Proceedings of the 33rd European Safety and Reliability Conference (ESREL 2023)
Edited by Mário P. Brito, Terje Aven, Piero Baraldi, Marko Čepin and Enrico Zio
 Published by Research Publishing, Singapore.
doi: 10.3850/978-981-18-8071-1_P217-cd

SafetyKube: Towards Orchestration at the Edge for Critical Production Systems

Yousuf Al-Obaidi, Ioannis Sorokos

Fraunhofer Institute for Experimental Software Engineering, Germany.
E-mail: {yousuf.al-obaidi, ioannis.sorokos}@iese.fraunhofer.de

Andreas Schmidt

Dependable Systems and Software, Saarland Informatics Campus, Germany.
E-mail: schmidt@depend.uni-saarland.de

Various trends, such as changeable lot-size-1 manufacturing, put production systems under pressure to become more flexible—a (r)evolution referred to as *Industry 4.0*. While this transformation is challenging for the physical assets, the same is true for the digital infrastructure that drives production. However, timely and flexible orchestration of computing, networking, and storage resources has been tackled by research and implementations in *cloud and edge computing*. What is missing are safety aspects that are essential in critical production environments. In this paper, we conduct a safety analysis of the orchestration task. We then propose new components for an established orchestration solution (Kubernetes)—allow handling of failure modes present in vanilla Kubernetes. Finally, we discuss the benefits and drawbacks of our approach and highlight future research directions to make safe orchestration a reality.

Keywords: Industry 4.0, Safety, Edge Computing, Internet of Things, Kubernetes, Containers, Orchestration

1. Introduction

Industrial manufacturing is behind consumer systems in terms of Information Technology (IT)—mainly due to manufacturing devices being designed for longer system lifetimes and the industry's scepticism about technology developed for consumer use cases. While this might have been legitimate when these technologies arose, this is no longer the case for various components (e.g. PROFINET is an industrial, Internet-inspired and -compatible network stack). Due to challenges in global supply chains and rapid shifts in customer demands, industry must develop reconfigurable production networks in contrast to static production lines—one facet of Industry 4.0 (I4.0). This development involves a paradigm shift in IT adoption: Instead of reinventing IT solutions, industry should look at existing, open-source solutions from the cloud and edge computing domain (cf. Shi and Dustdar (2016)).

However, the scepticism with respect to dependability (and in particular safety) is still legitimate when it comes to *orchestration*, a central task of edge/cloud computing. In operating systems, the ELISA Project (2022) is a first step to retrofit an open source component (Linux), which was not initially intended for safety-critical applications. The Apex.AI project (Pöhnl et al. (2022)) did a safety certification of the ROS2 middleware, which was also not developed for safety-critical use. In this paper, we follow a similar line by investigating what the open source system Kubernetes (K8S) is lacking to make it useable in safety-critical applications. e.g., I4.0. Eventually, this technical solution can contribute towards tackling the challenges in I4.0 safety assurance by Jaradat et al. (2017).

The contribution of this paper is three-fold: (1) We analyze generic edge computing workloads for hazards and risks related to the orchestration process. (2) We propose Kubernetes extensions that enable critical workloads (description files, custom controller, custom scheduler) to be managed with reduced risk. (3) We apply and discuss our approach on a worked example.

Next, we discuss background and related work in Sec. 2. A use case description (Sec. 3) is then followed by a safety analysis (Sec. 4). Our solution is described in Sec. 5 and discussed in Sec. 6. Sec. 7 concludes the paper and gives future work.

2. Background

Reconfigurable software systems regularly leverage the microservices architecture, container virtualization, and orchestration, such as Pallasch et al. (2018); Govindaraj and Artemenko (2018); Denzler et al. (2020) and KubeEdge (2022); StarlingX (2022); Azure IoT Edge (2022). Software containers are a promising technology for implementing edge computing in industrial settings—in particular due to their granular encapsulation. However, orchestrators currently lack essential properties for industrial applications, such as real-time (RT) operation and safety.

