoscaling of Pods or Nodes is based on misleading information
Race Condition	Concurrent actions whose final result depends on timing. E.g., in routing tables/connections
Unverifiable Certificate	Certificates cannot be verified or recognized (e.g., cert. rotation)
Bug	Bug in K8s, third-party, plugins, or underlying code (runtime, OS)
Human Mistake	Incorrect command or configuration including: 1) bad resource sizing of components or apps, 2) wrong or badly tuned settings
Unmanaged Upgrade	System specification or implementation changes, failing regression
Overload	Too many Pods or Pods with too many resources for a cluster/Node
Low-Level Issues	Faulty hardware or related drivers
Failing Application	Misbehaving application causing many events and/or failing Pods

(a)

Error	Error Description
State Retrieval	Irretrievable, stale, or corrupted state due to unavailability, delays or user commands
Misbehaving Logic	Components behave differently from expected, affecting the reconciliation actions
Communication	Networking delays or failures: DNS, routing, load balancing
Resource Exhaustion	Affected amount of available computational resources: number of available Nodes, Node/control plane resources, etc.
Control Plane Unavail.	Unhealthy control plane components are slowed down or cannot take actions
Local to worker Nodes	Errors in underlying software: container runtime, OS, image availability

(b)

Failure	Failure Description
None (`No`)	System recovered without any consequences, timely reaching the correct steady state
Timing Failure (`Tim`)	The creation/update of Pod or other resources took significantly longer than expected, e.g., due to component restarts or overfilled queues.
Fewer Resources (`FeR`)	One or a reduced number of services at steady state have permanently allotted fewer resources than planned, e.g. Pod number or resources
More Resources (`MoR`)	One or a reduced number of services has temporarily or permanently allotted more resources than needed, e.g. Pod number or Pod resources
Service Network (`Net`)	One or a reduced number of services have the correct amount of resources allotted but are incorrectly networked
Stall (`Sta`)	Cluster's ability to react to changes was compromised, but already-running services remained unaffected: e.g., new Nodes and Pods not spawned or configured.
Cluster Outage (`Out`)	A significant number or all the running services are compromised and unable to respond to application clients anymore

(c)

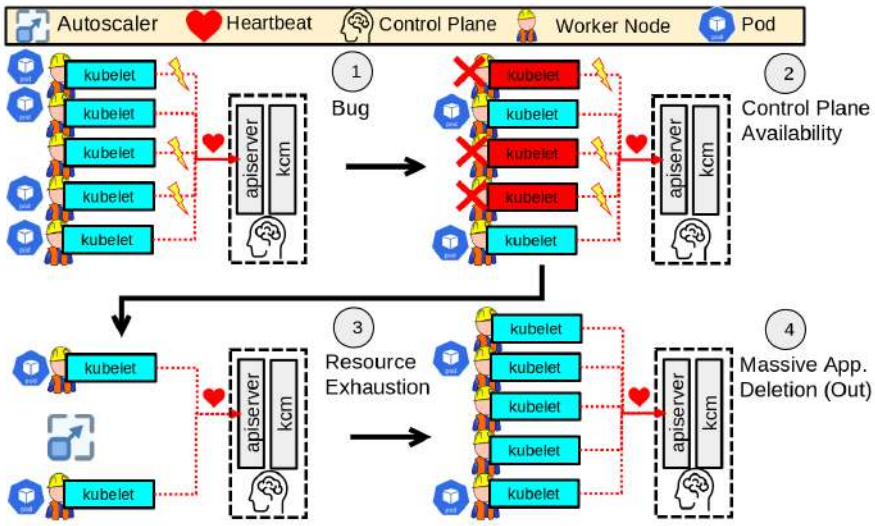


Figure 4.1. Example of cluster outage `Out` failure. A timeout during the control plane startup caused an intermittent Apiserver downtime. This caused Kubelets to be unable to report Node health, leading to a massive Node deletion and recreation by the Google Kubernetes Engine (GKE) autoscaler.

4.1.3 Orchestrator-level faults and errors

15 failures in total were `Out`. Hence, they are not infrequent but a major concern. The most severe faults/errors that caused them had the following causes (in parentheses, the categories from Table 4.1(a,b)): i) network manager failures that impacted the entire cluster (Communication); ii) massive numbers of unhealthy or deleted Nodes (Resource Exhaustion); iii) erroneous commands that deleted namespaces, clusters, or Etcd data (Human Mistake→State Retrieval); and iv) preemptions caused by infinite spawning (Resource Exhaustion). For example, Figure 4.1 illustrates a failure in which a fault hindered the Node heartbeat reporting, leading to massive Node deletion by the Google K8s Engine autoscaler, even if the Nodes were correctly running the applications [90].

Fault/error propagation

Finding 1 - Capacity-related failures

Misconfigurations can easily saturate all computing resources and overload the system, which does not detect hazardous user commands when managing resources at scale.

Misconfigurations (Human Mistake in Table 4.1(a)) caused 33 of the failures in my data set. 10 of them consisted of bad resource sizing of Nodes and services. If services had too few resources, the application failed (Human Mistake), if they had too many resources, Nodes failed (Overload→Resource Exhaustion). Specifically, 19 faults were misconfigurations of K8s, 3 misconfigurations of plugins, and 11 misconfigurations of external software. 13 incidents involved errors caused by bugs in K8s code (5), external software (4) (e.g., underlying OS), plugins (1), or custom code (3). Capacity issues were responsible for 21 failures; 11 of which were due to an overload of control plane components (Overload, Failing Application, Human Mistake→Control Plane Availability), which failed to timely reconcile the cluster state. 19 incidents involved communication errors (Communication in Table 4.1(b)): domain name system (DNS) resolution, a misbehaving network manager, black holes, latencies, and connection errors. They were caused by underlying OS race conditions or bugs, certificate rotations, human mistakes, or unmanaged upgrades. DNS-related issues have been deemed the most painful by multiple companies [88, 92].

Various incidents were caused by multiple interacting factors, which are troublesome in conditions rarely met in testing. Often, the alleged root cause of observed failures is a guess, which could be a propagated error of the actual unknown root cause. At other times, it is difficult to derive a cause from the available "story-telling". The lack of control motivated us to perform a systematic injection campaign in a controlled environment.

4.2 Fault Injection

This section presents the fault/error injection framework (in Figure 4.2), including workloads, the injector Mutiny, the fault/error injection campaign

manager, and the data collection.

4.2.1 The Mutiny injector

Basic principles

K8s is designed to withstand common errors and failures through a range of resiliency strategies, including heartbeats, redundancy, circuit breakers, failover, and stateless system components. Having stateless components is essential to have a robust system: action is based on observation rather than a state machine and, upon restart, a component only needs to get the current and the desired cluster states from the data store. The state dependency is thus moved away from the components to an external data store, which preserves the entire system state and hence becomes a dependability bottleneck. Any alteration of the data in the data store may propagate and cause failures in every system component that stores state information on Etcd. In the same way, since the control plane relies on the stored states to decide the actions to take, almost the totality of divergences from the desired state likely leaves some erroneous value on the datastore.

I do not care about the possible root causes of alterations: hardware faults, software bugs, misconfigurations, or other causes that somehow store an incorrect value.

In this sense, deliberately injecting state alterations can be seen either as a fault or error injection. Indeed, the injection can replicate a fault originating while data is being stored, or replicate the effect of some fault originating in another component, causing an erroneous value to propagate to Etcd. For example, injecting a bit-flip caused by hardware issues (like in [93]) replicates the originating fault that happened in the real world. On the other hand, a faulty autoscaling configuration can cause the creation of more Pods than expected, which can be replicated by directly altering the number of replicas on Etcd, i.e., the effect of the misconfiguration.

I show (as discussed in §4.2.4) that Etcd alterations can recreate a majority (54/81) of real-world failures analyzed in §4.1. For example, Nodes can become unhealthy because of a failing Apiserver or a bug in the Kubelet [90, 94]. Although the subtle behavior of a bug cannot be replicated, the injections can replicate the effect of having an unhealthy Node, e.g., targeting the heartbeat reporting system.

Mutiny - What, When, Where

Mutiny is an injector that can be integrated into K8s to alter the messages exchanged between components and, consequently, the current or desired cluster state. The fault/error injection framework allows us to systematically inject faults/errors into Etcd and assess system response (in terms of the orchestration actions and application behavior) in a controlled environment.

Three attributes characterize each fault/error injected by Mutiny: location (where?), type (what?), and trigger (when?).

Where is defined by a communication channel, a resource kind, and either a field value or the serialization protocol bytes of a message. I distinguish two types of communication channels: i) those from Apiserver to Etcd, or ii) those from another component to Apiserver. By injecting the data in the transactions from the Apiserver to Etcd, I directly alter the current or desired cluster state. This emulates faults/errors that originate in the Apiserver or other components, but propagate undetected to Etcd. With replicated control planes, the fault/error is injected before the consensus algorithm is run, so that all Etcd replicas agree on the value.

Messages directed from other components to the Apiserver undergo authentication, authorization, and admission control. Hence, a corruption of a message in this channel can make the message invalid and cause it to be rejected by the Apiserver. Admission control can change the message content, even through custom code, possibly introducing errors.

What consists of a value and fault/error type between bit-flip, data-type set, and message drop.

A bit-flip is an easy way to alter a correct value without understanding its semantics and hence allows for extensive fault/error injection campaigns. If the value must match regular expressions or ranges, the injected value is, with high probability, still valid but incorrect. Bit-flip faults in Etcd data were also reported by users [93].

A data-type set triggers data validations and integrity checks by setting extreme, invalid, or wrong values, dependent on the field type. Such values might include empty strings, 0 for integers, or unsupported values for fixed-set values.

A message drop emulates a state update that did not happen for some

reason: a failed request, software bug, updated system specification, or data loss [95]. It aims to stress the resiliency of level-triggered reconciliation. It is a commonly assumed failure mode in distributed systems [96].

When is defined by the occurrence of messages related to the same resource instance sent by the injected component, i.e., the index in the chain $[c_i^f, ..., c_j^f]$, where $c_{i,...,j}^f$ are the state changes (see §2.1.2) in which the injection target appears. The injection may have different effects depending on the current state of the instance and the next state changes. Moreover, a different occurrence index can correspond to a different action performed by the software, e.g., resource instance creation vs. update. This can influence the transiency of the effect.

4.2.2 Experimental method

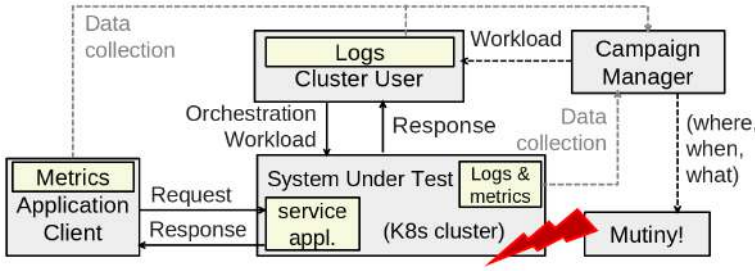


Figure 4.2. Mutiny fault injection framework. ©IEEE 2024.

Workloads

Orchestration workload The orchestration workloads perform operations on a service application used by a client to create activity in the orchestration system [2]. The workloads include i) "deploy", which creates new Deployments and related Pods; ii) "scale-up", which increases the replica number of existing

[2] I used synthetic workloads because there is a lack of benchmarks dedicated to the orchestration system. Well-known benchmarks for cloud microservices (e.g., *DeathStarBench*, available at `https://github.com/delimitrou/DeathStarBench`) do not necessarily generate representative orchestration workloads that trigger orchestration functions.

Deployments; and iii) "failover", in which a Node failure is simulated through a *NoExecution* taint, forcing the Pods running on the Node to be respawned onto available Nodes. The workloads are applied by *kbench*[3] acting as a cluster user (see Figure 2.4).

Applications The service application is a service exposed to the client. Its characteristics define the orchestration functionalities used.
The application client (AC) sends requests to the service application for a fixed period, monitoring its availability and response times.

Campaign manager

Campagin manager The campaign manager coordinates fault/error injection experiments, following the workflow in Figure 4.3. First, I recorded the fields of the resource instances sent to Etcd during the execution of a nominal orchestration workload, which comprises deploying, scaling, and updating Node states. Later, the injection campaign was generated, and finally the campaign manager drove the injection experiments.

Injection campaign The injection campaign includes injection experiments targeting i) a field of a message, ii) its serialization bytes, or iii) a whole message (for message drops). For each recorded integer field, I flipped a low- and a high-order bit (respectively, 1^{st} and 5^{th}), and I set the 0 data value. The reason for flipping those bits is that exchanged messages are serialized with the *Protobuf* protocol, and most such encoded integers are one byte long, with the 8^{th} bit used as a continuation bit.

For each recorded string field, I flipped the least significant bit of the first two characters, and I set the empty string data value. Injecting the least significant bit of a character still results in a character, and hence valid strings. Boolean fields are inverted. For each field, I ran an injection experiment for the occurrence indexes $1, 2$, and 3.

For each recorded resource kind, I performed a set of injections targeting random serialization Protobuf bytes to assess the system's response to incorrectly structured messages.

[3]The tool is available at `https://github.com/vmware-tanzu/k-benc`

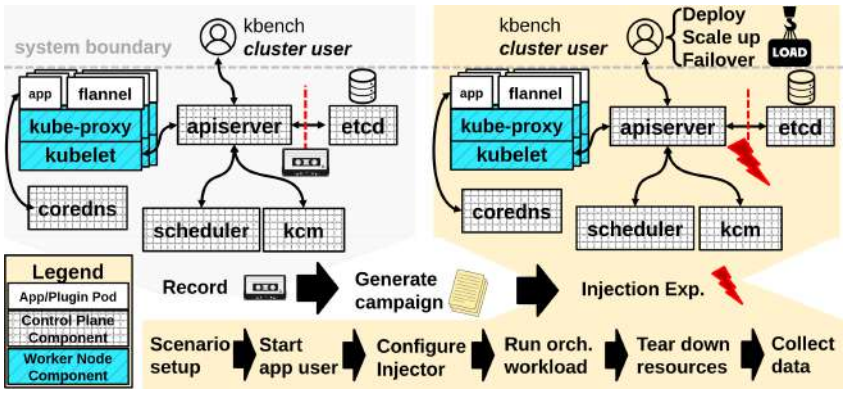


Figure 4.3. The three phases of the injection campaign workflow. ©IEEE 2024

For each recorded resource kind, I performed a message drop injection for the occurrence indexes from 1 to 10.

After the listed injections, I derived a set of *critical* fields, i.e., fields that caused `Out`, `Sta`, or unavailable service failures. I performed additional injection experiments with data values specific to each critical field semantics.

Injection experiment An injection experiment was composed of the following phases (Figure 4.3): K8s cluster restart, fault/error injection scenario set-up, application client workload start, injector programming, orchestration workload execution, and data collection. In each experiment, a single fault-/error was injected.

To restart the cluster, all the Nodes leave the cluster, the control plane Node resets the cluster and creates a new one, and, finally, all the Nodes join the newly created cluster.

The *scenario setup* creates all the resource instances that are required by the orchestration workloads before the injection.

Then, the application client workload starts performing requests to the service application. Next, the campaign manager configures the injection trigger by sending the triplet $(where, when, what)$ in an HTTP request to the injected component. Mutiny is implemented as a package in the K8s source tree. Any component can call it (with instrumentation < 10 LoC) to tamper a message serialized with Protobuf. For bit-flip and data-type set injections, Mutiny deserializes the message, modifies the content, and re-serializes it, re-

placing the original. For message drop injections, the calling function returns without any error before sending the message. In each experiment, I perform a single fault/error injection.

Data collection The data collection retrieved the logs of K8s control plane components (with verbosity level set at 6, i.e., debug), Kbench logs, response latencies experienced by the application client, and, finally, the metrics gathered from Prometheus with node_exporter and kube-state-metrics as sources.

Table 4.2. Client failure categories ©IEEE 2024

Failure category	Failure Definition
No significant impact (NSI)	The service is available and the response times seen by the AC are not significantly different from golden runs
Higher response times (HRT)	The service is available and the response times seen by the AC are significantly higher from golden runs
Intermittent availability (IA)	The application client experiences intermittent error responses from the service not due to request timeouts
Service unreachable (SU)	From a certain instant in time, the service is unreachable to the AC

4.2.3 Methodology of results analysis

Two levels of failures are considered: orchestration-level failures (OF, §4.1), and client-level failures (CF). CFs reveal the fault/error impact on application clients (AC) in terms of performance and availability. For both OFs and CFs, if a failure belonged to more than one category, I classified it as the most severe failure category.

In Table 4.2, I introduce the categories of client failures: no significant impact (NSI), higher response times (HRT), intermittent availability (IA), and service unreachable (SU). For each workload, I collected data from 100 golden runs without any faults/errors injected.

Orchestration-level failures To categorize orchestration-level failures, for every golden run I collected the number of ready replicas for each ReplicaSet, and the number of Service endpoints, every 3 seconds. I collected Kbench statistics regarding the number of Pods created/scheduled/running and the

Pods' *total startup times* [4]. I categorize the failures as follows, recalling Figure 4.1 (c).

• `Tim` failure: A service Pod is restarted, or the z-score relative to the golden distribution of either the worst Pod total startup time or the last Pod creation time is greater than 3.

• `FeR` failure: The number of ready replicas, created Pods, or endpoints is stable and lower than the baseline.

• `MoR` failure: The number of ready replicas, created Pods, or endpoints is higher than the baseline.

• `Net` failure: The number of ready replicas and Pods is correct, but some are not reachable or used in load-balancing.

• `Sta` failure: There is an uncontrolled spawn of Pods, control plane Pods are stuck, or networking Pods fail.

• `Out` failure: All the ReplicaSets are unreachable (including Prometheus), the DNS Pods fail, or the networking Pods fail and cause a disruption of the service application.

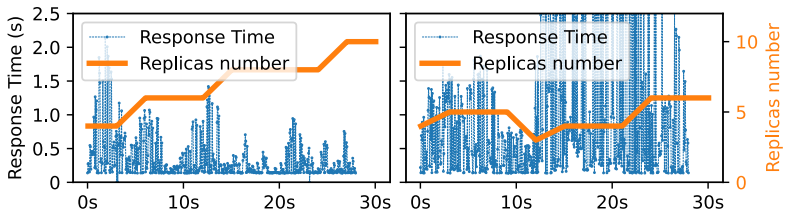


Figure 4.4. Example of response times seen by the client. On the left, a golden run time series ($z\_score = -0.2$). On the right, a time series during an injection experiment ($z\_score = 11.0$). ©IEEE 2024

Client-level failures To categorize client-level failures, I created a time series for each golden run containing the response time latency of the requests, ordered by the time of sending. I padded with 0 the response times of failed requests. I computed a *baseline* time series for each workload by averaging the golden run time series. I measured the mean absolute error (MAE) between each golden run time series and the baseline time series, obtaining a distribution of golden-run MAEs. For each injection experiment, I computed

[4] As defined in `https://github.com/vmware-tanzu/k-bench`

the MAE between the experimental time series and the baseline, and computed the z-score of the MAE against the distribution of golden-run MAEs. The z-score quantifies the impact on the application client (see Figure 4.4). I categorize the failures as follows.
• HRT failure: The z-score is greater than 2.
• IA failure: The application client experiences intermittent errors, not due to timeouts.
• SU failure: An application has no response from the service.

4.2.4 Experimental results

Experimental setup and parameters

The experimental setup consisted of a cluster running K8s v1.27.4 in the default *kubeadm* configuration. The cluster included 1 control plane Node and 4 worker Nodes, one of which was used for the application client and monitoring Pods The network manager was flannel v1.1.2. The cluster featured the default resiliency strategies described in §2.1.4. Unless differently specified (see §4.2.4) the cluster was managed by a single control plane Node running all control plane Pods (default configuration [5]). Although production environments commonly use multiple control plane Nodes, this improves the availability in case of a Node crash but provides no protection from faulty values on the datastore. Indeed, the Controller and Scheduler have only one active replica at any time, while the datastore replicas agree on the faulty value. The nodes were virtual machines (8 CPU, 4 GB RAM, Ubuntu 20.04, Linux kernel v5.4, containerd v1.7) communicating through an internal network, on top of VirtualBox 6 hypervisor in a cloud-tier environment (Intel Xeon E5-2695), where no other user application was running.

Based on the amount of resources of the setup, I parametrized the workloads as follows: the “deploy" workload created three Deployments, each with two replicas; the “scale-up" workload scaled two Deployments from two replicas each, to three replicas each, after 10 seconds to four each, and after another 10 seconds to five each; and the “failover" workload deals with three running Deployments with two replicas each. Kbench waited up to 40 seconds for each request to be completed. The service application was a *Flask*

[5] More info available at `https://kubernetes.io/docs/concepts/overview/components/`

Table 4.3. Mapping between orchestrator failures (OF) and client failures (CF). Percentages are of the total number of injections performed for that given workload. Percentages of single-digit numbers are omitted for readability ©IEEE 2024.

	Deploy				**Scale**				**Failover**			
	NSI	HRT	IA	SU	NSI	HRT	IA	SU	NSI	HRT	IA	SU
No	**1617 (62.2%)**	**84 (3.2%)**	0	0	**1382 (54.5%)**	**77 (3.0%)**	0	0	**2652 (72.7%)**	**137 (3.8%)**	11	0
Tim	28 (1.1%)	1	0	0	40 (1.6%)	8	1	0	18 (0.5%)	11	2	0
LeR	109 (4.2%)	138 (5.3%)	4	5	432 (17.0%)	63 (2.5%)	0	0	59 (1.6%)	10	1	0
MoR	368 (14.2%)	**12 (0.5%)**	2	0	303 (12.0%)	**41 (1.6%)**	7	0	531 (14.6%)	**31 (0.8%)**	0	0
Net	14 (0.5%)	7	6	**107 (4.1%)**	28 (1.1%)	46 (1.8%)	10	0	8	48 (1.3%)	**40 (1.1%)**	1
Sta	**81 (3.1%)**	**4**	0	0	**81 (3.2%)**	**5**	0	0	**66 (1.8%)**	**8**	0	0
Out	**10 (0.4%)**	**1**	0	**1**	**8**	**1**	0	**1**	**7**	**2**	2	**4**

web server, which read a seed for random numbers from a *Volume* during the startup, and responded to clients with the result of random computations. Its Pods had CPU and memory resource requests and limits, and default priority. The web server was stateless and did not require coreDNS name resolution. The application is used to trigger orchestration activity, through the workloads defined in §4.2.2: a stateful application with complex topology would complicate the application failure patterns but not the orchestration system ones. The application client sent 20 requests/second for 30 seconds.

Campaign details

I performed a total of 8,782 injection experiments based on the campaign described in §4.2.2, targeting the communication between the Apiserver and Etcd to directly alter the stored state and efficiently trigger failures. One fault/error was injected in each experiment.

The results are summarized in Tables 4.3, 4.4, and 4.5. Table 4.3 describes the propagation of OFs to CFs. For example, the cell intersecting the column HRT and row `MoR` contains the number of `MoR` failures that caused HRT. Tables 4.4, and 4.5 divide failures by workload and injection type, with percentages of categories, e.g., 2.8% of the experiments are `Sta`.

Table 4.4. Statistics on orchestrator-level (OF) failures observed in fault/error injection experiments. ©IEEE 2024

WL	Injection	Perf.	Orchestration-level Failures (OF)						
			No	Tim	FeR	MoR	Net	Sta	Out
Deploy	Bit-flip	1563	1097	17	135	210	58	45	1
	Value set	900	484	12	111	172	70	40	11
	Drop	136	120	0	10	0	6	0	0
Scale	Bit-flip	1522	950	29	260	190	45	47	1
	Value set	872	387	17	224	161	35	39	9
	Drop	140	122	3	11	0	4	0	0
Failover	Bit-flip	2132	1610	13	5	424	33	42	5
	Value set	1288	972	18	64	130	62	32	10
	Drop	229	218	0	1	8	2	0	0
	$\sum$	8782	5960	109	821	1295	315	245	37
	%	100%	**67.8%**	1.2%	9.4%	14.8%	3.6%	**2.8%**	**0.4%**

Analysis of OF and CF failures

Finding 2 - System-wide failures

3.2% of the performed injections of one value propagated to a system-wide failure, despite the resiliency strategies. 24.2% of injections resulted in service under/over provisioning, 3.6% in service networking problems. ≈70% of performed injections have no effect because they are either i) detected and mitigated by the health checks, like heartbeats; or ii) mitigated by natural system behavior (e.g., the value is overwritten).

In the experiments, a non-negligible number (3.2%, last two columns of Table 4.4) of fault/error injections of a single bit-flip or value set resulted in `Sta` and `Out` failures. `Sta` failures were caused by i) a control plane overload due to uncontrolled replication of resource instances (e.g., Pods); ii) a Scheduler or Controller that was unable to obtain a leadership role and perform state changes; or iii) a failure or deletion of networking Pods. On the other hand, the causes of cluster outages were i) uncontrolled replication of resource instances; ii) misconfigured networking daemons that caused a global network outage; or iii) failed or deleted coreDNS Pods.

An incorrect value error may occur for any of several reasons, but particularly interesting is the case of injections affecting the serialization protocol.

Table 4.5. Statistics on client-level (CF) failures observed in fault/error injection experiments. ©IEEE 2024

WL	Injection	Perf.	Client-level Failures (CF)			
			NSI	HRT	IA	SU
Deploy	Bit-flip	1563	1386	132	5	40
	Value set	900	720	105	6	69
	Drop	136	121	10	1	4
Scale	Bit-flip	1522	1379	133	10	0
	Value set	872	772	91	8	1
	Drop	140	123	17	0	0
Failover	Bit-flip	2132	1989	132	9	2
	Value set	1288	1139	100	46	3
	Drop	229	213	15	1	0
	$\sum$	8782	7842	735	86	119
	%	100%	89.2%	**8.4%**	**0.9%**	**1.4%**

They usually caused the resource instance to become undecryptable and be deleted (see §2.1.4), but in some cases, the resource instance remained decryptable and wrong. Because of how the protocol works, an injection can move a value from one field to another, and a required field could remain empty and trigger failures.

To give an example of system-wide failure, in the table below there is an example of uncontrolled replication. Corrupting the data of a service caused containers to be spawned in an infinite loop, leading to an overload and system-wide outage. The same pattern was observed in [87] as the result of incorrect container labels.

Example of uncontrolled replication

A single-bit corruption of the labels that associate a Pod with a DaemonSet leaves the Controller unable to identify the Pods belonging to the DaemonSet. That causes new Pods to be spawned, in an infinite loop. The system is overloaded and all the cluster computing resources are filled up. The DaemonSet Pods have high scheduling priority, so they terminate all application Pods to claim resources. Eventually, the disk of the control plane Node can fill up, stalling Etcd.

The results indicate that 3.6% of injections resulted in service networking problems (column 5 in Table 4.4), 24.2% in service under/over provisioning

(column 3,4). ≈70% of faults/errors across the three workloads had no perceivable effect (first column in Table 4.4). Both the injections recovered and the ones not activated belonged to this set. I define an injection as activated when the injected resource instance is requested after the injection. The activation rate is 82%. I have no control over the activation of a single field.

Examples of system recovery include i) overwriting of the injected data field with a correct value that is still stored somewhere in the system (e.g., some ReplicaSet fields, which cause a ReplicaSet recreation or the *PodIP*, which is overwritten by the correct value sent by Kubelets) or ii) the corrupted data have no immediate effect but remain latent.

For example, some data-structures have a version number. If the Controller does not detect any change in the number, it does not process the instance, preventing the injected value from being used. However, a subsequent update of the version number (e.g., by another request) triggers the errors caused by the injected value. For example, several injections targeting the networking DaemonSets can lead to a `Sta` or `Out` if triggered. Injections classified as `No` mostly did not propagate to clients (see `No`-NSI cell in Table 4.3). However, some of them led to `HRT` client failure. Those cases could be attributed to the natural nondeterministic orchestration times.

A non-negligible number (10.8%, last three columns of Table 4.5) of injections impacted the clients. Figure 4.5 shows the z-scores of response times

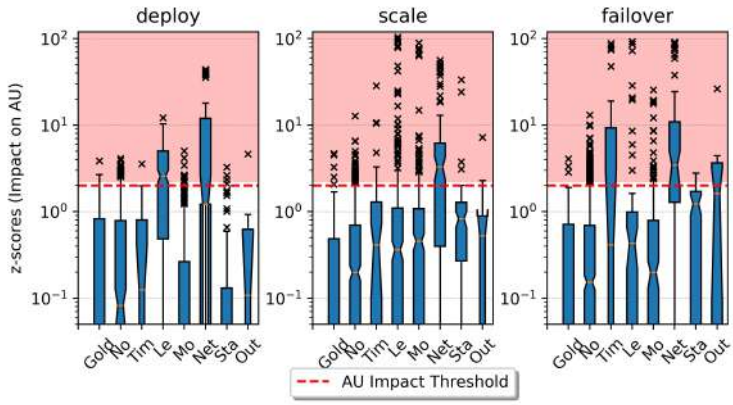


Figure 4.5. Impact on the application client measured through z-scores. ©IEEE 2024

observed by the clients for different orchestrator failures.

