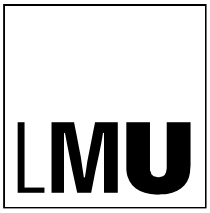


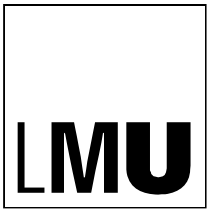
Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Implementation of a Reference Architecture for IIoT Platforms in Hybrid Cloud Environments

Luis Schweigard

Masterarbeit im Elitestudiengang Software Engineering

Software Engineering
Elite Graduate Program




Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Implementation of a Reference Architecture for IIoT Platforms in Hybrid Cloud Environments

Matrikelnummer:	2077806
Beginn der Arbeit:	Juli 2023
Abgabe der Arbeit:	Januar 2024
Erstgutachter:	Prof. Dr. Bernhard Bauer
Zweitgutachter:	Prof. Dr. Alexander Knapp
Betreuer:	Lars Gielsok, MaibornWolff GmbH

Software Engineering
Elite Graduate Program

EIDESSTATTLICHE ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Augsburg, den 12. April 2024	
Ort, Datum	Unterschrift

Abstract

In the age of digitalization, more and more devices are being connected to Internet of Things (IoT) platforms. It is estimated that the number of edge devices will grow immensely in the next years and that the market around edge computing will even overtake the market in the cloud. In industrial IoT (IIoT) scenarios, one of the main challenges is handling the enormous loads of data due to the high frequency of devices like sensors. While generating data on devices is simple, building large-scale IIoT platforms capable of handling such data is becoming increasingly difficult. Additionally, achieving full connectivity is a hard task due to the heterogeneity of devices and protocols. Another major challenge in building IIoT platforms is the high amount of environments spanning from edge devices across on-premise systems up to the cloud. Hybrid cloud platforms are necessary for several reasons in these scenarios. For example, edge devices may not be allowed to connect to the cloud due to security concerns, thus the need for an on-premises system. Additionally, latency-critical workloads cannot tolerate delays caused by networking into the cloud.

This work will deal with strategies, techniques, and architectures to solve these challenges. The current situation around industrial IoT will be evaluated and the major problems will be identified. Domains like edge computing, automation and orchestration, common IIoT patterns and architectures, and bare-metal machine management will be explored. To successfully implement industrial IoT systems that are capable of fulfilling all of these requirements, building upon reference architectures is essential. This thesis will discuss the shortcomings of many existing reference architectures, and then introduce a new reference architecture that is based on the concept "unified namespace", that aims to act as a basis for building modern industrial IoT systems at scale. All concepts necessary to understand and implement the architecture will be explored.

After the theoretical part, a strategy for implementing the newly introduced reference architecture for IIoT platforms in hybrid cloud scenarios will be introduced and then implemented based on the theory discussed in the previous chapters, followed by an evaluation of the system built as a proof-of-concept project. A wide range of technologies and techniques that help build the complete stack of this IIoT platform will be introduced and contextualized. Alternatives will be evaluated, given that the IIoT use case comes with many domain-specific requirements like security demands that must be supported. Finally, a high level of automation is essential for success, which will be the main focus of this work.

Contents

List of Figures

1 Introduction

The Internet of Things (IoT) is a transformative force within the landscape of technology, especially in the industrial sector. A key component of this technological journey is industrial IoT (IIoT), which brings digitalization into industrial manufacturing. Due to the rise of IIoT, we can see a rapid increase both in the number of connected devices and the data being created by them.

1.1 Motivation

It is apparent that building scalable IIoT systems that are capable of handling the sheer velocity and volume of data is getting more and more difficult. Being able to connect devices from different manufacturers using a wide and heterogeneous variety of protocols poses another demanding challenge. Due to modern requirements like low latency workloads or large-scale stream analytics, we also see that we have to deal with a large number of different environments from edge devices in a factory to on-premises systems and up to cloud systems which results in highly distributed systems and brings even more complexity into an IIoT platform.

This thesis aims to address these challenges, focusing on the creation and implementation of effective solutions for IIoT systems, especially in the context of hybrid cloud scenarios. Due to the significant expansion of IIoT across the world, it becomes imperative to provide solutions for these hurdles. Since we see a lack of standards, architectures and guidelines on the market that confront these emerging issues, we want to discuss a new reference architecture that is capable of serving as a basis for the implementation of modern IIoT systems. Even though the reference architecture dealt with in this work is already present, the absence of a strategy for implementing this architecture in real-world industrial IoT scenarios remains a significant gap. Thus, this thesis also intends to close this gap by offering a concrete yet adaptable strategy on the path from the high-level reference architecture to a fully functional IIoT system.

1.2 Goals

The primary goal of this work is to provide a comprehensive framework consisting of a reference architecture and a strategy for implementing it in the real world, tailored to tackle the challenges in the development of industrial IoT systems. For this, we aim to analyze the landscape around IIoT and identify the main challenges such as edge computing, automation,

orchestration, the lack of standards, architectures and guidelines, and the management of a large scale hybrid cloud setup, while providing guidance on how to solve them. By implementing a proof of concept for the infrastructure of the suggested architecture, we want to demonstrate the practical applicability of this strategy in a real project while introducing and contextualizing all relevant tools and concepts that are required. Furthermore, another goal of this thesis is to conduct a thorough analysis of the existing common strategies and architectures in the IIoT environment and to understand why they are no longer sufficient for current use cases.

1.3 Constraints

While this thesis addresses several perspectives of IIoT platforms, we also need to delineate the boundaries of this work. To begin with, this work does not deal with a large variety of IoT in general but has a predominant focus on industrial IoT in hybrid cloud scenarios. Due to the emphasis on the infrastructure of IIoT systems, the actual implementation of domain services running on an IIoT platform is also beyond the scope of this thesis, and thus is a great opportunity for further research. Lastly, while both the reference architecture and the proof of concept implementation have security features in mind, this work does not perform a comprehensive analysis regarding a detailed security strategy for an IIoT platform.

1.4 Structure

The work begins with the theoretical background in Chapter 2, exploring the current landscape of industrial IoT. This includes discussions on GitOps, orchestration, and finally common patterns and architectures in the domain of IIoT. Chapter 3 shifts the focus to a proposal for a modern reference architecture, detailing its requirements and components. Here topics such as the unified namespace, the application of GitOps in IIoT, and system management strategies are introduced. In Chapter 4 the topic of provisioning the infrastructure required for a large-scale IIoT system following the newly introduced reference architecture is addressed. The discussion revolves around topics like bare-metal infrastructure, Kubernetes, ClusterAPI, and virtualization. All of this can be seen in Chapter 5, where the proposed architecture was implemented in the form of a proof of concept project, displaying all of the components and techniques from the previous chapters. Finally, Chapter 6 concludes this thesis by summarizing the findings and potential areas for future work, along with acknowledgments to those who supported this work.

2 Theoretical Background

2.1 Current Situation

To understand the rationale of this thesis one must first look at the current situation of industrial IoT (IIoT). With movements like Industry 4.0 and Smart Manufacturing, the pace of digitalization in industrial production is higher than ever. While typical IoT systems are often uncritical systems like smart home devices, the application in the manufacturing sector imposes the inevitable need to solve new challenges. Dealing with the combination of the high frequency and volume of data recorded by manufacturing devices and the fact that systems often require real-time functionality with low latency communication means careful system engineering beginning with the setup of hardware and networking. Due to devices being very constrained regarding their computational resources highly distributed systems have to be built, which brings a high level of complexity into the system. Also since machines in manufacturing are built by many different vendors or just run old software, a very heterogeneous set of communication protocols makes building a uniform and central platform a difficult task. Due to the fact that the actual manufacturing process often depends on the IIoT system today, requirements like security and availability are also becoming increasingly important. Since malfunctions, vulnerabilities or downtimes of the system could lead to a halt in production, protecting systems from attacks and implementing state-of-the-art high availability is essential to fulfill the requirements of modern IIoT systems. Because of the strongly increasing amount of smart devices in production and the growing scale of IIoT systems which often include multiple environments like a cloud environment, one on-premises environment per production site and many edge environments per site, a high degree of automation is crucial for success. Maintaining the systems manually is often simply not possible due to the sheer scale of the systems. Onboarding and provisioning of infrastructure as well as software and hardware maintenance are just two examples of procedures that are infeasible to perform manually at this scale. Note that these are just a few of the many requirements that lay out the challenges of designing, building and operating IIoT systems in the modern world [10].

In the current state, we can see a shortage of standards and decision bases on the market, as will be further discussed in this work. While reference architectures that claim to tackle the typical challenges exist, many are not open source or are too generic to actually be helpful in real-world implementations. Others are often so tied to a particular vendor that they are unfeasible in many scenarios due to project-specific circumstances like existing hardware or cloud provider choice. There also exists a variety of architectures that work fine for the

production but fall short in other essential requirements like scalability or availability. Many architectures also struggle with the conjoining of the worlds of operation technology (OT) and information technology (IT) due to the very different viewpoints and requirements both worlds impose on an IIoT system. While the OT-oriented architectures are mainly focused on keeping the manufacturing process running, they often stifle innovation in the IT world e.g. by building point-to-point integrations between systems rather than having all data and communication centralized for everyone to work with easily at low integration cost. Lastly, a variety of reference architectures, especially those of modern cloud providers, fail to satisfy requirements like edge/on-premises computing (Section 2.2) or hybrid cloud (Section 3.2) and thus quickly become unsatisfactory for many projects.

For these reasons, the company "MaibornWolff GmbH" created a reference architecture for hybrid cloud IIoT systems (Chapter 3), that aims to satisfy the requirements and solve the challenges mentioned above. In this work, this reference architecture and all other practices and components necessary for it will be explained. Also, a strategy for realizing the architecture in the real world, in particular for setting up the required infrastructure, will be developed in this thesis and implemented in a proof of concept project, since the reference architecture only provides a target state without guidance on the actual implementation. Note that this work is conducted together with "MaibornWolff GmbH".

In this thesis, the plain term "reference architecture" will from now on always refer to the reference architecture created by "MaibornWolff GmbH" if nothing further is specified.

2.2 Edge Computing

In this chapter, we will introduce the concept of edge computing. We will explain the general term and put edge computing into context with current trends and movements in the industry. Then we will look at reasons why edge computing is becoming increasingly popular in industrial IoT environments.

2.2.1 Terminology and Definition

The term "edge computing" is very broad and not precisely defined in the literature. First, the term "edge" needs to be clarified. While some describe it as the last hop before smart- or end devices or even include end devices like IIoT devices themselves, the Industrial Internet Consortium (IIC) defines the edge as the boundary between the pertinent digital and physical entities, delineated by IoT devices. From a high-level perspective, edge computing can now be described as the opposite of the centralized data processing pattern, since as much as possible processing gets executed directly on the edge where data is created or events of interest occur, whereas in a centralized approach, most computation happens in a remote data center [11]. Another commonly used definition can be found in the "Industrial Internet Consortium (IIC) vocabulary", which defines "edge computing" as a form of distributed computing in which processing takes place on a set of networked machines that are near the

edge. The exact borders of edge computing are hard to identify and have to be defined by the requirements of the system in question [12, 13].

An edge computing system typically consists of components like edge devices, edge servers, edge gateways and edge connectors. Edge devices act as the connection between the physical things in the field, and are mostly sensors or actuators or servers with direct connections to these. Because these are as close as it gets to the physical world, systems with strict low latency requirements will typically run on them directly. Since edge devices in the real world are often not more powerful than a microcontroller, edge servers are responsible for more computationally expensive efforts. They are responsible for gathering data from edge devices and processing it in some way. Here, workloads that have requirements regarding low latency which are less strict than the ones of applications running directly on edge devices are located. Edge gateways are placed between the edge devices and the applications that use the generated data and are responsible for the delivery of data between these edge devices and applications. This can be used to overcome network borders or protocol limitations. In many cases, an edge server can act as the edge gateway, if the according network setup etc. allows it. Lastly, edge connectors, which are typically just software, deal with the heterogeneous set of protocols that typically exists in edge computing. They might for example forward data from a radio protocol to a different network via TCP/IP which is essential in scenarios like device-to-cloud communication and can be incorporated in any of the other edge components [14]. All of the components are generally located on-premises and hence require maintenance and operational efforts for the hardware. The set of all of the mentioned components can be seen as the "edge".

2.2.2 Application of Edge Computing in IIoT

Within the realm of IIoT, edge computing is a rapidly emerging trend. An article by Atos predicts that by 2023 more than half of new enterprise IT infrastructure will be deployed at the edge rather than centralized data centers compared to around 10% in 2021. According to Gartner, around 75% of enterprise-generated data will be created and processed at the edge by 2025 [14]. While the actual values differ in most predictions, the trend of growth in the edge computing context is a common understanding. The application of edge computing in IIoT has a set of reasons, the most obvious being the reduction of communication latency within the system. By shortening the distance between the creation of data or events of interest and the location where the actual computation happens, response times and latencies can be significantly decreased. For production critical services like visual inspection where high latencies may result in financial damage or could even endanger humans this is essential. By having full operational control over devices and networks that are participating in communication, strong security can be achieved which is especially necessary in the context of sensitive data. Another benefit of incorporating edge computing in IIoT is the more efficient usage of available resources. Since data is preprocessed (e.g. aggregated or filtered), less data has to travel through the network thus using less bandwidth, less energy and also producing less cloud cost. Not only is this more energy efficient and can even be

better in terms of sustainability, but can also be inevitable in large systems. IBM estimates that by 2025 each person will have at least one data interaction every 18 seconds. Many of these interactions can be traced back to the billions of IoT devices, which are expected to create over 90 zettabytes of data in the year of 2025. With that volume of data, a centralized approach is presumed to result in bandwidth, energy and latency issues making it unsustainable and even infeasible. With edge computing, the challenge of processing these massive amounts of data can be tackled. Also, requirements of an IIoT system like low cost, data sensitivity, network fault tolerance and resilience can be met, which is often impossible with the typical cloud computing paradigm [13, 15].

2.2.3 Infrastructure for Edge Computing

While edge computing opens many doors for IIoT systems, it also introduces complexities and increases the burden of operational maintenance. While managing hardware is not a new task in the IT environment, there are not yet enough techniques for management at this massive scale. The high amount of devices, heterogeneity of protocols, technologies and even environments (edge, cloud, etc.) make this even more difficult. Additionally, a thorough understanding and the capability of managing security and networking is essential, particularly using concepts such as zero trust and the principle of least privilege. Because of the growing interest in edge computing the industry's need for this is closer than many expect. Infrastructure, DevOps and platform practices like scheduling and orchestration, observability, etc. are essential and must be built as soon as possible. The automation of provisioning, scaling and maintaining such systems while keeping operational costs under control is a vital element for success in modern IIoT applications [13, 16]. In Chapter 4 and Chapter 5 suggestions for dealing with these challenges will be made by addressing them in a real-world implementation.

2.3 GitOps

GitOps is one of the main trends in the DevOps ecosystem and integrates into the Continuous Integration/Continuous Delivery (CI/CD) process. It can be described as an operational framework that applies DevOps best practices and proves to be optimal for automation in distributed environments and hence fits like a glove for the IIoT use case due to the highly distributed nature of such systems. In the context of GitOps, the following key concepts should be noted:

- Declarative description of the desired state
- Immutable and versioned
- Automatically pulled
- Continuously reconciled

First of all, the desired state of all deployment targets is expressed through declarative configuration files. By avoiding the use of imperative code for CI/CD, the process becomes more reproducible and less error-prone due to common issues like race conditions or faulty dependencies. As the name suggests, the configuration is stored in Git, which means, that the system is strictly versioned and each version is immutable, just through the usage of Git. This allows the use of known and battle-tested workflows like using Git pull requests, mandatory code reviews, branch rules and allowed merge time windows. Also, the system automatically comes with an immutable audit trail, which ensures a comprehensive record of changes, enhancing both accountability and traceability. Contrary to the typical CI/CD process of using pipelines that push to delivery targets, the desired state in Git is automatically pulled from the so-called GitOps controllers running on the delivery targets themselves. This not only means, that pipelines become obsolete for the delivery process, it also is beneficial for security. While pipelines need access to delivery systems which typically means opening ports in firewalls and setting up pipeline agents in the same network as their delivery targets, GitOps delivery targets only need to have access to the Git repository storing their respective desired state because of the shift from the push to the pull model. Since CI/CD agents are a very attractive target for attackers due to their often high level of permissions and access, this is a great benefit. Also, the computational power of the delivery targets is used, so that less strain is placed on the central CI/CD infrastructure, resulting in more efficient resource utilization and potentially lower operational costs. Lastly, the GitOps controllers continuously reconcile all deployments. The divergence between the desired state described in Git and the actual state in the system is consistently monitored. If such a drift is detected, the GitOps controllers apply the necessary changes to realign the system with the specified state in Git [17]. Here, the concept of eventual consistency comes in handy. Dependencies that would impose difficult challenges in building CI/CD pipelines can often be resolved by the automatic reconciliation of GitOps controllers. By retrying until success, many dependencies, which would have caused issues in typical pipelines, are resolved automatically.