Kubernetes (K8S) is the de facto standard, open-source orchestration tool used in cloud services (cf. Datadog (2020)). In addition, K8S is used in edge computing orchestration, due to its high configurability, extensibility, and modularity. K8S consists of one or more master nodes and multiple worker nodes (often just called *master* and *nodes*)—forming a *cluster*. The cluster orchestrates *resources* or *objects* (we avoid the term resources here, to avoid confusion with physical compute resources). *Pods* are objects representing the smallest unit of workload in K8S and consist of one or more containers (e.g., Docker containers). Pods often come wrapped in other K8S objects such as *deployments*. A deployment is a continuously running workload with a specified number of replicas in the cluster. The master node hosts the control plane of K8S, which consists of several modular components. The two components of interest to us are the *scheduler* and *controllers*. The scheduler, as the name suggests, schedules pods on nodes according to available resources. The controllers deal with specific K8S objects, such as *deployments*, and manage their lifecycles. K8S objects such as pods or deployments are usually created using a so-called *manifest file*. The file describes the to-be instantiated K8S object with a declarative API and specifies the object type.

Related Work Recently, many authors aim to bring K8S to industrial automation by enabling RT analysis during the pod scheduling phase. Struhár et al. (2021) have extended the architecture of K8S by defining a new interface for deployment of containers and admission control for RT and best-effort containers. A more recent and similar work by Fiori et al. (2022) has enabled K8S to deploy RT software containers with the use of a hierarchical scheduler and Linux's *earliest deadline first scheduling* policy (`SCHED_DEADLINE`). Barletta et al. (2022) have added criticality levels and network requirements in addition to RT analysis to the orchestration process of K8S.

Monaco et al. (2022) focused on resource sharing extensions for supporting mixed-criticality applications in K8S by stronger isolation and monitoring solutions.

Fewer authors address the safety concerns of edge computing software. Desai and Punnekkat (2019) researched the safety of fog-based automation systems and identified several threats to safety arising from *fog attributes* such as the overhead of the virtualization of control loops, RT response, and resource management. The authors have proposed design aspects for a conceptual safety framework at the fog level.

In our work, we address some of their highlighted concerns and focus, on a more practical level, by implementing a solution with the K8S orchestrator. According to Etz et al. (2020), the current static approach of designing safety-critical systems impedes the flexibility promises of I4.0. They propose self-organizing safety systems and automatic generation of suitable This is useful when considering the challenge of deploying safety-related software on a pool of computing resources. However, implementing such a method is out of scope of this paper, and we limit ourselves to manual configurations.

3. System and Use Case

Since safety is an emergent property of systems, it is useful to describe our problem as a *systems* problem and look at the existing hierarchy and emergence (as recommended by Leveson (2016)). Each hierarchical layer *controls* the layer below by enforcing *constraints* on the behavior. The application of human robot collaboration is taken as a use case and our system is illustrated in Fig. 1.