Timing failures generally had a limited impact on clients, but under the "failover" workload, they occasionally introduced a significant delay (reflected by a high z-score) because of control plane Pods' restarts. Scheduler restarts were caused by injections into the *nodeName* field of an existing Pod, which changed the value to a non-existing *nodeName*. Timing failures may cause delays in the orchestration times up to minutes.

Below, an example of this phenomenon.

Example of timing failure

The Scheduler detects a mismatch between the data in Etcd and its cache and, assuming a cache corruption, restarts. After a new leader Scheduler is elected (after 20 seconds, in the standard configuration), it starts scheduling Pods. The corrupted Pod remains pending for ≈50 seconds until the Controller deletes it and creates another one.

"Fewer resources" failures can have severe impacts on response times when the difference between the expected and used numbers of Pods is significant. I observed that part (≈40%) of MoR failures represent a negligible threat to clients because they are transient and involve little extra resource consumption. (The Pod number at steady state is correct, but the number of Pods spawned is greater than expected by less than three.) Interestingly, MoR failures can negatively impact the clients as well. When the system does not detect the resource overprovisioning for a service and uses fewer resources than allotted, a FeR and MoR failures are caused at the same time.

Example of undetected overprovisioning

When the *namespace* field of a Deployment is corrupted during the scale-up of the service application, fewer Pods are spawned, causing longer response times. Upon deletion of the resource instances, K8s starts reconciling a residual corrupted Deployment that is not even listed anymore, spawning Pods in the *terminating* state in an infinite loop. When the rates of terminated and created Pods become similar, the system reaches an equilibrium, but Etcd is filling up.

Service networking failures (Net) induced the majority of intermittent failures for clients (IA), and complete service outages (SU). Almost all SUs happened under deploy (see Table 4.3); the injections with index 1 during the deployment were "create" transactions, making the unwanted value changes barely detectable, unlike the injections in the following updates. Sta failures may or may not impact the application client response times, as said in §4.1.2, although the system eventually gets to a degraded state. Finally, Out failures in my data did not always impact response times because the service application does not require the *DNS*. This makes error propagation from the orchestrator to the clients difficult to identify.

I repeated the injections targeting the *critical* data fields (360 in total; see §4.2.4) in a cluster with three control plane Nodes with an Etcd replica on each control plane Node. The results showed no significant difference from the previous ones. The only component that actually works in a replicated fashion is Etcd, and values were injected before getting to it. A few additional experiments also showed that corrupting the data in Etcd at rest has a different propagation pattern from my injections because of the Apiserver cache. The cache is used intensively, and it is refreshed with Etcd data when needed, e.g., getting a resource instance. If the refresh does not happen before an update, the injected value in Etcd is overwritten, and a complete component restart may be needed to pick up the injected value. Furthermore, quorum reads mitigate corrupted values. In conclusion, i) corruption at rest is less likely to cause issues than errors that happen before a transaction; and ii) a corruption of the cache may overwrite a correct value on the data store if the right sequence of requests is triggered.

Critical field analysis

Finding 3 - Dependency relationships

51% of fault/error injection experiments that caused critical failures targeted the fields managing the dependency relationships among resource instances, revealing an inherent data weakness.

I analyzed the fields that caused the most severe failures when injected, i.e., Sta, Out, or SU. 360 injections were derived, which all affected the same 34

fields of different resource kinds. Out of them, 8 were related to metadata and 26 to technical specifications.

A subset of those fields is important because it constitutes the way in which K8s keeps track of the associations between multiple dependent resource instances of different kinds. These include: i) owner relationships with references to other resource instances, and ii) label relationships that use matching labels and selectors to create dynamic relationships. *labels*, *managed-by*, *targetRef*, and *ownerReferences* are metadata, while label selectors are specification fields. In total, 20 fields out of 34 belonged to this subset, representing 187 injections out of 360.

This finding is supported by an interesting real-world failure of a Reddit compute cluster [89]. In that case, the value of the role label of the control plane nodes changed from "*master*" to "*control-plane*" because of a K8s update. This relabeling led to 314 minutes of cluster downtime due to a system-wide network failure.

The injections that triggered uncontrolled replication of objects belonged to this category, revealing an issue in K8s: although these fields are important for the functioning and represent a risk because of the associated possible failures, there are not enough resiliency strategies in place to recover the system in case of errors. Other relevant fields (124 injections total) were *name*, *namespace*, and *uid*, which are the fields used by K8s to identify a resource and appear in its URL. The remaining fields included 5 related to networking (protocols, addresses, and ports); the replica number; and 2 specification fields of images and commands that prevent the start of critical Pods.

User error analysis

Finding 4 - User unawareness

The reconciliation of the observed cluster state and the desired state is postponed to a later time. If the state of the system diverges and never reaches the desired state because of failures, the user may be unaware of it unless proper monitoring alerts are set.

Figure 4.6 shows the number of injection experiments with failed user requests to the Apiserver (indicated by the *Error* label) as a fraction of the total

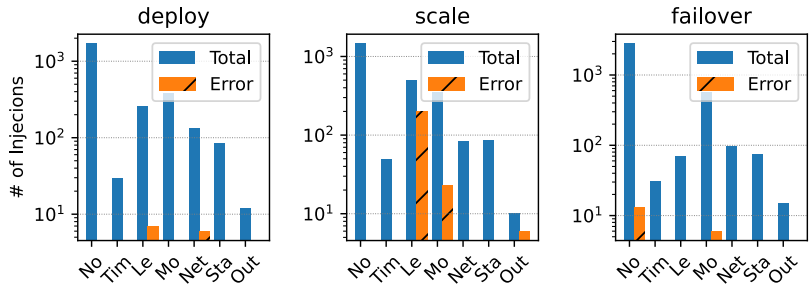


Figure 4.6. Number of total injection experiments vs. injection experiments in which the cluster user received an error (*Error*) in response to requests to the Apiserver. ©IEEE 2024

number of fault/error injection experiments, distinguished by orchestration failure types. For most fault/error injections that led to erroneous data store states and failures, the user did not receive any warning or error notification from the Apiserver. The “scale" workload shows that a significant number of experiments returned errors. The reason was that the workload generated multiple successive requests related to the same resource instances. In this context, a fault/error may compromise the resource integrity, triggering errors in subsequent requests.

In the case of the “deploy" and “failover" workloads, the majority of errors (29/32) were due to injections in the serialization protocol that prevented decoding and operations on the object. The reason for the small number of errors was the delayed reconciliation of the cluster state and the desired state on the data store. In fact, the Apiserver only acknowledges the requests for the modification of values on Etcd. However, this acknowledgment does not imply that the cluster state reached the desired state.

4.2.5 Comparison

Table 4.6 shows failures triggered by Mutiny compared to failures reported in the real world. Mutiny easily triggers errors related to logic, capacity, and control plane availability. For example, the uncontrolled replication analyzed in the example box in §4.2.4 can replicate the same pattern observed in [87], where container labels were not picked up, through the corruption

Table 4.6. Comparison between injections and the real world failures. **Bold** indicates what Mutiny can replicate, and *italics* indicates what is triggered by Mutiny and not present in the real world. ©IEEE 2024

Error	**Error Subcategories**
State Retrieval	State corrupted, erased, stale, unretrievable
Misbehav. Logic	**Wrong label**, **Wrong replica value**, **Request rejected**, **Lost update**, **Controller loop not executed**, **Relationship broken**
Commun. Problems	Connection delay, **Wrong IP address**, DNS resolution delay, **DNS not resolving**, Uneven load balancing, Endpoint delete after Pod kill, **Routes dropped**, ***New Nodes' routes not configured***, ***Routes not updated***
Capacity Exceed	**Overcrowding**, **Cluster out of resources**, **Worker nodes cannot join**, **Worker nodes unhealthy**
CP Availab.	**CP Pods crash loop**, ***CP Pods hang***, CP Pods deleted, **CP overload**
Local to Nodes	Kubelet delayed, Container runtime failure, **Pods not ready**, **Image Pull Error**, Slow/throttling
Failure	**Failure Subcategories**
Cluster Outage	**Cluster-wide networking drop**, Cluster-wide networking intermittent, **Massive Service Deletion**, **DNS resolution failure**
Stall	**Control Plane stuck**, **Control Plane slow**, Control Plane quorum unreachable, **New Services network not configurable**, ***New Nodes network not reconfigurable***
Service Networking	**Service Networking Drop Permanent**, **Service Networking Drop Intermittent**, Service Networking Delay
More Res.	**Pods not deleted**, **Too many Pods created**, ***More Pods Transient***, **More Resources Per Pod**
Fewer Res.	**Pods deleted**, **Pods not created**, **Pods crashloop**, **Fewer Resources Per Pod**
Timing	**Pods' Creation Delayed**, ***Pods Restart***

of a field of a ReplicaSet.

Conversely, it fails to trigger several errors local to the worker Nodes, because those errors are mainly due to local configurations and underlying software (e.g., kernel, runtimes) problems. For example, it falls short in inducing delays caused by DNS resolution, connection errors, arbitrary numbers (different from 1 and *all*) of unhealthy Nodes, and transient and intermittent network failures in general.

Nonetheless, almost all failure subcategories can be covered. At any rate, my aim is not to create a one-size-fits-all injector but rather to provide a framework capable of triggering unforeseen error patterns to test the system response and provide insights that can be used to devise methods for mitigating or recovering from potential failures.

4.3 Orchestrator Timing Analysis

The previous sections proved that misconfigurations or faults can cause unexpected load on orchestrators, overloads, and service disruption.

This section analyzes how orchestration times vary in error-free conditions when an increasing orchestration load is applied to the orchestrator, in order to answer the following research questions (RQs):

Q1: *Are commonly used orchestrators able to fully prioritize services by their criticality when the orchestration load increases?*

Q2: *What are key tuning knobs that can affect the orchestration times?*

Q3: *What are the sources of delay of orchestration times?*

4.3.1 Experimental setup

I used two setups: i) a *cloud cluster*, ii) an *edge cluster*. The cloud cluster included 5 virtual machine (VM)s (8 cores, 8 GB RAM each) running on multiple servers (Intel Xeon E5-2630L, SCSI storage). One of the 5 VMs is the control plane VM, and runs on a dedicated server. The edge cluster included a control plane workstation (Intel i7-4790, 16 GB RAM, SATA HDD) and 4 worker nodes, one of which is an *embedded device* (Raspberry Pi 4, 4 GB RAM), and 3 VMs configured as above. All nodes run Linux v5.15. Both clusters ran K8s v1.29 (`kubeadm` default configuration) with `Flannel` Container Network Interface, and `containerd` v1.7 as container manager.

4.3.2 Experimental method

Hereon I focus the analysis on the "failover" command since it involves multiple orchestrator subsystems: health monitoring, replication management, pod scheduling, creation, and networking. In K8s, a failover subsumes both deployments and scaling and Figure 4.7 shows that results are similar for the other orchestration commands and also for Docker Swarm.

To perform a failover experiment, I triggered the respawn of one or more pods, which were initially deployed on a set of failed Source worker Nodes (SN), to a set of Destination worker Nodes (DN). The pods to be respawned included 1 pod of a *test service*, which is the high-priority (i.e., critical) service, while the others are *nginx* web servers, chosen because of the minimal startup overhead. The test service was a stateless UDP echo server that responds to

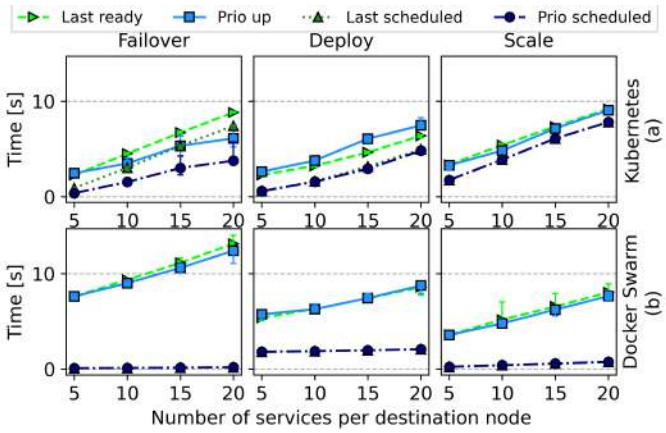


Figure 4.7. Orchestration times for failover, deploy, and scale commands, under increasing load with 2 destination nodes. "*Prio up*" is when the test service started to respond to client requests, "*Last ready*" is when the last pod was ready.

a client. The client sends an UDP packet each 50 *ms*. The client waits for the response to each packet sent and logs the timestamps once received. The client was deployed on a node not involved in the failover.

The test service was first placed on a *SN*. After the system stabilization, the failure of all *SN* was simulated through *NoExecution taints*[6] in K8s, and custom labels in Docker Swarm[7]. Through placement constraints, the test service was forced to respawn on a VM (in the cloud cluster) or the embedded device (in the edge cluster). The other Pods were free to be respawned on any *DN*. The test service was configured to have the highest *scheduling priority*[8]. The container images were already present (i.e., *pulled*) on the *DN* to exclude the variability of pulling times [97]. During each experiment, the logs of the control plane components were collected to obtain the event timestamps.

[6]Documentation available at `https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/`

[7]Documentation available at `https://docs.docker.com/config/labels-custom-metadata/`

[8]Documentation available at `https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/`

Experiment parameters Recalling that the default maximum number of Pods that can run on a node is 110, I set 60 as a reasonably high number of Pods to be orchestrated simultaneously for my testbed with 4 worker nodes. Previous works [24, 13] used similar numbers. To this aim, I ran the experiment with 5, 10, 15, and 20 Pods to be respawned for each destination worker node. I ran the experiment for a DN size of 1, 2, and 3. Hence, with 3 DN, the total number of Pods is respectively 15, 30, 45, 60. The Pods were grouped into Deployments of two Pods each. For each combination $(DN, pod\_number, setup)$, 30 experiment repetitions were performed for statistical purposes.

4.3.3 Experimental results

Answers to RQs

A1: No, commonly used orchestrators are not able to prioritize the critical services
A2: Rate-limiting thresholds, controller parallelism level, datastore, and the container manager are the key factors that affect orchestration times
A3: The sources of delay are asynchronous event management, datastore unpredictable operations, request queuing, scaling effects, and implementation choices

A1 - Service Prioritization

Figure 4.7 already showed that K8s and Docker Swarm are not able to prioritize a service. Further, Figure 4.8 shows the timing of events of interest for K8s. The time in the figures is the difference between the timestamp of the event and the one sampled before the *taint* request.

The results show an approximately linear increase of times as the number of Pods increases. In the cloud cluster, the contributions in terms of time spent on the control plane and worker nodes look roughly balanced. The time for the last Pod to become ready (the *"Last ready"* event) was ≈2 seconds after the *"Last scheduled"* event, which suffered from a steep slope when the Pod number increased up to 60 (i.e., 3 DN, 20 Pods per node). Despite the similar

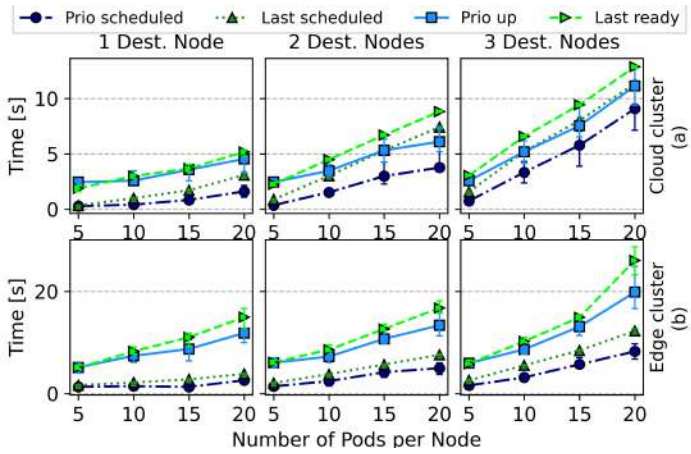


Figure 4.8. Timing of the K8s orchestration events during the failover experiment with increasing activity. Error bars represent the interquartile ranges. "*Prio up*" is when the high-priority Pod reacts to client requests, "*Last ready*" is when the last Pod was recognized as ready by the control plane. The "*scheduled*" event is when the control plane decides the worker node to spawn the Pod, and is the last event regarding the control plane.

trends in the edge cluster, spawning the Pods on the embedded device was the main bottleneck under heavy load.

Although a higher scheduling priority was set for the high-priority service, K8s did not prioritize it because the priority only affected the scheduling phase. As shown in Figure 4.8 (see *"Prio up"* event), the high-priority service Pod took as long as 20 seconds to respond in the worst case. Even worse, it was not the first ready Pod: as shown in Figure 4.13, when 20 Pods were scheduled, the first Pod was ready in ≈ 2 seconds.

One of the reasons for such behavior is the queuing delay that can affect event handling when the system load increases. Most of the queues in K8s are managed through rate-limited FIFO (e.g., in the Controller) or coarse-grained Fair Queuing (e.g., in the Apiserver) policies, preventing a full prioritization of a request more urgent than others.

Similarly, Docker Swarm is not able to prioritize a service over the others during orchestration. Indeed, it also uses FIFO queues and has no feature similar to scheduling priority.

One can argue that those times are just a matter of creating Pods. How-

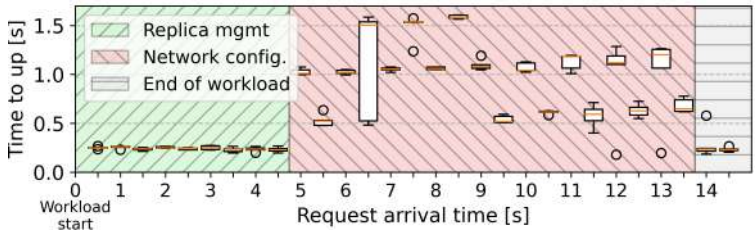


Figure 4.9. In K8s, time to start forwarding traffic to a running pod since a network configuration request, for different request arrival times since the start of concurrent workload. In the first seconds, the workload stresses the subsystem related to replica management, which creates the pods required to restore all the services with the desired number of pods running. Thus, the network configuration for the high-priority service is relatively fast. Next, during the red time window, the workload stresses K8s networking management subsystem, which must configure the network for all the pods created in the previous phase. Thus, the network configuration for the high-priority service becomes slower.

ever, I show that, even assuming an already running replica, the time to bring up the service can be up to 6 times longer under a heavy orchestration load. Figure 4.9 shows the results of an experiment where the time required to configure the network of the high-priority (already running) service is plotted against its request arrival time. When K8s components managing networking are under pressure because of concurrent low-priority workload (i.e., failover of 60 Pods), the time to bring up the service increases from $\approx 0.25s$ to $\approx 1.5s$. Hence, any worker node optimization to reduce start times, like container caching, cannot reduce this delay.

A2 - Key Tuning Knobs

Queue Scheduling I focus on three key parameters of the K8s configuration: *i) rate limit thresholds* for requests directed/coming to/from the Apiserver, *ii) weighted fair queuing request scheduling* configuration in Apiserver, and *iii)* the number of concurrent control loop threads.

Disabling request rate limits threatens production environments since orchestrator components may fail at random [98]. Similarly, exempting a set of requests from the weighted fair queuing can cause overloads and starve other requests. Nonetheless, I performed additional experiments on the cloud clus-

ter with request rate limits between the Controller and the Apiserver disabled and the request belonging to the high-priority service configured to be exempt from weighted fair queuing. The aim was to understand the impact of those key parameters on orchestration times.

Figure 4.10 shows that, when not rate-limited, the control plane scales better (see *"Prio scheduled"* and *"Last scheduled"* events) than the worker nodes (see *"Prio up"* and *"Last ready"* events), where most of the time is spent to spawn Pods. The delays on the worker nodes are amplified as the Pod number increases, becoming a bottleneck. Hence, the Kubelet must be designed to prioritize high-priority Pods. I observed a similar behavior in Docker Swarm (not reported for brevity), which has no rate-limiting parameter.

In particular, Figure 4.11a and Figure 4.11b show the histograms of runtimes of the control loops of K8s Controller, by enabling/disabling rate-limiting. The runtimes show that default K8s Controller used for the experiment in §4.3.3 handled requests as soon as possible until it started throttling them to respect the rate limit (i.e., 20 request per second by default). The *"Prio scheduled"* event in Figure 4.8 was affected by throttling, which does not account for priority.

Similar considerations hold for rate-limiting parameters and synchronization periods of Scheduler and Kube-proxy. The Scheduler never hit the rate limit and contributed for a few milliseconds to orchestration times in my experiments. Conversely, the Kube-proxy determined the orchestration times under heavy load in Figure 4.9 because of the synchronization period of the

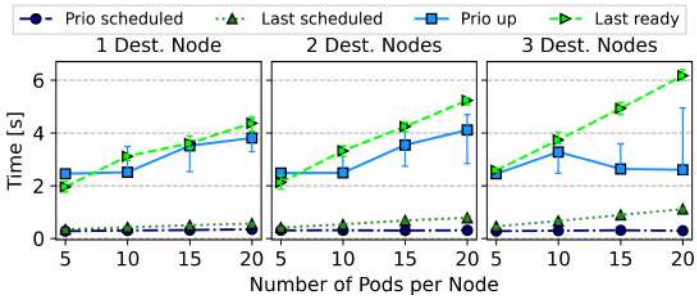


Figure 4.10. Failover experiment in cloud cluster, with Controller configured without rate limiting and high-priority service exempted from fair queuing in the Apiserver.

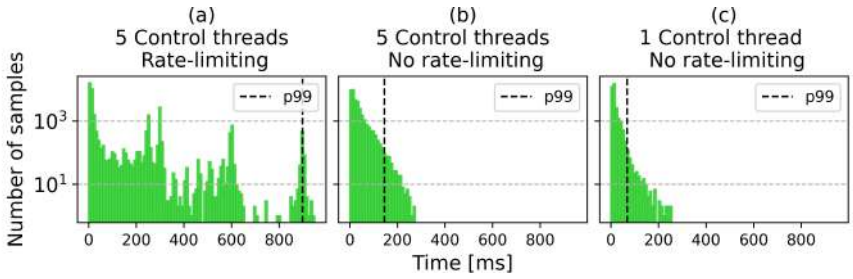


Figure 4.11. Control loop runtimes in the cloud cluster. 5 control loop threads and rate-limiting is default configuration.

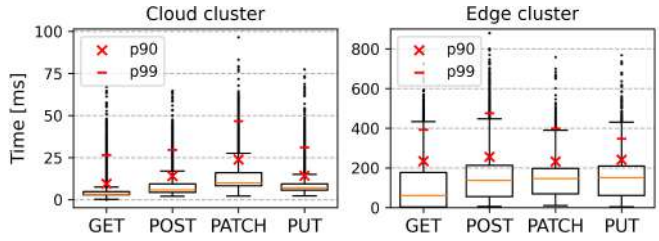


Figure 4.12. Latencies for the Apiserver replies in the two clusters divided by HTTP verb. The different timescales are caused by Etcd, which is much slower in the edge cluster. $p90$ and $p99$ are, respectively, the 90^{th} and 99^{th} distribution percentiles.

routing tables (i.e., 1s by default).

The K8s Controller has a control loop for each resource type, which spawns multiple control loop threads (5, by default). I further investigated the effect of the number of control loop threads with rate-limit disabled. By setting only a single control loop thread in the Controller (see Figure 4.11c) results show that the 99^{th} percentile of loop times are drastically lower than Controller with increased parallelism. The root cause is the asynchronous management of events. Indeed, the higher parallelism causes greater interferences due to the pressure on the orchestrator components because of the uncontrolled rate of handled requests. In addition, the asynchronous event management can cause priority inversions [99]: when a single control loop thread is free, an incoming request is immediately processed, and if there is a closely upcoming high-priority request, it is queued.

Datastore During previous experiments, the orchestration times related to the control plane events significantly differed between the two clusters. Figure 4.12 shows the distribution of response times of the Apiserver, which interacts with Etcd.

I observed that datastore response times affect the control plane timing behavior when not rate-limited, since Etcd is accessed by each action of an orchestration command chain (recall §2). For example, a failover in K8s requires at least 5 writes. Indeed, the control loop times (Figure 4.11) include one or more requests to the Apiserver. Response times can present orders of magnitude of variability due to load, underlying physical disk technology (e.g., HDD or SSD), and consensus protocols when replicated [100]. Furthermore, datastore response times have a skewed distribution, characterized by long tails exceeding the $p99$. Repeating the experiments with an in-memory Etcd in the cloud cluster, I obtained median latency reductions of $\approx$50% ($\approx 6ms$) for POST, PATCH, and PUT requests with a p-value of 10^{-7}.

Container manager I assessed the impact of the container manager when spawning multiple Pods on a single worker node. I compared `containerd` (K8s default choice) against `dockerd` (v24.0.7) in the cloud and edge clusters. The *DN* was a VM in the cloud cluster, while it was the embedded device in the edge cluster.

Figure 4.13 shows all Pod ready times distributions. As the Pod number increases, the time distributions present an increasing mode, and the ready time is clearly delayed for each Pod. The Pod spawn requests are served as soon as they arrive at the worker node, causing resource contention for container creation and configuration. This is particularly evident for `dockerd` on the embedded device. Pods including multiple containers add further delays.

A3 - Sources of Delay

I here summarize the sources of delays according to the insights of the timing analysis performed.

• **Indirect delays in asynchronous event management.** The state change events are often handled asynchronously, i.e., as soon as possible, in both worker nodes and the control plane. Asynchronous event handling creates interferences and resource contention that delay high-priority requests. Moreover, the requests can be throttled to respect possible rate limits. I define

indirect delays in both the control plane and the worker nodes the delays caused by asynchronous management, including resource contention, pressure on shared components, throttling, and priority inversion.

• **Datastore delays.** Orchestrators generally store the current and desired cluster state in a datastore [17] (Etcd for K8s), which is frequently accessed during orchestration commands. Commonly used datastores (like Etcd) do not distinguish the incoming requests to accommodate services SLOs and prioritize high-priority services. Replicated datastores can increase the delays because of the network delays that can affect the consensus protocol messages [100].

• **Queuing delay.** An event is generally handled through a chain of cascading actions. Events can suffer from queuing delay for each action of the cascading sequence. Indirect delays and unpredictable datastore response times can cause order inversions in the components' queues, amplifying the orchestration times for high-priority services.

• **Scaling delays.** Algorithms implemented in orchestrators may have runtimes that depend on the number of resource instances in the system, increasing the orchestration times as the system size increases. For example, the K8s Scheduler runtime increases with the number of worker nodes to evaluate (up to a configurable maximum) to place a Pod.

• **Implementation-related delays.** Delays due to the underlying software stack must be considered. Examples are *i)* code optimizations baked in the code to improve the scalability at the cost of response times, e.g., request

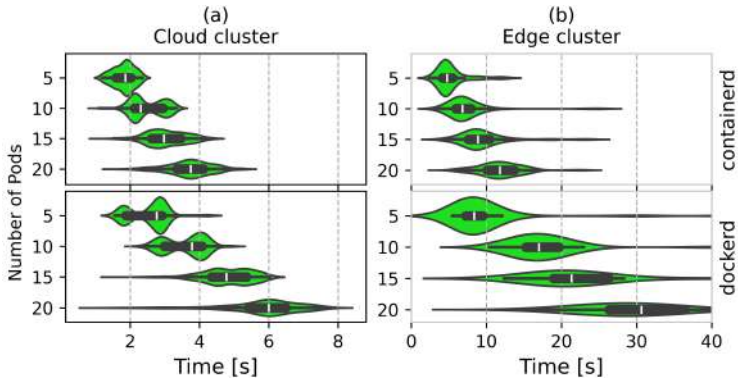


Figure 4.13. Pod ready times distribution with variable Pod number (1 *DN*).

batching; *ii)* communication over stacks without timing guarantees like HTTP over TCP; *iii)* language-runtime-related delays, e.g., the unpredictable routine scheduling of Golang for K8s.

4.4 Real-time Containers Timing Analysis

This section analyzes the timing behavior of real-time containers based on Linux when intense stress is co-located on the same node. The aim is to determine whether Linux-based containers can host real-time critical applications. I only study freedom from timing interference, overlooking freedom from failure, as Linux containers are known to suffer from reduced failure independence due to the shared underlying kernel [19].

4.4.1 Experimental setup

The experimental setup included three nodes of varied capabilities, representative of a heterogeneous industrial scenario: i) a Raspberry Pi 4B, ii) an HP Z230 workstation (Intel i7-4790, 16 GB RAM, SATA WDC WD10EZEX-60M HDD), and a iii) a DELL Optiplex (Intel i7-13700, 16 GB RAM, SN740 NVMe WD 1TB)

All the boards and workstations were configured for real-time: Linux was patched with the `PREEMPT_RT` patch and the patch in [101], power optimizations (including C-states and P-states) and frequency scaling were disabled at the hardware level, as well as hyperthreading. Debug options for real-time were disabled. All Linux kernels are set to version v5.15, and their configuration followed the kernel configurations advised by the official `PREEMPT_RT` configuration guide [9] (e.g., debug options disabled) and other general configurations advised for real-time systems from the Xenomai guide[10].