It is to be noted that GitOps is not a replacement but rather an addition to CI/CD pipelines. GitOps mainly deals with the delivery of software through configuration, whereas pipelines take over the continuous integration process, which yields artifacts required for the GitOps process. This includes tasks like building or compiling, running tests, building container images and changing configuration in Git. These artifacts are then applied by GitOps controllers based on the configuration and the artifacts provided by the pipelines earlier. Since this thesis focuses on the infrastructure of IIoT systems, the delivery aspect will be the main focus in this work though.

Because GitOps is only based on Git and a GitOps controller, the infrastructure to which GitOps can be applied is varied, including cloud environments, bare-metal servers and more. Compared to traditional CI/CD pipelining, this allows for a uniform delivery process across all environments. This is particularly useful in IIoT since workloads are expected to run on a variety of infrastructure, ranging from edge devices (Section 2.2) up to cloud or even

multi-cloud solutions, where standard pipelines begin to struggle. As mentioned in Section 2.1, one of the key factors in modern IIoT systems is automation. Many claim that the GitOps model is the key framework to achieve the necessary level of automation on such scale. Being able to roll out software to a large variety of delivery targets at once and keeping all targets in sync with the desired state from the Git repository as the single source of truth, while keeping operational costs (DevOps teams, CI/CD agents, deployment schedule, etc.) low is crucial. Scaling to such an extent is not only tedious but in many scenarios infeasible with traditional CI/CD pipelining [18]. Improving continuous delivery is not the only benefit of GitOps though. By using Git as the single source of truth for all desired state, development productivity can be increased by a lot as well. With proper automation in place, platform or DevOps teams no longer become the bottleneck in large-scale projects, since every developer with access to Git is now able to manage the desired state without needing assistance. The developer experience can be improved even further by providing templates or reusable components for common tasks [19].

However, with GitOps some new challenges are introduced as well. One of the most demanding ones is the chicken-egg problem introduced by having to run a GitOps controller on the infrastructure, that should be provisioned through GitOps. In practice, this is typically solved by setting up a temporary GitOps controller on an existing system or by bootstrapping the system manually once. Another limitation of this DevOps framework is that while it is one of the main trends in the DevOps community, it is mainly focused on the orchestration tool "Kubernetes" (see Section 2.4). While common GitOps technologies like "FluxCD" and "ArgoCD" fit perfectly for Kubernetes-based projects, systems using other orchestration technologies make the application of GitOps hardly possible. Lastly, provisioning infrastructure is in itself another challenge. For this, supporting tools have to be used. Infrastructure-as-Code (IaC) tools like "ClusterAPI", "Crossplane", "OpenTofu" or "Pulumi" enable developers to describe infrastructure in a declarative manner and manage the actual infrastructure under the hood. With such tools, not only software but also infrastructure can be managed through GitOps thus once again moving one step closer to the goal of storing "everything" as code [20]. How GitOps can be used to manage the infrastructure of a large-scale IIoT platform will be explored in Chapter 4.

2.4 Containerization and Orchestration

In the era of cloud-native development, containerized workloads are the standard approach for running software – be it on bare metal, VMs, orchestration tools like K8s or even in the cloud.

2.4.1 Containerization

By containerization, the process of bundling applications with all the components they require in order to run on any kind of underlying infrastructure is described. This is once again

a good match for IIoT systems since running workloads on a broad variety of infrastructure is a common practice there. The most common tool for this is Docker, which can be seen as an industry standard and is often even used as a synonym for containerization in general.

For running containers in production, custom operating systems (OS) that are optimized for hosting containerized applications are used. Since containers ship all required tools and libraries necessary to run their respective applications, everything that does not contribute to hosting containers can be seen as overhead, so minimal operating systems are often preferred. Not only does such an OS have a lower resource footprint and is hence better suited for efficient usage of available resources, it also has other benefits. A container-optimized OS for example provides a much smaller attack surface since the number of potential vulnerabilities that an attacker can exploit is drastically reduced by only having the bare minimum of components installed in the first place. Stronger security is also achieved by having locked-down firewalls and strict update mechanisms in place by default. Most operating systems of this type are also configured for automatic initialization so that no on-host setup is required. Common examples for such container-optimized operating systems include "Flatcar Linux", "K3OS", "Talox Linux", "Fedora CoreOS" and the meanwhile deprecated "CoreOS Container Linux" [21]. The upcoming Section 2.4.2 will discuss, how running and managing containerized workloads at scale can be achieved.

2.4.2 Orchestration and Scheduling

By using techniques like containerization, the underlying infrastructure poses less of a challenge, even when dealing with different processor architectures, edge devices, bare-metal machines or systems in the cloud. However, this does not solve the issue of orchestration and scheduling of services. Orchestration in such a distributed and dynamic environment poses a significant challenge and is infeasible to perform manually at scale. Tools to manage containers that automate tasks like scaling, rolling upgrades or high availability are essential to ensure consistent service delivery and maintain the overall health and performance of applications in such complex systems, where manual service management is often impossible. It hence doesn't come as a surprise that container virtualization and the orchestration of containers onto a heterogeneous set of delivery targets is a recent trend in the IIoT environment that shows high potential [22]. Since IIoT systems often come with a variety of environments (i.e. edge, fog and cloud) it is desirable to commit to one technology that is able to fulfill the requirements for all of the environments in order to build a uniform system. While tools like "Hashicorp Nomad" or "Docker Swarm" exist, the current industry-wide consensus focuses mainly on "Kubernetes", which this work will hence also focus on. Since a large amount of cloud-native technologies are compatible with or even based on Kubernetes, this is a safe choice for an orchestration tool. Kubernetes can support applications that are distributed over hundreds if not thousands of nodes on a connected network and has the potential to provide effective scheduling and orchestration functionality for resources even across the network edge. While Kubernetes brings the features necessary to ensure scalability and reliability across environments it is also challenging to work with

in resource-constrained environments like edge devices (Section 2.2). These edge devices or microcontrollers are often not powerful enough to run a full-fledged Kubernetes solution upon themselves. To address this issue, custom edge-optimized Kubernetes distributions are one solution. These lightweight variants perform well even in resource-constrained edge environments and solve the main problems that make running Kubernetes on such devices challenging e.g. by using overlay networks in favor of virtual private networks. These solutions also have lower memory and CPU requirements so that the available resources can be used for running workloads more efficiently. Distributions like Rancher K3s also reduce the complexity of Kubernetes by bundling all components into a single binary and thus simplify operations of multiple Kubernetes environments. Finally, edge-optimized Kubernetes distributions typically support processor architectures that are common in edge devices like the arm64 architecture [11, 18, 23]. An alternative to lightweight Kubernetes is provided by tools like "KubeEdge" that extend the scheduling and orchestration functionality to edge devices, without running Kubernetes on them directly. This typically works by having the Kubernetes control plane, which is responsible for tasks like scheduling and orchestration, in a cloud or data center where processing power is available, and only running a small agent that follows the control plane's signals on the actual edge device. The more common approach is running lightweight Kubernetes distributions however, and will also be the approach in this work.

While Kubernetes acts as a comfortable abstraction over the hardware in use, it remains important for engineers to stay aware of the underlying infrastructure. Not only do containers depend on the physical hardware and thus e.g. have to be built for the according processor architecture in use, some functionalities of Kubernetes like replication and redundancy, that usually increase robustness and reliability, also become irrelevant when running a single node Kubernetes instance. Engineers must hence stay close to the systems and have proper observability and monitoring in place to ensure the reliability of the system [11].

2.5 Common Patterns and Architectures for IIoT

In this section, we will explore existing patterns and architectures for IIoT systems on the market and evaluate them briefly.

2.5.1 Automation Pyramid

In IIoT the two worlds of IT and OT come together, as discussed in Section 2.1. It doesn't come as a surprise that the OT world has its own framework for IIoT. The "Automation Pyramid" is one of the central frameworks in the OT world.

What started with the ISA-88 standard from the '90s is now closely related to the ANSI/ISA-95 standard. With the automation pyramid, we are looking at a conceptual element that strives to assert the effective and efficient distribution and connection of components in IIoT enterprises, from the shop floor to the decision-making levels and hence tries to represent all company levels. It describes the enterprise in a layered architecture where

each layer can only communicate with its adjacent layers. The objective of the framework is to describe the production process from the organizational and the process perspective while providing common terminology that can be the basis for discussion [1, 24].

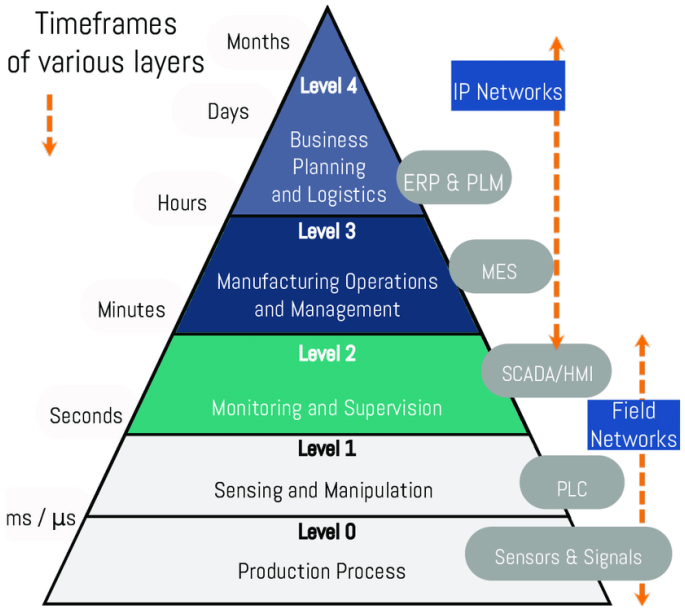


Figure 2.1: Automation Pyramid by United Manufacturing Hub [1]

The layers within the automation pyramid are closely linked to the varying levels of real-time responsiveness and decision-making requirements. A shift can be seen when looking at real-time process control and monitoring at the lower layers, towards more strategic and long-term planning and analysis at the higher layers. It is also to be noted, that there is a transition from OT (bottom) to IT (top) systems in the pyramid. This structure reflects the integration of both immediate operational needs and broader organizational objectives within the automation framework which can be seen in Figure 2.1. Starting from the bottom, the first layer represents components of the production process, which are mostly sensors or actuators. In this level (0), the actual production process is embedded which explains the high requirements regarding real-time capabilities while all subsequent levels deal with some form of automation components. The next level (1) is responsible for sensing and manipulation. Typical applications within this layer deal with Programmable Logic Controllers (PLCs) that run programs that read signals from sensors or write signals to actuators. The next level (2) deals with monitoring and supervision and is the place where systems like "Supervisory Control and Data Acquisition" (SCADA) or "Human Machine Interfaces" (HMI) are located. Those might for example supervise multiple PLCs in a production process that span over a hundred meters of an assembly line where one PLC is placed every 5 meters. HMIs can e.g. be found in a control room and allow humans to visualize a control panel or

to operate one or multiple machines at the same time in a remote location. The succeeding layer (3) has the responsibility of "Manufacturing Operations and Management", where systems like Manufacturing Execution Systems (MES) can be found. Here, management functions like generating orders are executed by the planning department. This level also houses the instruments for production and logistics, which for example includes responsibilities like decisions regarding what and when to produce. The last level (4) is called "Business Planning and Logistics". Here common systems for resource planning or product lifecycle management are located. This layer deals with long-term business/financial planning and is mainly responsible for organizational tasks. Inventories, billing, accounting and logistics are further things managed by this level. It is also used by many departments like finance, controlling, R&D or sales. Depending on the system, the transition between OT and IT happens somewhere around layer 2 or 3, however the exact line to distinguish the two worlds is often blurry [1].

As mentioned at the beginning of the chapter, the automation pyramid is a framework that mainly focuses on the OT world. It ensures that the production keeps running by combining all production-related components into autonomous layers that are not affected by casualties like IT outages. However, while this framework shines when looking at the production only, it imposes many limitations for projects dealing with large-scale IIoT systems. The main issue lies within the layered architecture. Since each component of each layer can only communicate with adjacent layers, each integration has to be built as a very specific point-to-point integration. This is very costly because many engineers with specific skill sets are required. It is also often required to use non-robust middleware to enable that communication, which can introduce many vulnerabilities into the system. The main downside of this is that it stifles innovation. Since every project comes with a high cost due to the requirement of building yet another point-to-point integration, development is slowed down immensely and thus time-to-market is heavily increased. Figure 2.2 shows the result of a growing system of point-to-point integrations.

Another issue is, that only the data that is relevant for the integration in question is sent. This means that while data is recorded at some layer, it might not be accessible on a different layer, because it was not a requirement in the integration that was built for the components. Also this results in data being siloed, which means that data recorded by one project is recorded but not available to all other projects for use due to the nature of point-to-point integrations. This is especially painful when decisions have to be made based on data of a layer, but the data is not available on the layer that is responsible for the decision yet. Lastly, the architecture leads to values based on data multiple times, since common values can't be reused by other components [1].

Overall it is clear that the automation pyramid is not suited to serve as a reference architecture for IIoT systems. It has drawbacks that limit scalability, innovation and other key factors and clearly only fits the OT side of things. Many are aware of its shortcomings and hence move to more modern hub-and-spoke architectures. It is important to note that it

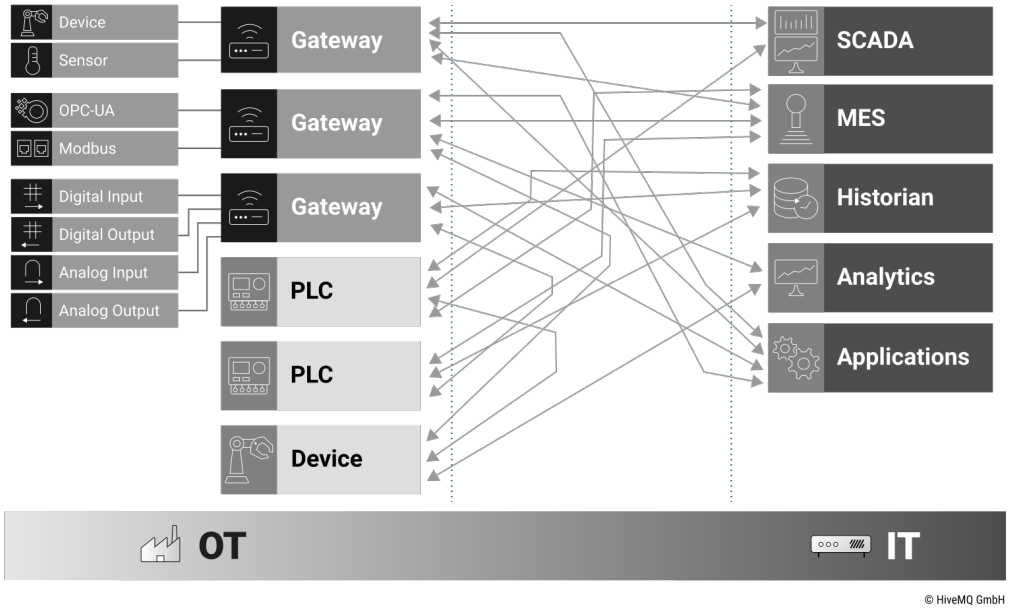


Figure 2.2: Smart Manufacturing Using ISA95, MQTT Sparkplug and the Unified Namespace [2]

brings useful components like the ISA-95 structure of levels which will be used in Section 3.3 and hence serves as the basis for terminology and structure in other reference architectures.