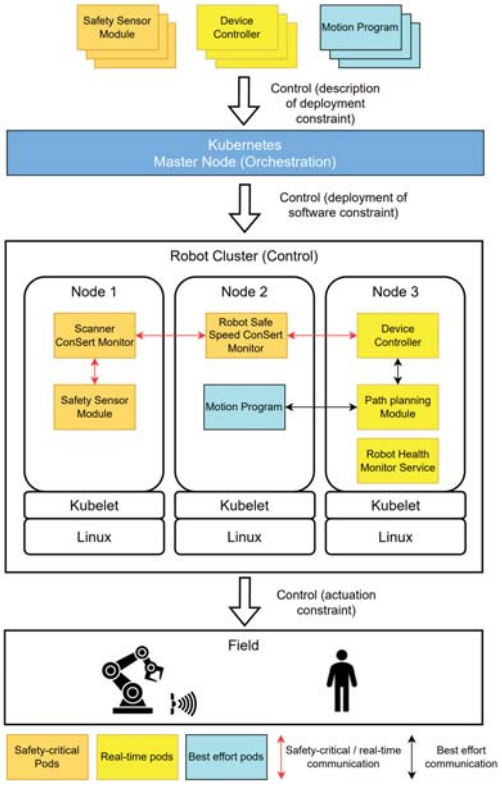

Fig. 1. System diagram showcasing the layers

The Field Layer is the lowest layer with devices and actors, such as a robot and a human worker. When we observe the integrating application (human robot collaboration for pick and place application), the safety problem emerges and we move upwards in the layer hierarchy. Typically, the emergence of safety issues is captured with a hazard analysis and risk assessment. For our use case, the hazards are collision (*H1*), pinching (*H2*), and overthrowing of objects by the robot (*H3*). To address these hazards, we have to impose constraints on upper layers.

The Robot Cluster Layer represents the software that controls the field. The control software is deployed as pods that run on the cluster's nodes. Decoupling software into pods allows for increased flexibility and configurability, as well as for update roll-outs of the robot system software. The robot system software and applications are comprised of many containerized modules deployed as pods. Several pods are responsible for monitoring and low-level control of the robot. These modules are abstracted in three pods: *Device Controller*, *Robot Health Monitor Service*, *Path Planning Module*. Applications can be written to manipulate and command the robot (*Motion Program*, *Linear Translation Motion Program*). A functional safety concept (namely step-wise speed scaling) was developed to reduce the risk of hazards and enforce limiting the robot's movement. The technical safety implementation is a sensor module (at the lower layer) and safety pods (to enforce the constraints on the robot). The sensor is responsible for detecting nearby humans and relaying a signal to the safety pods. The software in the pods implements ConSerts (Conditional Safety Certificates) monitors (cf. Schneider and Trapp (2013)), which exchange safety demands and service guarantees between the sensor and the robot. The ConSerts monitor pods (*Scanner ConSert Monitor* and *Robot Safe Speed ConSert Monitor*) are deployed to process sensor signals and send commands to slow or stop the robot based on human presence and distance respectively. With these two pods, proper constraint enforcement on the robot's movement can be achieved. For the collaborative application to be considered safe, any motion control pod must be accompanied by the *Robot Safe Speed ConSert Monitor* pod.

The Orchestrator Layer Typically, pods have safety requirements that must be fulfilled. These requirements involve the non-functional properties the system must provide, such as time criticality, failure handling and recovery, freedom from interference (FFI), etc. Fulfilling these requirements is achieved in an upper layer focusing on requirements and process management. The solutions are then built into the system by choosing the appropriate hardware and configuring it correctly, as well as employing an operating system that ensures the desired properties and constraints of the safety software's environment. Traditionally, the whole system is considered a single unit. Edge

computing follows a different approach in that software deployment is more flexible with the help of an orchestrator. The orchestrator is constraining the deployment of the pods and maintaining their desired safe state (as specified in the manifest file). This is achieved by *scheduling* the pods with the K8S scheduler on nodes that satisfy the pods' resource requirements, and by *maintaining* them with the controllers.

4. Safety Analysis

Requirements Safety is the absence of catastrophic consequences on the user(s) and the environment (cf. Avizienis et al. (2004)). K8S is a general-purpose orchestrator for cloud applications (microservice architecture) focused on scaling and reducing downtime. In industrial applications, software must be designed and implemented while respecting safety requirements, and this is where the following analysis becomes relevant. Our particular concern is the orchestrator's ability to *guarantee the correct deployment of software with sufficient resources onto the cluster nodes to facilitate safe execution*. To orchestrate safety-critical software components with K8S, several constraints and measures have to be implemented to adapt its functional model of deploying pods. The goal is to unambiguously and correctly specify the safety-related requirements of the orchestrated pods. The safety analysis focuses on identifying hazardous actions that can be taken by K8S. The safety pod requirements that K8S has to satisfy can be summarized as:

REQ1 No motion program pod shall be allowed to run without the safety pods.

REQ2 Safety pods shall always have the resources they require.

REQ3 The tasks in the safety pods shall be periodically executed and shall meet their deadlines.