The software setup included Docker v24.0.5 using runc v1.1.11. The Docker daemon was configured to have all the cpu-rt-runtime available in the configurations with `CONFIG_RT_GROUP_SCHED` enabled.

[9] `https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/preemptrt_setup`

[10] `https://v3.xenomai.org/tips/x86/common/`

4.4.2 Experimental method

I ran *rt-app* as a containerized benchmark application performing CPU, memory, and network activity to measure the interference from the co-located stress generated by *stress-ng*. Rt-app is an application that can create synthetic real-time workloads composed of one or more periodic threads with a configurable behavior in terms of phases of input/output (I/O) activity, memory accesses, CPU operations, and so on. Each thread logs in a pre-allocated memory area (avoiding disk interferences in the measurements) information including the measured activation latencies and slack times. *Stress-ng* is an application that allows spawning stressor threads to stress specific hardware/OS subsystems. Stress-ng offers more than 30 types of stresses belonging to the following categories based on the subsystem targeted: CPU, cache, memory, networking, I/O, interrupt, filesystem, and OS. A stress-ng thread continuously performs an action that depends on the selected stress type.

I ran rt-app pinned on one core, configured to execute a single thread scheduled with a fixed priority of 95. Rt-app was modified to allow network activity instead of standard I/O. The rt-app thread was periodic and had an execution loop that sequentially: *i)* wrote to the main memory, *ii)* ran CPU-bound operations, *iii)* slept, and finally iv) sent a result over the network interface. I set this loop to emulate a sequence of memory accesses to read inputs, computation, and sending of the results, which is representative of a variety of control loops and other cloud/edge services.

Each experiment was composed of a continuous run divided into sequential phases. Each phase was composed of 60 seconds of cooldown followed by 60 seconds with a co-located stress running. During each phase, a different stress type was applied. I collected the thread activation latency and the slack time gathered by rt-app along the run.

4.4.3 Experimental results

Experiment set 1 This set of experiments is a sensitivity analysis with an increasing stress intensity.

The rt-app thread had a period of 10 ms and an execution loop that sequentially: *i)* wrote 8192 bytes to the main memory, *ii)* ran CPU-bound operations for 1.2 ms, *iii)* sent 1024 bytes over the network interface. I applied 11 stress types: *cpu, udp, hdd, io, netdev, open, fork, memcpy, cpu + hdd, cpu*

+ udp, udp + io. The last three options combine stresses belonging to the three categories I/O, network, and CPU, to detect possible amplification or attenuation effects. I ran the experiment on the HP Z230 workstation for an increasing number of stressor threads: 1, 2, 4, 8, and 16. The results are presented in Figure 4.14.

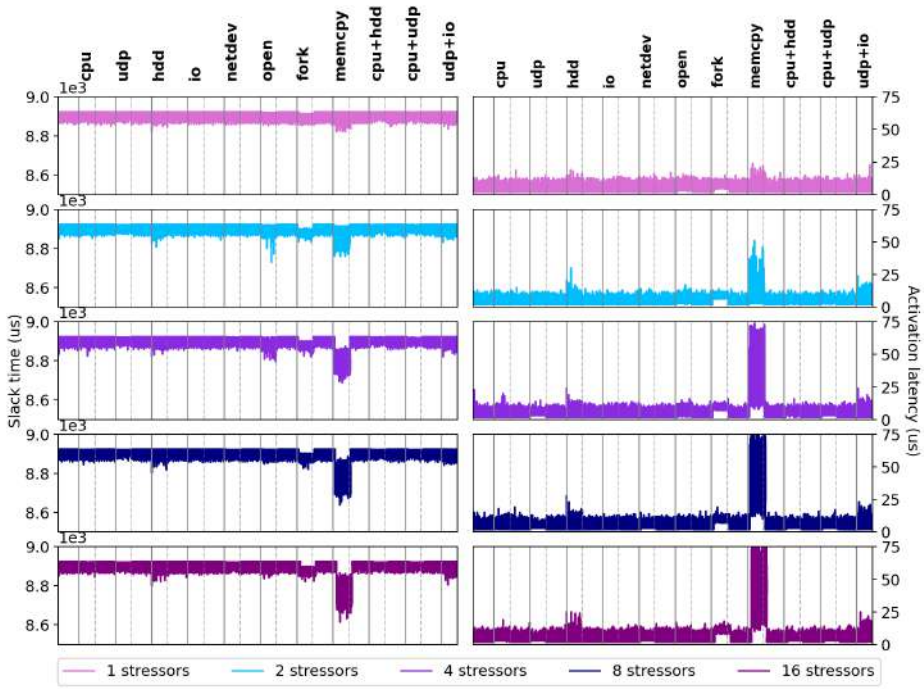


Figure 4.14. rt-app latencies measured on PREEMPT_RT (on HP Z230) with an increasing number of stressor threads (1, 2, 4, 8, 16 concurrent stressors)

Figure 4.14 shows how the effect of stressors on the slack time and activation latency depends on the number of concurrent stressors.

The results show a non-negligible interference on activation latency and slack times due to shared resources (CPU cache, network, memory). In particular, the *hdd* stressor caused higher activation latencies for those two configurations. Similarly, the *memcpy* stressor had the worst impact on rt-app. Hence, the *memcpy* stressor evicts cache lines of *rt-app*, causing both high activation latencies and reduced slack times.

A drop in performance, especially due to *memcpy* stressor, is evident with 4 or more concurrent stressors, i.e., when there is one stressor per physical CPU core. Indeed, in this case, stressor threads run on the same core as the rt-app real-time thread, causing L1 cache evictions.

Experiment set 2 This set of experiments compares the different nodes and configurations. On each node I used three configurations: i) *isolcpus* with interrupt redirect, ii) cgroup *cpuset*, and iii) thread pinning. *Isolcpus* is the most conservative configuration: it excludes the selected CPU from the Linux scheduler at boot times, redirects the interrupts to avoid the core on which rt-app is running, and uses the *NO_HZ_FULL* parameter. *Cpuset* is a hybrid solution, in which a group is created dynamically to reserve one core only for the execution of rt-app. Thread pinning instead only pins rt-app to a core, while relying on the Linux scheduler to share the core with other threads.

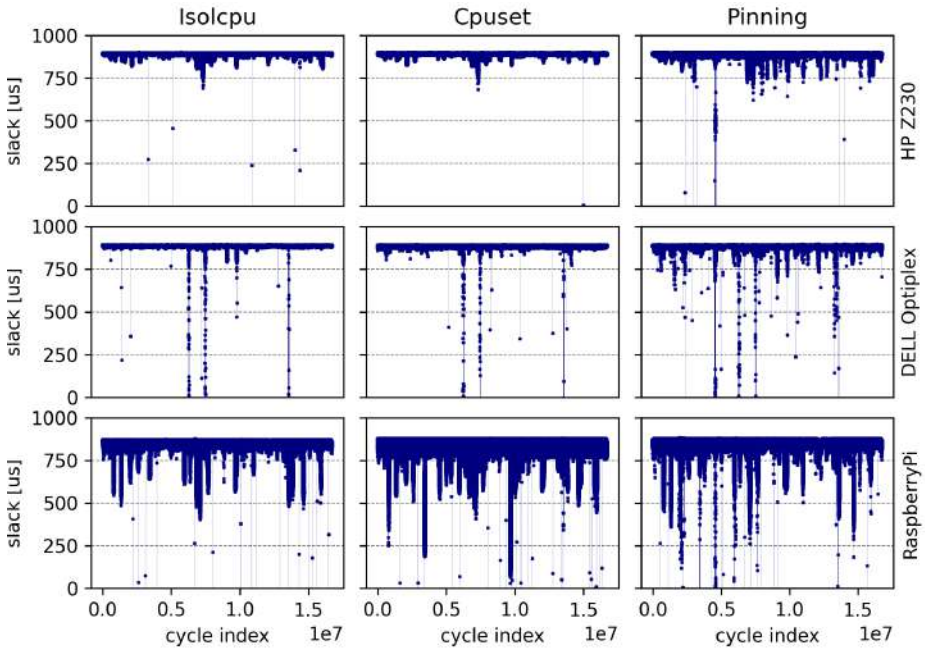


Figure 4.15. rt-app slack times for different hardware/software configurations. During each experiment, all the available stress types were applied sequentially.

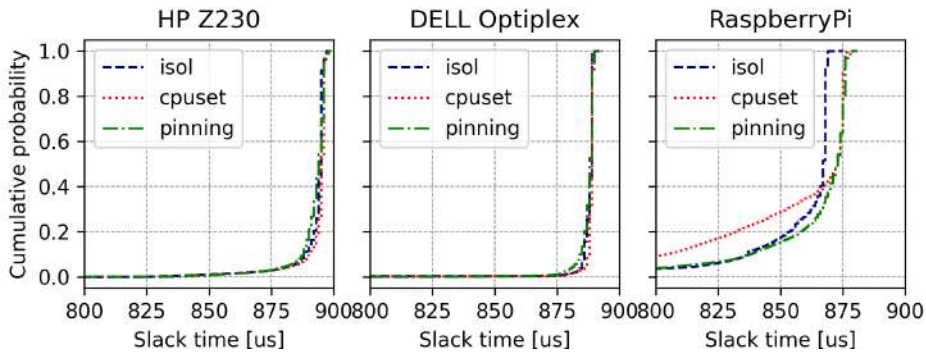


Figure 4.16. Cumulative distribution function of the rt-app slack time for different hardware and software configurations

The rt-app thread had a period of 1 ms and an execution loop that sequentially: *i)* wrote 8192 bytes to the main memory, *ii)* ran CPU-bound operations for 50μs, *iii)* slept for 50μs, and finally iv) sent 1024 bytes over the network interface. The parameters were selected to understand whether Linux can host real-time critical applications with tight timing requirements like 1 kHz frequency. The stress types applied were all the ones available in stress-ng (with 16 stressor threads), for a total of approximately 6 hours for each run.

The results are presented in Figure 4.15 and 4.16. The figures show that in all experiments there were deadline misses. Due to the low happening rate, I was not able to track the root cause of those deadline misses. Nonetheless, investigations through the *osnoise* tracer [102] hinted that the outliers should be caused by the hardware. Similarly, I could not determine a statistically significant correlation between isolated deadline misses and stress type.

Anyway, the results make clear that the hardware impacts the times more than the configuration, which can anyway guarantee some degree of improvement. Indeed, the Raspberry Pi shows drastically worse cumulative distribution functions for slack times, as it suffers more from co-located stress. Furthermore, different nodes suffer somewhat differently from co-located stress, showing a low correlation for multiple stress types among the nodes.

4.5 Discussion

Systematic resilience tests [11] Currently, companies use well-known techniques for testing services' resiliency (e.g., Chaos engineering), but there is limited understanding of how to test the resiliency of the orchestrator itself. At the moment, the responsibility for setting up proper monitoring alerts to detect failures and enforce manual mitigation belongs to cluster operators. Sometimes custom code is employed to prevent past failures from happening again; for example, validating a namespace deletion can prevent accidental deletion of a non-empty namespace together with all its Pods.

However, a post-incident manual procedure or code customization cannot be the answer to a trend that sees the use of K8s as a cloud OS for critical scenarios with tight non-functional requirements.

I argue that systematic orchestration resiliency tests should become an integral part of the development process to get quantitative metrics of resiliency strategies in place, similar to what is done for deployed services with Chaos engineering. A set of injectors, including Mutiny, can be used in testing clusters and/or in Chaos engineering processes to systematically evaluate the response to orchestration errors under realistic workloads, train the human cluster operators, and improve handbooks containing procedures to follow when a failure occurs.

Performing injections with Mutiny and real workloads would trigger new failure patterns not reflected in my experiments, due to the simple workloads that involved a limited set of resource kinds. For each critical failure pattern, appropriate and systematic countermeasures should be designed before deployment in production environments. Mutiny can be used to conduct a log analysis, check what K8s logs under injection, and possibly improve the logging when no traces of failures/errors are found. Furthermore, K8s auditing (currently a beta feature[12]) can be used together with injections to improve the cluster observability.

The proposed fault-injection methodology can be ported to other orchestrators, as they all share common architectural principles. Indeed, the study in

[11] This paragraph is taken from "Mutiny! How Does Kubernetes Fail, and What Can We Do About It?" ©IEEE 2024 (see author's publications section), with adaptations for this thesis.

[12] Additional information avaialable at `https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/`

[103] compares several orchestration frameworks (e.g., K8S, Docker Swarm, Mesos, Aurora, Marathon), observing that they all keep applications in their desired state by comparing the monitored and desired states [17] and rely on a data store (e.g., Etcd, Consul, Zookeeper) for storing the states.

Withstanding high loads Capacity-related system-wide failures were particularly troublesome in both real-world failures and fault injection experiments. These findings are not specific to K8s: they are an inherent threat to orchestrators because they manage resources at a huge scale. Furthermore, the timing analysis revealed that even keeping the system within a manageable load and without any error, transient high load conditions cause orchestration times up to tens of seconds. Such orchestration times are detrimental for services requiring strict SLOs including deadlines or availability. Experiments in Docker Swarm showed that also other orchestrators share with K8s common problems, mainly due to fully asynchronous even management that causes resource contention.

From this perspective, through injections or intense workloads, high-load conditions should be tested to determine the behavior and design proper countermeasures for critical services, mitigating the sources of delay.

Interferences on the nodes When a cluster is overcrowded, real-time applications can suffer from co-located stress. A benchmark should be used to understand the application behavior in the presence of faulty neighbors.

My experimental results showed that, in my setting, consolidating applications using Linux and general-purpose commercial-off-the-shelf (COTS) hardware cannot meet the requirements of real-time critical applications with high activation frequency. Indeed, the systems in the experimental setup considered did not have any hardware/software features to protect shared resources like cache or memory bandwidth, causing several deadline misses in runs of only a few hours. The authors of [104] also arrived at a similar conclusion, discouraging Linux containers for critical applications.

However, the real-time community is dedicating increasing efforts to tame hardware and software interferences even in COTS and Linux-based systems through software solutions [105, 106] or hardware features like Intel MBA. Therefore, Linux-based systems have been providing increasingly strict real-time guarantees. When deciding where to place a pod, the orchestrator must

be aware of *ad hoc* features to determine if a worker node can provide an assurance level suitable for the pod criticality.

Anyway, although the real-time community is considering running hard real-time tasks on Linux and COTS hardware, the dependability community is more skeptical. In fact, Linux is overly complex, and containers provide no failure isolation: if the kernel fails, all the tasks scheduled on it fail.

Advices for cluster managers [13] Since a single error can disrupt an entire cluster and large clusters can induce high orchestration times, cluster managers should prefer multiple clusters rather than a reduced number of high-scale clusters. Cluster managers should set and test upper resource limits in terms of Pod resources, number of Nodes in the cluster, request rates, and number of spawned Pods. *Namespace* features can limit resource counts and quotas[14] to somehow partition different tenant/service types and contain failures and transient load spikes. Cluster managers must be aware of the default parameters and the software specification in non-nominal conditions. Mechanisms like *MaxUnavailability*, *MaxSurge* (i.e., maximum Pod number that can be created over the desired one), and backoff timers should be tuned thinking at failures, despite slowing down daily operations. Stricter checks should be enforced: e.g., scaling of coreDNS to 0 should be denied, while adding the *MaxUnavailability* parameter could prevent outages. From a security perspective, access to Etcd must be strictly guarded by authentication and firewalls. K8s configures Etcd with client authentication, but not rarely administrators directly connect to Etcd. K8s security features can reduce the attack surface from unauthorized users, but cannot prevent errors generated in the authorized clients, e.g., Apiserver or Controller.

[13] This paragraph is taken from "Mutiny! How Does Kubernetes Fail, and What Can We Do About It?" ©IEEE 2024 (see author's publications section), with adaptations for this thesis.

[14] Documentation available at `https://kubernetes.io/docs/concepts/policy/resource-quotas/`

Chapter 5

Design of a Mixed-Criticality Orchestrator

We are engineers, not scientists. We have to get things working eventually. Either it works or it doesn't.

Marcello Cinque

This chapter introduces principles and architectures to build an orchestrator that supports real-time and mixed-criticality services, implementing the concepts of diversified orchestration introduced in chapter 3 to tackle the issues highlighted in chapter 4. To address the issues, the design is intended to be holistic: it accounts for the concepts of node assurance and pod criticality in all the subsystems from the ground up, involving all the phases of service orchestration: when orchestrating, when placing, and when running on the worker nodes, represented in Figure 5.1.

First, section 5.1 introduces the design principles of SLO-aware components and proposes three possible designs for an SLO-aware mixed-criticality orchestrator (i.e., *when orchestrating* and *when placing*).

Then, section 5.2 shows how critical pods can be built and how they can run on the worker nodes (i.e., *when running*), introducing the concept of partitioned containers, which are containers designed for critical services.

Finally, section 5.3 provides details about *Ulysses*, which is my K8s-based prototype, focusing on the SLO-aware Controller, SLO-aware Scheduler, and SLO-aware Kubelet.

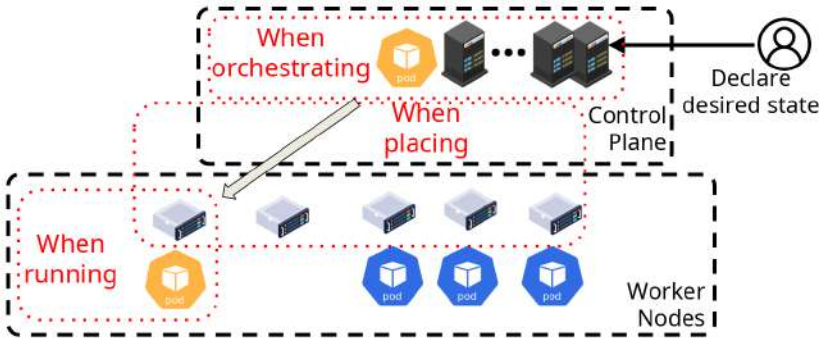


Figure 5.1. Orchestration aspects considered in the design of a mixed-criticality orchestrator.

5.1 Architectures for Mixed-criticality Orchestration

Chapter 4 showed that orchestration times and failures directly affect the services SLOs, thus some services may require differentiated and/or prioritized management. Without loss of generality, from now on I establish a one-to-one relationship between service criticality and SLOs, assuming that critical services also have stricter SLOs (like availability, failure probability, or probability of meeting response times).

This section proposes architectural designs and principles for mixed-criticality orchestrators derived from the key takeaways of chapter 4. In particular, section 5.1.1 provides some guidelines to drive the implementation of resilient orchestrators. Section 5.1.2 provides design principles to implement SLO-aware orchestrator components that mitigate the sources of delay identified in §4.3.3. Next, section 5.1.3 presents three possible architectures for SLO-aware orchestration systems, which leverage SLO-aware components.

5.1.1 Guidelines for resilient orchestration

Using the takeaways from the failure analysis in chapter 4, this subsection provides some guidelines to improve the fault tolerance of orchestrators.[1]

[1]This subsection is taken from "Mutiny! How Does Kubernetes Fail, and What Can We Do About It?" ©IEEE 2024 (see author's publications section), with adaptations for this thesis.

Runtime verification Since the orchestrator workflow can be modeled as a DAG and the services can be modeled with state machines, a nominal behavior of the system can be derived. Then, methods like model checking and runtime verification can be used to validate data exchanged between components and detect anomalies. Hence, circuit breakers must be systematically designed to prevent failure propagation when an anomaly is detected, especially for resource kinds that can cause overloads. Furthermore, when writing controllers that manage the replication of resource instances, the code should be designed to be resilient to a variety of faults/errors.

Critical fields The design of a resilient orchestrator must foresee *ad hoc* resilience techniques for critical resources and fields. For example, in K8s there is a massive use of labels because of their flexibility in grouping and selecting resources. I showed in §4.2.4 that this flexibility comes at the expense of resiliency because it is hard to validate their custom values. Recall the Reddit [89] failure discussed earlier, in which a single label tore down the entire cluster network. Hence, updates to critical fields and resources (e.g., control plane Pods or Nodes) should be logged. Upon change, the system behavior should be monitored to detect any degradation of the system's health, and changes to critical fields should be rolled back. Simple data redundancy mechanisms, like redundancy codes on critical fields, can protect the cluster from hardware faults with a negligible overhead in terms of resource usage (the critical fields identified in K8s are $< 10\%$ of total).

Data validation It is not enough to validate the data only once like in K8s, where data are validated by the Apiserver when stored. If for some reason an incorrect value propagates in the system (e.g., introduced by the Apiserver in K8s), escaping data validation, it can cause severe issues. The injection experiments in K8s proved that in some cases no circuit breaker or other resiliency strategies mitigate the impact on the system. Real-world incidents have proven that incorrect data can escape data validation, causing, for example, uncontrolled replication of resources.

5.1.2 Design principles for SLO-aware components

The following paragraphs provide a set of design principles for SLO-aware orchestrator components. The principles are in one-to-one correspondence

with the sources of delay identified in §4.3.3, and aim at addressing them.

Resource reservation Indirect delays affecting requests at high criticality levels can be mitigated for both control plane and worker node components through resource reservation. This includes: i) dedicated computational resources, or ii) real-time schedulers (e.g., real-time servers [42]) to synchronously handle requests.

Synchronous queue management can guarantee enough resources to reduce the interferences, e.g., when spawning a pod on a worker node, and mitigate the problem of priority inversion (recall §4.3.3). Indeed, similarly to non-work-conserving and anticipatory schedulers [107], synchronous queue management does not immediately serve low-priority requests, avoiding the potential priority inversion with a closely upcoming higher-priority request. Noteworthy, resource reservation and a bounded number of queued requests in the system can guarantee an analytical computation of the worst-case serving time for a request.

Low-latency datastores The datastore delays of critical requests can be mitigated by the use of real-time datastores [108] and better hardware. When the datastore is replicated, response times are higher since a consensus protocol must be executed. In those cases, response times also depend on the roundtrip times of messages over the network.

In specific scenarios, low latencies could be prioritized over persistency or availability, hence in-memory and/or non-replicated datastores could be considered to further reduce datastore response times.

Multi-priority queues Queuing delays can be bounded by allowing a finite population and using multi-priority management. For example, dispatching messages among control plane components (i.e., Apiserver in K8s) can use TSN schemes, including periodic, fixed-priority, and best-effort messages.

Bounded-time algorithms Scaling delays can be mitigated by using algorithms whose runtimes do not depend on the number of resource instances in the system. The extreme solution is to define the behavior of the components ahead of time for each possible case. For example, the scheduler and controller can leverage pre-generated rules that make the reaction to an event,

like deciding *a priori* where to re-schedule a pod upon a worker node failure.

Implementation choices The programming language used in an SLO-aware component can differ from vanilla components to leverage real-time scheduling and avoid language-runtime–related delays. Moreover, all the optimizations baked into the implementation details of a component can be designed to prefer low latencies over high throughput. For example, writing a component from scratch in a system language like Rust enables real-time scheduling for the component threads and a simpler implementation without unpredictable optimizations.

5.1.3 Architectural designs

The design principles and mitigations listed in §5.1.2 can be implemented (all or a subset of them) in what I call orchestrator *SLO-aware components*, such as: the *controller*, the *scheduler*, the *datastore* and the *worker node agent*. This component list is non-exhaustive and aims at expressing general concepts to integrate into the logic architecture of a generic orchestrator.

I propose three architectural designs that use SLO-aware components (illustrated in Figure 5.2): a **co-orchestrator** (① in Figure 5.2), an orchestrator with **additional components** (②), and a **patched orchestrator** (③).

I define a *"critical node"* as a worker node running an SLO-aware worker node agent and providing a higher assurance level compared to other worker nodes. Critical pods should be scheduled only on critical nodes. I define the other worker nodes as *standard nodes*.

A worker node range can provide a greater assurance level by relying on a range of containerization technologies[2]. Each technology provides a different assurance level suitable to host services of different criticality levels. For example, a worker node supporting partitioned containers (presented later in this chapter) can host critical pods.

The proposed architectural designs do not depend on a specific orchestrator: several products, like Docker Swarm, Apache Mesos, Openwhisk [109], Oakestra [110], share similar architectures [12], with a main datastore and controllers and schedulers acting on it.

The following subsections examine the three architectures in detail.

[2]I refer to §7.3 for a complete overview of containerization technologies.

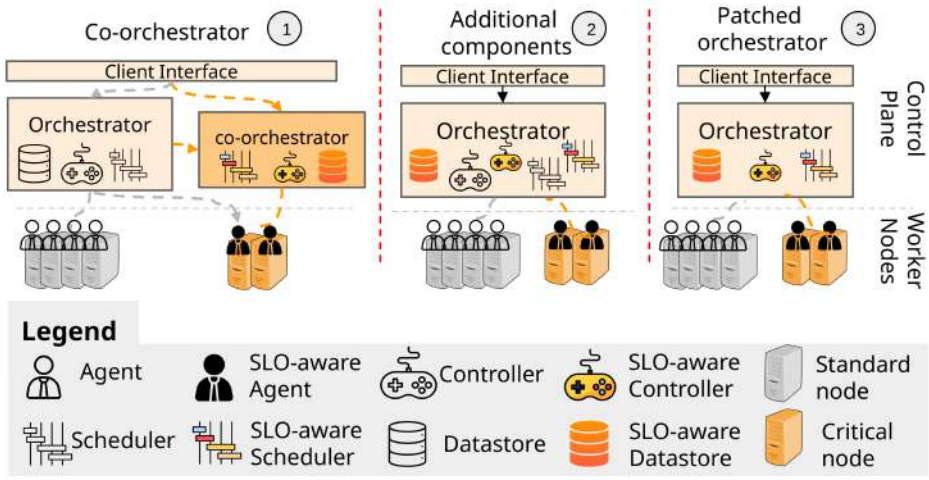


Figure 5.2. Three possible SLO-aware architectures for orchestrators.

Co-orchestrator

A *co-orchestrator* managing critical services operates alongside the *vanilla orchestrator* (hereon shortened as *orchestrator*), which remains unmodified. The system interface discerns requests and dispatches them to the orchestrator or co-orchestrator according to the criticality level. The idea is similar to real-time co-kernels for operating systems: they intercept interrupts to serve them with low latency through *ad hoc* kernel subsystems implementations.

When the co-orchestrator manages resources shared with the orchestrator, like critical nodes, the view of the node status must be coherent with the orchestrator. Hence, events regarding those nodes must be sent to the co-orchestrator, which propagates them to the orchestrator when possible, similarly to interrupts in real-time co-kernels.

The co-orchestrator includes an SLO-aware scheduler, controller, and datastore that can run on a dedicated node or use resources isolated from the vanilla control plane.

Co-orchestrator components are *dedicated*, written from scratch, and use all the mitigations listed in §5.1.2. Hence, co-orchestrator components can be written in a different programming language from orchestrator components, enabling sound real-time scheduling of requests. Moreover, while the

orchestrator components work in a *declarative* way [111, 83], co-orchestrator components can work imperatively to reduce orchestration times. Despite the high implementation effort, handling the critical services' requests through an end-to-end path of actions involving components with an *ad hoc* design achieves the best performances and guarantees. Indeed, an issue in the vanilla orchestrator, including overloads or component failures, will not affect the co-orchestrator. Co-orchestrator components can be designed to be simple and resilient, removing unnecessary features and implementing the fault tolerance strategies pointed out in chapter §4.

Additional components

Some components are duplicated, and a set of *ad hoc* (written from scratch) SLO-aware components are integrated into the vanilla orchestrator architecture (e.g., scheduler, controller). Those components handle a set of resource instances disjoint from the set handled by vanilla components. For example, the vanilla scheduler may manage non-critical services, while the SLO-aware scheduler manages critical services. Duplicated components, like the two schedulers, must use suitable caching policies to see a coherent state of common resources (e.g., standard nodes).

Conversely, other components (e.g., the datastore and the interconnecting component like the Apiserver in K8s), are shared between the action paths handling critical and non-critical requests. Those components must be configured or patched to differentiate the management of critical and non-critical requests, to reduce delay for critical requests, while not overly affecting non-critical services.

Compared to the co-orchestrator design, this design is simpler since it does not require implementing all components from scratch. On the other hand, the disadvantage is that shared components, like the datastore, still handle the load of the entire system, and in non-nominal conditions may not be able to provide guarantees for critical services.

Patched orchestrator

All the orchestrator components are shared between critical and non-critical action paths. The components are patched to be SLO-aware, and provide differentiated guarantees according to the criticality of requests. Hence,

the components must provide all the features required by non-critical services while preserving critical services. This design is the simplest of the three, and the idea is similar to a Linux patched with `PREEMPT_RT`.

This design does not require implementing components from scratch, but existing ones must be patched, reducing the implementation effort. On the other hand, fully-featured components can be bloated and unable to provide the same guarantees of *ad hoc* components. For example, it is difficult to fully address indirect, implementation-related, and scaling delays by only patching a fully-featured component: the sources of possible delays are baked and scattered all over the component source code.

5.2 Partitioned Containers

This section introduces the concept of partitioned containers as the convergence between orchestrators and partitioning hypervisors to enable the orchestration of critical pods. Partitioned containers must be seen as part of a wide spectrum of technologies, and the notion of pod criticality is independent of the technology used. In the rest of the dissertation, when I refer to critical pods, I do not necessarily imply the use of partitioned containers.