2.5.2 OPC-UA

The open platform communications unified architecture (OPC-UA) is published by the OPC Foundation, which with over 900 members is one of the world's leading organizations for interoperability solutions based on OPC specifications. It is an open information exchange standard for secure, reliable, manufacturer- and platform-independent industrial communications and has not only been the connectivity standard for nearly three decades but is still the most widely used open standard in the manufacturing industry.

The first OPC standard was named **O**LE **P**rocess **C**ontrol with "OLE" standing for Object Linking and Embedding. It enabled PLC controllers to deliver live data, alarms and historical data already back in 1996. In 2003 the OPC foundation started separating services from data and created OPC-UA as a service-oriented architecture which was later officially released in 2008 after verification with the goal to deliver a secure and reliable information exchange from sensors through to IT enterprise systems. In 2012 the standard was accepted as an International Electrotechnical Commission (IEC) standard called IEC62541.

OPC-UA aims to address typical needs in smart manufacturing or Industry 4.0 like the need for standardized data connectivity and interoperability while being independent of

both the device manufacturers and the operating systems used on the devices. Also, both horizontal (same layer, e.g. machine to machine on a shop floor) and vertical (across layers, e.g. device to cloud) data communications are supported by OPC-UA. Other interesting features of OPC-UA include the ability to scale from small microcontrollers with a footprint of around 15 kB up to multicore hardware. OPC-UA also provides mechanisms for application and user authentication, both for software and hardware systems. With object modeling, data can be described in a semantic way. Resources, also referred to as nodes, in the system are represented as objects with variables, methods, events and relationships to other objects. OPC-UA also supports automatic service and machine discovery while keeping independence of the transport method. Note that these are just some of the many requirements OPC-UA aims to satisfy. While this obviously results in a very rich and featureful standard, it also introduces lots of complexity and overhead that might be overkill in many scenarios. The OPC-UA is hence composed of 24 parts and the specification over 1200 pages long [25]. When numerous and heterogeneous applications and devices from a variety of vendors are connected, OPC-UA becomes more and more challenging due to the high complexity and sheer length of the specification [26].

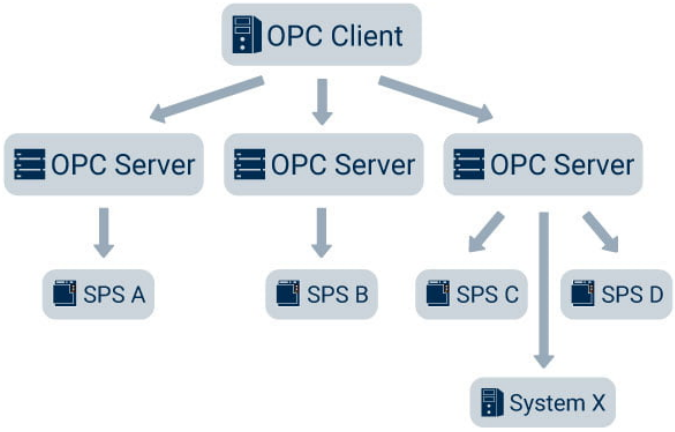


Figure 2.3: OPC-UA Client Server Architecture [3]

OPC-UA has discovered and dealt with the shortcomings like point-to-point integrations of the automation pyramid covered in Section 2.5.1. The classic OPC-UA system is designed upon a client/server architecture as shown in Figure 2.3 and uses TCP/IP and HTTP/SOAP. OPC-UA servers are responsible for converting the hardware communication protocols so that device data can be accessed by OPC-UA clients in a standardized way. Clients can decide when and which data the servers should retrieve from underlying systems like PLCs by polling on a configured interval. This is very simple since all resources (nodes) are individually addressable and (optionally) structured as data objects since OPC-UA supports object modeling. Nodes in the system are also automatically discovered.

This client/server model has its downsides though, one of them being, that the issue of point-to-point integrations is still a thing. While OPC-UA provides a standard format for communication, OPC-UA clients still communicate through direct bidirectional connections which results in similar problems as described for the automation pyramid in Section 2.5.1. Apart from the point-to-point integrations, the biggest issue is scalability, which can be traced back to the architectural decision for a client-server model. Each OPC-UA client creates n direct subscriptions/sessions to m OPC-UA servers. The clients periodically request updates which is often referred to as "polling". As the number of clients and subscriptions increases the servers will experience higher processing loads while the network traffic becomes more demanding. This can quickly lead to scalability issues [26, 27].

For these reasons, today's most widely used variant of the OPC standards is "OPC-UA PubSub over MQTT" which was released in 2018. The protocol MQTT will be further discussed in Section 2.5.3. With this variant, a separation between publishers and subscribers is implemented. The system is also re-engineered from a client/server architecture to an event-driven architecture, which addresses the scalability challenges. For today's IIoT systems, the PubSub variant of OPC-UA should always be preferred over the client/server model. It enables OPC-UA systems to scale from the edge up to the cloud. An MQTT broker acts as the single source of truth and enables all components in the system to access all data. This system architecture and its benefits will be discussed further in Section 3.3. The modern OPC-UA PubSub over MQTT standard also provides reference architectures for the large cloud providers like "Microsoft Azure", "Amazon Web Services" or "Google Cloud Platform". Through this new standard, most issues of the automation pyramid (Section 2.5.1) like data silos, point-to-point integrations and stifling of innovation are solved [27].

In summary, OPC-UA is an open standard that aims to fulfill as many requirements of industrial IoT as possible. Because of standards in communication, object modeling, flexibility in protocols and acceptance in the industry, OPC-UA is still the most widely used open standard in IIoT systems in manufacturing. While the client/server model of OPC-UA shines in simplicity, the OPC-UA PubSub with MQTT model should be preferred for today's systems, since the client/server model is not fit for today's scale of such complex projects. Overall, while OPC-UA is capable of fulfilling almost any use case one might have regarding an IIoT system, it is a very large and complex standard and its complexity must be taken into account when thinking about applying OPC-UA in a project.

2.5.3 Message Queuing Telemetry Transport and Sparkplug B

This section will deal with Message Queuing Telemetry Transport (MQTT) and the addition of Sparkplug B since they offer an interesting alternative to OPC-UA (Section 2.5.2).

MQTT

MQTT is a simple and lightweight protocol. It was developed for highly constrained devices and unreliable networks with high latency and is widely accepted in the world of IoT today, especially since the release of MQTT 5. The protocol guarantees a secure and reliable data exchange in industrial automation and is hence perfectly suited for use in the digitalization of manufacturing and production. Similar to OPC-UA (Section 2.5.2) MQTT is battle-tested and has even celebrated its 20th anniversary in 2019. The core principle of MQTT is publish-and-subscribe (pub-sub) where publishers send messages that are distributed to all consumers (subscribers) that are interested in the data. This is achieved using a hub-and-spoke model in which all communication is handled by a message-oriented middleware called the "MQTT broker". The broker handles needs such as redundancy, failover, high availability and scalability on the given infrastructure. MQTT clients (publishers and subscribers) in IIoT can for example be IoT gateways, edge devices or domain applications. They are loosely coupled since they only communicate via messages and only with the broker, not directly with each other. The broker can also manage MQTT sessions which allows for message persistence beyond the duration of client connections, which is optimal for resource and network-constrained scenarios. When a client reconnects to the broker after a network disruption, it will receive all missed messages from the broker's persistence layer. Since all data is sent via the broker, the MQTT broker becomes the single source of truth for all data. Also, through the change of communication structure the problem of point-to-point integrations that was an issue in architectures following the automation pyramid (Section 2.5.1) can be solved.

The messages sent by publishers are addressed to so-called "topics". Subscribers can then subscribe to these topics and will then only receive messages addressed to topics they are interested in. A topic is a UTF-8 string that uses the "/" separator to support multiple levels where a level might for example be the name of a device or its location. MQTT topics are simple and lightweight - the client does not need to create the desired topic before they publish or subscribe to it. The broker accepts each valid topic without any prior initialization. MQTT also supports wildcards for one or more levels so subscribers can e.g. subscribe to all topics related to a certain hierarchy level, which in practice could for example be the name of a physical location. Since topics are very lightweight they can be used in a very specific manner, for example by creating one topic per device [28]. A common pattern for topic structures is using the ISA-95 standard which can also be seen in the automation pyramid in Section 2.5.1. Topics can be structured based on the ISA-95 hierarchical structure "Enterprise/Site/Area/Line/Cell", which appears to be a perfect

fit for MQTT topics in IIoT and will be further discussed in Section 3.3. The payload of messages in MQTT is not restricted by the specification in any way and can be filled with any data format, where typically a format like "Protobuf" or "JSON" is used in practice. In IIoT, all components that used to communicate through bidirectional channels now only talk directly to the MQTT broker. This decouples the components of the system and solves the issue of point-to-point integrations mentioned earlier. Because MQTT was designed for scenarios with unreliable communication, the standard offers three quality of service (QoS) levels. Level 0 follows "at most once" semantics, which means that senders attempt to send their messages to the receiver once, without verifying success ("fire and forget"). Note that the receiver can be either a broker, when sending data from a client, or also a client, when the broker delivers data to a client. This level requires the least amount of bandwidth and energy and can be used when the devices are heavily constrained in terms of computational power or network stability and when the loss of a message is not critical. QoS level 1 is the middle ground between 0 and 2 and sends messages with an "at least once" approach. Here, senders retransmit messages until they receive an acknowledgment from the receiver. This guarantees that messages will be delivered but comes with the risk of sending duplicates. The highest QoS is level 2 which has "precisely/exactly once" semantics. Through a four-way handshake, successfully received messages are first acknowledged by the receiver who also stores the state for the message in order to filter out potentially upcoming duplicates. After the sender acquires the acknowledgment, it then sends a packet instructing the receiver to get rid of its state since it is no longer needed. The sender finishes the four-way handshake by acknowledging that instruction as well. Note that all messages are retried until they are successfully delivered while asserting proper handling of duplicates in this level. This QoS requires the most bandwidth and energy but also has the most intuitive semantics [26].

Compared to OPC-UA, using the protocol MQTT for central communication in an IIoT system shines through its simplicity. Not only is the protocol MQTT easy to understand, the specification is also strikingly short with only around 80 pages. The use of MQTT in the commonly used standard "OPC-UA PubSub with MQTT" shows that MQTT is a good choice for IIoT systems. Despite its simplicity, MQTT still provides many more useful features for IIoT. Through "retained messages", the broker can persist the most recent message of a topic. This message, which might for example be the latest reading of a sensor in an IIoT system, is always delivered to new subscribers of the topic, even when no messages are sent by the client after the subscription started. By using "persistent subscriptions", the broker can store messages for a client that has a persistent subscription but is offline. Once the client comes back online, all messages stored on the broker will be delivered. This allows for delivery guarantees even in systems with unreliable networks. Lastly, by using the "Last Will and Testament" (LWT) functionality, clients can specify a message that will be automatically published by the broker on their behalf, if or when an unexpected disconnection occurs. While devices can be automatically discovered by subscribing to the topic they publish to, their state remains unknown. By sending a message to a topic when coming online and immediately specifying an LWT message for when the client goes offline,

each device's state can be automatically discovered by subscribing to the according MQTT topics [26].

Sparkplug B

While MQTT is fully capable of being the central communication technology of an IIoT system, the "Eclipse Foundation's Sparkplug Working Group" created the project "Sparkplug B" that aims to improve this capability even further. The Sparkplug B specification describes how edge gateways, native MQTT-enabled endpoints and MQTT applications can communicate using MQTT while keeping focus on IIoT. Its goal is to provide implementation standards based on MQTT to fulfill interoperability requirements in industrial automation. This is achieved by three separate aims of the specification.

The first aim is to define topic namespaces. MQTT allows any UTF-8 string to be a topic, which stands contrary to the goal of achieving standardization. Sparkplug B solves this by defining a fixed topic structure:

$$namespace/group\_id/message\_type/edge\_node\_id/device\_id$$

The enforcement of this topic structure ensures that every communication partner uses the same topic structure thus ensuring standardized communication. While most of the hierarchical layers in the standard are just IDs or names, the "message_type" is another addition by Sparkplug B which can transmit metadata as well as information about the state of a device. The first message type defined in Sparkplug B includes "birth/death certificates" for announcing the online/offline status of devices. "Data messages" are used for sending standard data payloads. Since only changes in data are sent (report by exception) messages of this type only impose a very small load on the system. "Commands" are used to send commands to edge-of-network components or devices which is especially useful in cloud-to-device communication. Lastly, "state messages" are used for transmitting the status information of IIoT hosts in environments with multiple, non-cluster-capable brokers. The message type is optional and not used very often.

The second aim of Sparkplug B is to standardize payload data structures. While MQTT doesn't impose any limitations on the payload format, this can lead to inconsistencies. While JSON is the most popular data format via MQTT, in large projects other formats might be used. Because of this Sparkplug B defines a hierarchical structure for payloads including at least a timestamp, the metric itself and a sequential ID. It is also defined that the data is encoded using "Protocol Buffers", more commonly referred to as "Protobuf".

Finally, Sparkplug B defines a standard for state management. It specifies an ontology for all communication partners on how to transmit their state over the network. This includes states like the client status and offline-/online state. In practice, this is realized by sending birth/death certificates, last will and testament and retained messages all included in the MQTT standard.

In summary, Sparkplug B tries to solve MQTT's limitations like lack of object modeling or state management for industrial automation use cases [26]. It is to be noted that in a Sparkplug B-based system, all communication partners must use that standard, whereas sending plain MQTT messages will no longer be supported since it doesn't fit the standards set by Sparkplug B. In Section 3.3 it will be further discussed whether Sparkplug B should be used in a modern IIoT system and which implications it has on such a system in the real world.

2.5.4 High-Level Reference Architectures

In this section, we will take a look at high-level reference architectures that are often mentioned in the context of IIoT. These are less focused on hardware, bare-metal and platform infrastructure but more on business, strategy and manufacturing. In practice, they are rather used for designing, planning and discussing IIoT systems than for the actual implementation.

Industry 4.0: Reference architecture model for Industry 4.0 (RAMI 4.0)

In order to be able to achieve success in the modern "Industry 4.0" and smart manufacturing movements the German federal government in collaboration with German manufacturing companies created the reference architecture model for Industry 4.0 (RAMI 4.0). The model aims to combine all central elements of Industry 4.0 into a three-dimensional layered model in order to get a uniform structure and a ubiquitous language for all company and technology levels to base their communication on. Since we already discovered in Section 2.5.1 that there are several user perspectives in IIoT, especially when looking at the worlds of IT and OT, having such a basis for common understanding is crucial for success. In RAMI 4.0, production items are recorded and uniformly represented in IT throughout their entire lifecycle, ranging from components and machinery to interconnected production facilities. This allows for the integration of various sectors, for example IT and OT, by combining the real production item with its virtual representation, similar to a digital twin. RAMI 4.0 is designed with the objective of addressing the following goals:

- providing an open, modular and scalable architecture
- supplying a basis for common understanding and serving as a basis for discussion
- depicting the value progression throughout the entire product lifecycle
- integration of production, facilities, processes, etc. in one model

Figure 2.4 shows the three dimensions of the layered RAMI 4.0 model. The typical properties of a layered architecture like interchangeability of layers and defined interfaces between layers apply here as well and will not be explicitly discussed here. To understand the figure, we have to take a look at the different axes of the three dimensions, starting with the vertical

axis called “Layers”. Starting from the bottom, the first layer starts with assets, which depict the devices, machines, and components in the physical world. Above that layer, the integration layer represents the transition from OT to IT and contains the digital part of assets. The following communication layer ensures that access to data conforms to the Industry 4.0 standard and is the basis for standardized communication between parties in the system ranging over all ISO/OSI layers. The information layer describes data that is used or created by the technical functionality of the system. The subsequent layer is the functional layer and is responsible for the technical functionality of assets. Lastly, the business layer represents the business and operational components of the IIoT system. In summary, the vertical axis shows a digital image of the system layer by layer with production components and their functionality and data.