REQ4 The safety pods shall be able to communicate with other pods while respecting the maximum tolerable latency.

REQ5 In any case where the safety pods fail, all other pods shall be stopped and prevented from starting until the safety pods are up and running again.

REQ6 FFI shall be satisfied for safety pods.

We assume security requirements are satisfied and we do not address malicious attacks.

Failure Mode and Effects Analysis (FMEA) We apply a functional qualitative FMEA to determine K8S behaviors that could potentially lead to the hazards (*H1-3*). Failure modes are derived from K8S's nominal function (cf. the official Kubernetes (2023) documentation) and from possible violations of safety pod requirements. The full FMEA table can be found online[a].

The analysis shows a lack of system or application-wide failure policies, in the sense that the dependency on the safety pods cannot be described, let alone their relevant failures handled (failures 1–5). In addition, the implicit or inadequate description of K8S objects leads to problems with meeting required resources after deployment (failures 6–8). K8S' lack of awareness of the context of the deployed pods results in possible unplanned or unverified coexistence of pods, which cannot be prevented (failure 10). This potentially violates the FFI assumption of a safety pod (failure 9) or leads to a false safety claim. By *context* we refer to a group of pods with a common safety goal. This means that the collection of software applications within a certain context must always be safe to be deployed together. Any pod that does not belong to that context shall not be allowed to run along the ones that do. Several failure modes can be mitigated or prevented with a common solution. To address the hazardous failure modes in the orchestration process of K8S, we propose the following extensions: a) a strict interface to create and describe K8S safety-critical objects, b) a safety-context-aware custom K8S controller, c) RT schedulability analysis capable scheduler, and d) runtime overload monitoring.

5. Implementation

The first three extensions are implemented using native customization of objects, controllers, and

[a] https://gitlab.cc-asp.fraunhofer.de/yalobaid/safety-kube/-/tree/main/FMEA

schedulers. Custom objects are created with a custom resource definition (CRD) and allow new objects to be managed by the K8S API. A custom controller is implemented via K8S client libraries to manage objects. Custom schedulers are created using the modular scheduling plugins framework. The fourth extension is a reactive overload monitor with minimal overhead—implemented using Linux kernel tracing. We open source the code [b].

Critical Deployment Custom Object (CDCO) defines an interface (an abstraction) for creating and describing critical objects in the cluster. A CRD object is used to create the new CDCO. After that, the CDCO is available for Create/Retrieve/Update/Delete operations in the K8S API. The CDCO is defined by the following properties in the manifest: safety context, target node, RT period, prerequisites, interfaces, max. memory, max. network delay, worst-case execution time (WCET), service object, and failure policy.

The safety context and prerequisites both ensure a safe deployment. The former is responsible for setting the object's safe context and protecting its FFI assumption. The latter enables the implementation of a safety dependency on the orchestration level between the critical deployments. For example, a critical deployment with `context1` as safety context and `safetymonitor` as a prerequisite, means that the related object can only be deployed when the cluster is running the `context1` safety context, and the `safetymonitor` critical deployment is also running. The manifest file also contains RT properties necessary to determine whether RT tasks will meet their deadlines (i.e. period, WCET, network delay). The current implementation assumes that each pod has one container with one RT task (in future work, we plan to lift this to more than one RT task or container). A failure policy is an application-wide policy to be applied when a failure occurs (e.g. shutting down all the pods when a certain pod fails).

Critical Deployment Controller manages the new custom object. This controller extends the K8S control plane, responsible for both the pods' and the overall application's safety. During the critical deployment object's lifecycle, the controller continuously monitors it and takes actions as appropriate. The controller checks the safety contexts and prerequisites, creates pods to meet the specified resource requirements, and handles failures. The currently implemented failure policies are the following: a) *ContextShutdown*: terminates all critical deployments with the same context. Requires manual restart. b) *AffectedShutdown*: terminates the failed deployment and its dependencies only. c) *ContextRestart*: restarts all critical deployments. d) *AffectedRestart*: restarts the failed deployment and its dependencies only.