A partitioned container is an application running in isolated partitions controlled by a partitioning hypervisor[3], but managed as any other container by orchestrators. The main aim of partitioned containers is to provide a way to pack and deploy critical industrial services with tight real-time and dependability requirements. Indeed, relying on partitioning hypervisors provides partitioned containers with two essential properties: i) improved freedom from interference from co-located applications on execution times compared to other containerization technologies, and ii) isolation from cascading failures of other co-located VMs (including the privileged VM).

In chapter 2 containers were defined as "standard unit of software that packages applications and all their dependencies". Following this definition, a real-time application (e.g., an industrial control loop) could be packed either together with its Linux libraries to become a Linux container or together with its dependencies to run upon a minimal application runtime (e.g., a min-

[3]Partitioning hypervisors are a small software layer that exploits hardware-assisted virtualization to statically allocate hardware resources to VMs so that VMs execute with almost no hypervisor interposition [112]. I refer to §7.3 for additional details.

imal RTOS such as Zephyr or FreeRTOS, or a WASM runtime) and become a partitioned container that executes bare-metal.

Cloud orchestrators can manage partitioned containers as any other container thanks to compliance with the OCI standard for both the container runtime [21] and image specification [20]. Indeed, although commonly adopted container images may be bloated and split into many layers, images composed of a few (even a single) files are also supported by the OCI standard.

In this perspective, partitioned containers realize the vision of diversified containers introduced in chapter 3. Indeed, an orchestrator can seamlessly manage the same application deployed as a standard Linux container or as a partitioned container that has dedicated resources and provides more guarantees. For instance, an orchestrator can create a partitioned container with a dedicated CPU core, a sensor and/or a field network (e.g., CAN bus), while leveraging paravirtualization to exchange best-effort network traffic through a privileged VM to have a virtual network like traditional containers.

RPU-based partitioned containers

Although partitioned containers can run on any hardware/board supporting a partitioning hypervisor and an RTOS, their core value is given by the adoption in mission-critical systems of boards that pack on the same chip heterogeneous hardware specifically designed to host critical applications.

For example, Figure 5.3 shows an architectural diagram of a popular Multi-Processor System on a Chip (MPSoC), i.e., the Zynq UltraScale+. The Zynq UltraScale+ features 2 real-time processing unit (RPU) designed to host real-time critical applications. Those units are designed to have highly predictable computation times, dedicated interrupt channels, and a small dedicated tightly coupled memory (TCM) to avoid interferences from the application processing unit (APU). Moreover, the two RPUs of the Zynq UltraScale+ can also work in lockstep to provide dual redundancy (or TMR in more recent boards), protecting the computation from transient faults. These dependability features are essential to run critical and certified applications.

The RPUs do not have a complex memory management unit (MMU), but only a simple memory protection unit (MPU) to increase predictability. Hence, RPUs can host single binary applications compiled against simple library RTOS like Zephyr, but they cannot run complex OSes like Linux.

To take advantage of heterogeneous hardware the *Omnivisor* introduced

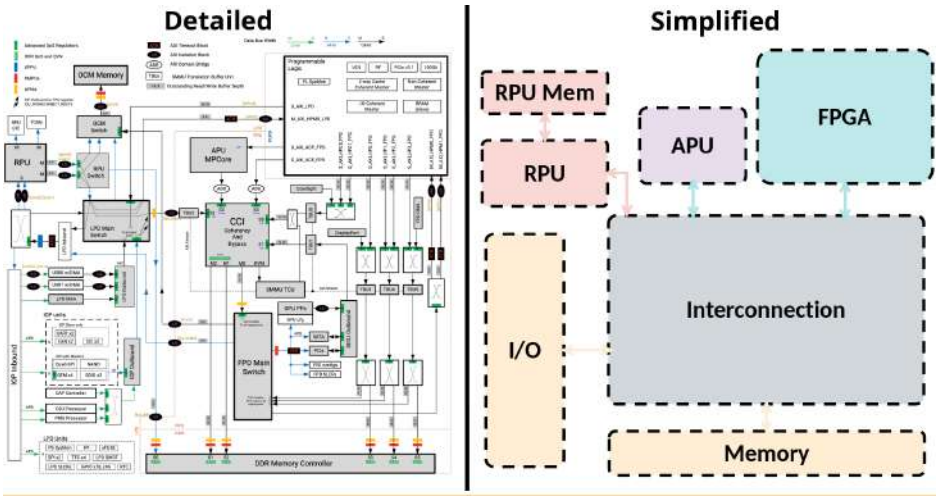


Figure 5.3. Detailed and simplified diagrams of a Xilinx Zynq UltraScale+ MPSoC ZCU104. The simplified view overlooks some components like graphic processing unit (GPU) and board management unit as are not of interest for this section.

in [113] extends the static partitioning hypervisor model, enabling the partitioning also on co-processors without MMU, such as RPUs or FPGA-based soft-cores (for more information, see §7.3).

In this perspective, partitioned containers running on top of a partitioning Omnivisor [113] provide easy and seamless integration of single-binary real-time applications running on dedicated RPUs into orchestrators.

5.2.1 Building partitioned containers images

When building a container image, the platform (as defined in §3) is determined by the host OS and ISA. This means that an application is compiled for the target ISA, and then it is packed within the image together with its OS-dependent required libraries. The OCI standard specifies how to describe these images, which are divided into layers that can be shared and reused by multiple containers.

Conversely, partitioned containers run bare metal with direct hardware access, relying on a minimal application runtime shipping the device drivers to abstract hardware resources. The application runtime is either supplied by

the node on which the partitioned container is spawned or shipped within the partitioned container image.

In the first case, the partitioned container image depends only on the application runtime. Hence, the platform is determined by the application runtime and the image is portable. An example of such bare-metal runtime could be a WebAssembly (WASM) runtime. However, WASM support for real-time applications is still immature and bare-metal WASM runtimes are still at the dawn at the time of writing [114].

In the second case, a libOS RTOS acts as the minimal bare-metal runtime. In this context, the application can be compiled against a libOS-RTOS (e.g., Zephyr, FreeRTOS, NuttX) to create a single bare-metal binary, which includes the OS and its drivers. The building process relies on the platform descriptor typically shipped with the RTOS to target a specific hardware/board. As a result, a distinct container image must be created for each target board. Nevertheless, platform-specific development and testing would be anyway necessary for a practical critical scenario.

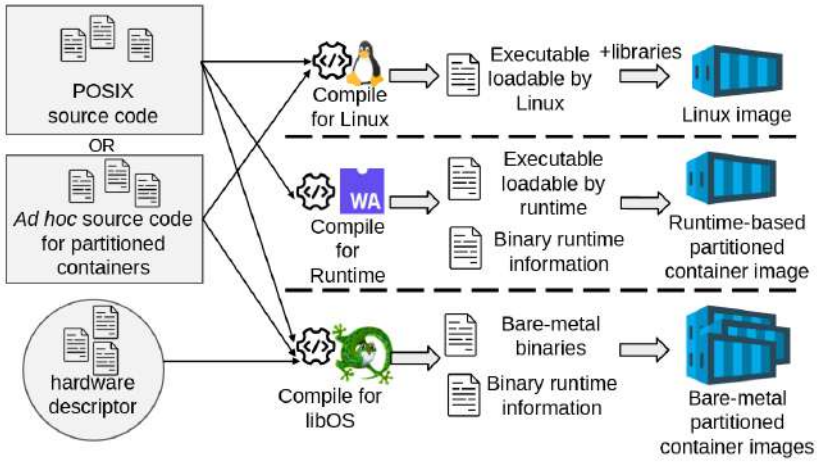


Figure 5.4. Image building workflow for partitioned containers.

The source code used to build partitioned container images can be i) the same POSIX-compliant source code used to build a Linux container image (through a POSIX application runtime); or, in order to increase diversity, ii) an *ad hoc* source code targeting the runtime. In the first case, the POSIX

application must be simple enough to accommodate the constraints of library RTOSes, which often do not support complex software or hardware features like a filesystem or an MMU.

The second case is instead the one of the coupled AI and safety-critical controllers presented in §3.2. Sometimes, *ad hoc* application source codes can take advantage of the frameworks written for library RTOSes. For example, Robot Operating System (ROS) applications can take advantage of microROS and AI-powered applications can leverage TensorFlow lite to run on Zephyr.

The described building combinations are depicted in Figure 5.4. The building process of partitioned containers outputs, besides application binaries, *binary runtime information* files. These files contain the necessary information for the hypervisor to start a partitioned container. For example, the binary runtime information file may include the virtual address space layout of the bare-metal binary, as described in §5.2.2.

5.2.2 Running partitioned containers

This subsection introduces the worker node stack needed to run partitioned containers, comparing it with the architecture of a Linux container stack (see Figure 5.5) that was introduced in §2.1.3.

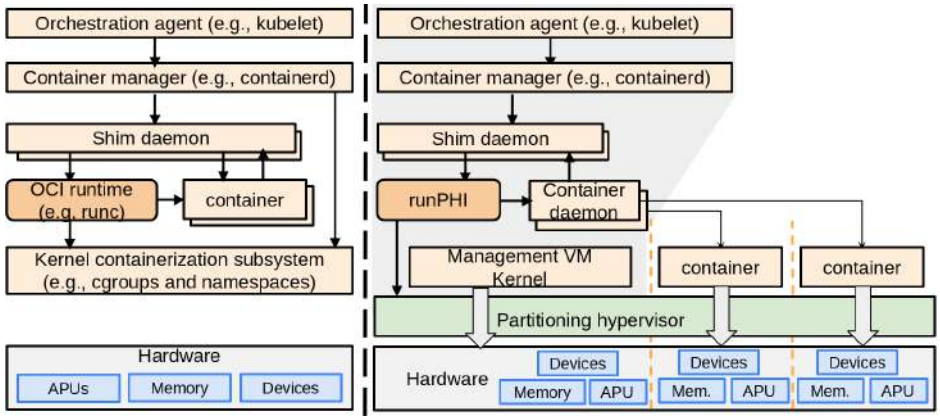


Figure 5.5. Linux container node stack (on the left) vs. partitioned container node stack (on the right).

The stack runs in a management VM that can control the lifecycle of other

partitions. The container manager is responsible for downloading images suitable for the specific platform. Next, *runPHI*, the core of the proposed stack, is invoked. runPHI is an OCI-compliant container runtime, responsible for creating, starting, killing, and deleting partitioned containers. Instead of relying on the OS containerization subsystem, runPHI configures the partitions of a static partitioning hypervisor (or Omnivisor) to host the containers.

Once a partition is configured and started, runPHI relies on custom daemons to maintain the communication channels and the status consistency of the partitioned containers. The partitioned container does not rely on the management VM because it has direct access to hardware resources, and it can also survive a crash of the management VM.

Although partitioning hypervisors fall short in virtualizing network interfaces with real-time guarantees, several approaches can overcome this issue. For example, one approach is to use a broker to manage the network [115]. Another method involves using a board with multiple network interfaces partitioned and assigned to different VMs. Finally, hardware support such as SR-IOV with real-time capabilities can be used [116].

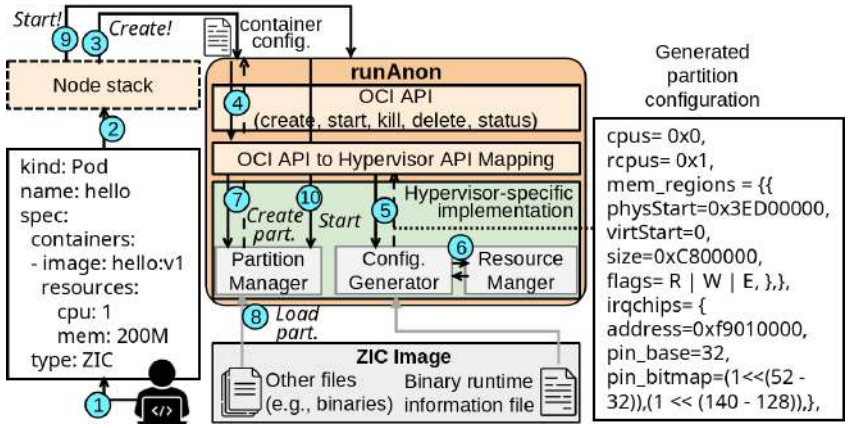


Figure 5.6. High-level architectural description of runPHI, highlighting the interactions of the configuration generator and partition management.

Figure 5.6 depicts a high-level and hypervisor-independent architecture of runPHI internals. When a user submits a manifest file ① to the orchestrator to request a container creation, the upper layers of the partitioned containers stack receive the information ②, and send a *create* command to runPHI

③. The topmost layer of runPHI parses the OCI-compliant command line arguments. Next, an intermediate layer addresses the semantic mismatch between container-related configuration parameters and concepts associated with partitions ④. The intermediate layer relies on the bottom layer, the implementation of which is hypervisor-dependent. This layer provides functions to manage the partition lifecycle ("*Partition Manager*" in the figure), to configure partitions ("*Configuration Generator*"), and to keep track of available/used resources ("*Resource Manager*"). The Configuration Generator is responsible for creating a partition configuration file ⑤. The file contains all the details required to create a partition, including information provided by the Resource Manager ⑥, the container configuration coming from upper layers ③, and the binary runtime information file shipped within the partitioned container image. After generating the configuration file, the Partition Manager loads the application (and runtime) into the partition ⑧, and finally, starts it ⑨⑩.

Networking

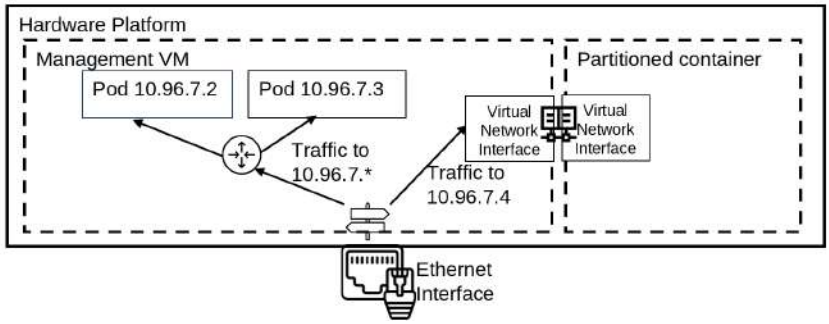


Figure 5.7. Networking configuration of a partitioned contianer.

In order to make the partitioned container compliant with the network virtual address seen by K8s, the network can be configured to forward the traffic directed to the partitioned container through network address translation (see Figure 5.7). The network translation can be implemented in the management VM or even in a dedicated bare-metal partition, like the broker used in [115], that owns the network interface. In the former case, a failure of the management VM causes a network failure. The latter case is more dif-

ficult to implement but guarantees that the networking survives the failure of the management VM.

5.3 Prototype Implementation

This section describes the implementation of a K8s-based mixed-criticality *Patched Orchestrator* prototype, named *Ulysses*. I selected this design to show the benefits of the design principles with the lowest implementation effort, compared to ad hoc components written from scratch. Moreover, the patch keeps all K8s functionalities out of the box, fostering a quick uptake of the solution by practitioners.

The prototype is based on the model and functionalities introduced in chapter 3. The prototype assumes that each resource instance has a priority level, which can either be defined in the *manifest* file used to create it or inherited from related resource instances following the rule defined by an *aggregation policy*. For example, I defined the ReplicaSet priority as the highest priority of its Pods.

My modifications and configurations involve *Kube-controller-manager* (shortened as *Controller*), the *Kube-apiserver* (shortened as *Apiserver*), the *Kube-scheduler* (shortened as *Scheduler*), and the *Kubelet*. The modification for the control plane consists of ≈1200 lines of code in total, while the container runtime for partitioned container consists of other ≈3700 lines of code

The following subsections delve into the modifications of each component, heavily referring to the specific terminology of K8s introduced in §2.1.4.

5.3.1 SLO-aware Controller

The SLO-aware Controller aims to reduce delays for requests regarding critical services, prioritizing them to get to the worker nodes sooner. The Controller behavior is depicted in Figure 5.8.

Multi-priority queues The queues of the controllers managing the main resource kinds are turned into multi-priority queues. The queues, containing the resource instance on which the controllers operate, have 3 priority levels. I added an optional parameter to the functions operating on the queues to specify the priority level of a resource instance when enqueued. Queue oper-

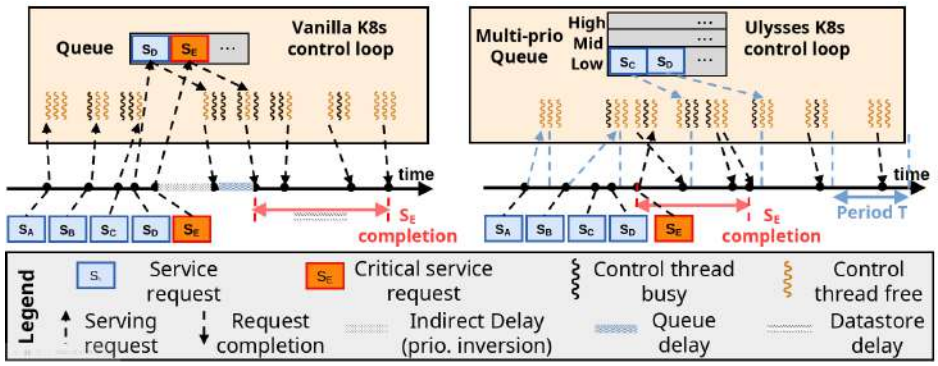


Figure 5.8. Vanilla Controller (on the left) compared to the SLO-aware Controller (on the right). Blue lines are synchronous actions that happen on the period tick. In K8s vanilla, S_A, S_B, and S_C are immediately taken in charge by the control threads (respectively $CT1$, $CT2$, and $CT3$). When S_D arrives, it is enqueued because there are no free control threads, and the same happens to S_{Crit}. Assuming that $CT1$, which processes S_A, is the first CT to finish, it immediately starts processing S_D. When $CT2$ is freed, it starts processing S_{Crit}. Hence, S_{Crit} is delayed by S_A, S_B, and S_C because they are served as soon as they arrive and there is no free CT left. S_{Crit} is subject to a queueing delay due to S_D too. With the SLO-aware Controller, when S_B, S_C, and S_D arrive, they are enqueued, waiting for the next period to be served. When S_{Crit} arrives, it is immediately served by a dedicated CT. Since the service of S_A and S_B is controlled in time, S_{Crit} undergoes minimal delay. Therefore, the overall turnaround time of S_{Crit} is shorter.

ations related to resource instances that do not specify any priority level have the lowest level to keep retro-compatibility with K8s vanilla code.

Synchronous queue management The queues in the Controller are modified to be managed synchronously. In particular, I turn the control loop threads into periodic tasks, resulting in a *periodic Controller* with synchronous handling of low-priority requests. Conversely, additional asynchronous control loop threads are dedicated to handling high-priority events. Although Golang (the K8s programming language) does not support sound real-time scheduling, I implemented a solution similar to polling servers [37] through *channels* and *tickers*. Figure 5.8 shows the behavior of the implemented solution. The control loop threads have a phase $\phi = k * T/N$, where k is the thread index, T is the period, and N is the number of active threads. Hence,

a request is processed every T/N. In this perspective, the Controller periodicity can be seen as a constant, fine-grained, and controllable throttling to respect the rate-limiting threshold and overcome the issues highlighted in chapter 4.3. Indeed, periods can be controlled at resource type granularity, improving the control over the system compared to a unique rate-limiting for the entire Controller. Furthermore, some controllers may be configured with higher frequency compared to others. This is fundamental when certain resource types have more instances than others. The periods can be tuned to have a throughput similar to the vanilla Controller, balancing interference due to short periods and performance degradation due to long periods.

5.3.2 SLO-aware Scheduler

The SLO-aware Scheduler aims at i) placing the Pods on a worker node that offers a suitable assurance level, ii) performing real-time schedulability tests for the Pod when necessary, and also iii) performing a schedulability test for the special-purpose network that a Pod may require. In other words, the SLO-aware Scheduler satisfies all the Pod requirements defined in chapter 3.

From a timing perspective, K8s already supports the *schedulingPriority* that defines the priority in the scheduling queue. Hence, I do not modify the Scheduler to improve orchestration times.

The implemented prototype of the SLO-aware Scheduler takes advantage of the plugin-based architecture of the K8s Scheduler to extend its functionalities, together with the use of taints and annotations (defined in §2.1.4).

In particular, I provide three *scheduler plugins* that implement the criticality-aware scheduler, real-time scheduling support for hierarchical deferrable servers, and a TDMA-enabling network manager.

Criticality-aware placement

I implemented the *criticality-aware* placement by using labels, and annotations, and extending the sorting, filtering, and scoring phases of the K8s Scheduler. The Pods with criticality level $C \neq NO$ are labeled with their criticality level, and the worker nodes are annotated with the vector of assurance levels $\vec{A}_n$ that they can offer for each resource. During the sorting phase, the Pods are ordered by criticality, so that critical Pods are scheduled as soon as possible. During the filtering phase, the Scheduler filters out the worker

nodes that do not satisfy the assurance requirements specified in the Pod aggregation policy. The implemented aggregation policy is a simple threshold comparison. Next, the Scheduler scores the worker nodes in decreasing order of assurance offered to a Pod.

During the Pod execution, a K8s daemon running on the worker nodes updates the assurance scores based on the status of the worker node. When the scores are updated, a custom controller checks whether a Pod must be rescheduled because the worker node no longer respects the Pod constraints.

Real-time support

The real-time scheduler plugin extends the filtering and scoring phases of the K8s Scheduler, and leverages Pod annotations. During the scoring phase, the plugin looks for real-time requirements in the Pod annotations. The Scheduler forwards the Pod requirements to a *scheduling agent* running on each worker node labeled as real-time. The scheduling agent determines whether the worker node has enough resources to host the Pod, and based on the agent response, the Scheduler can filter out the worker node. During the scoring phase, the plugin scores the worker nodes using the remaining available utilization for the least loaded core, giving higher scores to less loaded worker nodes.

Several solutions are possible to manage the real-time scheduling of worker nodes: one possibility is to install a scheduling agent on each worker node. Another is to install a single scheduling agent on a control plane node. Intermediate solutions foresee a scheduling agent managing a subset of nodes. For example, each agent can be in charge of managing the nodes requiring a specific schedulability test. For example, an agent manages the nodes supporting ARINC-653–compliant temporal partitioning schedulers and another agent manages the nodes supporting server-based hierarchical scheduling with dynamic priority at OS-level.

The implemented prototype installs a *scheduling agent* for each real-time worker node due to reduced complexity: as long as a worker node provides an agent that respects the protocol, any schedulability test can be enforced, depending on the hypervisor/OS.

Special-purpose network support

The network plugin extends the filtering and reservation phases of the K8s Scheduler and leverages Pod annotations to guarantee to a Pod the correct amount of networking resources (e.g., a slot in a TDMA network, a priority flow in TSN, a frequency in a wireless network).

The plugin parses network requirements from Pod annotations and filters out worker nodes that do not support the network type required by the Pod. Next, the network plugin performs a schedulability test for that type of network. For example, the implemented prototype considers a TDMA network and looks for a suitable network time slot for each TDMA network in the cluster where at least one worker node is eligible for the Scheduler. If the network under exam has no available time slots matching the requirements, all the worker nodes on that network are filtered out.

During the reservation phase, when a Pod must be spawned on a worker node, a networking agent configures networking resources for the Pod. My prototype implements a networking agent supporting RTnet, a TDMA-based network protocol shipped with Xenomai [117].

5.3.3 SLO-aware Apiserver

The Apiserver remains unmodified but is configured not to delay high-priority requests due to weighted fair queuing through an *exempt flow-schema* (see §4.3.3). The exempt flow-schema prevents requests and events belonging to a dedicated *namespace* from being subject to weighted fair queuing. Such namespace is dedicated to hosting critical services, i.e. services at medium or high criticality.

5.3.4 SLO-aware Kubelet

The SLO-aware Kubelet aims at reducing the interference of concurrent pod spawning, prioritizing critical pods, and using a suited containerization runtime to execute them.

Multi-criticality and synchronous event management I modified the asynchronous request handling (i.e. as soon as received) of the Kubelet for creation, deletion, and update. My SLO-aware Kubelet distinguishes between

high-priority (i.e., critical) and low-priority (i.e., non-critical) requests. High-priority requests are immediately served to reduce the latency. Conversely, the SLO-aware Kubelet introduces a delay between the handling of two low-priority requests, postponing the processing of the latter (see Figure 5.9). This timing policy limits the interference (and delays) affecting high-priority requests by limiting the service rate of low-priority requests.

Since high-priority requests are still handled asynchronously, in order to avoid the interference effects of the vanilla Kubelet, a minimum inter-arrival time of high-priority requests must be guaranteed. This constraint can be enforced by the control plane, which is not supposed to handle many high-priority requests concurrently. In other words, if every request has high priority, no request has high priority.

Two possible policies to determine the waiting time between low-priority requests are: *i)* a *fixed*, in which the waiting time between requests is constant; *ii)* an *exponential decay*, in which the waiting time between requests varies accordingly to a decreasing geometric progression, and it is reset after a period of inactivity. The fixed policy guarantees lower interferences at the cost of lower throughput. The exponential decay guarantees low interference only to the first requests of a burst, allegedly more critical than the later ones because of control plane prioritization.

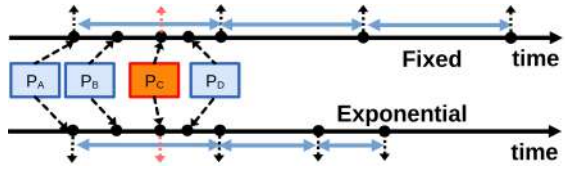


Figure 5.9. The proposed timing policies for the Kubelet.

Support for partitioned containers The SLO-aware Kubelet can optionally rely on a container runtime for partitioned containers. To this aim, I implemented a Jailhouse-based prototype of runPHI (recall §5.2.2). I chose Jailhouse because it was already modified in [113] to comply with the omnivisor model, providing seamless management of RPUs. The current runPHI implementation consists of ≈3700 lines of Rust lines. It creates a cell configuration file, compiles it, loads a binary into an RPU or APU, and finally starts the partitioned container. The network is configured according to Figure 5.7

by a container networking interface (CNI) plugin for partitioned containers that consists of ≈100 bash lines.

I remark that the multi-criticality and synchronous event management of the Kubelet is independent of the container runtime. Thus, partitioned and standard Linux containers can be both considered critical by the Kubelet to be prioritized at need.

5.3.5 Configurability

Listed modifications reduce delays of high-priority services and extend the support for Pods with special requirements, like real-time scheduling or industrial networks. Nonetheless, some of the modifications can be detrimental for orchestration delays and throughput of low-priority ones. For example, synchronously managing the queues by deliberately introducing delays can reduce orchestration flexibility and throughput in some conditions. In other cases, the fixed priorities used for the queues could even cause starvation of low-priority services when a high-priority service is misbehaving.

Since K8s allows dynamic redeployment of single control plane components, I implemented a set of components so that the Ulysses modifications can be selectively enabled. For example, the vanilla Controller is used until a critical service requiring low orchestration times is admitted to the system. After that, the multi-priority (and optionally periodic) Controller is enabled.

Chapter 6

Experimental Evaluation

There is no better way to understand dependability characteristics of computer systems than by direct measurement and analysis.

Ravishankar Iyer

This chapter evaluates the implemented prototype of the mixed-criticality orchestrator introduced in the previous chapter.

Section 6.1 evaluates how the proposed modifications affect orchestration times under increasing load conditions of the orchestrator. The experiments contain an ablation study of the possible configurations range using SLO-aware components. In the most extreme configuration, orchestration times for critical services remain constant despite the increasing concurrent workload for the orchestrator.

Then, section 6.2 evaluates the criticality-aware placement in terms of schedulability of pod sets and assurance guaranteed to pods. The results show that the assurance can be optimized without penalizing the pod acceptance rate and real-time scheduling prevents from scheduling real-time pods that would not have enough resources to guarantee meeting the deadline.

Afterward, section 6.3 evaluates how different containerization technologies for real-time containers behave in terms of boot times and assurance of real-time guarantees under stress conditions. The experiments highlight that partitioned containers provide boot times comparable with Linux containers while offering independence from failure and timing interference.

6.1 Orchestration Times Evaluation

In this section, I evaluate the improvement given by Ulysses in terms of orchestration times, by analyzing the contribution of each proposed modification (§6.1.1) and using Ulysses for a cloud-native 5G scenario (§6.1.2).

The experimental setup used in the experiments is the same as reported in §4.3.1 for the previous set of experiments.

6.1.1 Ablation study

I repeated the experiments performed in §4.3.2 in the configurations reported in Table 6.1 to assess the impact of each implemented modification.

Experiment Parameters

Config. #	Kubelet	Apiserver	Controller
1	Vanilla	Vanilla	Vanilla
2	Patched	Vanilla	Vanilla
3	Patched	Multi-priority	Vanilla
4	Patched	Multi-priority	Multi-priority Asynch.
5	Patched	Multi-priority	Multi-priority Periodic

Table 6.1. Experimental configurations for the orchestrator components.