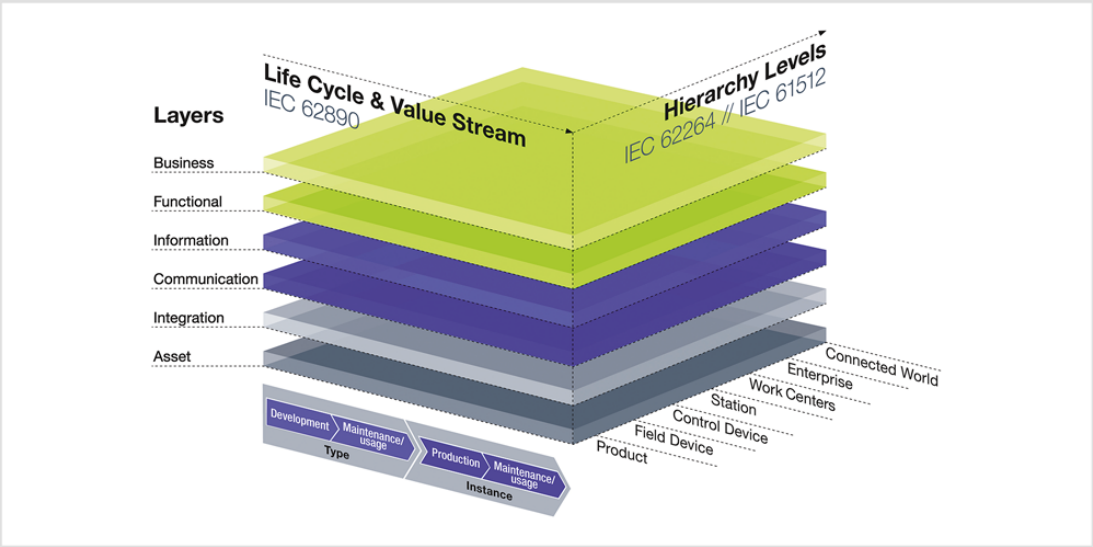


Figure 2.4: RAMI 4.0 as a layered model [4]

The left horizontal axis “Lifecycle & Value Stream” shows the lifecycle of products in the IIoT system, especially with regard to data collection, and follows the IEC 62890 standard. The lifecycle is grouped into two sections, starting the “Type”. This section represents products that are prototypes and still in development which is why it is further subcategorized into the parts “Development” which deals with actual development and construction of the production and “Maintenance/Usage” which concerns with software updates, manuals or product changes. The second section called “Instance” deals with products that are finished already. It also has two subcategories, in particular “Production” which deals with production-relevant data like quality measurements, serial numbers, etc. and “Maintenance/Usage”, again focusing on operational tasks like maintenance, optimizations, updates and more.

Lastly, the right horizontal axis is named "Hierarchy Levels" and follows the IEC 62264 / IEC 61512 standards which include an international norm for the integration of enterprise IT and control systems. The IEC 62264 standard also includes the automation pyramid discussed in Section 2.5.1 which is why strong similarities to the automation pyramid can be found in the legend of this axis. Apart from the new parts "Product", which focuses on the actual product under observation rather than sensors or actuators interacting with it, and "Connected World" which allows to portray today's internet-based approach to connecting production facilities over the network. Overall this layer shows the different functionalities within a production facility/ecosystem.

As mentioned earlier, the RAMI 4.0 model provides a basis for discussion. For instance, this allows to place other standards or norms into the three-dimensional model. This can help to detect overlaps or even gaps when evaluating or discussing frameworks or technologies. In summary, the model fits the use case of discussing, planning and designing IIoT systems perfectly, but is not aimed towards actual implementation and is hence not well suited for the purposes of the implementation-focused and practical goals of this thesis [5].

The Industrial Internet Reference Architecture (IIRA)

Another common standard in the field of reference architectures for IIoT is the "Industrial Internet Reference Architecture" (IIRA). The IIRA aims to address the need for a common architecture framework to develop interoperable IIoT systems for diverse applications across a broad spectrum of industrial verticals in the public and private sectors to achieve the true promise of IIoT, with focus on broad applicability, interoperability and identification and satisfaction of stakeholder needs. It was created by the "Industry IoT Consortium" and is based on the ISO/IEC/IEEE 42010:2011 standard [29]. Seen from a high-level perspective, the IIRA highlights the important architectural concerns of IIoT systems and classifies them into so-called "viewpoints" along with their respective stakeholders like product managers, developers or system operators. Based on this classification the IIRA proceeds to provide guidance to resolve the concerns in these viewpoints.

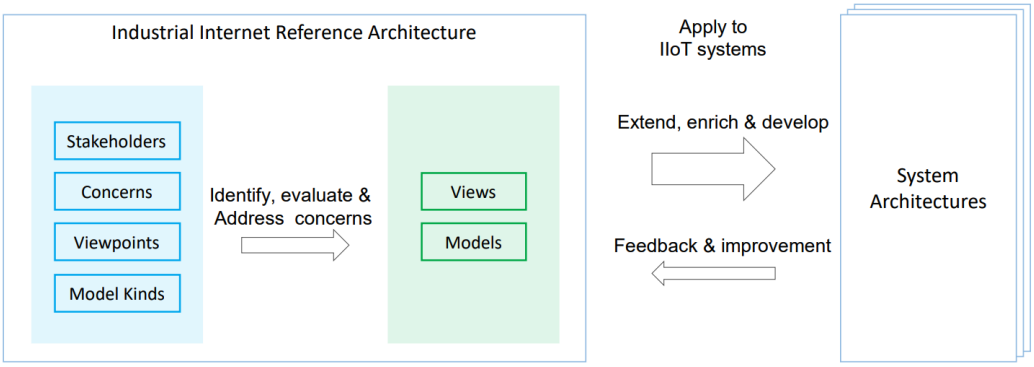


Figure 2.5: IIRA Constructs and Application [5]

Figure 2.5 shows the workflow of using the IIRA. In the first step, stakeholders and their concerns are identified. Based on these identified concerns, the architect maps the stakeholder concerns into a set of predefined viewpoints. These viewpoints are the business viewpoint, which focuses on business-related aspects of the system, the usage viewpoint, which looks at how users will interact with the system, the functional viewpoint, which concerns with the system's functional requirements and capabilities and finally the implementation viewpoint that revolves around the technical realization of the system. Note that in the IIRA, the viewpoints are arranged in an order that reflects the general interaction pattern between them. Decisions from a higher-level viewpoint (e.g. business viewpoint) guide and impose requirements on the viewpoints below it while lower viewpoints validate and in some cases cause revisions to the analysis and decisions in the viewpoint above it. Some crosscutting concerns might also intersect multiple or even all viewpoints, such as safety and security. As the last component of the left section in the figure, the IIRA provides a set of predefined model kinds that are the types or categories of models that are relevant to the viewpoints. A model kind can also be described as a blueprint for a model that specifies syntax, semantics, rules and more. Based on all of these parts, specific "views" and "models" are developed for each viewpoint. Views are representations of the system from the perspective of a specific viewpoint hence they are also specific to one or more stakeholders. An example is a view for a security viewpoint, which would for example include security concerns like authentication, authorization and confidentiality. Models on the other hand are detailed representations within the views. They are developed based on the defined model kinds for the according viewpoint and are typically structural, behavioral or domain-specific. Following the security viewpoint example, a model might show, how a user's identity is verified in the IIoT system. As with any kind of reference architecture, the model is then used to extend, enrich and develop the real system architecture. From learnings during development, the model based on the IIRA can be adapted through the feedback & improvement step, thus resulting in a continuous cycle of refinement [29].

While the process described so far mostly deals with requirements engineering, the IIRA also recommends well-established architecture patterns for IIoT including a three-tier architecture, pub-sub architectures or layered data bus that verify concepts mentioned in this thesis, especially in Chapter 3. The three-tier architecture pattern comprises edge, platform and enterprise tiers and suggests dividing the system into three "tiers", where each tier has its own responsibilities. This for example allows for practices like edge computing mentioned in Section 2.2 and cloud computing. The pub-sub architecture is the basis for the communication protocol MQTT which was discussed in Section 2.5.3 and is also mentioned as an example concrete technology for the message/data broker functionality in the IIoT system. Lastly, the layered data bus pattern suggests having multiple data buses, which might be a message broker like an MQTT broker, on different levels, where levels might, for example, be instances of the tiers of a three-tier architecture. It allows to have local communication compared to a centralized, often cloud-based, data bus, where all communication has to go to a central location first. Communication between layers only happens between

data buses, so no direct communication channels are used which helps with decoupling system components. By applying the layered data bus pattern, requirements like low latency, peer-to-peer communication, ability to work offline (important for manufacturing) or secure network segregation can be fulfilled. All of the just mentioned patterns will also be applied in the architecture proposal of this thesis in Chapter 3, which confirms that the suggested architecture proposal is well-aligned with established practices and principles in the field. Note that the IIRA mentions more patterns or capabilities relevant to IIoT systems like time-series databases or stream analytics which go beyond the scope of this work.

In summary, the IIRA outlines key concepts pertinent to IIoT systems, offering wide-ranging applicability across various industrial sectors. Its focus is primarily on the overarching system architecture and the generic aspects of IIoT systems. This contrasts with RAMI 4.0 detailed in Section 2.5.4, which is specifically tailored to smart manufacturing and the core principles of Industry 4.0 and less towards system architecture. The IIRA mentions patterns that are similarly structured to modern IIoT architectures and speaks about their benefits. In contrast to older architectures like the automation pyramid discussed in Section 2.5.1 it also supports requirements for modern IIoT systems like scalability or even concepts like edge computing. The IIRA is however at a level of abstraction, that excludes architectural elements that are only available in concrete systems and is not at a depth sufficient to implement a real system. The reference architecture is best used for planning and designing a large-scale IIoT system whereas other guidance should be used for the actual implementation of the system [29].

2.5.5 Cloud Reference Architectures for Industrial IoT

In this section, we will examine the reference architectures for IIoT as designed by major cloud providers such as Microsoft Azure, Amazon Web Services, and Google Cloud Platform. Our focus will be on identifying their limitations and exploring why they may not entirely meet the evolving requirements of contemporary IIoT systems thus emphasizing the need for a modern reference architecture in this sector.

One of the most common design frameworks used for IIoT scenarios is the “Microsoft Azure IoT Reference Architecture”. The framework is targeted to IoT in general and will be refined towards IIoT later with the “Microsoft Azure Industrial Internet of Things”. The reference architectures of other cloud providers are often similar and only differ in details, which will be mentioned later.

Figure 2.6 shows a high-level overview of the architecture in question. Similar to what was recommended in the IIRA in Section 2.5.4 the structure is separated into three tiers, where each tier has its own responsibilities. The architecture describes four kinds of components within the system: devices, a cloud gateway service, stream processors and a user interface. Devices and/or on-premise edge gateways are responsible for registering (often referred to as onboarding) within the cloud system and having connectivity options with the cloud for sending and receiving data.

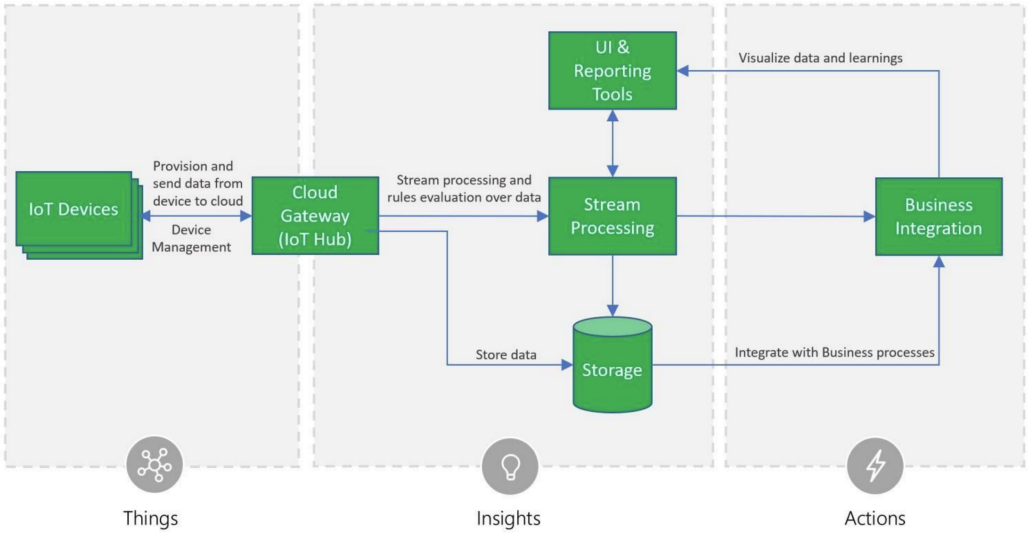


Figure 2.6: Microsoft Azure IoT Reference Architecture: Overview [6]

The receiver of this data is a cloud gateway service, also known as a hub. Its tasks include device management and securely communicating with both upstream and downstream components, while most importantly acting as the bridge to the cloud. Here we can see the main architectural difference to the three-tier architecture described in the IIRA in Section 2.5.4, which is, that all data is sent directly from the devices to the cloud for further processing, whereas in the IIRA, the "platform tier" might also be built on-premises. From here on managed cloud services can be used which is a major benefit when dealing with big data, since a large amount of processing power, storage and network bandwidth is available. Stream processors are components that consume data, integrate with business processes and place data into storage. Lastly, a user interface is required to visualize telemetry data and facilitate device management. Apart from the system architecture itself, the Azure IoT reference architecture also deals with other concerns, mainly managed cloud resources. Since we will later see that the reference architecture is not sufficient for many use cases, and managed resources are very cloud provider-specific, this is out of the scope of this work.

Diving deeper, one of the main components used in the Azure IoT architecture is the "Azure IoT Hub". It acts as the cloud gateway service and is a managed, highly scalable service that acts as a central message hub for communication. It behaves similarly to an MQTT broker (see Section 2.5.3) but apart from MQTT also supports data transfer with the protocols "Advanced Message Queuing Protocol (AMQP)" and HTTPS. It also integrates nicely with other cloud services by Azure, like Azure Event Grid for reacting to events, Azure Machine Learning for implementing AI solutions or Azure Stream Analytics for real-time analytic computations. Since it runs in the cloud, it's possible to subscribe and react to

events on the factory floor from anywhere in the world. However while typical IoT devices like smart home devices can easily run software that can communicate using protocols like MQTT, this is often not possible in IIoT due to reasons like resource or network constraints as discussed in Section 2.1 [6].

For applying this architecture to industrial IoT scenarios, Microsoft Azure provides guidance with the "Microsoft Azure Industrial Internet of Things". It is described as a suite of Azure cloud microservices and Azure IoT Edge modules and is said to provide the ability to integrate data from assets, sensors and existing systems into the Azure cloud using protocols like OPC-UA which was discussed in Section 2.5.2. A system following this architecture is constructed of at least one Azure IoT Hub, one or more Azure IoT Edge devices and various Azure IoT Edge modules. The IoT Hub, which has been previously described, acts as the central communication node and thus requires no further explanation. An Azure IoT Edge device is again composed of two components, namely an edge device and edge modules. The edge device itself is physical hardware running the software Azure IoT Edge on top of Linux or Windows, which can be described as a runtime that enables cloud functionality to be extended to local or edge devices. It allows developers to deploy, run and monitor containerized Linux workloads, which follows the same idea as orchestration tools discussed in Section 2.4. Devices with this runtime can be remotely monitored and managed through a cloud-based interface. On this runtime, so-called "IoT Edge modules", which represent units of execution, implemented as Docker-compatible containers, that run business logic at the edge, are deployed. While Azure provides prebuilt modules, custom ones can be built and deployed as well, which helps with fulfilling the requirement of running workloads at the edge as analyzed in Section 2.2.

An example use case for an edge module is the preprocessing and compression of data, since sending raw sensor readings across the communication link to the cloud is not desirable, especially in network-constrained scenarios. Note that these IoT edge devices don't necessarily have to be where the data is actually created. Actual components like sensors that might communicate using protocols like S7 or even OPC-UA (Section 2.5.2) can send their data to the IoT edge device, where protocol adapters, which are deployed as containerized applications, can then handle the translation to a protocol supported by the receiving Azure IoT Hub. This shows, that the reference architectures by cloud providers as expected often focus on the cloud side while neglecting the on-premises side which often requires other strategies like using OPC-UA [6] [30] [31].