Real-time Scheduler is developed based on the plugins framework of the K8S scheduler. The schedulability analysis uses *partitioned fixed priority rate monotonic scheduling* (cf. Liu and Layland (1973)). For each node, we isolated cores and labelled them accordingly. Without the full view of the RT task set at each scheduling cycle, the scheduler assigns the pod's task a *halfway* Linux RT priority based on its period (from 1 (lowest) to 99 (highest)). For example, the first task to be scheduled on a core receives priority 50, the second task would either receive 25 (if its period is larger) or 74 (if it is shorter), and so on. If more than one task shares the same period, the tie break would be based on Linux's FIFO scheduler, i.e., the task that acquired the CPU first retains it.

The schedulability analysis works as follows: First, we apply the necessary utilization check ($Utilization \leq 1$ for a single core). If the check passes, we assign priority according to rate-monotonic scheduling and check the sufficient condition of Liu and Layland (1973). If it holds, we deem the pod schedulable on that node's core. Otherwise, we follow the classical response time analysis Joseph and Pandya (1986), to see whether the schedule is still feasible with the new task. If not, we proceed to the next core. The schedulability test is applied at the *filter plugin* stage. If the node fails the test on all of its cores, it is eliminated from the K8S scheduling

[b] https://gitlab.cc-asp.fraunhofer.de/yalobaid/safety-kube/

process. The winning node would pin the pod's process on the specified core and runs it with the assigned priority that makes the schedule feasible. Eventually, RT communication is considered, as message exchanges between nodes must also comply with safety-critical thresholds. As our solution is independent of specific RT networking approaches (e.g. Time-Sensitive Networking (TSN)), we do not yet consider this and an integration is future work.

Overload Monitor uses kernel tracing solutions to check that tasks are not missing deadlines during runtime. The process uses tracepoints in the Linux scheduler for task entry and exit times. Based on this information, the monitor updates the critical deployment objects when necessary, for the controller to take appropriate actions.

Intended Workflow The K8S manifest files can be exploited as the safety contracts of pod specifications. These manifest files are examined and used in the safety argument for the execution infrastructure and the execution context. The manifest files contain a complete specification of the non-functional properties and capabilities that the safety-critical software requires to behave as intended and keep the overall system within its safe states. The intended workflow of a safe orchestration is: a) a distributed containerized application is developed, b) the container's resource requirements and safety context are specified and tested on target hardware platforms, c) critical deployment manifest files are created to run the pods and their containers safely, and d) the critical deployment objects are deployed in the cluster.
In the current solution, the deployed pods are continuously running programs. The containers of the pods do not terminate to indicate the completion of an operation. Safety-related pods are not called upon to be deployed by a process or a situation. Still, they are deployed together with the rest of the distributed application, and they are intended prerequisites for deploying and running application pods. The critical deployment controller ensures that only when prerequisites of a pod are running, the pod can be scheduled. In case of hazardous failures, the failure-handling policies are executed by the critical deployment controller to restore the distributed applications to a safe state.

6. Discussion

Application of Workflow to Use Case The use case that was discussed previously in Sec. 3 is used to demonstrate our safety extensions to K8S. At design time, the safety-critical software is developed, to implement safety and control actions according to the safety requirements. In our demonstrator, this represents creating the containers showcased in Fig. 1. Here, the safety context is determined as `CollaborativePickAndPlace`. Any software that does not implement this context shall be prevented from running in the same cluster as the ones that do. In addition, since the stepwise speed scaling and safety scanner functionality are decoupled from the motion and control program, it has to be made a specific prerequisite. This way, the motion program pod is stopped or prevented from being deployed when the ConSert monitors are not running. At configuration time, the critical deployment manifest files are created. A requirement for this is determining the target nodes, required peripheral devices, and the WCET on the nodes. The software must be deployed to the correct node that meets the required resources and allows execution without violating the specified WCET. The critical deployment manifest must specify a list of cluster nodes where the pod's execution is permitted. The *labeling* mechanism of K8S is then used in node assignment. With this workflow, introducing new software components requires returning to the design stage to reanalyze the safety implications of the new software on the given safety context.