The parameters were the same as described in §4.3.3. Rate-limiting was enabled, as usual in production environments. I repeated and compared the experiment on both cloud and edge clusters for each configuration in Table 6.1. *"Kubelet patched"* means both multi-priority and synchronous. *"Multi-priority Apiserver"* means Apiserver configured according to §5.3.3. *"Multi-priority asynchronous Controller"* means a vanilla Controller with only modifications for multi-priority queues. The Scheduler was not considered in this set of experiments as the SLO-aware Scheduler affects the functional behavior but provides no benefit to scheduling times. In preliminary experiments, the vanilla Scheduler proved to have a minimal contribution to the orchestration times ($< 10ms$) due to the reduced cluster size.

I configured the Kubelet exponential decay policy in the following way: *i)* I set $700ms$ as initial waiting time, reduced by 50% at each request, for worker nodes in the cloud cluster; ii) I set $1.5s$ as initial waiting time, reduced by 40%

at each request, for worker node of the edge cluster. The initial waiting was reset after 5*s*. All these parameters were set according to the time figures measured in §4.3.3. The control loop parameters were configured as follows: 5 threads (K8s default) for each control loop (i.e., for each resource type), one of which was asynchronous and dedicated to high-priority requests. I configured *Nodes*, *Endpoints*, and *EndpointSlices* controller to serve a request each 50*ms*; and *ReplicaSet*, *Deployment*, and *Services* controllers to serve a request each 100*ms*. These period parameters were chosen to have the Controller maximum request limit similar to K8s default one (i.e., 20 per second). The different periods were meant to accommodate the needs of the different controllers due to their number of resource instances managed.

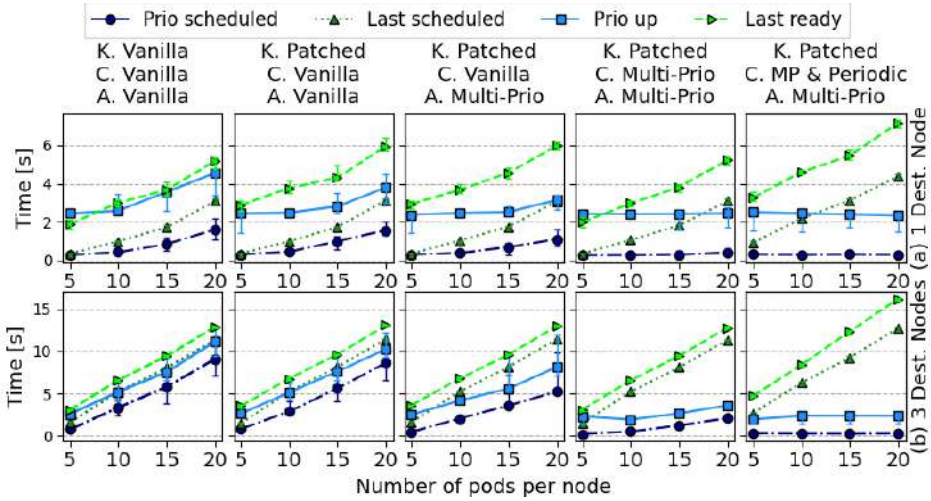


Figure 6.1. Comparison of the configurations in terms of orchestration times in the cloud cluster with throttling enabled. *"K."* is Kubelet, *"C."* is Controller, *"A."* is Apiserver, *"MP"* is multi-priority.

Results

Orchestration times Figure 6.1 shows the results for the cloud cluster for 1 (a) and 3 (b) Destination Nodes (*DN*). The modified Kubelet improves the time to spawn the high-priority Pod on the worker node: the Pod starts serving the client ≈ 2.1*s* after the "*Prio scheduled*" event, despite the increasing

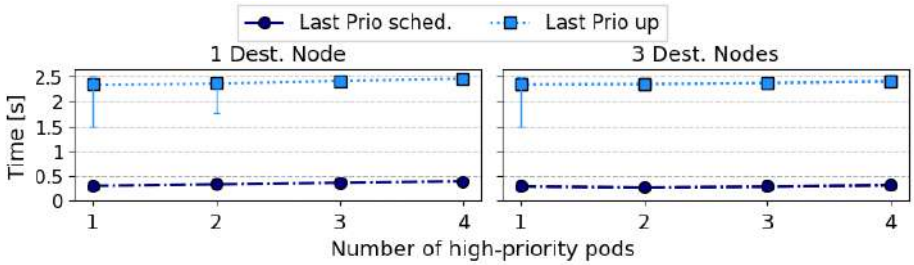


Figure 6.2. Orchestration times of multiple critical Pods, under a load of 20 low-priority Pods per *DN*, for 1 and 3 *DN*.

number of Pods. However, the Kubelet cannot avoid the control plane delays, especially when it hits the rate-limiting threshold (see 3 *DN*).

In the 3 *DN* scenario, while the multi-priority Apiserver (column 3 in the figure) gives some improvement, adding a multi-priority Controller (column 4) significantly improves the time to schedule the high-priority Pod. Nonetheless, the "*Prio scheduled*" event still shows an increasing slope when the control plane hits the rate limits and triggers throttling. The multi-priority periodic Controller (column 5) solves this issue by accurately controlling request rates. It ensures a constant orchestration time of the high-priority Pod in all tested conditions, with a reduction up to 78% in the scenario with 3 *DN*, 20 Pods per node. The *"Last ready"* event timing remains roughly the same in all cases, except for the periodic Controller, which presented a slowdown of up to $\approx 3s$ (median 14.5%) in the worst case (3 *DN*, 20 Pods per node).

For this latter configuration, I repeated the experiment with multiple critical Pods, with 1 and 3 *DN* and 20 Pods per node. The results in Figure 6.2 show only a slight increase, while the slowdown for other Pods is unchanged.

Control loop runtimes I compared the control loop runtimes of the periodic multi-priority Controller and the vanilla Controller in the cloud cluster. Figure 6.3 shows that the modified Controller reduces the interferences among control loop threads and prevents throttling. In the experiments, it caused a 96% reduction of the *p99* of runtimes. The standard deviation was lower than 1 control loop thread (in Figure 4.11). The median value of memory usage percentage increase due to Ulysses was +3.2%.

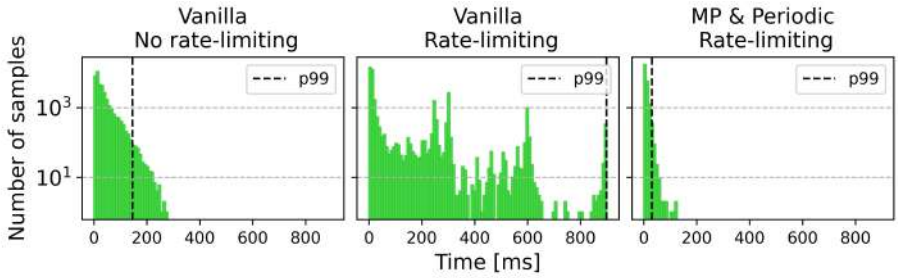


Figure 6.3. Comparison of control loop runtimes for 5 control loop threads (i.e., default). Vanilla with rate-limiting is the default K8s configuration. The spikes are caused by the throttling, which is necessary to respect the rate limit.

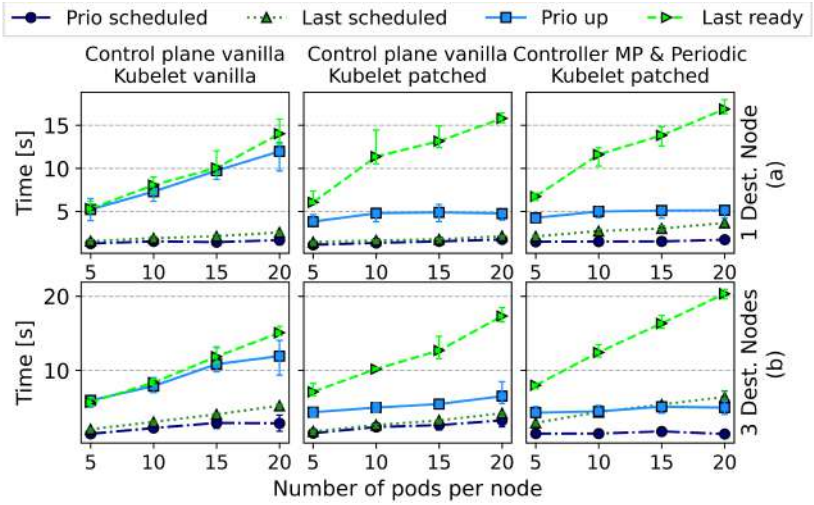


Figure 6.4. Comparison of the configurations on the edge cluster without control plane rate-limiting. The Kubelet alone can provide significant improvement.

Effect of patched Kubelet I disabled the rate-limiting on the control plane to more accurately assess the isolated effects of the Kubelet modifications. I considered both vanilla and multi-priority periodic Controllers. Figure 6.4 shows the results for the edge cluster. Patching only the Kubelet (column 2) provides noticeable improvements. The *"Prio up"* time remains almost flat and ≈40% less than the vanilla case. A multi-priority periodic Controller

introduces further benefits at the cost of slight performance degradation.

Kubelet waiting time policies I measured the Pod ready times when setting vanilla, exponential decay, and fixed (configured with constant 1.2*s* waiting time) waiting time policies on the embedded device in the edge cluster. I targeted the embedded device because it amplifies the differences in the ready time distributions. Similar but rescaled effects were observed in a cloud node.

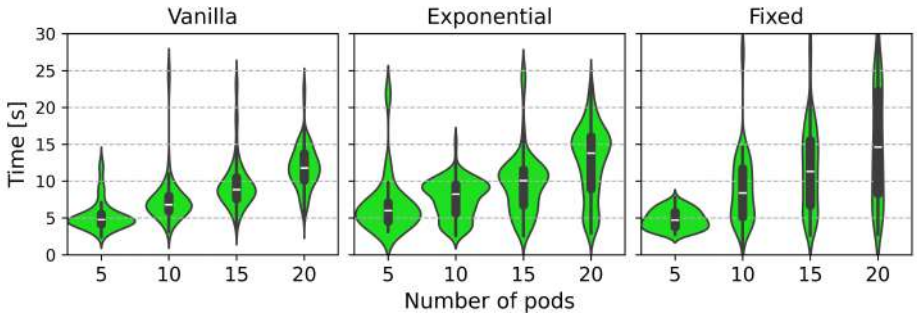


Figure 6.5. Distributions of Pod ready times on the embedded device with the different Kubelet waiting time configurations.

Figure 6.5 shows the results. The distributions for *Exponential* and *Fixed* policies show that more Pods become ready earlier than *Vanilla*. The *Fixed* configuration is more conservative, exhibiting an almost uniform distribution and higher maxima (i.e., last Pod ready). The mode of the *Exponential* policy is similar to *Vanilla* in the upper part of plots because of the increasingly short waiting times between Pods' requests. The exponential policy is a good tradeoff between throughput and interference.

6.1.2 Cloud-native 5G

I aim to prove the benefits of Ulysses in a realistic scenario with multiple services and differentiated SLOs. I joined the edge and cloud clusters, obtaining a cluster with 8 VMs, 1 workstation, and 1 embedded device. The K8s control plane node was a cloud VM as before. I used *Open5GS* as a softwarized 5G core network [1]. The 5G control plane was deployed over three cloud VMs, while the user plane function (UPF) of the data plane had high

[1] `https://github.com/open5gs/open5gs`

priority and is deployed on the edge embedded device. I used UERANSIM [2] (deployed in an edge VM) as a commonly used 5G simulator for user equipment and radio access networks.

Train-ticket [3] was used as a realistic microservice-based application, with less stringent SLOs than the 5G network functions. Train-ticket microservices were deployed across 5 cluster nodes (nodes used for 5G control and data plane were excluded), with K8s being free to decide the scheduling.

Two edge nodes, including the one where the UPF was deployed, were tainted to trigger activity in the orchestrator (similarly to what is performed in §4.3.2). The UPF Pod must be prioritized over the concurrent orchestration load because it may provide connectivity to some User Equipment (UE) requiring low latency and high availability. I repeated the experiment 30 times for statistical purposes for each configuration, i.e. K8s vanilla and Ulysses periodic and multi-priority.

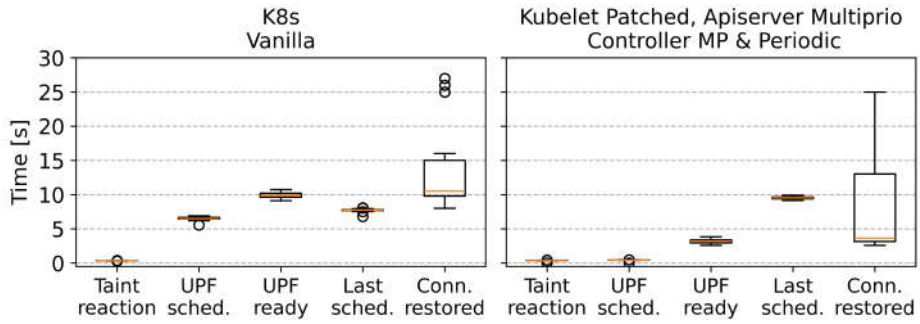


Figure 6.6. Main orchestration events for the 5G data plane failover example.

Figure 6.6 shows the timing of the main events, including the 5G connection recovery. The time to schedule the UPF Pod was drastically reduced ($\approx -92\%$), as well as the UPF ready time ($\approx -68\%$). Although the connection recovery times ("Connection Restored" in Figure 6.6) varied due to the configuration of a new connection, the median time was approximately the same as the median of UPF ready times. This matters in scenarios where the mean time to recovery for UE connections must be as low as possible [118].

[2] `https://github.com/aligungr/UERANSIM`
[3] `https://github.com/ovkulkarni/train-ticket`

6.2 Criticality-aware Placement

This section compares the criticality-aware placement policy with the default K8s Scheduler policy (in the following, *vanilla* Scheduler).

Acceptance rate

Experimental method I simulated three possible vanilla Scheduler configurations[4], and compared them with the criticality-aware placement policy: i) *LeastAllocated*: it prefers nodes with more free resources to reduce interferences among Pods; ii) *MostAllocated*: it prefers the nodes with less free resource that can host a Pod; iii) *RequestedToCapacityRatio*: it prefers nodes with less percentage of free resources after allocating the Pod.

The criticality-aware placement policy was designed as a multi-objective optimization, encompassing multiple functions, each designed to be maximized and with a codomain laying in $[0,1]$. The objective function was a weighted sum of the following functions multiplied by the acceptance rate:

• Acceptance rate: percentage of Pods that the scheduler managed to assign to its nodes. This function forces the scheduler to assign jobs instead of rejecting them. It is mathematically expressed as: $\frac{\sum_n |J\_on\_n|}{m}$, with m number of pending + active Pods, and J_on_n the Pods running on node n.

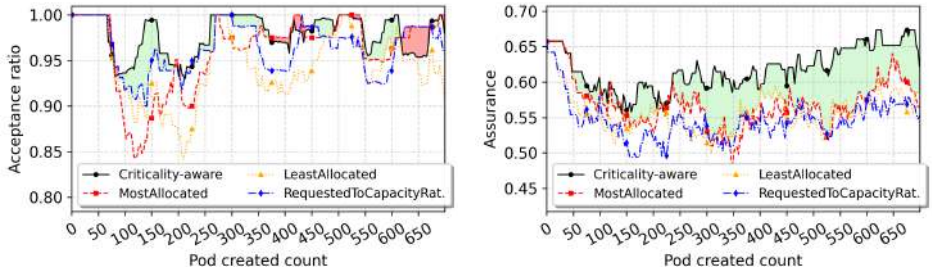
• Node assurances: cluster average assurance. It ensures that incoming critical Pods find nodes with high assurance to host them. To simplify the simulation, the assurance value is a percentage of $\alpha_n + \beta_n$ that assumes discrete values. The value is defined by the minimum Pod criticality hosted on the node, and whether the resources guaranteed to the Pod are the minimum or maximum requested by the Pod. It is expressed as: $\sum_{n:RT_n=1} A_n(t) / \sum_{n:RT_n=1} 1$

• Residual capacity: sum of nodes' squared free resources. The squared terms want to achieve an effect similar to the *MostAllocatedFirst* K8s policy, leaving some nodes free to host any incoming heavy Pod. It is mathematically expressed as: $\sum_n \sum_{z=1}^{|B\vec{R}_n|} (B\vec{R}_n(z) - \sum_{l \in J\_on\_n} BR_l(z)_{limit})^2 \Big/ \sum_{z=1}^{|B\vec{R}_n|} (B\vec{R}_n(z))^2$

In the simulation, all four policies first filtered out the worker nodes not eligible to host the was chosen according to the policy. Default K8s filtering

[4] `https://kubernetes.io/docs/reference/scheduling/config/#scheduling-plugins`

was extended with: i) assurance compliance $\forall_{k \in J\_on\_n \cup \{l\}} A^*_n \geq \theta_k$ where A^*_n is the assurance level of node n if pods l were assigned to it, and θ_k is the assurance threshold required by Pod k based on its criticality level; and ii) real-time compliance: $\neg RT_l \vee RT_n$.



(a) Comparison of acceptance rate. **(b)** Comparison of the assurance values.

Figure 6.7. Simulated placement metrics. Green and red areas are where criticality-aware placement is respectively better and worse than the best of the others. The weights for criticality-aware policy were: 52.5, 42.5, and 5.

The simulation started with a random number of nodes (between 4 and 10) and Pods (between 300 and 1000). During the experiment, other nodes were randomly added to the cluster and pods were randomly terminated, freeing the resources allotted to them. Each scheduling algorithm received the same input sequence, in terms of nodes and Pods.

Results The plots in Figure 6.7 show that co-optimizing the *assurance level* ensures more guarantees to critical Pods without necessarily penalizing the acceptance rate compared to default schedulers. Indeed, while the green and red areas in the acceptance ratio plot are approximately balanced (the criticality-aware policy is not statistically different from the others), the assurance plot shows almost everywhere green areas (i.e., improvements of the criticality-aware policies compared to the best of the others).

Schedulability

Experiment method I compared the schedulable sets of Pods between vanilla K8s and Ulysses. I simulated the placement of Pods in a cluster made up of

5 nodes with no real-time guarantees, 2 nodes with medium real-time isolation guarantees that allow a *budget/period* scheduling interface for containers (with the schedulability test used in [72]), and 1 node with high real-time isolation guarantees that is characterized by a hierarchical deferrable server scheduling. The set of Pods to be scheduled was composed of: i) 30 best-effort Pods with random CPU requirements between 8% and 25%, ii) 30 real-time Pods with medium isolation requirements, with random real-time CPU requirements between 8% and 25%, and iii) a random number between 6 and 12 of real-time Pods with high isolation requirements, created with the algorithm presented in [119] such that their total utilization is 180%.

Results In the simulation, K8s considered all the 200 sets of Pods as schedulable, while Ulysses scheduled only 108 of the 200 sets. For the unschedulable pod sets not enough resources could be guaranteed to complete the execution within the deadline in the worst case. In particular, 61 sets were rejected by the hierarchical deferrable server schedulability test, and 45 were rejected by the EDF utilization bound. Hence, although K8s schedules all the Pods, their execution would lead to hazardous latencies and deadline misses.

6.3 Evaluation of Real-Time Containers

I evaluated boot times, temporal isolation, and failure isolation for different containerization technologies. In particular, I compared three containerization technologies: Linux-based real-time containers, Xenomai-based real-time containers, and Jailhouse-based partitioned containers.

6.3.1 Experimental setup

The experiments of this section were executed on a Xilinx Zynq UltraScale+ MPSoC ZCU104 (hereafter ZCU104), and on the HP Z230 workstation already described in §4.4.1.

The ZCU104 was configured in the same way as the previous hardware: at the hardware level the processor frequency was fixed and the idle states disabled. The kernel was a Linux v6.1 patched with `PREEMPT_RT`, with debug disabled and all other tweaks used for the other kernels. For Jailhouse, I used the default configuration of the installation of v0.12 patched with the

Omnivisor extension [113].

The results were also compared with my previous proposal of Xenomai-based real-time containers presented in [117]. The Xenomai co-kernel used was v3.1 patched as in [117], configured following the official guide[5].

6.3.2 Boot times comparison

I compared the boot times of partitioned containers with Linux containers, running the experiments on the ZCU104. Although a Xenomai configuration on the board was not available, the boot times of Linux containers can approximate the boot times of Xenomai-based containers, as Xenomai-based containers are standard Linux containers with real-time threads scheduled by the Xenomai co-kernel.

I performed the experiment on top of the same Linux kernel v6.1 patched with the `PREEMPT_RT` patch and configured with all tweaks for real-time kernels for both real-time Linux containers and partitioned containers, I used Docker to trigger the container creations, and I measured the boot times as the difference between two timestamps: the *start* time and the *ready* time. The start time was sampled right before the *create* request to the low-level container runtime (i.e., runc for Linux containers, runPHI for partitioned containers). The ready time was sampled as the first function call executed by the main function of a harness empty application used as the container entry point. The harness empty application was compiled as both a Linux executable to be packed within an Ubuntu Linux container (27 MB) and a bare-metal binary including Zephyr v3.22.1 to run as a partitioned container (12 KB) over the RPU of the board.

In the first set of experiments, I dropped the Linux (i.e., the host or management VM) caches at the end of each measurement in both configurations to have independent measurements. In the second set, I did not drop the caches between measurements to estimate average boot times. The caches were dropped by executing the command *"sync; echo 3 > /proc/sys/vm/drop_caches"*. I performed 200 measurements for each configuration.

Figure 6.8 shows the results. The time to boot a partitioned container is comparable to a standard Linux container. The results for partitioned containers are notably better when the compiler is cached since most of the boot

[5] `https://v3.xenomai.org/tips/x86/common/`

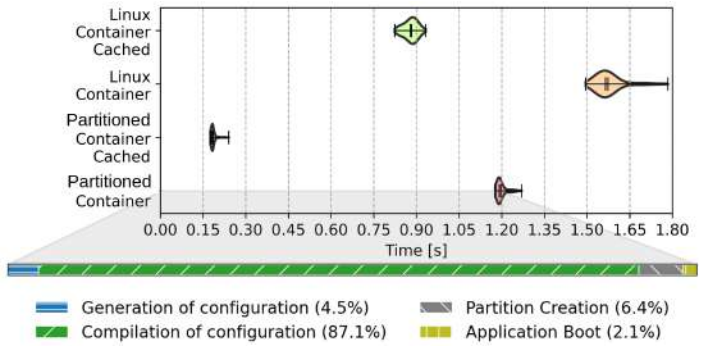


Figure 6.8. Distribution of boot times for partitioned containers and Linux containers, with the breakdown of the runPHI runtime into its phases.

time is spent compiling the Jailhouse partition configuration file.

The size of the selected container images did not affect the experimental results. Indeed, although the authors of [113] showed that the times the Jailhouse Omnivisor takes to load applications linearly grow with the container image size, the loading time is a fraction of the "partition creation" time, representing a negligible contribution to the overall boot time. Conversely, the authors of [97] showed that the file system size is not linearly correlated with boot times for Linux containers.

6.3.3 Temporal isolation comparison

This subsection analyzes and compares the behavior in terms of temporal interferences of real-time containers based on different technologies when intense stress is co-located on the same node.

Experimental method The experimental method was the same as the experiment presented in §4.4.2. I briefly recall it here: I ran *rt-app* as a containerized benchmark application performing CPU, memory, and network activity to measure the interference from the co-located stress generated by *stress-ng*. I ran rt-app pinned on one core, configured to execute a single thread scheduled with a fixed priority of 95.

Following the parameters of the *Experiment set 2* in §4.4.3, the rt-app thread had a period of 10 ms and an execution loop that sequentially: *i)*

wrote 8192 bytes to the main memory, *ii)* ran CPU-bound operations for 1.2 ms, *iii)* sent 1024 bytes over the network interface.

I applied the following stress types: *cpu, udp, hdd, io, netdev, open, fork, memcpy, cpu + hdd, cpu + udp, udp + io*.

In addition to the thread activation latency and slack times gathered by rt-app, I also collected the timestamp of packets sent by rt-app through *tcpdump*. I used the packet timestamp to measure the time between consecutive packets (from here on, defined as inter-sending time). The inter-sending times can reveal potential jitters that affect network packets, which are expected to be periodic like the thread and correlated with slack oscillations.

I ran the experiments on the HP Z230 workstation and the ZCU104 for each configuration available on each node. Those configurations were: i) real-time containers on `PREEMPT_RT` Linux on the HP Z30 workstation, ii) Xenomai with RTnet and real-time containers for the HP Z30 workstation, iii) `PREEMPT_RT` Linux with rt-containers for the ZCU104, and iv) Jailhouse with partitioned containers running over the ZCU104 RPU. To have an even comparison with partitioned containers, in the ZCU104 `PREEMPT_RT` configuration, rt-app ran on a core isolated with *isolcpus*. For `PREEMPT_RT` and Xenomai, I compiled rt-app respectively as a Linux or Xenomai application to be executed within an Ubuntu Linux container. For the Jailhouse-based partitioned containers, I created a porting of rt-app for Zephyr v3.22.1 that entirely fit the RPU TCM of the ZCU104.

In the Xenomai with RTnet configuration, the TDMA cycle was set at 5 ms, with 1 ms time slots. The TDMA regulator that sent the *sync* packets for a TDMA cycle was the same workstation on which I ran the experiment.

Results Figure 6.9 shows the activation latency, slack time, and inter-sending of rt-app with 16 concurrent stressor threads across the four target worker node configurations. Although the absolute time values depend on the compilers, hardware, and libraries involved, which had to differ across the configurations[6], some insights on the deviation and stress sensitivity can be extracted. The results exhibit a non-negligible interference on activation latency

[6]The libraries differ between PREEMPT_RT HP Z230 and Xenomai HP Z230 since Xenomai relies on an own library. The libraries and the compiler differ between Zephyr ZCU104 and PREEMPT_RT ZCU104 as Zephyr relies on its building toolchain. The compilers and the hardware obviously differ between the two machines.

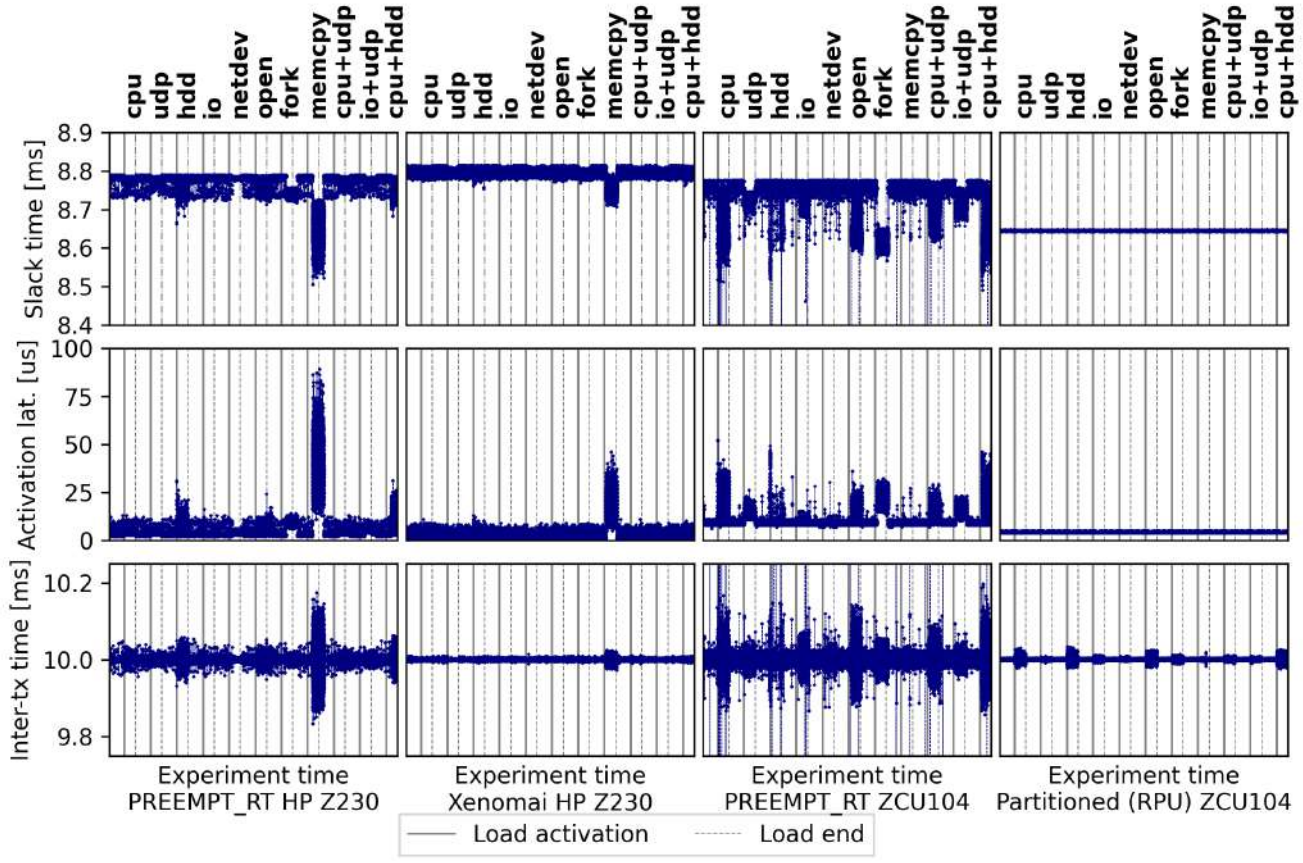


Figure 6.9. Comparison of thread activation latencies, slack times, and packets inter-sending times between PREEMPT_RT, Xenomai, and RPU partitioned containers. The latencies are plotted over time, with co-located stresses indicated above for each phase.