As mentioned at the beginning of this section, IIoT reference architectures of other big cloud providers are of very similar structure. The solution of Amazon Web Services (AWS) for example uses "AWS IoT Core" as its cloud gateway instead of the Azure IoT Hub and "AWS Greengrass" as its edge runtime as a substitute for Azure IoT Edge, where both are very similar to their counterparts. AWS' IIoT solutions have a small advantage over the Azure solutions when it comes to flexibility since they support uncontainerized workloads and any kind of programming language whereas Azure IoT solutions only support containerized workloads written in a specified set of languages. This advantage is however irrelevant when it comes to the shortcomings of the reference architectures [32]. Much alike is the

IoT solution of Google called "Google IoT Core". Here, the edge runtime is called "Gateway Device" and communicates with the cloud gateway called "Google Cloud IoT Core". Again, the structure is very similar to the solutions from Azure and AWS. Note that after around five years of lifetime, Google informed its customers that the IoT core service would be shut down, hence it may not be available at the time of reading this thesis [33].

While all of the mentioned reference architectures by the three big cloud providers sound promising, they all have almost the same shortcomings when it comes to modern IIoT systems, with probably the most critical one being their edge runtime (Azure IoT Edge, AWS Greengrass or Google Gateway Device). First of all, the edge runtime practically has to run on every device, that needs to be able to communicate with the cloud. In many scenarios, this is not feasible due to resource constraints, security, existing software or just operational effort. Especially when the actual requirement is just sending data to the cloud, having an entire edge runtime running on a device is often overkill. Also, while the documentation suggests using a cloud-native orchestration tool like Kubernetes in the IIoT system, similar to what was discussed in Section 2.4, the edge runtime adds much complexity and increases operational effort into the system by adding another orchestration tool on top of the existing ones in the cloud or on-premises. To get back to the industry standard of using Kubernetes as the orchestrator with this architecture, in order to only have to manage a uniform technology across all environments, Azure makes the suggestion to use the CNCF project "KubeVirt". This project allows us to run the edge runtime inside a virtual machine, that runs inside a container which yet again runs inside a Kubernetes Pod. This introduces even more unnecessary levels of complexity and inflates resource requirements even further.

The architecture recommended by Azure involves a notable transition when shifting from the cloud provider-specific edge runtime to other environments like Kubernetes in the cloud. This transition is not just architectural but also involves a change in communication protocols. While Azure typically recommends using OPC-UA for on-premises solutions, it switches to MQTT starting from the edge runtime. This leads to challenges in maintaining uniformity in data communication and access across different environments. It is also obvious, that the cloud reference architectures favor the device-to-cloud communication path over other use cases. This kills important requirements like time-critical services or services with the capability to run offline on the edge that require data from the whole factory before it hits the cloud with delay since the edge runtimes are only designed to communicate with the cloud and not with each other. Since production critical parts like a SCADA system need to stay functional, they still have to be deployed on-premises and connected to the systems with point-to-point integrations. The actual manufacturing system in an IIoT system is mostly ignored by these cloud-centric architectures, which suggests that using the reference architectures doesn't solve issues that already existed in the automation pyramid, which we discussed in Section 2.5.1 already [6, 30, 34].

2.5.6 State of Reference Architectures

Overall we can see a shortage in reference architectures on the market, that are capable of satisfying the requirements of modern IIoT systems. We discovered that older designs like the automation pyramid (Section 2.5.1) mainly struggle with scalability and innovation due to point-to-point integrations. Regarding architectures, we saw that high-level reference architectures (Section 2.5.4) like RAMI 4.0 and the IIRA are not concrete enough for actual implementation, but still contain many ideas and concepts that are very relevant to this day and already solve many issues of systems strictly following the automation pyramid. Finally, we discovered that the reference architectures designed by the large cloud vendors (Section 2.5.5) like Microsoft Azure not only impose a vendor lock-in but are also very limiting and make common use cases like edge computing hard, if not impossible, to implement. The upcoming Chapter 3 will introduce a new, modern reference architecture that aims to solve this blank spot in the market.

3 Architecture Proposal

In this chapter we will discuss the recommended reference architecture for hybrid cloud industrial IoT systems designed by the company MaibornWolff GmbH. We will first explore the structure of the architecture followed by an in-depth exploration of essential topics that are necessary to be able to apply the architecture in a real world project. Apart from other sources, this chapter will follow the "Building IIoT 2023" conference presentation [7] by Sebastian Wöhrl and Marc Jäckle (MaibornWolff GmbH).

3.1 Requirements for the Hybrid Cloud IIoT Reference Architecture

Before introducing the hybrid cloud IIoT reference architecture, we must first think about the requirements, that the reference architecture must satisfy in order for it to be useful. Since the architecture in question is mainly focused on the actual implementation of such a system, we will concentrate on identifying and fulfilling common use cases whereas the architectures discussed in Section 2.5.4 rather focused on requirements engineering and planning.

A reference architecture fundamentally has a set of domain-independent requirements, providing a standardized framework applicable across various contexts. It provides a common vocabulary for all stakeholders to base their discussions on, reusable designs and building blocks and industry best practices that are used as a constraint for designing more concrete architectures. Unlike solution architectures, they are designed to function as a blueprint that can be applied to multiple projects within a specific domain, whereas solution architectures are tailored to address the unique needs of a single project, focusing on specific requirements and solutions. Leveraging reference architectures as foundational templates for creating custom solution architectures can significantly lower development costs and shorten project timelines [35].

Focusing more on the IIoT domain, many more requirements arise and are mostly dependent on the project. Concerns like edge computing, low-latency workloads, high availability & scalability, security & compliance, heterogeneous protocols and many more typically have to be addressed and must be dealt with in an IIoT reference architecture. We discovered in Section 2.5.1 that systems based on the automation pyramid which is seen as a standard in the world of operational technology have fundamental shortcomings like stifling of innovation and high cost due to point-to-point integrations between systems in the layered design, a new reference architecture must solve these known issues. High-level reference architectures discussed in Section 2.5.4 provide a good basis for discussion of an overall design, but lack

specificity when it comes to the actual implementation of a real system. Lastly, more concrete reference architectures designed by cloud providers as discussed in Section 2.5.5 often appear to lack flexibility and often kill important use cases like edge computing, which the reference architecture focused in this work must consider. It becomes evident that there is a notable gap in reference architectures on the market that offer a good balance between those that mainly serve as a high-level discussion and design basis and those that are concrete enough for implementation but often too specific and hence not suitable for many projects. The reference architecture that will be introduced in the next chapters aims to fill that gap.

3.2 Structure

Figure 3.1 shows a visual representation of the proposed hybrid cloud IIoT reference architecture. At first glance, we can see that the system is designed as a three-tier architecture, similar to the design in the IIRA in Section 2.5.4. These "tiers" will be referred to as "environments" from now on. The reference architecture describes the three tiers as "edge", "fog" and "cloud" environments, whereas each environment plays specific roles in processing data and control flows and has its own responsibilities. Note that the public literature often refers to the edge environment as the "far edge" and the fog tier as the "near edge". We can also see that all environments are based on the orchestration tool Kubernetes (see Section 2.4) which results in having a uniform technology across the whole system. Let us take a look at the responsibilities of each environment now.

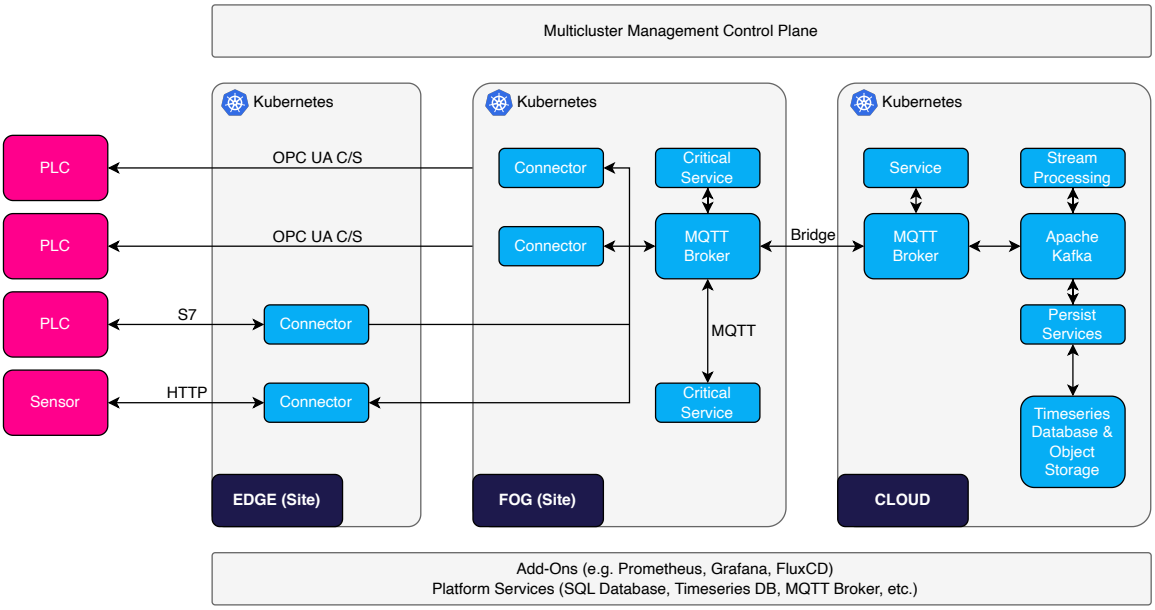


Figure 3.1: Hybrid Cloud IIoT Architecture by MaibornWolff GmbH [7]

The edge environment is the closest environment to the actual production site. It can exist many times per site and is mainly used for edge computing, which is especially relevant for low-latency workloads like visual inspection with machine learning, where latencies could cause damage to both humans and the production site. An edge environment typically consists of a bare-metal edge device running a single-node Kubernetes cluster, where an edge-optimized distribution of Kubernetes is used (Section 2.4). Typical edge computing use cases include data preprocessing like aggregation, which is useful for countering high sampling rates that typically are an issue in IIoT systems or simple data analytics. Another very common use case for the edge environment is running adapter/connector services for IIoT components like sensors or PLCs that do not natively support the central communication protocol - in this case MQTT (see Section 2.5.3) - where the adapter service performs a protocol translation e.g. from Bluetooth Low Energy or LoRa over to MQTT. One scenario where such connectors are required is when common industry standards dealing with security and compliance like the cybersecurity norm "IEC62443" are enforced. In this norm, the segmentation of networks based on principles like least privilege, i.e. limiting the amount of network participants with access to data as much as possible, is required. A practical application of this in the real world happens when dealing with the S7 protocol, which is very common in the OT world and transmits data without any encryption. Following the cybersecurity norm, communication is then only allowed between communication partners that reside in the same network segment [36]. An edge device in the same network segment as the IIoT device (e.g. a PLC) can be used to receive and send data via the unencrypted S7 protocol and forward it into other network segments using an encrypted protocol thus being compliant with the IEC62443 norm.

The successor of the edge environment is the fog environment, often referred to as the near-edge. It typically exists once per production site and consists of a highly available Kubernetes cluster running on bare-metal servers or virtual machines. It serves as the optimal environment for workloads that demand latency levels less stringent than those on the edge, yet more critical than those in the cloud. Apart from typical data aggregation or analytics use cases, this environment houses production critical services. Since the environment runs fully on-premises, operability even during internet outages is guaranteed. The fog environment collects data from and sends data to all edge environments in the same site and communicates bi-directionally with the cloud environment. Unlike implementations following the IIoT reference architectures by cloud providers (Section 2.5.5) running services that require data from the whole factory before it hits the cloud with delay is easily feasible with this architecture. Similar to edge environments, having data just in an on-premises system also solves some confidentiality issues, e.g. by preprocessing data before storing it in the cloud or by just having stronger network security. Finally, the fog environments host MQTT brokers (Section 2.5.3). The MQTT broker in this environment is the central communication medium in the production site and should thus be deployed in a highly available fashion. The fog environment is also the default environment for machine connection to IoT-ready devices, which in our case means devices with support for MQTT, that directly send their data to this MQTT broker. Devices that need adapters for protocol translation

from a protocol that is encrypted like OPC-UA to the MQTT standard can send their data to connector/adapter services which then forward the translated data to the broker. Devices that communicate using unencrypted protocols must send their data to a connector/adapter service in the edge environment, which then yet again forwards data to the MQTT broker. We will discuss the MQTT brokers further in Section 3.3.

The third and last environment is the cloud environment which only exists once per IIoT system following this reference architecture. It will typically consist of a Kubernetes cluster managed by a cloud provider and a set of other supporting cloud resources. In this environment, typical domain services that have more relaxed requirements regarding latency and confidentiality are deployed. Access to the managed resources of cloud providers like databases, data/stream analytics, machine learning or storage enables developers to build services at an enormous scale. Infrastructure services, which the cloud provider does not offer as managed resources, can be efficiently operated within the managed Kubernetes cluster, offering significant scalability. Because of the high amount of computational power, the cloud environment is also ideal for shared services like running CI/CD agents, hosting a private Git instance or having a central command & control system with capabilities like observability (Section 3.5) and multi-cluster management (Section 3.6) for operating the growing amount of environments. Apart from shared services, the cloud environment hosts the central MQTT broker, which all MQTT brokers in the fog environments communicate with, exchanging their data bi-directionally. We will discuss this MQTT broker as well in Section 3.3.

Note that this reference architecture is just a base recommendation and must be adjusted according to the requirements of the project. For example while Figure 3.1 shows an instance of the distributed event streaming platform "Apache Kafka" in the cloud environment, Kafka can also be deployed in the fog environment of a site if the functionality is required there as well. Also while the architecture suggests using MQTT as the central protocol that also bridges data between fog environments and the cloud, it is absolutely possible to use Kafka for this functionality, if it is necessary for fulfilling the requirements of the system in question. Being able to schedule any workload on all environments is mainly possible due to the central orchestration technology Kubernetes (Section 2.4) which is the basis of all environments and creates a uniform, homogeneous and highly flexible system. While different Kubernetes distributions like an edge-optimized distribution on edge and fog environments and a managed solution in the cloud environment are used in practice, the environments still offer a uniform set of interfaces for systems or humans to interact with [7].

3.3 Unified Namespace as the Basis for IIoT Systems

It can be said that the success of a digital transformation strategy depends on how well-integrated organizational data is across business units and technology domains. In Section 2.5.1 we discovered that traditional and especially OT-based architectures in the industrial IoT domain have major disadvantages that often make them infeasible in modern systems. We discussed how point-to-point integrations between components of adjacent layers hinder scalability from both the business (high cost for every integration) and the technology (network and performance issues) perspective but also innovation and productivity. These hundreds, if not thousands of point-to-point integrations are a set of implementations only the domain specialists of each integration understand, which almost guarantees to result in technical debt at some point. We also found that this structure leads to data gaps, as only the data necessary for specific integrations is passed around. It also results in data silos, because of data being captured by individual components but not made accessible to other parts of the system. Both of these issues – data gaps and data silos – make some use cases like accessing device data from the cloud, which requires integrations between every level, very cumbersome. Lastly, we saw that the heterogeneity of data formats in IIoT is a challenge that needs to be tackled in order to be able to design, develop and maintain a modern IIoT system at scale [8].