Test Environment The implementation is realized and tested on a cluster with one master and three Revolution Pi Core as nodes. Each node runs a Linux kernel with the `preempt-rt` patch and has two isolated cores. The control plane runs on the master node, which is hosted on an Ubuntu 20.04 virtual machine. The extensions are

deployed in the master node and start watching for critical deployment objects. All members of the cluster connect via wired Ethernet. The critical deployment objects are created in the K8S cluster via the `kubectl` command line interface. The custom controller detects their creation and creates the corresponding pods. Upon creation, the RT scheduler analyzes the candidate nodes with respect to both schedulability and available resources—and assigns the pod to a node.

Failure Handling Performance Fast failure handling response is sometimes necessary to quickly execute failure policies and return to a safe state. Our proposed solution does not account for timing constraints with respect to failure handling. However, we have made a preliminary investigation into the matter. In native K8S, status monitoring (node health, pods status) can be configured to high frequencies, but this is doubtful, as official documentation only mentions failure detection in the range of seconds. In our experiments, the minimum node health check interval is 400 milliseconds—anything less would cause instability. This is, however, achieved with a control plane hosted on a virtual machine run by a machine with modest capabilities. If the safety requirements for failure-handling response times are stringent, then K8S-based orchestration is, in any case, not an ideal choice. Redundancy can be used to reduce the probability of complete service failure. There is native support for redundancy in K8S by means of replica sets. Replicas can be configured to spread out across the nodes, which lowers the probability of complete failure of a safety application. Nevertheless, software redundancy with orchestration is only possible on systems that are not dependent on existing wiring and hardware and, in some cases, (standby redundancy) require redundancy protocols as described by Johansson et al. (2022).

Standard Compliance A major challenge of orchestration-based edge computing in industrial automation is safety certification. All the solutions in this work leverage Linux features. Although its kernel is often used in industry, to this day, there is no certified Linux for safety applications. The ELISA Project (2022) develops a framework to certify a Linux configuration. An essential aspect of software safety is achieving FFI (cf. IEC 61508-3). As scheduling pods and deploying them on specific nodes is done at runtime and without a predetermined schedule, this becomes particularly relevant. The containerization process in Linux provides low-overhead isolation between processes (termed 'weak isolation' when compared to hypervisors and virtual machines, because processes share the kernel). Spatial memory isolation is done inherently in Linux by means of virtual memory spaces and limiting a pod's maximum amount of memory. The RT behavior of the Linux kernel is crucial to realize temporal isolation, as it assures deterministic scheduling of critical tasks. Our evaluation uses the `preempt-rt` patch to achieve low-latency responses. However, the `preempt-rt` patch relies only on testing statistics for assuring the RT behavior, i.e., testing latency with various loads and priorities for a long duration. Furthermore, the RT processes were isolated and pinned on separate isolated cores (spatial isolation). In addition, containers in Linux hide the filesystem, restrict the view of the system's other running process, and constrain interprocess communication as well as access to peripheral devices.

7. Conclusion

We present SafetyKube, a set of Kubernetes extensions for safer orchestration of critical production systems. We review concrete safety-critical applications and use safety analysis to derive requirements for our proposed solution—including extended application description files with information such as worst-case execution time or real-time execution period. A critical deployment controller leverages this information to place parts of the distributed application on nodes that enable a safe collaboration. An RT scheduler assesses and enforces that the parts of the application have enough time to execute their function safely. Lastly, we specify a monitoring component capable of measuring and reporting incidents of system overload, where safety is under risk.

Together, these parts allow safeguarding against certain faults occurring from orchestration.

In future work, we look further into the question of *what is needed to bring K8S to critical environments*. While we extend K8S in this work, it remains open whether changing or rewriting its core services is a more suitable and sustainable approach. Finally, we intend to investigate how to automate the generation of safety configurations.