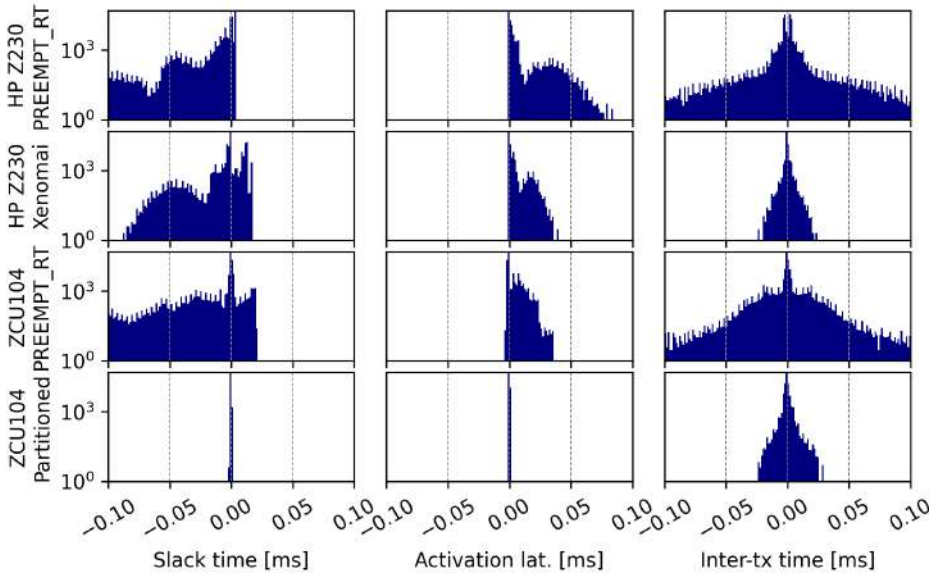


Figure 6.10. Comparison of distributions of thread activation latencies, slack times, and packets inter-sending times between PREEMPT_RT, Xenomai, and RPU partitioned containers. The distributions are normalized to have zero median value. The distributions cover the entire experiment and thus include all the co-located stress types. Note the logarithmic scale of the y-axis.

and slack times due to shared resources (CPU cache, network, memory) for PREEMPT_RT (recalling §4.4.3), and in a reduced way for Xenomai. In particular, the *hdd* stressor caused higher activation latencies for those two configurations. Similarly, the *memcpy* stressor had the worst impact on rt-app due to cache eviction, as previously motivated in §4.4.3. This behavior is expected since neither PREEMPT_RT nor Xenomai use cache isolation mechanisms by default, like *cache coloring* [120]. Nonetheless, the Xenomai configuration suffers slightly less than the PREEMPT_RT. This is due to the way in which Xenomai schedules Linux tasks along with the Xenomai real-time tasks. It is worth saing that when I ran the same application on the ZCU104 with PREEMPT_RT outside the container, it showed far less timing variability. In this sense, the impact of the libraries (e.g., glibc) used inside the container should be further investigated.

Conversely, the slack times and activation latencies for the partitioned

container did not suffer from any of the co-located stresses and are much more predictable. Indeed, the partitioned container was deployed on the RPU, which has reserved resources with regard to the Linux system running on the rest of the board. Although the RPU RAM uses the same RAM as Linux, the RPU has a dedicated port to access it. In addition, the RPU features a processor that is designed to be predictable, thus also the slack times without any co-located stress are more predictable (times vary by $\pm 2\mu s$).

Furthermore, Figure 6.9 shows that, across all the target configurations, the inter-sending time between network packets is correlated to the slack times and activation latencies. Still, the deviation and spikes do not necessarily match the ones of those other two metrics. Hence, different resources can present independent spikes and different isolation degrees, and they must be isolated and evaluated independently. For example, although the slack times of Xenomai and PREEMPT_RT are comparable, the network metrics show almost no interference for Xenomai. This freedom from interference is due to RTnet, which asynchronously manages the packet transmission based on the TDMA time slots. Interestingly, also the RTnet inter-sending times suffer slightly from the *memcpy* co-located stress load.

On a similar note, the inter-sending times for the partitioned container show little interference, which is not present in the slack times. This interference is caused by Linux, on which the partitioned container relies to send packets over the network asynchronously to the rt-app executing on the RPU.

Findings Based on the results, the findings are summarized as follows:

• Different task-level metrics regarding shared resources can be impacted independently by co-located stress loads.

• Special purpose hardware can be integrated into cloud management systems to isolate workloads from co-located intense stresses.

• The same hardware can provide different assurance levels depending on the software stack and configuration. In the same configuration, different resources provide different assurance levels (e.g., CPU, network, memory).

6.3.4 Failure isolation comparison

This subsection analyzes and compares the resilience of real-time containers based on different technologies.

Experimental method I wrote an application that performs I/O activity on a serial device and compiled it twice to pack it as a Linux or partitioned container. I packed it in an Alpine container for Linux and I compiled it together with Zephyr to become a partitioned container.

I deployed a service with one replica of the application in two different configurations: i) a pod can be deployed on the ZCU104 and on a workstation, both running `PREEMPT_RT` Linux, ii) a pod can be deployed on the ZCU104 and on a workstation, being the ZCU104 managed by Jailhouse and the workstation by `PREEMPT_RT` Linux.

After 30 seconds from the beginning of the experiments, a software failure of the Linux kernel of the ZCU104 was simulated by injecting a kernel panic.

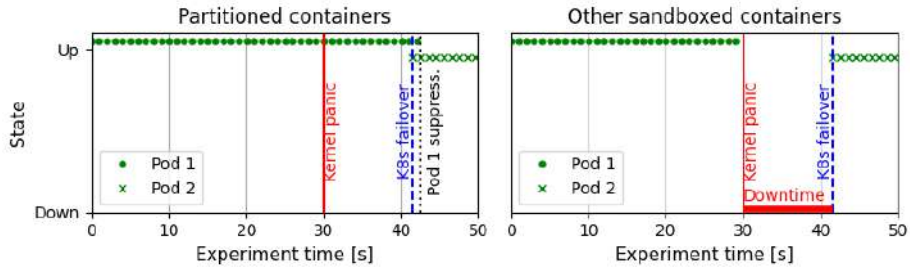


Figure 6.11. Availability comparison in case of failures between partitioned containers and other containers. Pod 1 is the Pod running before the failure, Pod 2 is the Pod spawned by K8s after detecting the failure. The kernel panic is injected on the host/service VM of the node hosting Pod 1. The partitioned container survives the crash, even if the networking through Ethernet cannot be used.

Results Figure 6.11 shows the results. The partitioned container survives the crash since it does not depend on Linux except for the management networking via Ethernet. Conversely, Linux and other sandboxed containers failed because they depend on the Linux kernel, either because it is the hosting kernel or the service VM providing access to hardware.

Chapter 7

Related Work

The only constant in our field is change.

Ravishankar K. Iyer

This chapter provides an overview of related research work regarding the timeliness and dependability of both container orchestration systems and containers, with a particular focus on works targeting industrial settings.

The related work is divided into three sections, including the subsystems involved in the orchestration workflow of a container (see Figure 5.1), recalling the structure of chapter 5. Each section contains two parts: the related work on dependability and the ones regarding timing aspects. Some of the works analyzed may lay in the intersection between those two aspects, and thus are analyzed in both parts.

Section 7.1 focuses on the work done on the control plane and its orchestration commands, including both functionalities implemented (e.g., container state migration, pod state machine replication) and non-functional characteristics. Section 7.2 focuses on the work done regarding the placement policies, including new placement algorithms that account for additional information besides constraints and resource availability. Section 7.3 focuses on the stack of technologies of the worker node used by a container, including a wide range of virtualization technologies and networking stacks, each with different characteristics.

7.1 Related Work Targeting the Control Plane

7.1.1 Dependability assessment and improvement

This subsection focuses on: i) the dependability assessment and error/-failure analysis of the control plane logic, and ii) control plane features to improve services' dependability. A failure affecting the control plane logic can cause incorrect reconciliation of current and desired states or control plane unavailability.

Real-world data analyses From an FFDA standpoint, following our research, in [121] the authors analyzed 210 operator bugs from 36 K8s operators, investigating root causes, manifestation, impact, and correction. The study revealed that most of the operator bugs can be triggered with no more than 3 operation requests, and that half of the operator bugs involve manipulating built-in resources. Among the most impactful bugs, 31 caused operators to repeatedly crash and restart and 10 caused the cluster to constantly switch between different states.

I remark that the control plane dependability assessment must be considered only partial information in a more complex cloud system. Indeed, the authors of [122] analyzed production failures, showing that they cannot be understood by analyzing a single system in isolation.

From an analytic point of view, stochastic models (like reliability block diagrams and continuous time Markov chains) were used to analyze the availability of OpenStack in [31], performed failure detection and classification [123], and analyzed the dependability bottlenecks of two software defined network (SDN) controllers [124]. Although SDNs are not container orchestration systems, they have a similar architecture, as they feature a distributed control plane and a data plane.

However, since complete real-world data are not often available, the use of stochastic models is often coupled with fault injection to accelerate errors and failures. Furthermore, cloud systems are designed to be resilient and withstand errors and failures because of their frequency. Therefore, fault injection is also used to effectively test the adequacy of fault tolerance mechanisms.

Fault injection for bug discovery The authors of [125] created a testing framework for custom K8s controllers by perturbing the controller's view of

the current cluster state through stale-state, crash, and unobserved states. That was followed by [126], which describes how end-to-end tests can be generated to trigger state changes from states different from the initial one. Oracle state checking was used to detect misbehavior. The approach described in [127] looked for crash-recovery bugs in distributed systems by injecting crashes at precise points identified through the analysis of meta-information used by nodes. Similarly, in [128], time-of-fault bugs (i.e, bugs in which the timing of a crash is critical to determine manifestation) were found by identifying conflicting operations based on correct runs, and exploiting the ones not covered by fault-tolerance mechanisms. The authors of [129] injected network partitions to discover partition bugs. The injections were performed when consistency invariants were violated. Among the cloud systems targeted, there was Cassandra and YARN. Simple random partitions were revealed to be useful in [130] in which the probability bounds for discovering bugs were derived. The authors of [131] argued that partial histories, including staleness, time traveling, and observability gaps, are an inherent threat to distributed systems that locally cache their state, as K8s does. They can be used to improve the tests of the components, which must assume that the state they are seeing may differ from the state on the data store. The authors of [132] described how violations of invariants can be used to detect deterministic bugs, and how semantic-equivalent input transformation through symbolic execution can be performed to recover dynamically.

Fault injection for resilience assessment Past cloud management platforms like OpenStack were widely analyzed in terms of resilience and failure modes. The authors of [133] performed a fault injection campaign and subsequently analyzed the resulting failures, showing that OpenStack exhibits a non-fail-stop behavior and failures propagate undetected in the system. The analysis was extended in [123], in which the authors proposed an unsupervised machine learning algorithm (i.e., Markov models and clustering) to automatically derive the failure modes from the data. The authors ultimately claimed the need for thorough run-time verification and fault containment. The authors of [134] analyzed the robustness of OpenStack-based SDN controllers, including VM spawn time in the presence of network delay and bandwidth limitations. Unfortunately, modern container-based systems are different from past cloud management platforms, in terms of both system scale and

functionalities of the system.

The authors of [135] used fault injection including anomalous resource consumption, long response times, and error response codes to study the effectiveness of K8s at handling the aging and faults of deployed microservices. They concluded that probes fall short in detecting several failure modes.

In 2016, Netflix introduced Chaos engineering [35], which automatically, randomly, and deliberately introduces faults through injections in production systems to find and improve dependability bottlenecks. Chaos engineering effectiveness relies on the simple fault/error model that can be applied without being aware of services' semantics, highlighting the bottlenecks of a complex system topology.

Improving control plane fault tolerance The papers [136] provided reviews of the most common fault-tolerance mechanisms used in cloud environments. While most of the recent research focused on improving services' fault tolerance and resilience, some works also targeted the fault tolerance of the control plane itself. In [137] the entire K8s control plane was made Byzantine fault tolerant through state machine replication. The work in [138] did something similar for SDN control planes.

Improving services fault tolernace In [139, 140], K8s was modified to offer automatic and transparent state machine replication to provide replicated services with high availability along with Byzantine fault tolerance and strong consistency. The proposal integrated both services coordination and a leader election protocol.

The authors of [52] argued that the automated workflows implemented by the orchestration commands are a chance to improve the recovery performance in a distributed control system scenario, complementing the manual replacement of failed hardware controllers. Nonetheless, the authors did not modify the vanilla orchestrator.

Conversely, multiple research works tried to increase the services' availability by reducing the repair times [141, 50, 142]. In [141], the authors proposed a custom K8s controller to manage stateful and high availability (HA) services. The controller distinguishes an active from a standby Pod, and provides a mechanism for the active Pod to replicate its state data to its standby Pod. Upon failure of the active Pod, the controller inserts the standby pod

in the Endpoints for the HA Service, routing the service traffic to it. In [50], the authors extended K8s with custom controllers to perform state migration of industrial controllers to support seamless application updates with strict timing requirements. Similarly, in [143] the authors proposed a stateful service migration solution for industrial machine learning applications, and, in [144], the authors used remote direct memory access (RDMA) to migrate the state of industrial virtual PLCs. For this latter work, the integration into orchestrators like K8s was reported as future research.

In [145] the authors implemented a custom K8s controller to integrate the support of checkpoint and restore in user space for K8s services. The controller automatically performs periodic checkpoints of the running pods.

In [142], the authors proposed an architecture to support the seamless migration of offloading services while serving moving users with the nearest edge server.

In [146], the authors modelled an application as a DAG of microservices. Assuming either replication or temporal redundancy (i.e., request re-send) as fault tolerance methods, they determined the deployment that can guarantee a given response time while minimizing the number of replicated tasks.

7.1.2 Orchestration times analysis and improvement

This subsection focuses on works that analyze the timing of the control plane or modify it to speed up the scaling, deployment, and migration of services. In this perspective, the timing characteristics of the control plane affect the applications' non-functional properties. Hence, some of the works reported in this subsection were mentioned in the previous sections from a dependability perspective, as they lay at the intersection between dependability and timeliness.

Reducing downtimes In [50], the authors set strict timing constraints for the state migration of the industrial controllers. In particular, they target sub-cycle timing requirements ($\approx 10ms$) for the state migration in order to avoid control disruption. Similarly, the work in [144] targeted "failover times in the single millisecond area", and used custom high-frequency heartbeats and state synchronization with a hot spare replica to reach the goal. In [143] the authors divided the state into a hot state and a cold state, where the hot state is the state part necessary to restart the service. During the migration, the pod

can be started after moving only the hot state to reduce downtimes, while the cold state is eventually moved. The authors also used destination container pre-creation to reduce migration times. In [141] the failover completely relied on K8s control plane, hence measured times are in the order of seconds. However, the times are measured in low-load conditions.

In [142], the authors leveraged the division of container images in layers to reduce the amount of data to transfer and thus reduce the handoff times in constrained bandwidth conditions.

Reducing startup times Startup times of containers directly impact the services SLOs, above all when frequent scaling is required. Increased startup times (e.g., when *cold starts* happen) are, for example, a serious issue for FaaS paradigms, in which the software units to be run are short-lived and are frequently orchestrated. In FaaS case, a cold start includes the time to allocate and start a container, and the time to bootstrap the runtime for the function. Startup times can be reduced by borrowing the solutions investigated in several works concerning FaaS cold starts. Those solutions can be divided [147] into application-based [148, 149], checkpoint-based [150], prediction-based [151, 152, 153], and cache-based [154]. Those techniques involve pre-warming containers, scheduling, and reusing containers, using the pool of warm containers, and keeping containers alive. They aim to reduce as much as possible the activities to be performed to start a new container, including downloading it, setting up a sandbox, initializing a language runtime and the application code.

The work in [111] showed that pre-creating containers' network and directly contacting the container manager on the worker nodes in K8s (i.e., not using orchestration functionalities) enables meeting deadline-driven SLOs.

Analyzing orchestration times While previous works extended orchestrators to add features to support state migration or standby replicas, they did not modify the core components of the orchestrator control plane.

Hence, the orchestration times directly affect the failover, scaling, and deployment times of those solutions (when they rely on the K8s control plane). Therefore, it is essential to analyze the orchestration times.

The authors of [13] designed a tool for systematic benchmarking of different orchestrators. In particular, the tool defines some orchestration com-

mands common to multiple orchestrators (e.g. start, restart, failover, crash, update, remove), and implements those commands for K8s and Nomad in a parametric way, catching the timing of main events.

In [24] the resource consumption and the performance of both control plane and worker nodes were compared for different lightweight K8s distributions. Remarkably, those distributions reuse the K8s code base, stripping down functionalities and re-packaging the components to reduce resource consumption. The authors reported that K3s and K0s showed the highest control plane throughput (in terms of pod creation) and MicroShift showed the highest data plane throughput.

The authors of [100] showed that the strong consistency of Etcd in K8s is a main limitation for scalability in edge environments where consensus messages must be exchanged over the network. Thus, they proposed to turn the datastore into an eventually consistent store. Nonetheless, the proposal was not implemented and tested.

In [111] and [83] the declarative nature of K8s was judged as a threat to predictable orchestration times. In particular, the authors of [111] stated that "non-deterministic delays appear when multiple containers are created in parallel". Deployment time was also a concern in [110], where the authors showed that a hierarchical orchestrator enhances scalability in edge computing environments.

7.1.3 Relationship of this dissertation to existing work

Dependability This dissertation performs a resiliency assessment of orchestrators and provides a taxonomy of their failures, similarly to previous studies targeting past cloud platforms [133, 155, 123]. However, this work faces different issues and provides dissimilar insights due to the inherent differences concerning those systems, which were characterized by a lower system scale.

Unlike resilience assessment studies involving orchestrators, like Chaos engineering, this work does not target the deployed services or the computing infrastructure. Conversely, I injected faults/errors into the orchestrators' internals, leveraging the system knowledge to design *ad hoc* methods. Indeed, I claim that Chaos-engineering–like methods should be applied to the orchestrator components themselves and integrated into the business processes. However, standard Chaos engineering tools are not suited for orchestrators, as they inject simple faults/errors (e.g., latencies, crashes, HTTP er-

rors) that are agnostic of the microservices and effective in a complex interaction topology. In K8s, the limited number of components and the well-known interaction patterns make simple errors well-tolerated.

Similarly to us, other works have already used *ad hoc* injections into orchestrators' internals. However, I focus on the propagation of faults/errors and the failure modes of orchestrators, while they aim primarily at bug discovery. Indeed, they leveraged the inherent weaknesses of distributed systems to inject simple faults (e.g., crash and stale states) in strategic ways and expose bugs and flaws of components within an orchestrator. In addition, compared to [121], which mainly analyzed the effects of bugs, this dissertation also considers that the interaction of flawless components can lead to system failures due to hardware faults.

Finally, I remark that even Byzantine fault-tolerant[1] state machine replication for either services or the control plane does not guarantee complete fault tolerance. Indeed, state machine replication cannot mitigate common cause failures, e.g., deterministic failures due to misconfigurations, mistakes, bugs, and upgrades.

Orchestration times Recent research focused on services' SLOs by carefully managing state transfer, replicas, and pre-warmed sandboxes to increase availability and reduce scaling and startup times [141, 50, 111, 154]. However, those works either avoid relying on K8s or extend K8s but leave untouched the core components of the control plane on which they rely.

Extending K8s inherently neglects the variability of orchestration times due to the complexity of the core components and their interactions. The time required by the control plane management to handle evens and state migrations can be delayed due to several factors. For example, upon a pod failure, switching the traffic to one of the replicas requires some control plane management, which can be delayed when the orchestrator is busy managing concurrent orchestration workloads (recall Figure 4.9).

This dissertation tackles the problem of nondeterminism and delays introduced by core orchestration components in the presence of errors or overloads, complementing the mentioned works. Nondeterministic orchestration times were already highlighted in [111] for K8s. The authors of [111] worked around the problem by directly issuing commands to the worker nodes instead

[1] Byzantine faults are faults with completely unpredictable behavior.

of the orchestrator. Nonetheless, this solution manually acts instead of the orchestrator, preventing the user from using K8s orchestration functionalities. Conversely, I performed a fine-grained timing analysis of orchestration times and carefully analyze the sources of nondeterminism proposing architectural modification.

7.2 Related Work on Placement Algorithms

The placement is one of the essential tasks of an orchestrator since it decides the worker node where to place a pod. The majority of the research papers analyzed this aspect. This section reports the most relevant works to this dissertation, while the reader can refer to the dedicated surveys ([156, 157]) for a complete portrait of the state of the art.

7.2.1 Dependability-aware placement

The authors of [158] stressed that in edge and industrial environments, reducing container startup latency is increasingly important to ensure low end-to-end latency. They considered pulling times and claimed that orchestrator schedulers must account for task dependencies, placing a task at a node with the maximum number of the task's dependencies stored locally.

The authors of [159] designed a K8s scheduler that aims to improve the reliability of edge applications through the placement of spare backup resources. The scheduler considers the latency constraints of the services, tracking the communication delays between nodes.

In [160] the authors proposed a scheduling for executing in cloud real-time applications represented as DAG. The authors optimized the frequency in order to meet the deadline while keeping the requested reliability. The proposed policy dynamically calculates an optimal number of task recoveries, maintaining the deadline and MTTF threshold.

In [161] the authors formulated the placement problem as a reliability-aware placement of k-out-of-n serial-parallel block diagram, guaranteeing a minimum throughput under failures. Monte Carlo simulations were used to calculate reliability-related parameters based on independent failures of fog devices.

In [162], the failure-repair of container replication configurations was analyzed via a state-space and a non-state-space model.

Older works considered similar problems for VM placement. For example, in [163] an integer linear programming (ILP) model was proposed to find the optimal node where to deploy a VM to minimize the MTTR.

7.2.2 Placing containers with timing requirements

Recent work in literature extended K8s to support real-time containers by integrating real-time schedulability tests [72, 75, 71, 164, 165, 76]. They are individually examined in the following paragraphs.

The authors of [72] introduced RT-Kubernetes: a modified version of K8s that supports the real-time CPU resource. The Kubelet of RT-Kubernetes was modified to manage real-time containers (more details are presented in §7.3), while the K8s Scheduler was enhanced to keep track of the real-time CPU usage of the nodes. The Scheduler performs a utilization-based schedulability test of the real-time containers that relies on period and runtime, following the HCBS theory.

Similarly, the work in [71, 75] modified the K8s Scheduler to perform admission control through schedulability analysis of real-time containers, co-located on the same worker nodes with best-effort containers. In addition, the paper presented a system of multiple controllers to dynamically resize the resources of real-time containers and meet timing requirements. Real-time containers were implemented in the same way as RT-Kubernetes.

The authors of [164] proposed something similar to [71, 75], but they aimed to overcome the problem that "the offline phase does not scale for real industrial systems and that the online adaptation of containers is based on simple metrics that do not capture the utility of computation results after a deadline miss". To this aim, they based the orchestration decisions on the time-utility functions and proposed three heuristics to overcome scaling problems.

In [165], K8s was extended to support mixed-criticality robotic applications. In addition to the utilization-based schedulability analysis, they also modified the K8s scheduler to match Pods' criticality with nodes' criticality. The criticality levels are taken from the automotive safety integrity levels (ASIL) standard. For low-criticality tasks, suitable worker nodes with higher utilization are preferred, while for high-criticality tasks, worker nodes with

low utilization are preferred. The authors also implemented a runtime migration mechanism that is triggered when a real-time container presents many deadline misses.

Several extensions to the K8s Scheduler were proposed in [76] in order to deal with inherently shared resources, such as cache and memory bandwidth. The authors divided the resources into discrete bands and resized Pod resources for a collaborative QoS reduction.

Previously, similar attention was given to cloud management platforms like OpenStack [66, 73]. A modified version of OpenStack was proposed in [66], where real-time containers have been integrated to foster the NFV management of next-generation networks. Field data was compared with analytic queuing theory models, showing results close to the expected. In [73] OpenStack was modified to integrate real-time Virtual Machines for Xen, using the compositional schedulability analysis.

From a placement algorithm perspective, recent works formulated resource allocation problems to keep into account network latencies [166, 167, 159, 168, 169]. The work in [166] presented an architecture for real-time FaaS. The authors designed an analytical model that considers the end-to-end latency of real-time request/response pairs and provided three possible architectures for real-time networking in the cloud (i.e., hardware, software, and hybrid). Furthermore, the authors used a partitioned-EDF scheduler and designed a partitioning algorithm designed to minimize the number of used compute nodes. The scheduling algorithm takes into account the different execution times of a task on different worker nodes.

The authors of [167] proposed a greedy border allocation algorithm to minimize communication costs in fog/edge environments for industrial automation contexts. The cost function was expressed in terms of a generic network distance, which may be a function of the amount of data transferred between sensing, computing, and actuating devices, as well as communication latency. The work in [159] (already analyzed for the placement of spare replica) also considered the communication latency between nodes. On a similar note, the work in [168, 169] extended the K8s scheduler to be aware of the network latencies. In particular, the network latencies between nodes are periodically updated through probes, and the affinity rules that tend to co-locate pods on the same worker node are updated dynamically to account for the communication between two pods. Differently from other works account-

ing for latencies, in [168, 169] network conditions are periodically evaluated to trigger pods' rescheduling when the communication cost can be further diminished. Similarly, the work in [170] extended the K8s Scheduler to account for the amount of traffic exchanged between communicating services in order to colocate them and provide an overall lower latency.

Although focusing on looser requirements, the authors of [171] proposed a placement algorithm to minimize service latencies, by predicting the position of vehicles and migrating containers to the server that can offer minimum latencies to users.

Other works, like [172, 173, 174, 175] analyzed the interference-aware placement and resource allocations, which can cause increased latencies to the services when co-located with other applications that cause resource contention. However, those works are not the main focus of this dissertation.

7.2.3 Relationship of this dissertation to existing work

The mentioned research works try to extend the current cloud platforms to support real-time services or services with latency requirements. However, they neglect the heterogeneity of the industrial environments in multiple ways, which is a focal point of this work.

Only the authors of [165] mentioned services for robotic applications with different criticality levels, which must match worker node criticality levels. Nonetheless, the authors considered only real-time containers based on Linux even for high criticality levels, and they did not provide any information on how to evaluate the criticality of a worker node. In other words, they do not have a holistic approach as this dissertation does. Moreover, it is worth noting that the work in [165] is later (or concurrent) than the works presented in this dissertation.

In the same way, also the other works rely on Linux-based containers. Although real-time containers based on Linux are a practical solution, I proposed solutions to support also *ad hoc* hardware like RPUs and lightweight RTOSes and fully the heterogeneity. Indeed, in many industrial settings, Linux is not yet a viable option to host critical tasks, instead hypervisors, dedicated hardware, and certified software systems is preferred.

On a similar note, the reliability-aware placement algorithms were mainly designed for cloud environments composed of similar worker nodes. Hence, they generally rely on measurement-based reliability estimation, which does

not catch the heterogeneity of mission-critical and edge environments.

In addition, only the authors of [166] considered for the real-time schedulability analyses an application execution time that depends on the worker node. This is paramount in a heterogeneous environment where hardware can range from embedded boards to micro-datacenters.

7.3 Related Work on Container Technologies

7.3.1 Improving container dependability

Sandboxed containers The definition of container introduced in §2.1 does not necessarily imply the use of OS-level virtualization. Hence, containers can be implemented using other virtualization/sandboxing technologies. Indeed, OS-based containerization can suffer from a reduced level of security, failure, and performance isolation due to the kernel shared between applications. This jeopardizes the application of OS-based containers for industrial environments and opens to alternative containerization technologies, e.g., hypervisor-based containers.

I refer to [176] for an in-depth analysis of containerization technologies, while I summarize the main alternatives to OS-based containerization. I also refer to [177] for a complete survey of real-time virtualization technologies used in industrial environments.

Sandboxed containers run unikernels or applications inside VMs with minimal kernels (i.e., *microVMs*) to offering improved isolation guaranteed by hypervisors and guest kernels while exhibiting overhead and startup times comparable to containers. A container runtime transparently sets up the environment, exposing no or little differences to the upper layers of the container software stack (recall Figure 2.3).

Sandboxed containers can offer multiple advantages: from a security perspective, applications rely on their kernel rather than sharing it with all colocated applications; from a performance perspective, a minimal kernel (e.g., unikernel) can provide faster boot times, better scalability, and reduced virtualization overhead compared to containers. Moreover, hypervisors are generally characterized by a reduced code base compared to fully-fledged OSes like Linux. This reduces the trusted computing base for an application.