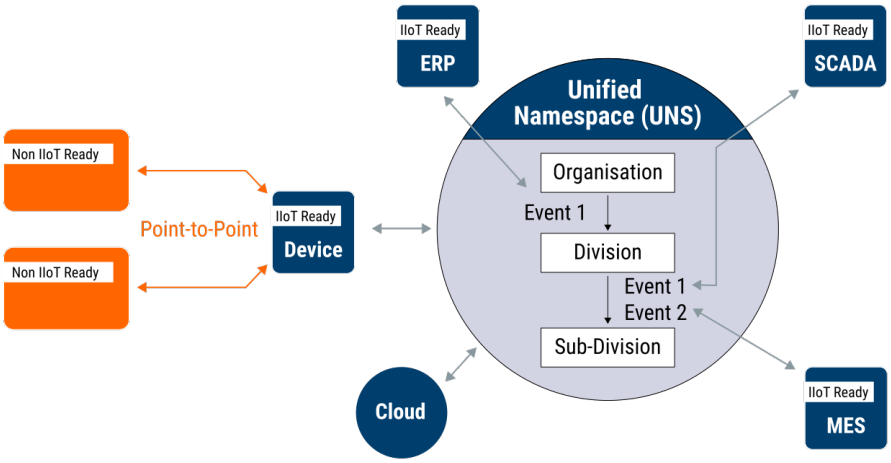


Figure 3.2: Unified Namespace [8]

In Section 2.5 we already saw that many more modern architectures follow a pattern where a central message-oriented middleware is used in order to build a uniform basis for data exchange, which has proven to be a more suitable approach for a smart manufacturing or IIoT architecture [2]. Examples of this design are the modern standard “OPC-UA Pub-

Sub over MQTT" (see Section 2.5.2) with a central MQTT broker or the "Microsoft Azure Industrial Internet of Things" reference architecture which uses the Azure IoT Hub as a central communication node. A modern and increasingly common approach based on this concept is the "Unified Namespace" (UNS) which is used in the proposed reference architecture shown in Figure 3.1. It follows the idea of having all data in a centralized location with a standardized and uniform structure - the unified namespace - and can be seen in Figure 3.2. This centralized location is a central hub or broker which becomes the single source of truth for the whole IIoT system and is structured in a way that represents the actual structure of the business. A common basis for the structure is the use of ISA-95 layers from the automation pyramid which was discussed in Section 2.5.1. This shows that despite its shortcomings, the automation pyramid can still be complementary to a modern IIoT architecture [2]. After applying the shift of communication structure from point-to-point integrations to a centralized messaging system all components of the system communicate through a central repository of information using a uniform protocol thus solving the issues regarding point-to-point integrations. Components that are not capable of communicating in the uniform protocol are placed behind gateways or adapter/connector services that then bidirectionally communicate with the UNS. The uniform protocol in a system based on the unified namespace must use a publish-and-subscribe (pub-sub) architecture like the standard OPC-UA PubSub with MQTT (Section 2.5.2) or the protocol MQTT (Section 2.5.3), which leads to a loosely coupled, scalable, highly-available and fault tolerant system supporting millions of nodes. Apart from scalability and high availability, the use of this architecture has several beneficial implications. Using this structure, all entities in the system get immediate access to all relevant data by just communicating with the UNS which leads to a simplified integration into the system and thus a reduction of cost since no specialized engineering effort for building custom integrations is required. The ability to access all IIoT data in real-time also enhances development speed, boosts productivity, and opens up greater opportunities for innovation [7, 8].

Let us now look at the requirements that a unified namespace must meet. First of all the UNS should be built upon open-source components in order to assert interchangeability and expandability. The UNS system must also be lightweight since IIoT systems often live in both resource- and network-constrained environments as discovered in Section 2.1. Apart from the edge-driven and push-based approach of the "PubSub" architecture which was identified as one of the best communication strategies in IIoT, this can be taken even further by applying "Report by Exception" which means that data producers publish data to the UNS only when they detect changes in the monitored value thus reducing network traffic and requirements regarding processing power even further. The following paragraphs will describe how the UNS can be implemented based on the reference architecture shown in Figure 3.1.

The most common open specifications used for implementing UNS projects are MQTT and OPC-UA. In the proposed reference architecture of this thesis, MQTT is used to re-

alize the unified namespace as Figure 3.1 shows. As analyzed in Section 2.5.3, MQTT shines through simplicity and understandability which is the main reason why the protocol is chosen in this reference architecture. Its flexible and hierarchical topic structure fits the ISA-95 standard used for structuring the unified namespace perfectly. Following the ISA-95 layers, messages are addressed to the according topics based on the structure "Enterprise/Site/Area/Line/Cell". This simplifies the uniform data access even further since data can be accessed or produced by just subscribing or publishing to the desired topic. Even accessing all data of one or more layers of the ISA-95 standard is possible by using MQTT wildcards. Note that while the ISA-95 standard is a common recommendation for the implementation of a unified namespace, a custom structure should be used if the standard does not fit the structure of the business in question [8]. Regarding performance, MQTT has been proven to fare well when benchmarked against competing protocols like OPC-UA [37] and hence represents a solid choice for central communication technology.

In Section 2.5.3 we discussed the open-source specification "Sparkplug B" on top of MQTT which claims to be a framework for seamless integration of applications, sensors, devices and gateways within an existing MQTT infrastructure in a bidirectional and interoperable way. By defining a fixed topic structure, a payload format, a set of commands and a strategy for dealing with state, the specification aims to help with achieving standardization across a large environment, for example an IIoT system. However with the use of Sparkplug B also come disadvantages. First of all the integration of the common standard ISA-95 with hierarchical MQTT topic structures is not compatible with the standardized topic namespaces of Sparkplug B. As a reminder, the Sparkplug B topic namespaces are defined as "namespace/group_id/message_type/edge_node_id/device_id". To tackle this issue, Sparkplug B provides two distinct methods. The "Parris Method" by Matthew Parris fits the entire ISA-95 structure into the "group_id" level of the Sparkplug B topic namespace by using a custom delimiter rather than the topic delimiter "/" of MQTT which might for example look like "spBv1.0/Plant1:Area3:Line4:Cell2/NDATA/edge1.2.1" where the "namespace" field is set as the "Enterprise" level of ISA-95. While this provides a simple fix for combining ISA-95 and Sparkplug B, it falls short since MQTT does not support publishing or subscribing to substrings. This means that applications can for example no longer subscribe to all data of a production site with a wildcard directly, but need to subscribe to all topics of the namespace (in this case "spBv1.0/#") and destructure messages themselves thus resulting in much higher data volume and use of computational power. The other method provided by Sparkplug B is the "Schultz Method" developed by David Schultz which doesn't just use a single broker but multiple brokers at various levels (e.g. Area, Line, Cell) of the enterprise to build the UNS. Lower-level brokers will publish their data to the higher-level broker to achieve a single source of truth in the end again. While this method is also feasible, it again adds lots of additional complexity and operational overhead.

Another issue is that Sparkplug B uses Protobuf as the encoding for the payload. Although it offers greater technical efficiency, the binary encoding used by Protobuf lacks the human readability of the industry-standard JSON. In the realm of IIoT, where data frequency is

often a more significant concern than data volume, the larger size of a JSON message is usually not an issue. More drawbacks include the fixed Quality-of-Service levels defined in Sparkplug B which might not fit every use case, the limited command set which misses common use cases like backfilling historical data and the lack of a specification about time series compression [7, 8, 37]. Since Sparkplug B shows several shortcomings when it comes to implementing a unified namespace for an IIoT system, and its useful features can be implemented when needed anyway, plain MQTT is chosen in favor of Sparkplug B in this reference architecture.

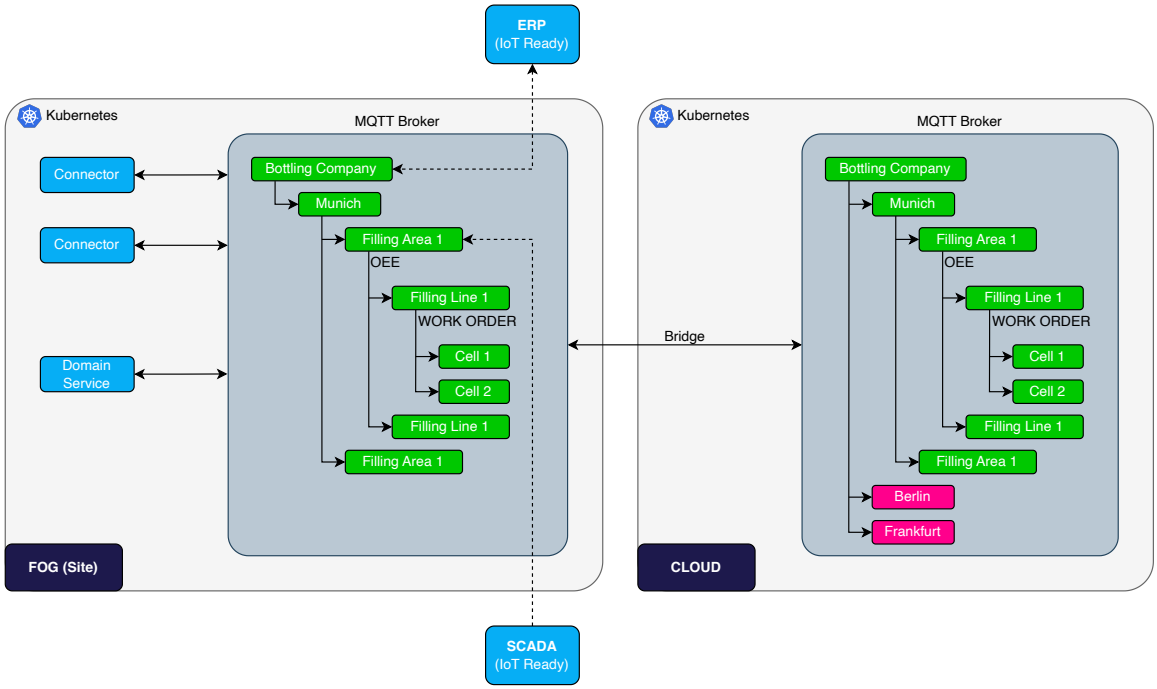


Figure 3.3: Concrete UNS implementation following the reference architecture [7]

Next, we explore how the implementation of the UNS is realized when following the proposed reference architecture. Figure 3.3 shows one fog environment (left) and the central cloud environment (right) of an exemplary implementation. Starting with the fog environment, we can see the sample structure of the UNS realized within an MQTT broker based on the hierarchical ISA-95 topic structure. The graphic not only illustrates how devices interact with the UNS through connectors, but also shows how common IIoT components (see Section 2.5.1) can easily connect to the IIoT system by consuming and/or producing data to topics of interest in the UNS. Examples of this are a SCADA system, that subscribes to all data required to run the production and publishes commands back to the UNS for others to receive, or an ERP system, that subscribes to the data of the whole production site in order to calculate and publish key performance indicators (KPI) [8]. Since this environment

runs fully on-premises, every IIoT component or service can instantly access the production site's data, avoiding the latency associated with communication to the cloud environment. The figure also shows that the cloud system also hosts a unified namespace based on an MQTT broker. This strategy is called "layered databus" and was already mentioned in the IIRA in Section 2.5.4. The idea here is, that each production site has its own UNS for the communication within the site. Even if the connection to the cloud were to be disrupted, the production site would remain operational. The MQTT brokers representing the unified namespaces of the production sites forward their data to the central MQTT broker using an MQTT bridge, which consumes all topics of the production site broker and publishes the messages to the same topics on the cloud broker. Note that depending on the use case, it might be desirable to use a technology with even stronger message delivery guarantees like Apache Kafka for the bridge between the on-premises sites and the cloud. This reference architecture chooses MQTT over Kafka mainly for standardization purposes across a large ecosystem. The cloud broker, serving as the enterprise-wide single source of truth, enables workloads operating in the cloud environment to efficiently perform operations on extensive enterprise data by simply subscribing to the topics of interest. Additional supporting tools, such as data streaming applications like Apache Kafka, and long-term storage databases like TimescaleDB or OpenSearch, can also process or persist the data managed by the central cloud broker. Note that the data in the cloud broker may not be the most recent due to latencies or even connectivity issues, which is why low-latency workloads should run in the fog or even the edge environment. The MQTT bridge can be implemented bidirectionally, enabling use cases such as issuing commands from the cloud to edge devices. Executing a command becomes as straightforward as publishing a message to the appropriate topic in the cloud environment. This message is then seamlessly forwarded through the bridge to the on-premises MQTT broker, where the corresponding edge device can receive it [7].

3.4 CI/CD and GitOps

In Section 2.3 we discovered that the GitOps framework can be very useful in large-scale platforms with an enormous set of heterogeneous delivery targets. In this section, we will look at how the reference architecture proposes to employ GitOps in a hybrid cloud IIoT system.

Once an IIoT platform surpasses a few production sites, the amount of environments in the system increases rapidly. Classic CI/CD pipelines then quickly begin to cause issues not only in terms of scale but also in their complexity. For a large-scale IIoT platform, having a simple and scalable deployment mechanism for all Kubernetes-based targets distributed over a large set of environments is desirable. Common use cases like batch deployments to many environments at once and sharing as much configuration between deployment targets to reduce code duplication should be possible with the chosen strategy. The reference architecture suggests using the GitOps model which we already discussed in Section 2.3 by having a GitOps controller running on each environment's Kubernetes instance which is configured

to watch the parts of a Git repository relevant to the according environment. The implementation of the pull model in GitOps comes with significant scalability, making it highly effective for large-scale systems since the computational power of each environment is used, rather than using central and shared computational power like in common CI/CD pipeline implementations. Eventually, the GitOps controllers will sync the desired state from Git and the actual state of the delivery targets. Through this model, the system gains additional security since information is only pulled from Git rather than being pushed from pipelines, which allows engineers to further lock down their systems since no access from outside is required for the delivery anymore. In the context of Kubernetes, this means that the Kubernetes API does not need to be exposed to the CI/CD pipeline agents but can remain only accessible to the developers, which increases security even further. The reduction of code duplication is typically achieved by using techniques like "Helm Templates" or configuration management and overlay tools like "Kustomize". These offer ways to share as much code as possible between environments while giving developers the possibility to customize the configurations per environment where needed - all based on configuration files that again can be stored in Git and pulled by GitOps controllers running within the environments. Using shared code, batch deployments can easily be made since the configuration is shared across many environments anyways [7].

To effectively implement the GitOps framework, every component of the system must be describable in a declarative format and reconcilable by some sort of controller. While this is simple for typical workload deployments, it gets more challenging for infrastructure. To address this challenge, Kubernetes-native tools such as Crossplane, Pulumi, ClusterAPI, Custom Operators and many more options can be used. These tools integrate Infrastructure-as-Code capabilities into the GitOps framework by providing custom GitOps controllers for the according configuration types. An example for this is using custom resource definitions for managing cloud provider resources such as managed databases. Domain-specific custom operators even allow to store configuration like RBAC management of applications such as databases as configuration in Git. The target of this is that everything, be it deployments, infrastructure, cloud resources and more, is stored in Git as the single source of truth in the end. To be able to store secrets in Git in an encrypted manner, tools like "Mozilla SOPS", "Sealed Secrets" or "External Secrets Operator" are commonly used. Depending on the project it can make sense to give all developers access to this single source of truth so that teams can manage infrastructure etc. through some sort of agreed upon Git workflow (pull requests, code reviews, etc.), which reduces the risk of the platform team becoming the bottleneck of the project.

It is also essential to mention that the use of the GitOps framework which the reference architecture suggests is not a replacement for CI/CD pipelines. While GitOps controllers manage continuous delivery tasks, pipeline implementations remain responsible for integration activities such as building projects, running tests, and publishing artifacts. These pipelines generate configurations, such as Helm charts, as artifacts, which are from then on managed only by the GitOps controllers. This structure leverages the strengths of both

approaches: it utilizes the continuous integration capabilities of pipelines during the development phase and employs the continuous delivery mechanisms of the GitOps framework for deployment.

3.5 System Observability

This section deals with the observability of a system built upon the proposed reference architecture. At the latest when a system surpasses the Proof-of-Concept (PoC) stage and grows beyond a few environments, having a central observability system that reaches across the whole system becomes a non-negotiable requirement, especially due to the highly distributed nature of the system. The suggestion of the reference architecture to build the system based on a uniform technology, in particular Kubernetes, allows for applying a standardized strategy for observability encompassing all environments. The significance of observability extends beyond typical monitoring scenarios, as highlighted by its inclusion in the "Open Web Application Security Project Top 10" (OWASP Top 10), which is a standard awareness document for developers and web application security. It represents a broad consensus about the most critical security risks to web applications and includes "Insufficient Logging & Monitoring" as one of the most critical ones. While the lack of observability is hardly represented in CVE (Common Vulnerabilities and Exposures) or CVSS (Common Vulnerability Scoring System) data, failures in this category directly impact visibility, incident alerting and forensics, which indicates the importance of having proper observability set up [38].

In the context of observability, it is often spoken of the "three pillars" of observability which are metrics, logs and traces. Since this thesis focuses more on the infrastructure and less on the domain-specific applications of an IIoT system, and traces are mostly used for application development while metrics and logs give insights about infrastructural components, this work will exclude traces and leave it as further work. Note that just having metrics and logs is not enough - supporting systems and strategies have to be applied in order to fully use their potential.