Acknowledgement

The German Federal Ministry for Economic Affairs and Climate Action (BMWK) supported this work within the research project "FabOS" under grant 01MK20010A. The European Union Horizon 2020 programme supported this work within the "Secure and Safe Multi-Robot Systems (SESAME)" grant agreement 101017258. This work was funded by the German Research Foundation (DFG) grant 389792660 as part of TRR 248 – CPEC (see `https://perspicuous-computing.science`).

References

Avizienis, A., J.-C. Laprie, B. Randell, and C. Landwehr (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing 1*(1), 11–33.

Azure IoT Edge (2022). Microsoft. `https://azure.microsoft.com/en-us/products/iot-edge/`. Accessed: 2022-10-12.

Barletta, M., M. Cinque, L. De Simone, and R. Della Corte (2022). Introducing k4. 0s: a model for mixed-criticality container orchestration in industry 4.0. *arXiv preprint arXiv:2205.14188*.

Datadog (2020). Container report. `https://www.datadoghq.com/container-report-2020/`. Accessed: 2022-10-12.

Denzler, P., J. Ruh, M. Kadar, C. Avasalcai, and W. Kastner (2020). Towards consolidating industrial use cases on a common fog computing platform. In *International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 172–179. IEEE.

Desai, N. and S. Punnekkat (2019). Safety of fog-based industrial automation systems. In *Proceedings of the Workshop on Fog Computing and the IoT*, pp. 6–10.

ELISA Project (2022). Elisa: Enabling linux in safety applications. `https://elisa.tech`. Accessed: 2022-10-12.

Etz, D., T. Frühwirth, and W. Kastner (2020). Flexible safety systems for smart manufacturing. In *International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1123–1126. IEEE.

Fiori, S., L. Abeni, and T. Cucinotta (2022). Rt-kubernetes–containerized real-time cloud computing.

Govindaraj, K. and A. Artemenko (2018). Container live migration for latency critical industrial applications on edge computing. In *International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 83–90. IEEE.

Jaradat, O., I. Sljivo, I. Habli, and R. Hawkins (2017). Challenges of safety assurance for industry 4.0. In *2017 13th European Dependable Computing Conference (EDCC)*, pp. 103–106. IEEE.

Johansson, B., M. Rågberger, T. Nolte, and A. V. Papadopoulos (2022). Kubernetes orchestration of high availability distributed control systems. In *Proc. ICIT*.

Joseph, M. and P. Pandya (1986). Finding response times in a real-time system. *The Computer Journal 29*(5), 390–395.

KubeEdge (2022). Why kubeedge. `https://kubeedge.io/en/docs/kubeedge/`. Accessed: 2022-10-12.

Kubernetes (2023). Kubernetes. `https://kubernetes.io/docs`. Accessed: 2023-02-28.

Leveson, N. G. (2016). *Engineering a safer world: Systems thinking applied to safety*. The MIT Press.

Liu, C. L. and J. W. Layland (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *JACM 20*(1), 46–61.

Monaco, G., G. Gala, and G. Fohler (2022, 01). Extensions for shared resource orchestration in kubernetes to support rt-cloud containers.

Pallasch, C., S. Wein, N. Hoffmann, M. Obdenbusch, T. Buchner, J. Waltl, and C. Brecher (2018). Edge powered industrial control: concept for combining cloud and automation technologies. In *Int. Conf. on Edge Computing (EDGE)*, pp. 130–134. IEEE.

Pöhnl, M., A. Tamisier, and T. Blass (2022). A middleware journey from microcontrollers to microprocessors. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 282–286.

Schneider, D. and M. Trapp (2013). Conditional safety certification of open adaptive systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS) 8*(2), 1–20.

Shi, W. and S. Dustdar (2016). The promise of edge computing. *Computer 49*(5), 78–81.

StarlingX (2022). Documentation. `https://docs.starlingx.io/`. Accessed: 2022-10-12.

Struhár, V., S. S. Craciunas, M. Ashjaei, M. Behnam, and A. V. Papadopoulos (2021). React: Enabling real-time container orchestration. In *International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8. IEEE.