Table 7.1 summarizes the main OCI-compliant runtimes. Some rely on

Table 7.1. State of the practice solutions for sandboxed containers.

Solution	Guest Type	Hypervisor	Orchestration
runc, crun, etc. (Container)	Linux app.	No	OCI compliant
vSphere VIC	VMs and Containers	VMware ESXi	vSphere
LightVM	microVM/Unikernel	Xen	N/A****
Firecracker	Light VM	KVM	N/A****
runnc (Nabla Container)	Single binary (Unikernel)	Nabla Tender**	OCI compliant*
runsc (gVisor)	User-space kernel + app.	No	OCI compliant
KubeVirt	VMs	KVM	Kubernetes
runV (Kata Container)	microVM (Linux)	QEMU, ACRN, Firecracker (KVM)	OCI compliant
runX	microVM	Xen	OCI compliant
runu (Unikraft)	Single binary (Unikernel)	QEMU (KVM)***	OCI compliant*
runwasi WASMedge	WASM runtime + WASM app.	No	OCI compliant*

* Although is OCI compliant, it can only handle container images containing *ad hoc* binaries (e.g., unikernel, WASM).

** Tender is a user-space unikernel monitor rather than a hypervisor.

*** Although Unikraft unikernels also support Xen and Firecracker, runu only provides an implementation for QEMU.

**** The solutions propose new VMMs rather than container runtimes for orchestration. They can be integrated into orchestrators through external projects.

user-space runtimes (Nabla, gVisor, and WASMedge), which forward selected syscalls to the host OS. Some others rely on type-2 hypervisors (i.e., QEMU, Firecracker, KVM) which directly depend on the host kernel. Finally, other solutions use type-1 hypervisors (ACRN, Xen) with paravirtualized guests, which rely on PVM drivers to access hardware resources.

LightVM [178] runs unikernels or lightweight VMs based on stripped-down Linux kernels, managed by a modified Xen to improve scalability. Amazon Firecracker is a virtual machine monitor (VMM) that leverages the KVM hypervisor to run miroVMs based on minimal and optimized Linux kernels [179]. Firecracker replaces QEMU with a minimal VMM providing only essential functionalities for serverless. *IBM Nabla*[2] [180] packs applications as

[2] `https://nabla-containers.github.io/`

unikernels, that execute on top of a *tender*, i.e., a monitor that uses no virtualization but executes unikernels as OS processes allowing only 7 system calls. *Google gVisor*[3] creates a dedicated guest kernel for running containers, that intercepts application system calls through *system call interposition*, and only a limited subset of system calls are managed by the host OS. Both *KubeVirt*[4] and *vSphere Integrated Containers (VIC)*[5] integrate VMs and containers under a single orchestration infrastructure. *Kata Containers*[6] place a containerized application in a dedicated VM with a minimal kernel. Xilinx proposed *runX*[7], which uses Xen hypervisor to run containers in multiple separate VMs, either with the provided custom-built Linux-based kernel, or with container-specific kernel. The authors of Unikraft [181] implemented *runu* to make containers shipping unikernels OCI compliant and start them on QEMU [8]. Industry players like Bosch and Ericsson explored WebAssembly (WASM) for real-time industrial applications [182, 51]. WASM runtimes provide sandboxing, memory safety, control flow integrity, fault isolation, and no access to code addresses. Several projects were created to WASM runtimes OCI-compliant for orchestration. I refer to [176] for further details.

The listed solutions do not target real-time requirements, which directly depend on the real-time capabilities of the hypervisor/host used.

Partitioning Hypervisors Partitioning hypervisors are a small software layer that exploits hardware-assisted virtualization (e.g., Intel VT-x, ARM VHE) to statically allocate hardware resources to VMs [183, 184, 185, 186, 187]. This design avoids the need for hardware emulation drivers and resource scheduling, allowing a minimal code base that executes only when a VM tries to access unauthorized resources. Once a partition for a VM is created, the VM executes with almost no hypervisor interposition. Hence, the hosted VMs directly control hardware resources allotted to them. In [113] the authors proposed the Omnivisor model. The model extends the capability of static partitioning hypervisors to run VMs on co-processors such as RPUs or softcores while keeping the partition isolated from a temporal and spatial point

[3] `https://gvisor.dev/`
[4] `https://kubevirt.io/`
[5] `https://vmware.github.io/vic-product/`
[6] `https://katacontainers.io/`
[7] `https://github.com/Xilinx/runx`
[8] `https://unikraft.cloud/blog/containers-and-unikernels/`

of view. I refer to [112] for a complete analysis of the current status of partitioning hypervisors.

Partitioning hypervisors are designed specifically for industrial environments, due to the high level of resource isolation and the reduced code base. They prioritize simplicity over features, following the guidelines of industrial standards to cope with certification. Nevertheless, partitioning hypervisors require a remarkable manual configuration effort, and are not supported by cloud orchestration tools.

Some examples of partitioning hypervisors are the following. *Jailhouse* [184] by Siemens enables asymmetric multiprocessing to assign hardware resources to isolated partitions at runtime. It leverages Linux only for booting to keep a minimal codebase. After the hypervisor activation, Linux acts as a management VM that manages the lifecycle of other partitions. Jailhouse was extended in [113] to comply with the Omnivisor. *Bao* [183] is an open-source hypervisor for edge devices that does not rely on Linux, but the partitions are statically defined at boot time. Similarly, *ACRN* [186] targets edge devices based on Intel processors, but it can have both partitioned or paravirtualized VMs, alongside a management VM that has the same role as in Jailhouse. Unlike general-purpose hypervisors, a failure of the management VM in Jailhouse or ACRN does not cause the partitioned VMs to fail, since they directly access hardware resources. *Xtratum* [188] by fentISS and *PikeOS*[9] by SYSGO are instead popular commercial partitioning hypervisors.

7.3.2 Real-time containers

Consolidated real-time operating systems like VxWorks by WindRiver recently introduced the support of an OCI-compliant container engine [7]. Due to the closed-source implementation, it is hard to get technical details. Besides industry products, research studies also explored the use of containers to run real-time tasks. I refer to the surveys in [189, 190] for a complete review regarding real-time containers for industrial environments. Summarizing, the most popular solutions include i) the use of Linux with the `PREEMPT_RT` patch [191]; and ii) the use of real-time Linux co-kernels (e.g., RTAI and Xenomai [192, 117]), and hierarchical schedulers or distributed monitors to manage the hard real-time tasks managed by the co-kernel.

[9] `https://www.sysgo.com/pikeos`

In particular, the proposals in [72, 165, 75, 66] use Linux-based real-time containers, which leverage the *rt-cgroups* to group real-time processes and schedule them with a patched version of `SCHED_DEADLINE`, modified to be hierarchical in [101]. In [75], a monitor allows dynamic resize of the CPU bandwidth allotted to the real-time containers to reduce the timing failures.

The authors of [64] designed an allocation algorithm of the CPUs and CPU time of a worker node to real-time containers, prioritized containers, and best-effort containers. The authors use the virtual Base-Band Units used for networking as real-time containers. The algorithm shows improvements in terms of tail latencies of the applications compared to the default allocation scheme on a real-time kernel.

In addition, recent research works proposed different architectures to integrate TSN into orchestrators [55, 54, 53, 193, 166] to support real-time applications and services, with particular focus on industrial applications, where TSN is becoming a standard. These works mainly rely on strategies like kernel bypassing and/or OS-level daemons to perform packet scheduling. The problem of the virtualization of a TSN network was also examined in [194], where authors demonstrate that latency-sensitive applications can execute in virtual machines without disruptions to their network operations. In the paper, the authors rely on a precise clock synchronization method to support the TSN protocol inside the VMs. On a similar note, the integration of RTnet based on TDMA protocol shipped with Xenomai into real-time containers was proposed in [117]. RTnet works with a pluggable real-time Medium Access Control (MAC) policy on standard Ethernet hardware.

Although not focusing on the implementation of container runtimes, there are some works worth noting that deal with real-time unikernels. In [74], the author compared real-time applications that use different virtualization technologies and guest OSes. The results showed that KVM with a suitable microVM and the OSv unikernel are viable alternatives to traditional VMs or containers for real-time virtualization in terms of activation latencies and boot times. Nonetheless, in [74] there is little characterization of the stress load competing with the real-time tasks.

The authors of [195] also showed the feasibility of scheduling paravirtualized unikernels with a deferrable server scheduler in Xen, deriving a worst-case response time analysis for self-suspending tasks.

7.3.3 Relationship of this dissertation to existing work

From a temporal perspective, containers rely on kernel mechanisms that do not support advanced isolation techniques (e.g., cache partitioning, memory bandwidth regulation, and hardware virtualization support). Similarly, the sandboxed containers do not support advanced isolation techniques to mitigate the interference on shared resources[10].

From a dependability perspective, co-located containers run on a shared kernel and sandboxed containers depend on a common host kernel or management VM. This dependency represents a security and reliability issue: real-world stories showed that a guest can escape the VM, and cause host resource exhaustion and host crashes, which affect all hosted applications[11].

Although runX and KataContainers use type-1 hypervisors (Xen and ACRN), which can be configured as partitioning hypervisors (e.g., Xen Dom0-less configuration), those runtimes only support paravirtualized Linux VMs, which depend on a single management VM, and do not support other minimal RTOSes.

The proposals of [74] and [195] are interesting research proposals, but they leave an open gap with technologies of industrial settings, which demand the highest level of dependability

Conversely, the proposed partitioned containers take advantage of partitioning hypervisors, which are specifically designed for industrial environments. Partitioned containers do not depend on any other kernel when they run and can take advantage of advanced isolation techniques of partitioning hypervisors. This guarantees an improved freedom from interference, in terms of execution times interference and cascading failures, which is required by industrial standards. Moreover, partitioning hypervisors prioritize simplicity over features, allowing them to cope with certification (e.g., ARINC-653, DO-178C, ISO 26262). runPHI tries to overcome the problem of remarkable manual configuration efforts for partitioning hypervisors, providing a way to integrate partitions into orchestrators.

[10] Xen supports cache coloring, but runX is a no longer maintained prototype.

[11] Some examples can be found at `https://github.com/firecracker-microvm/firecracker/issues/1088`, `https://github.com/firecracker-microvm/firecracker/issues/3542`, `https://forum.manjaro.org/t/solved-kvm-causes-kernel-panic/50812/4`, `https://googleprojectzero.blogspot.com/2021/06/an-epyc-escape-case-study-of-kvm.html`, `https://github.com/kata-containers/kata-containers/issues/3373`

Chapter 8

Conclusions

Once you have known what it is like to fly, you will walk on the ground looking up to the sky, because that is where you have been and that is where you will wish to return.

Leonardo Da Vinci

This dissertation investigated whether current container orchestration systems meet mixed-criticality requirements and how they can be modified to support them. An analysis of real-world failures of Kubernetes (i.e., the most widely used container orchestration system) and fault/error injection experiments revealed that, although it can tolerate a variety of faults and errors, even a single error can disrupt an entire cluster. Besides this, experimental results of a timing analysis showed that the orchestration times of critical services may be delayed by even tens of seconds, affecting the application SLOs. Those results suggest that Kubernetes is not ready yet to host real-time and critical applications. However, a careful investigation indicates that those problems are not a matter regarding only Kubernetes. Indeed, the issues are in the architecture, underlying assumptions, and models of the environment, which are common to several container orchestration systems.

On the one hand, the architecture of container orchestration systems is designed to achieve unprecedented system scales. Thus, an error can propagate, be amplified, and cause overloads.

On the other hand, container orchestration systems are not able to pri-

oritize the management of a critical service because they assume the services to be equally critical and the nodes to provide similar guarantees. Indeed, although there is some basic support for different container runtimes or scheduling priorities, those concepts are not used as main design principles in all system components, and can only be leveraged for the placement through manual customizations.

For this reason, I proposed a model for mixed-criticality services, introducing the notion of service criticality and node assurance as core concepts upon which an orchestrator must be built. The model enables a new set of orchestration commands that can benefit mixed-criticality systems, like diversified replication, seamless migration, and diversified rolling updates. Those commands are intended to embed the notion of criticality in every aspect of the orchestration.

Hence, I designed architectures for mixed-criticality orchestrators and partitioned containers to implement the proposed model. Partitioned containers are applications running in an isolated partition managed by a partitioning hypervisor, but managed as normal containers by the orchestrator. The orchestrator architectures are based on the concept of separation of concerns to isolate critical services, preventing failure propagation and temporal interferences. The architectures include SLO-aware components that implement differentiated policies to manage services based on their criticality.

An implemented prototype of the proposed solutions based on Kubernetes showed that mixed-criticality orchestration prevents temporal interferences caused by concurrent orchestration workload that affect the orchestration times of critical services. In addition, partitioned containers exhibited timing predictability and isolation from timing interference and failures when heterogeneous processing units, like RPUs, are involved.

8.1 Future Directions

This dissertation proposed the integration of systematic injections into the testing and Chaos-engineering disciplines to detect misconfigurations early and increase the overall system resilience. However, the Chaos-engineering approach is not viable for critical systems, as it deliberately introduces faults and errors. In other words, there is still an open question on how to build resilient orchestrators.

Furthermore, as applications evolve towards the FaaS paradigm, other layers that can be prone to failures are stacked upon an already complex infrastructure. For example, current FaaS platforms are built upon Kubernetes. How orchestrator failures propagate in such paradigms must be investigated.

The other two architectures (besides the patched orchestrator) for mixed-criticality orchestrators could be implemented to compare them and quantify the difference in terms of isolation.

Besides architectural aspects, the diversified orchestration commands introduced and the concepts of mixed-criticality orchestration can potentially lead to formal analyses and models dedicated to orchestration logic. Methods should be developed to automatically decide the optimal number, criticality, and redundancy scheme of pods in order to guarantee the desired dependability of critical services.

From a theoretical point of view, the orchestration of real-time components raises a non-trivial problem: the real-time tasks cannot have an WCET analysis for all the hardware platforms available in the environment. While recent research neglected this problem because it targeted soft timing requirements, the shift to stricter timing requirements for industrial applications needs a clear answer. In this sense, the position paper [196] provides a great overview of current problems.

Furthermore, from the perspective of virtualization technologies, solutions should be further explored for partitioning hypervisors to cope with hardware resources that cannot be reserved only for one partition (e.g., a network interface), like in [115].

In addition, our proposal of partitioned containers raises the following question for future virtualization technologies: "*what actually is a container*?". Indeed, although partitioned containers currently do not use the filesystem layers of other OCI containers, they still enable the seamless orchestration of critical applications over special-purpose hardware. In this sense, current containerization solutions blend two different properties: the *portability*, which allows writing the application once and running it everywhere and translates into the OCI image standard, and the *orchestrability*, which provides a standard way of integrating an application into orchestration tools and translates into the OCI runtime standard.

Bibliography

[1] A. Burns and R. I. Davis, "Mixed criticality systems-a review," *Department of Computer Science, University of York, Tech. Rep*, 2022.

[2] Robert N. Charette, "This car runs on code." `https://spectrum.ieee.org/transportation/systems/this-car-runs-on-code`. Accessed June 12, 2025.

[3] D. McCandless, P. Doughty-White, and M. Quick, "Million lines of code." `https://informationisbeautiful.net/visualizations/million-lines-of-code`. Accessed June 12, 2025.

[4] J. Morgan, M. Halton, Y. Qiao, and J. G. Breslin, "Industry 4.0 smart reconfigurable manufacturing machines," *Elsevier Journal of Manufacturing Systems*, vol. 59, pp. 481–506, 2021.

[5] A.-W. Colombo, S. Karnouskos, and J.-M. Mendes, "Factory of the future: A service-oriented system of modular, dynamic reconfigurable and collaborative systems," in *Springer Artificial intelligence techniques for networked manufacturing enterprises management*, pp. 459–481, Springer, 2010.

[6] M. Bortolini, F. G. Galizia, and C. Mora, "Reconfigurable manufacturing systems: Literature review and research trend," *Elsevier Journal of manufacturing systems*, vol. 49, pp. 93–106, 2018.

[7] Windriver, "Container Support Architected for Embedded. Built on Standards." `https://www.windriver.com/containers`. Accessed June 12, 2025.

[8] G. Gala, G. Fohler, P. Tummeltshammer, S. Resch, and R. Hametner, "RT-cloud: Virtualization technologies and cloud computing for railway use-case," in *2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 105–113, IEEE, 2021.

[9] P. Veloso Teixeira, D. Raposo, R. Lopes, and S. Sargento, "Software defined vehicles for development of deterministic services," *arXiv e-prints*, pp. arXiv–2407, 2024.

[10] J. Vikberg, G. Hall, T. Cagenius, R. Wang, and J. Schultz, "Robustness Evolution: Building robust critical networks with the 5G System," *Ericsson Technology Review*, vol. 2021, no. 11, pp. 2–12, 2021.

[11] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1657–1681, 2017.

[12] M. A. Rodriguez and R. Buyya, "Container-based cluster orchestration systems: A taxonomy and future directions," *Software: Practice and Experience*, vol. 49, no. 5, pp. 698–719, 2019.

[13] M. Straesser, J. Mathiasch, A. Bauer, and S. Kounev, "A systematic approach for benchmarking of container orchestration frameworks," in *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE)*, pp. 187–198, 2023.

[14] I. M. Al Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, and A. Palopoli, "Container orchestration engines: A thorough functional and performance comparison," in *2019 IEEE International Conference on Communications (ICC)*, IEEE, 2019.

[15] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the tenth european conference on computer systems (EuroSys)*, pp. 1–17, ACM, 2015.

[16] A. Khan, "Key characteristics of a container orchestration platform to enable a modern application," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 42–48, 2017.

[17] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade," *ACM Queue*, vol. 14, no. 1, pp. 70–93, 2016.

[18] Docker, "Use containers to Build, Share and Run your applications." `https://www.docker.com/resources/what-container/`. Accessed June 12, 2025.

[19] A. Randal, "The ideal versus the real: Revisiting the history of virtual machines and containers," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–31, 2020.

[20] OpenContainers, "OCI Image Format Specification." `https://github.com/opencontainers/image-spec`. Accessed June 12, 2025.

[21] OpenContainers, "OCI Runtime Specification." `https://github.com/opencontainers/runtime-spec`. As of June 12, 2025.

[22] The Linux Foundation, "Kubernetes Home Page." `https://kubernetes.io/`, 2022. Accessed June 12, 2025.

[23] Cloud Native Computing Foundation, "CNCF Annual Survey 2021." `https://www.cncf.io/reports/cncf-annual-survey-2021/`, 2022. Accessed June 12, 2025.

[24] H. Koziolek and N. Eskandani, "Lightweight Kubernetes distributions: a performance comparison of MicroK8s, k3s, k0s, and Microshift," in *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE)*, pp. 17–29, 2023.

[25] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX annual technical conference (USENIX ATC 14)*, pp. 305–319, 2014.

[26] A. Avizienis, J.-C. Laprie, B. Randell, *et al.*, "Fundamental concepts of dependability," *Technical Report Series-University of Newcastle upon Tyne Computing Science*, 2001.

[27] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 610–621, IEEE, 2014.

[28] H. Khazaei, J. Mišić, V. B. Mišić, and N. B. Mohammadi, "Availability analysis of cloud computing centers," in *2012 IEEE Global Communications Conference (GLOBECOM)*, pp. 1957–1962, IEEE, 2012.

[29] R. Ghosh, F. Longo, F. Frattini, S. Russo, and K. S. Trivedi, "Scalable analytics for iaas cloud availability," *IEEE Transactions on Cloud Computing*, vol. 2, no. 1, pp. 57–70, 2014.

[30] L. De Simone, M. Di Mauro, R. Natella, and F. Postiglione, "A latency-driven availability assessment for multi-tenant service chains," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 815–829, 2022.

[31] M. Faraji Shoyari, E. Ataie, R. Entezari-Maleki, and A. Movaghar, "Availability modeling in redundant OpenStack private clouds," *Wiley Software: Practice and Experience*, vol. 51, no. 6, pp. 1218–1241, 2021.

[32] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, pp. 1–55, 2016.

[33] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *IEEE Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[34] R. Moraes, R. Barbosa, J. Durães, N. Mendes, E. Martins, and H. Madeira, "Injection of faults at component interfaces and inside the component code: are they equivalent?," in *Proceedings of 6th European Dependable Computing Conference (EDCC)*, pp. 53–64, IEEE, 2006.

[35] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[36] J. A. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems," *Computer*, vol. 21, no. 10, pp. 10–19, 1988.

[37] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*, vol. 24. Springer Science & Business Media, 2011.

[38] S. Bozhko, F. Marković, G. von der Brüggen, and B. B. Brandenburg, "What Really is pWCET? A Rigorous Axiomatic Proposal," in *2023 IEEE Real-Time Systems Symposium (RTSS)*, pp. 13–26, IEEE, 2023.

[39] R. I. Davis and L. Cucu-Grosjean, "A Survey of Probabilistic Schedulability Analysis Techniques for Real-Time Systems," *Leibniz Transactions on Embedded Systems*, vol. 6, no. 1, pp. 04:1–04:53, 2019.

[40] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *28th IEEE international real-time systems symposium (RTSS)*, pp. 239–243, IEEE, 2007.

[41] Z. Deng and J.-S. Liu, "Scheduling real-time applications in an open environment," in *Proceedings of the 1997 Real-Time Systems Symposium (RTSS)*, pp. 308–319, IEEE, 1997.

[42] R. I. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *26th IEEE International Real-Time Systems Symposium (RTSS)*, pp. 10–pp, IEEE, 2005.

[43] I. Shin and I. Lee, "Compositional real-time scheduling framework," in *25th IEEE International Real-Time Systems Symposium (RTSS)*, pp. 57–67, IEEE, 2004.

[44] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *24th IEEE Real-Time Systems Symposium (RTSS)*, pp. 2–13, IEEE, 2003.

[45] S. C. Ergen and P. Varaiya, "TDMA scheduling algorithms for wireless sensor networks," *Springer Wireless networks*, vol. 16, pp. 985–997, 2010.

[46] J. Kiszka and B. Wagner, "RTnet-a flexible hard real-time networking framework," in *2005 IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, pp. 8 pp.–456, IEEE, 2005.

[47] Y. Koren, U. Heisel, F. Jovane, T. Moriwaki, G. Pritschow, G. Ulsoy, and H. Van Brussel, "Reconfigurable manufacturing systems," *Elsevier CIRP annals*, vol. 48, no. 2, pp. 527–540, 1999.

[48] J. Mellado and F. Núñez, "Design of an IoT-PLC: A containerized programmable logical controller for the industry 4.0," *Elsevier Journal of Industrial Information Integration*, vol. 25, p. 100250, 2022.

[49] D. J. Perez, J. Waltl, L. Prenzel, and S. Steinhorst, "How real (time) are virtual plcs?," in *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8, IEEE, 2022.

[50] H. Koziolek, A. Burger, and A. P. Peedikayil, "Fast state transfer for updates and live migration of industrial controller runtimes in container orchestration systems," *Elseier Journal of Systems and Software*, p. 112004, 2024.

[51] Balador, Ali and Eker, Johan and Ul Islam, Raihan and Mini, Raquel and Nilsson, Klas and Ashjaei, Mohammad and Mubeen, Saad and Hansson, Hans and Arzen, Karl-Erik, "AORTA: Advanced Offloading for Real-time Applications." `https://lucris.lub.lu.se/ws/portalfiles/portal/161851101/AORTA_RT_Cloud_2023_4_.pdf`, 2023.

[52] B. Johansson, M. Rågberger, T. Nolte, and A. V. Papadopoulos, "Kubernetes orchestration of high availability distributed control systems," in *2022 IEEE International Conference on Industrial Technology (ICIT)*, pp. 1–8, IEEE, 2022.

[53] D. Balla, I. Moldován, M. Máté, M. Maliosz, and J. Harmatos, "Time Sensitive Industrial Applications in Kubernetes," in *2024 IEEE Network Operations and Management Symposium (NOMS)*, pp. 1–7, IEEE, 2024.

[54] A. Garbugli, L. Rosa, A. Bujari, and L. Foschini, "KuberneTSN: a deterministic overlay network for time-sensitive containerized environments," in *2023 IEEE International Conference on Communications (ICC)*, pp. 1494–1499, IEEE, 2023.

[55] L. Rosa, A. Garbugli, L. Patera, and L. Foschini, "Supporting vPLC networking over TSN with kubernetes in industry 4.0," in *Proceedings of the 1st Workshop on Enhanced Network Techniques and Technologies for the Industrial IoT to Cloud Continuum*, pp. 15–21, 2023.

[56] L. Orciari, D. Raggini, and A. Tilli, "Taming edge computing for hard real-time advanced control of mechatronic systems," *IEEE Transactions on Industrial Informatics*, vol. 20, no. 8, pp. 9898–9906, 2024.

[57] A. Biskupovic, M. Torres, and F. Núñez, "Automatic synthesis of containerized industrial cyber-physical systems: A case study," *IEEE Transactions on Industrial Informatics*, vol. 19, no. 7, pp. 8262–8273, 2022.

[58] W.-T. Tsai, Q. Shao, X. Sun, and J. Elston, "Real-time service-oriented cloud computing," in *2010 6th World Congress on Services*, pp. 473–478, IEEE, 2010.

[59] W. Tsai, Y.-H. Lee, Z. Cao, Y. Chen, and B. Xiao, "RTSOA: real-time service-oriented architecture," in *2nd IEEE International Symposium on Service-Oriented System Engineering (SOSE)*, pp. 49–56, IEEE, 2006.

[60] T. Cucinotta, A. Mancina, G. F. Anastasi, G. Lipari, L. Mangeruca, R. Checcozzo, and F. Rusinà, "A real-time service-oriented architecture for industrial automation," *IEEE Transactions on Industrial Informatics*, vol. 5, no. 3, pp. 267–277, 2009.

[61] J. Larrea, A. E. Ferguson, and M. K. Marina, "Corekube: An efficient, autoscaling and resilient mobile core system," in *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, pp. 1–15, 2023.

[62] J. Aelken, J. Triay, B. Chatras, and A. M. de Nicolas, "Toward Cloud-Native VNFs: An ETSI NFV Management and Orchestration Standards Approach," *IEEE Communications Standards Magazine*, vol. 8, no. 2, pp. 12–19, 2024.

[63] R. Botez, J. Costa-Requena, I.-A. Ivanciu, V. Strautiu, and V. Dobrota, "SDN-based network slicing mechanism for a scalable 4G/5G core network: A Kubernetes approach," *MDPI Sensors*, vol. 21, no. 11, p. 3773, 2021.

[64] A. F. Ocampo, M.-R. Fida, J. F. Botero, A. Elmokashfi, and H. Bryhni, "Opportunistic CPU sharing in mobile edge computing deploying the cloud-RAN," *IEEE Transactions on Network and Service Management*, vol. 20, no. 3, pp. 2201–2217, 2023.

[65] A. F. Ocampo, M.-R. Fida, A. Elmokashfi, and H. Bryhni, "Assessing the Cloud-RAN in the Linux Kernel: Sharing Computing and Network Resources," *MDPI Sensors*, vol. 24, no. 7, p. 2365, 2024.

[66] T. Cucinotta, L. Abeni, M. Marinoni, R. Mancini, and C. Vitucci, "Strong Temporal Isolation among Containers in OpenStack for NFV Services," *IEEE Transactions on Cloud Computing*, vol. 11, no. 1, pp. 763–778, 2021.

[67] J. Lex, M. Ulrich, R. Mader, and D. Fey, "HyFAR: A hypervisor-based fault tolerance approach for heterogeneous automotive real-time systems," *Elsevier Journal of Systems Architecture*, vol. 156, p. 103263, 2024.

[68] H. Sami, A. Mourad, and W. El-Hajj, "Vehicular-OBUs-as-on-demand-fogs: Resource and context aware deployment of containerized micro-services," *IEEE/ACM transactions on networking*, vol. 28, no. 2, pp. 778–790, 2020.

[69] P. Laclau, S. Bonnet, B. Ducourthial, T. Lin, and X. Li, "Experimental validation of user experience-focused dynamic onboard service orchestration for software defined vehicles," in *Proceedings of 2024 IEEE International Conference on Intelligent Transportation Systems (ITSC)*, 2024.

[70] N. Nayak, D. Grewe, and S. Schildt, "Automotive container orchestration: Requirements, challenges and open directions," in *2023 IEEE Vehicular Networking Conference (VNC)*, pp. 61–64, 2023.

[71] V. Struhár, S. S. Craciunas, M. Ashjaei, M. Behnam, and A. V. Papadopoulos, "React: Enabling real-time container orchestration," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8, IEEE, 2021.

[72] S. Fiori, L. Abeni, and T. Cucinotta, "RT-Kubernetes—Containerized Real-Time Cloud Computing," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pp. 36–39, 2022.

[73] S. Xi, C. Li, C. Lu, C. D. Gill, M. Xu, L. T. Phan, I. Lee, and O. Sokolsky, "Rt-open stack: Cpu resource management for real-time cloud computing," in *2015 IEEE 8th International Conference on Cloud Computing*, pp. 179–186, IEEE, 2015.