> "Logs, metrics, and traces are useful tools that help with testing, understanding, and debugging systems. However, it's important to note that plainly having logs, metrics, and traces does not result in observable systems." – Cindy Sridharan, Google [39]

Let us explore how to effectively create and store metrics and logs and how they can be used efficiently in a system. Metrics are numerical representations of a value of a point in time stored as time series data. Components within the system, such as the MQTT brokers of the unified namespace (Section 3.3) can provide metrics like network throughput and messages per second, offering valuable data for other components to utilize. Domain-specific applications can be instructed to provide both predefined metrics like resource and storage

usage and domain-specific metrics that depend on the use case. The metrics are typically processed in a push model, where services push their metrics into a centralized store, or a pull model, where an agent is configured to scrape (pull) metrics from all configured targets to then persist them. Compared to the other pillars of observability, metrics are cheap to store while still providing valuable information, especially for monitoring and alerting. The reference architecture suggests storing metrics of all system components in a centralized store in the cloud. Each environment will first store its own metrics in a local time series database which acts as a buffer, and will remotely write the data into the centralized store in the cloud [7]. This strategy ensures that monitoring persists during network disruptions by locally buffering metrics, which are then forwarded once the network connection is reestablished, preventing the loss of data that would occur without such a buffer. Finally, a user interface in the cloud environment is used to visualize metrics of all environments in the system to enrich their value even more.

Logs on the other hand are omnipresent in software and are of tremendous help when trying to understand a system and its behaviour. While metrics are continuously present, logs only appear occurrence-based and capture information on what and when something happens in the system. Since the sheer volume of logs can cause large amounts of storage and network usage, it is often infeasible to centralize the log aggregation of a full system which is why the reference architecture suggests only aggregating them per production site with sane retention times [7]. The storage for logs is typically realized by using a search engine like OpenSearch or ElasticSearch, which are time series databases optimized for efficiently persisting ("indexing") and querying string-based data, which fits perfectly for logs. Each machine in each environment runs an agent called a "log shipper", which forwards logs generated by applications running on the machine to the search engine. Using the search engine, developers can easily query the long-term storage to acquire the desired logs of any component running within the according production site. In order to maximize the value of logs, structured logging should be used which describes writing logs in a well-structured and consistent format that can be easily read, searched, and analyzed by any application or any interested individual. The most common structured logging format is using JSON [40]. To enhance querying capabilities, the JSON fields should be uniform across the environments. A minimal log entry could e.g. be defined as a JSON document containing a "timestamp", "message" and "environment" field.

By having a centralized cloud-based store for metrics and a search engine storing logs per production site and one search engine in the cloud for logs occurring in the cloud environment, the two pillars of observability "metrics" and "logs" are sufficiently covered.

3.6 Multicluster Management

While having good observability (see Section 3.5) certainly helps with monitoring and understanding a large-scale IIoT system, the management of the high amount of environments still poses a challenge. It is desirable to be able to manage each system from a central location, especially in critical topics like upgrading Kubernetes versions, configuring role-based access, single sign-on and more. Since all environments are based on Kubernetes a uniform solution can be used for this, which is where "Multicluster Management" solutions come in handy. Since this component is mostly just operating on top of the infrastructure, this work will only briefly mention some common solutions rather than diving deeper into the topic.

One of the most common open-source multicluster management solutions in the Kubernetes environment is Rancher. The developers describe Rancher as "a complete software stack for teams adopting containers. It addresses the operational and security challenges of managing multiple Kubernetes clusters while providing DevOps teams with integrated tools for running containerized workloads" [41]. Another similar enterprise product is "D2iQ Kubernetes Management Platform - DKP" which follows a common goal. Overall, the solutions allow to centrally operate many Kubernetes clusters by e.g. providing tooling to configure role-based access management, network access and cost analysis across all managed clusters. This is typically achieved by running a control plane on a dedicated management cluster, that is only used for the management of other so-called workload clusters. Each workload cluster runs an agent which creates a tunnel back to the control plane, often involving a proxy so that all clusters can be accessed via the management cluster. This is especially interesting with restricted networks like in cloud-to-edge scenarios since tunnels are securely encrypted and thus allow for safe access [7]. Many multicluster management solutions like Rancher also have features that allow for provisioning clusters onto different infrastructure ranging from cloud providers to bare-metal. Unfortunately, the multicluster management platforms in their current state often lack extensibility, which makes use cases like provisioning Kubernetes onto own bare-metal hardware difficult if not infeasible.

Another tool for multicluster management is "ClusterAPI" which has emerged as an increasingly prominent project in the domain of cloud computing and orchestration [42]. It mainly focuses on simplifying the management of the full lifetime of workload clusters including provisioning, operating and upgrading by abstracting the underlying complexity. ClusterAPI will be further discussed in Section 4.2.3.

In summary, a multicluster management solution is helpful with a growing amount of environments and helps to manage each environment by providing a central control plane that facilitates oversight, coordination, and operation across the variety of environments. This enhances efficiency and consistency regarding policy enforcement across all environments and thus helps to satisfy the requirement of central management across an entire IIoT system.

3.7 Identity and Access Management

Similar to how having a uniform strategy for observability across all environments is required, it is also essential to have such a strategy for identity and access management in all tooling in order to be able to maintain an IIoT system at scale. Since the proposed architecture employs GitOps (see Section 2.3) as the continuous delivery strategy, the already existing role-based access control (RBAC) mechanisms of the Git repository hosting service (e.g. GitHub or GitLab) can be used for both code access and continuous delivery. In a scenario where the "everything as code" approach is fully implemented, those developers granted the necessary Git permissions are able to alter the configurations across the entire system. By having one GitOps controller per environment, or even per team per environment, the segregation of teams can easily be realized. Kubernetes offers a rich set of RBAC functionality with service accounts, roles and role bindings that can be managed through manifests in Git. By employing a multicluster management solution (see Section 3.6) the RBAC configuration for access to the Kubernetes instances can be simplified and centralized across all clusters.

Lastly, access to applications in the system like observability tools can be unified by relying on a common standard like OAuth 2.0 and OpenID Connect. This allows to use a central, preferably already existing, identity provider like Azure Entra ID or the open-source project Keycloak for authentication and authorization across the whole system.

4 Infrastructure Provisioning

In this chapter we will take a look at the core question of how the infrastructure necessary for realizing the reference architecture from Chapter 3 can be provisioned. Since provisioning managed cloud services is a solved problem, this chapter will focus on provisioning on bare-metal machines while taking into account that a uniform strategy for provisioning the infrastructure of all environments (edge, fog and cloud) should be chosen.

4.1 Advantages and Challenges in Bare-Metal Infrastructure

In IIoT systems having own hardware is almost always a non-negotiable factor, especially when it comes to edge computing (see Section 2.2), which is why thinking about bare-metal infrastructure is inevitable in IIoT systems. Before taking a look at solutions for bare-metal infrastructure provisioning, we must first look at why bare-metal is such a challenge in the first place. The layering of abstractions is a recurring strategy in the history of computing. However, building abstractions over bare-metal machines is still a central problem that needs to be addressed, since physical machines are the fundamental pieces that underlie everything in the software ecosystem. The need for properly configured and managed hardware that is solid and capable of fulfilling today's requirements regarding load, stress, strain and even external influences like fire, earthquakes or a pandemic is higher than ever. The main issue regarding building a solid abstraction over bare-metal nowadays is the lack of standardization in APIs for provisioning and management. APIs that exist, OpenStack being the most common one, are often extremely complex and impose huge amounts of operational effort [43]. Other than that, maintaining bare-metal infrastructure has several other drawbacks. These include the high effort and complexity, the necessity for dedicated specialists, higher initial cost and lower flexibility and scalability compared to cloud resources. Once the machines are provisioned, the lack of services managed by cloud providers like object storage, virtual networks or databases poses another challenge.

Using bare-metal infrastructure also brings several beneficial factors however. First of all, it makes use cases like edge computing and low-latency applications possible, which are common requirements in IIoT. Owning hardware can also be more cost-effective compared to utilizing cloud provider resources, especially after the initial investment (buying hardware, engineering cost for abstractions, etc.) is amortized. This is especially noticeable in performance efficiency, since the full computational power of the hardware is available, compared to cloud-managed machines which often experience overhead due to virtualization. Furthermore, bare-metal machines offer higher standards when it comes to security,

isolation and compliance as there is more control over both physical and network access. Moreover, while resources are shared among multiple tenants in a cloud system, which can lead to fluctuations in performance ("noisy neighbor problem"), bare-metal machines provide dedicated resources and thus offer more predictable performance. Custom hardware also performs better for workloads that are not fit for virtualized environments like database servers or GPU-heavy workloads for machine learning. Lastly, having open solutions built upon own hardware results in no dependence on a cloud provider thus minimizing the risk of a vendor lock-in [43].

4.2 Provisioning Kubernetes

In Chapter 3 we introduced a reference architecture that revolves around Kubernetes as its central technology. This section will deal with strategies that will allow us to provision Kubernetes on a wide variety of infrastructure ranging from physical hardware in the edge and fog environments up to cloud providers in the cloud environment. To understand this section in its entirety, a basic understanding of Kubernetes is required.

4.2.1 Required Infrastructure

As mentioned earlier, while having own bare-metal machines imposes many benefits, it comes at the cost of having to manage the machines across their full lifecycle. This begins with provisioning the machines, which typically means to perform tasks like

- picking a suitable server from a pool of available servers
- applying specific configuration for hardware (like network interfaces or RAID controllers) and low-level software (like firmware or BIOS)
- loading appropriate software (operating system, drivers, applications) as well as user data into it
- applying necessary updates to minimize vulnerabilities in software
- configuring network stack, monitoring, applications, etc. [43]

In order to achieve this, we once again aim to find a uniform solution that not only works for bare-metal machines but for all environments of the proposed reference architecture from Chapter 3. The solution should also have domain knowledge regarding our central technology Kubernetes and its operational tasks like provisioning and upgrading. Lastly, the solution should work in a declarative manner in order to be compatible with our GitOps setup.

Let us take a look at what we need to provision infrastructure-wise in order to be able to implement the reference architecture. As already mentioned above the basis for any

bare-metal machine is configuring the hardware and installing the necessary software, which at least includes an operating system (OS), ideally one that is optimized for hosting containerized applications (see Section 2.4.1). With the OS installed, the system must now be configured for production readiness regarding the hosting of Kubernetes. This includes network configuration, disabling swap memory, configuring the kernel and the firewall and many more configurations domain-specific to Kubernetes. The system is now ready for the installation of Kubernetes. On top of Kubernetes, a network plugin and a storage plugin must be chosen and installed depending on the infrastructure that K8s runs upon.

This setup is simpler for the cloud environment since most cloud providers offer managed instances of Kubernetes (Azure Kubernetes Service, Amazon EKS, Google Kubernetes Engine, etc.) out of the box. If managed virtual machines are chosen, the steps of the previous paragraph, starting with configuring the OS, apply. Once the environments are all running Kubernetes, a GitOps controller (see Section 2.3) needs to be deployed in each Kubernetes instance and configured to reconcile the according desired state from Git. From then on, all deployments, configurations, etc. can be done through GitOps by adding changes to Git.

Since we seem to encounter different types of scenarios regarding infrastructure we will explore these briefly. The easiest scenario is provisioning managed Kubernetes within a cloud provider, which is a solved problem and can be done with a variety of tools. The next scenario are servers, both bare-metal or virtual in the cloud, that already have an operating system provisioned, which can be handled by using remote connection protocols like SSH or standard IT automation tools. Similar to that are scenarios where a virtualization platform (see Section 4.3) like VMware vSphere is used since these often provide sane APIs to create virtual machines, which brings us back to the scenarios that were already mentioned. The last scenario is having to provision bare-metal machines which is also the most challenging due to the lack of tooling and standardization.

4.2.2 Technical Solutions and Tooling

Let us now explore potential technical solutions (i.e. tooling) that are capable of provisioning and managing infrastructure in all of these scenarios. For such tasks, IT automation tools like Ansible, Chef or Puppet quickly come to mind. With these, it is easy to automate tasks like provisioning Kubernetes onto machines or a managed Kubernetes instance of a cloud provider. In the real bare-metal world, these tools however are often infeasible to use, since they expect machines to already be provisioned with an operating system which often is not the case. To tackle this challenge, we could provision an operating system by network booting the machine using technologies like “PXE Boot” and “IPMI” (see Section 4.2.4) and then use an automation tool like Ansible. However, there are even more shortcomings with such tools. Configuration automation tools mostly operate on a push-based approach. This method lacks continuous reconciliation and monitoring capabilities, which are e.g. essential for automatically provisioning new hardware without manual intervention, for taking action in case of hardware or software failures or in general detecting and handling a drift between

the desired and actual state. They are also not Kubernetes-aware, which makes operational tasks like upgrading Kubernetes more demanding.

Another strategy following the principle of immutable infrastructure, where components are replaced rather than changed or updated in place, would be to install a prebuilt OS image via network boot. This OS image could already contain everything required for running a production-ready instance of Kubernetes (see Section 4.2.1) and would require no further modifications after booting. This method is similar to the previous one, but also lacks continuous reconciliation and an understanding of Kubernetes, making it unviable for large-scale systems. It also introduces a serious security risk, since the OS image would also contain sensitive information like the Kubernetes join token. Consequently, anyone who gains access to this image could potentially join the Kubernetes cluster, posing a substantial vulnerability. Also, the use of a manually started network boot prevents us from fully committing to the GitOps framework.

An ideal solution that fulfills all requirements is the project "ClusterAPI", which we will explore in detail in the following section.

4.2.3 ClusterAPI

Let's consult the official description provided by the developers of ClusterAPI (CAPI) to understand how they characterize the project:

> "Cluster API is a Kubernetes subproject focused on providing declarative APIs and tooling to simplify provisioning, upgrading, and operating multiple Kubernetes clusters. Started by the Kubernetes Special Interest Group (SIG) Cluster Lifecycle, the Cluster API project uses Kubernetes-style APIs and patterns to automate cluster lifecycle management for platform operators. [...] This enables consistent and repeatable cluster deployments across a wide variety of infrastructure environments." – ClusterAPI, GitHub [44]

This description gives us a great overview of the project. It offers a standardized solution for the challenge of managing and operating Kubernetes clusters throughout their lifecycle at scale by abstracting the underlying complexity. Note that the Kubernetes SIG is also the maintainer of well-known projects like "kOps" and even "kubeadm", both of which offer tooling for creating production-ready Kubernetes clusters.

ClusterAPI is a Kubernetes-native project and consists mainly of custom Kubernetes Operators, which are software extensions to Kubernetes that make use of custom resources to manage applications and their components [45]. The operators run instances of so-called custom controllers, which are responsible for the continuous reconciliation of the custom resources like "Cluster" or "Machine". In this process, each controller continuously watches for drift between the desired and the actual state described in these custom resources and takes according actions when a drift is detected. Since these custom objects are nothing but

Kubernetes manifests (i.e. YAML files) we can store them in Git, thus providing a great integration into our GitOps (see Section 2.3) setup. When using ClusterAPI, a separate Kubernetes cluster is used as a "Management Cluster". This cluster is mainly responsible for managing the lifecycle of the so-called "Workload Clusters". It can also host common infrastructure like central observability (see Section 3.5) or a multicluster management solution (see Section 3.6) but must not be used for running domain-specific workloads [42].

Let us now take a look at how ClusterAPI can solve all of the different provisioning scenarios mentioned in Section 4.2.1 that we typically find in the real world. The project achieves this by providing flexible extensibility through a provider system, which in practice behaves similarly to a plugin design. The documentation states, that "Cluster API can be extended to support any infrastructure (AWS, Azure, vSphere, etc.), bootstrap or control plane (kubeadm is built-in) provider" [44].