[74] L. Abeni, "Virtualized real-time workloads in containers and virtual machines," *Elsevier Journal of Systems Architecture*, vol. 154, p. 103238, 2024.

[75] V. Struhár, S. S. Craciunas, M. Ashjaei, M. Behnam, and A. V. Papadopoulos, "Hierarchical resource orchestration framework for real-time containers," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 1, pp. 1–24, 2024.

[76] G. Monaco, G. Gala, and G. Fohler, "Shared resource orchestration extensions for kubernetes to support real-time cloud containers," in *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 97–106, IEEE, 2023.

[77] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[78] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Proceedings of 8th IEEE International Symposium on Fault-Tolerant Computing (FTCS-8)*, vol. 1, pp. 3–9, 1978.

[79] E. Cittadini, M. Marinoni, A. Biondi, G. Cicero, and G. Buttazzo, "Supporting AI-powered real-time cyber-physical systems on heterogeneous platforms via hypervisor technology," *Springer Real-Time Systems*, vol. 59, no. 4, pp. 609–635, 2023.

[80] J. B. Dugan and M. R. Lyu, "System reliability analysis of an N-version programming application," *IEEE Transactions on Reliability*, vol. 43, no. 4, pp. 513–519, 1994.

[81] L. Abeni, R. Andreoli, H. Gustafsson, R. Mini, and T. Cucinotta, "Fault tolerance in real-time cloud computing," in *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 170–175, IEEE, 2023.

[82] K. S. Trivedi and A. Bobbio, *Reliability and availability engineering: modeling, analysis, and applications*. Cambridge University Press, 2017.

[83] Q. Liu, D. Du, Y. Xia, P. Zhang, and H. Chen, "The gap between serverless research and real-world systems," in *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC)*, p. 475–485, ACM, 2023.

[84] Cisco Systems Inc., "Converged core Data UPF failover time with RCM is between 8 - 11 seconds." `https://bst.cloudapps.cisco.com/bugsearch/bug/CSCwa21578`. Accessed June 12, 2025.

[85] Kubernetes, "Kubernetes scalability and performance SLIs/SLOs." `https://github.com/kubernetes/community/blob/master/sig-scalability/slos/slos.md`. Accessed June 12, 2025.

[86] H. Jacobs, "Kubernetes Failure Stories." `https://k8s.af/`, 2023. Accessed June 12, 2025.

[87] Airbnb, "10 More Weird Ways to Blow Up Your Kubernetes." `https://www.youtube.com/watch?v=4CT0cI62YHk`, 2021. Accessed June 12, 2025.

[88] S. Visvanathan and N. Venkatachalam, "101 Ways to "Break and Recover" Kubernetes Cluster." `https://www.youtube.com/watch?v=likHm-KHGWQ`, 2018. Accessed June 12, 2025.

[89] J. Howard, "You Broke Reddit: The Pi-Day Outage." https://www.reddit.com/r/RedditEng/comments/11xx5o0/you_broke_reddit_the_piday_outage/, 2023. Accessed June 12, 2025.

[90] Venafi, "How a Simple Kubernetes Admission Webhook Lead to a Cluster Outage." https://venafi.com/blog/gke-webhook-outage/, 2019. Accessed June 12, 2025.

[91] Google, "All incidents reported for Google Kubernetes Engine." https://status.cloud.google.com/products/LCSbT57h59oR4W98NHuz/history, 2022. Accessed June 12, 2025.

[92] L. Bernaille and R. Boll, "10 Ways to Shoot Yourself in the Foot with Kubernetes," 2020. Accessed June 12, 2025.

[93] Kubernetes repository users, "Kubernetes Issue 69579." https://github.com/kubernetes/kubernetes/issues/69579, 2018. Accessed June 12, 2025.

[94] Zalando, "Let's talk about Failures with Kubernetes - Hamburg Meetup." https://www.slideshare.net/try_except_/lets-talk-about-failures-with-kubernetes-hamburg-meetup, 2019. Accessed June 12, 2025.

[95] M. Cebula and B. S. Airbnb, "10 Weird Ways to Blow Up Your Kubernetes." https://www.youtube.com/watch?v=FrQ8Lwm9_j8, 2020. Accessed June 12, 2025.

[96] D. Dolev, R. Friedman, I. Keidar, and D. Malkhi, "Failure detectors in omission failure environments," tech. rep., Cornell University, 1996.

[97] M. Straesser, A. Bauer, R. Leppich, N. Herbst, K. Chard, I. Foster, and S. Kounev, "An empirical study of container image configurations and their impact on start times," in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 94–105, IEEE, 2023.

[98] Kubernetes developers, "Kubernetes Issue 18266." https://github.com/kubernetes/kubernetes/issues/18266. Accessed June 12, 2025.

[99] Ö. Babaoğlu, K. Marzullo, and F. B. Schneider, "A formalization of priority inversion," *Springer Real-Time Systems*, vol. 5, no. 4, pp. 285–303, 1993.

[100] A. Jeffery, H. Howard, and R. Mortier, "Rearchitecting Kubernetes for the edge," in *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, pp. 7–12, 2021.

[101] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based real-time scheduling in the linux kernel," *ACM SIGBED Review*, vol. 16, no. 3, pp. 33–38, 2019.

[102] D. B. de Oliveira, D. Casini, and T. Cucinotta, "Operating system noise in the linux kernel," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 196–207, 2022.

[103] E. Truyen, D. Van Landuyt, D. Preuveneers, B. Lagaisse, and W. Joosen, "A comprehensive feature comparison study of open-source container orchestration frameworks," *MDPI Applied Sciences*, vol. 9, no. 5, p. 931, 2019.

[104] A. Daichendt, F. Wiedner, J. Andre, and G. Carle, "Applicability of hardware-supported containers in low-latency networking," 2024.

[105] A. Zuepke, A. Bastoni, W. Chen, M. Caccamo, and R. Mancuso, "Mempol: policing core memory bandwidth from outside of the cores," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 235–248, IEEE, 2023.

[106] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 155–166, IEEE, 2014.

[107] S. Iyer and P. Druschel, "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O," in *Proceedings of the 18th ACM symposium on Operating systems principles (SOSP)*, pp. 117–130, 2001.

[108] R. Andreoli, T. Cucinotta, and D. Pedreschi, "Rt-mongodb: A nosql database with differentiated performance," in *Proceedings of the 11th International Conference on Cloud Computing and Services Science-CLOSER*, pp. 77–86, Science and Technology Publications (SciTePress), 2021.

[109] I. Baldini, P. Castro, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, and P. Suter, "Cloud-native, event-based programming for mobile applications," in *Proceedings of the 2016 International Conference on Mobile Software Engineering and Systems*, p. 287–288, ACM, 2016.

[110] G. Bartolomeo, M. Yosofie, S. Bäurle, O. Haluszczynski, N. Mohan, and J. Ott, "Oakestra: A lightweight hierarchical orchestration framework for edge computing," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 215–231, 2023.

[111] E. H. Beni, E. Truyen, B. Lagaisse, W. Joosen, and J. Dieltjens, "Reducing cold starts during elastic scaling of containers in kubernetes," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pp. 60–68, 2021.

[112] J. Martins and S. Pinto, "Shedding light on static partitioning hypervisors for arm-based mixed-criticality systems," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 40–53, IEEE, 2023.

[113] D. Ottaviano, F. Ciraolo, R. Mancuso, and M. Cinque, "The omnivisor: A real-time static partitioning hypervisor extension for heterogeneous core virtualization over mpsocs," in *36th Euromicro Conference on Real-Time Systems (ECRTS)*, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

[114] I. Kadusale, G. Gala, and G. Fohler, "Wasm and containers for real-time serverless edge computing," *JRWRTC 2024*, p. 11, 2024.

[115] G. Schwäricke, R. Tabish, R. Pellizzoni, R. Mancuso, A. Bastoni, A. Zuepke, and M. Caccamo, "A real-time virtio-based framework for predictable inter-vm communication," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, pp. 27–40, IEEE, 2021.

[116] Z. Jiang, N. Audsley, P. Dong, N. Guan, X. Dai, and L. Wei, "MCS-IOV: Real-time I/O virtualization for mixed-criticality systems," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, pp. 326–338, IEEE, 2019.

[117] M. Barletta, M. Cinque, L. De Simone, and R. Della Corte, "Achieving isolation in mixed-criticality industrial edge systems with real-time containers," in *34th Euromicro Conference on Real-Time Systems (ECRTS)*, Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2022.

[118] ETSI, "5G; Management and orchestration; 5G end to end Key Performance Indicators (KPI) (3GPP TS 28.554 version 16.7.0 Release 16)," Tech. Rep. ETSI TS 128 554, ETSI, 2021.

[119] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pp. 6–11, 2010.

[120] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European conference on Computer systems (EuroSys)*, pp. 89–102, 2009.

[121] Q. Xu, Y. Gao, and J. Wei, "An Empirical Study on Kubernetes Operator Bugs," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1746–1758, 2024.

[122] L. Tang, C. Bhandari, Y. Zhang, A. Karanika, S. Ji, I. Gupta, and T. Xu, "Fail through the cracks: Cross-system interaction failures in modern cloud systems," in *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*, pp. 433–451, ACM, 2023.

[123] D. Cotroneo, L. De Simone, P. Liguori, and R. Natella, "Fault injection analytics: A novel approach to discover failure modes in cloud-computing systems,"

IEEE transactions on dependable and secure computing, vol. 19, no. 3, pp. 1476–1491, 2020.

[124] P. Vizarreta, K. Trivedi, V. Mendiratta, W. Kellerer, and C. Mas-Machuca, "DASON: Dependability Assessment Framework for Imperfect Distributed SDN Implementations," *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 652–667, 2020.

[125] X. Sun, W. Luo, J. T. Gu, A. Ganesan, R. Alagappan, M. Gasch, L. Suresh, and T. Xu, "Automatic reliability testing for cluster management controllers," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 143–159, 2022.

[126] J. T. Gu, X. Sun, W. Zhang, Y. Jiang, C. Wang, M. Vaziri, O. Legunsen, and T. Xu, "Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management," in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, pp. 96–112, ACM, 2023.

[127] J. Lu, C. Liu, L. Li, X. Feng, F. Tan, J. Yang, and L. You, "Crashtuner: detecting crash-recovery bugs in cloud systems via meta-info analysis," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 114–130, ACM, 2019.

[128] H. Liu, X. Wang, G. Li, S. Lu, F. Ye, and C. Tian, "FCatch: Automatically detecting time-of-fault bugs in cloud systems," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 419–431, 2018.

[129] H. Chen, W. Dou, D. Wang, and F. Qin, "CoFI: consistency-guided fault injection for cloud systems," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 536–547, ACM, 2020.

[130] R. Majumdar and F. Niksic, "Why is random testing effective for partition tolerance bugs?," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–24, 2017.

[131] X. Sun, L. Suresh, A. Ganesan, R. Alagappan, M. Gasch, L. Tang, and T. Xu, "Reasoning about modern datacenter infrastructures using partial histories," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, pp. 213–220, ACM, 2021.

[132] Z. Zhou, T. A. Benson, M. Canini, and B. Chandrasekaran, "Tardis: A fault-tolerant design for network control planes," in *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, pp. 108–121, ACM, 2021.

[133] D. Cotroneo, L. De Simone, P. Liguori, R. Natella, and N. Bidokhti, "How bad can a bug get? an empirical analysis of software failures in the openstack

cloud computing platform," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 200–211, 2019.

[134] A. Malik, J. Ahmed, J. Qadir, and M. U. Ilyas, "A measurement study of open source SDN layers in OpenStack under network perturbation," *Elsevier Computer Communications*, vol. 102, pp. 139–149, 2017.

[135] J. Flora, P. Gonçalves, M. Teixeira, and N. Antunes, "A study on the aging and fault tolerance of microservices in kubernetes," *IEEE Access*, vol. 10, pp. 132786–132799, 2022.

[136] P. Kumari and P. Kaur, "A survey of fault tolerance in cloud computing," *Journal of King Saud University-Computer and Information Sciences*, vol. 33, no. 10, pp. 1159–1176, 2021.

[137] G. M. Diouf, H. Elbiaze, and W. Jaafar, "On byzantine fault tolerance in multi-master kubernetes clusters," *Elsevier Future Generation Computer Systems*, vol. 109, pp. 407–419, 2020.

[138] E. Sakic, N. Deric, E. Goshi, and W. Kellerer, "P4bft: Hardware-accelerated byzantine-resilient network control plane," in *2019 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–7, IEEE, 2019.

[139] H. V. Netto, A. F. Luiz, M. Correia, L. de Oliveira Rech, and C. P. Oliveira, "Koordinator: A service approach for replicating docker containers in kubernetes," in *2018 IEEE Symposium on Computers and Communications (ISCC)*, pp. 00058–00063, IEEE, 2018.

[140] H. V. Netto, L. C. Lung, M. Correia, A. F. Luiz, and L. M. S. de Souza, "State machine replication in containers managed by Kubernetes," *Elsevier Journal of Systems Architecture*, vol. 73, pp. 53–59, 2017.

[141] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "A Kubernetes controller for managing the availability of elastic microservice based stateful applications," *Elsevier Journal of Systems and Software*, vol. 175, p. 110924, 2021.

[142] L. Ma, S. Yi, N. Carter, and Q. Li, "Efficient live migration of edge services leveraging container layered storage," *IEEE Transactions on Mobile Computing*, vol. 18, no. 9, pp. 2020–2033, 2018.

[143] P. Bellavista, S. Dahdal, L. Foschini, D. Tazzioli, M. Tortonesi, and R. Venanzi, "Kubernetes Enhanced Stateful Service Migration for ML-Driven Applications in Industry 4.0 Scenarios," in *2024 IEEE Annual Congress on Artificial Intelligence of Things (AIoT)*, pp. 25–31, IEEE, 2024.

[144] T. Kampa, A. El-Ankah, and D. Grossmann, "High availability for virtualized programmable logic controllers with hard real-time requirements on cloud infrastructures," in *2023 IEEE 21st International Conference on Industrial Informatics (INDIN)*, pp. 1–8, IEEE, 2023.

[145] H. Schmidt, Z. Rejiba, R. Eidenbenz, and K.-T. Förster, "Transparent fault tolerance for stateful applications in kubernetes with checkpoint/restore," in *2023 42nd International Symposium on Reliable Distributed Systems (SRDS)*, pp. 129–139, IEEE, 2023.

[146] R. Andreoli, H. Gustafsson, L. Abeni, R. Mini, and T. Cucinotta, "Optimal deployment of cloud-native applications with fault-tolerance and time-critical end-to-end constraints," in *Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing*, pp. 1–10, 2023.

[147] A. Ebrahimi, M. Ghobaei-Arani, and H. Saboohi, "Cold start latency mitigation mechanisms in serverless computing: taxonomy, review, and future directions," *Elsevier Journal of Systems Architecture*, p. 103115, 2024.

[148] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 467–481, ACM, 2020.

[149] X. Liu, J. Wen, Z. Chen, D. Li, J. Chen, Y. Liu, H. Wang, and X. Jin, "Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–29, 2023.

[150] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 559–572, ACM, 2021.

[151] R. B. Roy, T. Patel, and D. Tiwari, "Icebreaker: Warming serverless functions better with heterogeneity," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 753–767, ACM, 2022.

[152] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[153] D. Bermbach, A.-S. Karakaya, and S. Buchholz, "Using application knowledge to reduce cold starts in faas services," in *Proceedings of the 35th annual ACM Symposium on Applied Computing*, pp. 134–143, ACM, 2020.

[154] H. Yu, R. Basu Roy, C. Fontenot, D. Tiwari, J. Li, H. Zhang, H. Wang, and S.-J. Park, "RainbowCake: Mitigating Cold-starts in Serverless with Layer-wise Container Caching and Sharing," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 1*, pp. 335–350, ACM, 2024.

[155] X. Ju, L. Soares, K. G. Shin, K. D. Ryu, and D. Da Silva, "On fault resilience of OpenStack," in *Proceedings of the 4th annual Symposium on Cloud Computing (SoCC)*, pp. 1–16, ACM, 2013.

[156] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Computing Surveys (CSUR)*, vol. 55, no. 7, pp. 1–37, 2022.

[157] Z. Rejiba and J. Chamanara, "Custom scheduling in Kubernetes: A survey on common problems and solution approaches," *ACM Computing Surveys (CSUR)*, vol. 55, no. 7, pp. 1–37, 2022.

[158] S. Fu, R. Mittal, L. Zhang, and S. Ratnasamy, "Fast and efficient container startup at the edge via dependency scheduling," in *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.

[159] L. Toka, "Ultra-reliable and low-latency computing in the edge with Kubernetes," *Springer Journal of Grid Computing*, vol. 19, no. 3, p. 31, 2021.

[160] M. Ghose, K. P. Pandey, N. Chaudhari, and A. Sahu, "Soft Reliability Aware Scheduling of Real-time Applications on Cloud with MTTF constraints," in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 459–468, IEEE, 2023.

[161] S. Pallewatta, V. Kostakos, and R. Buyya, "Reliability-aware Proactive Placement of Microservices-based IoT Applications in Fog Computing Environments," *IEEE Transactions on Mobile Computing*, vol. 23, no. 12, pp. 11326–11341, 2024.

[162] S. Sebastio, R. Ghosh, and T. Mukherjee, "An availability analysis approach for deployment configurations of containers," *IEEE Transactions on Services Computing*, vol. 14, no. 1, pp. 16–29, 2018.

[163] M. Jammal, A. Kanso, and A. Shami, "CHASE: Component high availability-aware scheduler in cloud computing environment," in *2015 IEEE 8th International Conference on Cloud Computing*, pp. 477–484, IEEE, 2015.

[164] S. Walser, J. Ruh, and S. S. Craciunas, "Real-time container orchestration based on time-utility functions," in *2024 IEEE 20th International Conference on Factory Communication Systems (WFCS)*, pp. 1–8, IEEE, 2024.

[165] F. Lumpp, F. Fummi, H. D. Patel, and N. Bombieri, "Enabling Kubernetes Orchestration of Mixed-Criticality Software for Autonomous Mobile Robots," *IEEE Transactions on Robotics*, vol. 40, pp. 540–553, 2024.

[166] M. Szalay, P. Matray, and L. Toka, "Real-time faas: Towards a latency bounded serverless cloud," *IEEE Transactions on Cloud Computing*, vol. 11, no. 2, pp. 1636–1650, 2022.

[167] R. Eidenbenz, Y.-A. Pignolet, and A. Ryser, "Latency-aware industrial fog application orchestration with Kubernetes," in *2020 5th International Conference on Fog and Mobile Edge Computing (FMEC)*, pp. 164–171, IEEE, 2020.

[168] A. Marchese and O. Tomarchio, "Extending the kubernetes platform with network-aware scheduling capabilities," in *International Conference on Service-Oriented Computing*, pp. 465–480, Springer, 2022.

[169] A. Marchese and O. Tomarchio, "Application and infrastructure-aware orchestration in the cloud-to-edge continuum," in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, pp. 262–271, IEEE, 2023.

[170] Ł. Wojciechowski, K. Opasiak, J. Latusek, M. Wereski, V. Morales, T. Kim, and M. Hong, "Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pp. 1–9, IEEE, 2021.

[171] W. Zhang, J. Luo, L. Chen, and J. Liu, "A trajectory prediction-based and dependency-aware container migration for mobile edge computing," *IEEE Transactions on Services Computing*, vol. 16, no. 5, pp. 3168–3181, 2023.

[172] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices," in *14th USENIX symposium on operating systems design and implementation (OSDI)*, pp. 805–825, 2020.

[173] B. Cai, X. Wang, B. Wang, M. Yang, Y. Guo, and Q. Guo, "A self-stabilizing and auto-provisioning orchestration for microservices in edge-cloud continuum," *Elsevier Computer Networks*, vol. 242, p. 110279, 2024.

[174] L. Zhao, Y. Cui, Y. Yang, X. Zhou, T. Qiu, K. Li, and Y. Bao, "Component-distinguishable co-location and resource reclamation for high-throughput computing," *ACM Transactions on Computer Systems*, vol. 42, no. 1-2, pp. 1–37, 2024.

[175] H. Li, H. Liu, C. Liu, A. Chen, Z. Niu, and J. Du, "NeiLatS: Neighbor-Aware Latency-Sensitive Application Scheduling in Heterogeneous Cloud-Edge Environment," in *Proceedings of the 52nd International Conference on Parallel Processing*, pp. 615–624, ACM, 2023.

[176] R. Vaño, I. Lacalle, P. Sowiński, R. S-Julián, and C. E. Palau, "Cloud-native workload orchestration at the edge: A deployment review and future directions," *MDPI Sensors*, vol. 23, no. 4, p. 2215, 2023.

[177] M. Cinque, D. Cotroneo, L. De Simone, and S. Rosiello, "Virtualizing mixed-criticality systems: A survey on industrial trends and issues," *Elsevier Future Generation Computer System*, vol. 129, pp. 315–330, 2021.

[178] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My vm is lighter (and safer) than your container," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pp. 218–233, ACM, 2017.

[179] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight Virtualization for Serverless Applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 419–434, USENIX Association, 2020.

[180] D. Williams, R. Koller, M. Lucina, and N. Prakash, "Unikernels as Processes," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pp. 199–211, ACM, 2018.

[181] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, Ş. Teodorescu, C. Răducanu, *et al.*, "Unikraft: fast, specialized unikernels the easy way," in *Proceedings of the 16th European Conference on Computer Systems (EuroSys)*, p. 376–394, ACM, 2021.

[182] W. Zaeske, S. Friedrich, T. Schubert, and U. Durak, "WebAssembly in Avionics: Decoupling Software from Hardware," in *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*, pp. 1–10, IEEE, 2023.

[183] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems," in *Workshop on next generation real-time embedded systems (NG-RES 2020)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[184] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look mum, no vm exits!(almost)," *arXiv preprint arXiv:1705.06932*, 2017.

[185] P. Lucas, K. Chappuis, B. Boutin, J. Vetter, and D. Raho, "VOSYSmonitor, a TrustZone-based Hypervisor for ISO 26262 Mixed-critical System," in *2018 23rd Conference of Open Innovations Association*, pp. 231–238, IEEE, 2018.

[186] H. Li, X. Xu, J. Ren, and Y. Dong, "ACRN: a big little hypervisor for IoT development," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 31–44, ACM, 2019.

[187] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, "Xtratum: a hypervisor for safety critical embedded systems," in *RT Linux Workshop*, pp. 263–272, Citeseer, 2009.

[188] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned embedded architecture based on hypervisor: The XtratuM approach," in *2010 European Dependable Computing Conference (EDCC)*, pp. 67–72, IEEE, 2010.

[189] V. Struhár, M. Behnam, M. Ashjaei, and A. V. Papadopoulos, "Real-Time Containers: A Survey," in *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)* (A. Cervin and Y. Yang, eds.), vol. 80, pp. 7:1–7:9, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

[190] R. Queiroz, T. Cruz, J. Mendes, P. Sousa, and P. Simões, "Container-based Virtualization for Real-time Industrial Systems—A Systematic Review," *ACM Computing Surveys (CSUR)*, vol. 56, no. 3, pp. 1–38, 2023.

[191] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time linux kernel: A survey on preempt_rt," *ACM Computing Surveys (CSUR)*, vol. 52, no. 1, pp. 1–36, 2019.

[192] M. Cinque, R. Della Corte, A. Eliso, and A. Pecchia, "RT-CASEs: Container-Based Virtualization for Temporally Separated Mixed-Criticality Task Sets," in *31st Euromicro Conference on Real-Time Systems (ECRTS)*, Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2019.

[193] M. Eppler, J. Schenk, T. Gruner, A. Zirkler, H. Mueller, and A. Blenk, "Fabos: Hooking up container platforms with time-sensitive networks," in *Proceedings of the 1st Workshop on Enhanced Network Techniques and Technologies for the Industrial IoT to Cloud Continuum*, pp. 29–34, ACM, 2023.

[194] A. Garbugli, L. Rosa, L. Foschini, A. Corradi, and P. Bellavista, "A framework for TSN-enabled virtual environments for ultra-low latency 5G scenarios," in *2022 IEEE International Conference on Communications (ICC)*, pp. 5023–5028, IEEE, 2022.

[195] K.-H. Chen, M. Günzel, J. Boguslaw, M. Buschhoff, and J.-J. Chen, "Unikernel-based real-time virtualization under deferrable servers: Analysis and realization," in *34th Euromicro Conference on Real-Time Systems, ECRTS 2022*, pp. 6–1, Dagstuhl, 2022.

[196] M. Nasri and J. Voeten, "A Position Paper on Transforming Embedded Real-Time Systems to the Cloud: Challenges and New Research Directions," in *2024 IEEE Workshop on Design Automation for CPS and IoT*, pp. 30–32, IEEE, 2024.

Author's publications

Part of the research work contained in this dissertation is contained in the following publications in international journals:

1. M. Barletta, M. Cinque, and C. Di Martino "SLA-Driven Software Orchestration in Industry 4.0", IEEE Internet of Things Magazine, Vol. 5 (4), pp. 136-141, 2022, DOI: 10.1109/IOTM.001.2200216

2. M. Barletta, M. Cinque, L. De Simone, and R. Della Corte "Criticality-aware monitoring and orchestration for containerized industry 4.0 environments", ACM Transactions on Embedded Computing Systems, Vol. 23 (1), pp. 1-28, 2024, DOI: 10.1145/3604567

Part of the research work contained in this dissertation in contained in the following publications in international conferences' proceedings:

1. M. Barletta, M. Cinque, and R. Della Corte "Hierarchical Scheduling for Real-Time Containers in Mixed-Criticality Systems", 32nd IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) Wuhan, China, Oct. 2021, pp. 286-287, IEEE, DOI: 10.1109/ISSREW53611.2021.00082

2. M. Barletta, M. Cinque, L. De Simone, and R. Della Corte "Achieving isolation in mixed-criticality industrial edge systems with real-time containers", 34th Euromicro Conference on Real-Time Systems (ECRTS 2022), Modena, Italy, July 2022, pp. 15:1-15:23, Schloss-Dagstuhl-Leibniz Zentrum für Informatik, DOI:10.4230/LIPIcs.ECRTS.2022.15

3. M. Barletta, M. Cinque, L. De Simone, R. Della Corte, G. Farina, D. Ottaviano, "RunPHI: Enabling Mixed-criticality Containers via Partitioning Hypervisors in Industry 4.0", 2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)Charlotte, NC, USA, October 2022, pp. 134-135, IEEE, DOI: 10.1109/ISSREW55968.2022.00058.

4. M. Barletta, M. Cinque, L. De Simone, and R. Della Corte, “Introducing k4. 0s: a Model for Mixed-Criticality Container Orchestration in Industry 4.0”, 2022 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech) Falerna, Italy, September 2022, pp. 1-6, IEEE, DOI: 10.1109/DASC/PiCom/CBDCom/Cy55231.2022.9927896.

5. M. Barletta, M. Cinque, L. De Simone, R. Della Corte, G. Farina, and D. Ottaviano, “Partitioned Containers: Towards Safe Clouds for Industrial Applications”, 2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S) Porto, Portugal, June 2023, pp. 84-88, IEEE, DOI: 10.1109/DSN-S58398.2023.00029.

6. M. Barletta, L. De Simone, R. D. Corte and C. Di Martino, "Failover Timing Analysis in Orchestrating Container-based Critical Applications”, 2024 19th European Dependable Computing Conference (EDCC) Leuven, Belgium, April 2024, pp. 81-84, IEEE, DOI: 10.1109/EDCC61798.2024.00026

7. M. Barletta, M. Cinque, D. De Vita, "Orchestrating Mixed-Criticality Cloud Workloads in Reconfigurable Manufacturing Systems”, 2024 19th European Dependable Computing Conference – Fast Abstracts and Student Forum Proceedings , Leuven, Belgium, April 2024,
DOI: 10.48550/arXiv.2403.19042

8. M. Barletta, M. Cinque, C. Di Martino, Z. Kalbarczyk and R. Iyer, "Mutiny! How Does Kubernetes Fail, and What Can We Do About It?", 2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Brisbane, Australia, June 2024, pp. 1-14., IEEE,
DOI: 10.1109/DSN58291.2024.00016

Two replication packages related to conference papers were published in order to support open and replicable science.

1. M. Barletta, M. Cinque, L. De Simone, and R. Della Corte, “Achieving isolation in mixed-criticality industrial edge systems with real-time containers (Artifact)”, 34th Euromicro Conference on Real-Time Systems (ECRTS 2022), Modena, Italy, July 2022, pp. 1:1-1:12, Schloss-Dagstuhl-Leibniz Zentrum für Informatik, DOI: 10.4230/DARTS.8.1.1

2. M. Barletta, M. Cinque, C. Di Martino, Z. Kalbarczyk and R. Iyer, "Mutiny! How Does Kubernetes Fail, and What Can We Do About It? (Artifact)", 2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Brisbane, Australia, June 2024,
DOI: 10.5281/zenodo.10275036

Part of the research work contained in this dissertation may be protected by the following patent:

1. WO2024175183A1 - “SLA-Driven Orchestration of Software Containers”. Inventors: Catello Di Martino, Marco Barletta, September 2024.