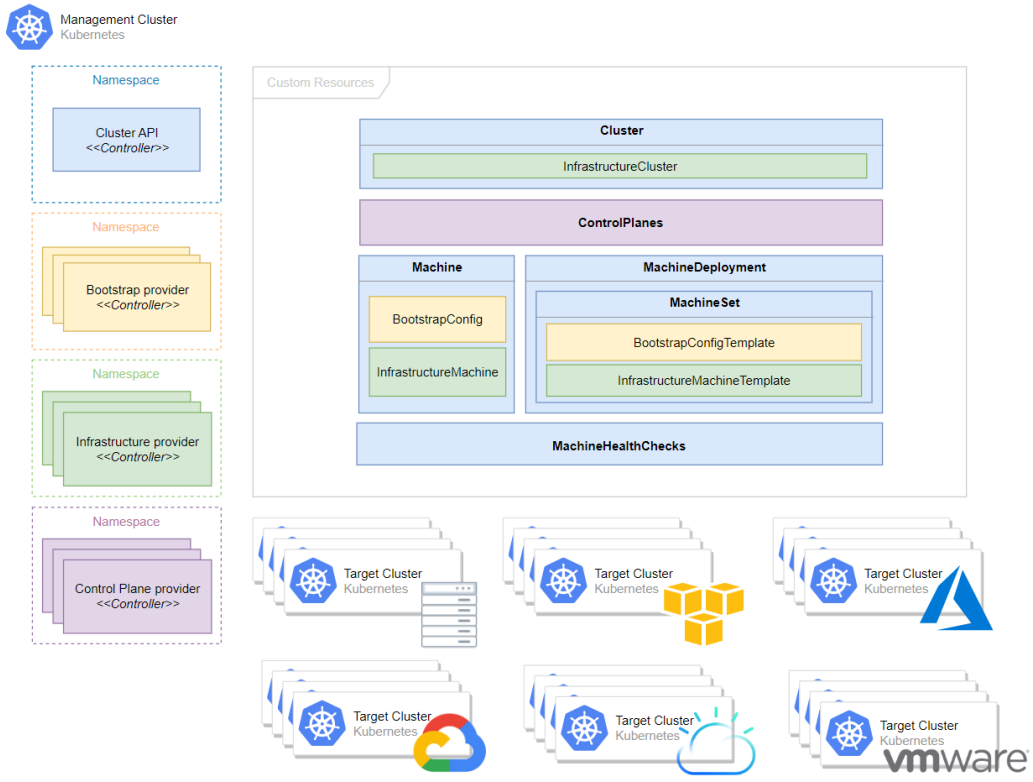


Figure 4.1: ClusterAPI Concepts [9]

Figure 4.1 shows the architecture that makes this high degree of extensibility possible. The top right section shows components and custom resource definitions that ClusterAPI ships by default. These can be seen as the most generic components that basically provide interfaces

for provider-specific components to implement as we will see later.

The most important custom resource definitions for this work are the "Cluster", "ControlPlane" and "Machine"-resources. The "Cluster" resource is a resource that is responsible for grouping all other related resources to managed Kubernetes clusters and is the most abstract of them all. Within the cluster, we can see an "InfrastructureCluster", which in practice is a reference to a specific cluster resource from an "Infrastructure provider". Below that we see the "ControlPlanes" which again is a high-level abstraction over a Kubernetes control plane. In practice, the according "Control Plane provider" specific control plane resource is located within the cluster resource as a reference. Lastly, the "Machine" custom resource definition is a "declarative spec for an infrastructure component hosting a Kubernetes Node" [9]. We observe two instances of provider-specific references in each machine object: one is related to a resource from the "Bootstrap provider", and the other to a resource of the "Infrastructure provider" [46]. All of these providers are explained further in the next paragraph. From the perspective of ClusterAPI, all "Machine" objects are immutable: once they are created, they are never updated, only deleted. For this reason, so-called "MachineDeployments", which are similar to Kubernetes Deployments, are preferable and used in practice rather than using "Machine" objects directly [9]. The other components in this section of Figure 4.1 are not relevant for this work. It is hence up to the reader to explore these independently, according to their interest or need.

Since we saw that the ClusterAPI resources are mostly high-level abstractions that include references to provider-specific implementations, we must now understand what these providers are responsible for. The providers allow developers to provision workload clusters on their desired infrastructure and abstract away all necessary interaction with the actual infrastructure, ranging from on-premises bare-metal systems up to cloud providers [42]. The ability to create infrastructure that can be maintained, destroyed and re-deployed with little human interaction embodies the philosophy of treating servers as "cattle, not pets", ensuring that systems are easily manageable, rather than being treated as irreplaceable individual entities. This is the dream in the environment of physical hardware since it offers interaction with the infrastructure that feels very similar to using a cloud provider [47].

For this interaction, the "Infrastructure provider" holds all responsibility. Depending on the provider it has the capabilities necessary to provision managed resources in the cloud, virtual machines in a virtualization platform or even physical bare-metal machines that are required by other ClusterAPI objects. The "Bootstrap provider" on the other hand is responsible for turning a provisioned machine into a Kubernetes node, which is typically done using cloud-init. This mainly involves creating the necessary configuration to bootstrap the Kubernetes control plane and worker nodes, generating cluster certificates and joining nodes to the cluster. The "Control Plane provider" is very similar since it is responsible for building and reconciling the configuration for the Kubernetes control plane nodes. Through references the provider-specific resources are mapped in a one-to-one relationship where one ClusterAPI resource refers to one specific provider resource and vice versa [9, 46].

Using this flexibility we can now provision Kubernetes clusters in all infrastructure scenarios discovered in Section 4.2.1: cloud providers, virtualization platforms and bare-metal machines both with and without an operating system installed just by choosing the appropriate providers. In Chapter 5 we will provision a managed cluster in the Microsoft Azure cloud and a bare-metal cluster onto a physical machine using such providers. Overall ClusterAPI is a proven solution for managing Kubernetes clusters that fits this use case perfectly. Companies like Mercedes-Benz use ClusterAPI to manage hundreds to thousands of Kubernetes clusters in production which further validates the use of CAPI [48]. Another validating example is "Gardener" which is a common tool for managing the full lifecycle of conformant Kubernetes clusters as a service. The documentation states that the project, while not using it directly, is heavily inspired by ClusterAPI and still follows it with great interest to this day [49]. Similarly, the popular multicluster management platform "Rancher" by SUSE leverages ClusterAPI under the hood for provisioning their Kubernetes distribution RKE2 [50].

4.2.4 Out-of-Band Management and Network-Based Boot

While we have found a tool that is capable of uniformly managing Kubernetes-based infrastructure across all environments in our systems with ClusterAPI, we still have not taken a look at how ClusterAPI provisions and manages physical servers, which is the actual challenge when dealing with bare-metal machines. While the previous section only explained this with the use of a provider/plugin system, this section will deal with the technical details.

Out-of-Band Management

The centerpiece of any proper bare-metal management solution is a microcontroller called baseboard management controller (BMC) within the hardware, which makes it possible to remotely manage and monitor hardware independent of its operating system or status, even when it is powered off. The ability to perform such operations is often referred to as "Integrated Lights out Management" (ILOM) or "Out-of-Band Management" (OoB) [51]. These operational features are exposed for external access via the network by an interface, the most common one being the IPMI (Infrastructure Platform Management Interface). Note that all of this depends on the hardware and is only possible if a BMC is available.

The interface was first released in 1998 and was developed by Intel, Hewlett-Packard, Dell and NEC with the goal of making remote management of servers possible - even when they are powered off or have no operating system installed. Using this interface to access the BMC, operators can for example monitor sensor data of the system and install software, drivers or updates. The most important feature in the context of this work however is the ability to power the machine on or off remotely [52]. This feature allows for remotely powering machines on or off, enabling them to be booted up by ClusterAPI infrastructure providers as needed, thereby preparing the machine for the subsequent steps of provisioning.

Network Boot

Once the system is booting, an operating system needs to be provisioned. Typically, ClusterAPI infrastructure providers designed for bare-metal machines will choose a Kubernetes optimized OS like Flatcar Linux, Fedora CoreOS or Talos Linux as described in Section 2.4.1. In order to achieve full automation during the OS installation, most ClusterAPI providers rely on the strategy of booting the machines via the network. One of the most popular models for this is the Preboot Execution Environment (PXE) which we will briefly discuss in this work.

The client-server model PXE which was originally developed by Intel is an open industry standard that enables network-capable machines, which are referred to as PXE clients, to boot via a server in the local network. The process requires a machine with a PXE-enabled network interface card (i.e. the PXE client), a custom configuration in the network's DHCP server and a TFTP server.

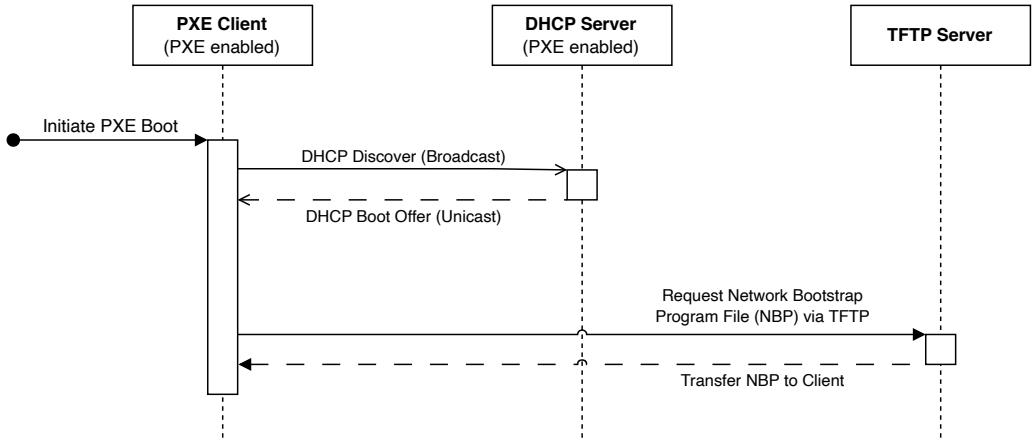


Figure 4.2: PXE Boot Process

Figure 4.2 shows the steps of a network boot using PXE. The PXE-Client initiates the process by sending a standard DHCP discover packet with PXE-specific boot parameters and system information set as a broadcast into the local network. A DHCP-Server that supports PXE and is configured properly will reply to the network boot request broadcast with the necessary information to perform a network boot, whereas other DHCP-Servers will simply ignore the flag and continue with the standard four-way DHCP handshake. This implies that setting up additional DHCP servers in the network, specifically dedicated to handling boot requests, is practical. These additional servers are commonly referred to as "Proxy DHCP-Servers". This approach allows for the separation of boot request management from IP address allocation or just using PXE even though the pre-existing DHCP server does

not support it. If the PXE/DHCP server is in a separate network, which causes DHCP broadcasts to not reach their intended target destination, many routers offer a functionality in the form of a router rule called "ip helper", which configures the router to forward these packets to their according target in the according network.

The response (boot offer) from the PXE-enabled DHCP server contains two necessary DHCP options that have to be configured in the DHCP server in advance:

- Option 66 (TFTP Server Name): Hostname or IP address of the TFTP server
- Option 67 (Boot File Name): Path of the boot file on the TFTP server

The last step for the PXE client is to reach out to the TFTP server (DHCP Option 66) and download the boot file (DHCP Option 67) from it into memory via the TFTP protocol. This boot file is also referred to as the network bootstrap program (NBP) and has the task of preparing the client for running the specified operating system. The client boots using this NBP, which typically leads to the download and finally the launch of the actual operating system [53]. Similar to what was described regarding out-of-bands management in the previous Section 4.2.4, ClusterAPI infrastructure providers specialized towards bare-metal infrastructure will typically either ship their own or have integrations with external DHCP and TFTP servers. Once the operating system is running, ClusterAPI can continue to provision Kubernetes as described in Section 4.2.3. Note that this was a concise overview of the PXE boot process. For more detailed information, readers are encouraged to further explore the topic.

Shortcomings of PXE and IPMI

While PXE and IPMI are very common solutions, they are both old and no longer up-to-date to today's standards and will be replaced by more modern successors like HTTP Boot for PXE and Redfish for IPMI eventually.

PXE has a major issue regarding security since it offers no mechanism for encryption or authentication natively, which makes the standard vulnerable to attacks like a man-in-the-middle attack using a rogue DHCP server that hands out malicious boot information. Since no authentication or encryption is in place, this can hardly be prevented. Another drawback of PXE is the poor scalability due to TFTP timeouts and UDP packet loss (DHCP uses UDP). With a larger scale, these issues will occur more and more often resulting in an unreliable network boot setup. Common solutions include booting into an intermediate and more featureful network boot firmware like "iPXE" that from then on tackles these limitations and adds functionality like booting via HTTP or even via a wireless network whereas only LAN boots via a TFTP server are possible with plain PXE. This process is often referred to as "chainloading". This however imposes the use of yet another piece of software that brings complexity. The more modern approach is using HTTP Boot which acts as a replacement

for PXE and tries to address these issues. HTTP Boot either uses a preconfigured or a, via DHCP, auto-discovered URL to either boot an NBP or an ISO image directly. The main difference however is its reliance on HTTP. This solves the issues of PXE regarding security by using the encrypted HTTPS protocol and the issues regarding scalability using HTTP load balancing and the more reliable TCP protocol as a layer 4 protocol base in favor of UDP [54].

Similarly to PXE, the interface for out-of-band management IPMI also has shortcomings regarding security. Not only does it lack modern security best practices, but there are also known vulnerabilities in the IPMI specification like the "IPMI 2.0 RAKP Authentication Remote Password Hash Retrieval" that allows remote attackers to obtain password hashes and perform offline password guessing attacks. Due to its age, IPMI struggles with modern architectures like multi-node server clusters. It is also not UEFI-aware, which means that it cannot interface directly with modern UEFI settings like boot order or secure boot. Lastly, IPMI suffers from limitations in terms of scalability, since it was mainly designed for managing individual servers rather than a large-scale distributed environment consisting of hundreds if not thousands of servers. The modern "Redfish" specification aims to address these flaws by providing a rich RESTful interface over HTTPS with scalability and security built-in by design [54].

Note that while today it would be beneficial to choose the modern stack with Redfish and HTTP Boot, this choice depends heavily on the hardware. If the hardware in question only supports PXE and IPMI, then only these can be used. When using ClusterAPI, an infrastructure provider must be chosen that fits the hardware that should be provisioned.

4.3 Virtualization

We discovered in Section 4.2.1 that a common scenario when provisioning infrastructure on own hardware in IIoT systems is the provisioning of virtual machines in a virtualization platform. In this section, we want to understand when this makes sense and how it is done in practice.

> "Virtual machines have finally arrived. Dismissed for a number of years as merely academic curiosities, they are now seen as cost-effective techniques for organizing computer systems resources to provide extraordinary system flexibility and support for certain unique applications." – Robert P. Goldberg, 1974 [55]

Virtual machines have been seen as a great opportunity from very early on. By hosting multiple virtual machines on a single physical machine, the hardware can often be used in a more efficient and flexible way. At the core of virtual machine technology is the hypervisor, often referred to as the virtual machine monitor (VMM). It is a software layer responsible for creating and running virtual machines, serving as an interface between the hardware and

the virtual environments. If an operating system is installed on the machine, the hypervisor interfaces between the OS and the virtual machines. The VMM manages and virtualizes the hardware in a way that is transparent for the virtual machines. It is not differentiated between physical and virtual hardware, which means that the hardware interface provided to the operating system of the virtual machines is indistinguishable from a bare-metal server from the VM's perspective. The VMM also fully isolates virtual machines from one another in terms of resources and security, i.e. regarding their processing power, memory, storage or network traffic. While virtualization used to cause a loss of performance, the technology is optimized well today and hardly causes any overhead, mainly due to hardware support for virtualization like custom CPU instructions [55].

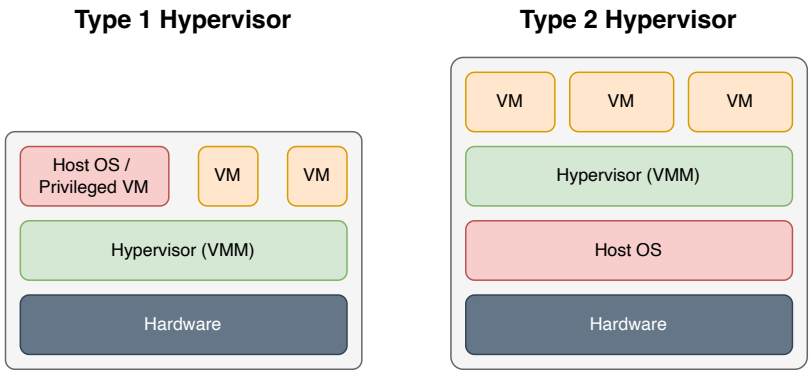


Figure 4.3: Type 1 and Type 2 Hypervisors

Typically a distinction is made between two types of hypervisors as shown in Figure 4.3. Type 1 hypervisors, also known as bare-metal hypervisors, lay directly on the hardware platform and create and run virtual machines directly on top of the hardware. This has the main benefit of offering high performance for VMs and is usually used for server systems where a high number of virtual servers is expected. In such systems, the VMs are typically centrally and remotely managed through APIs for efficient scaling capabilities. Since the hypervisor cannot rely on an underlying operating system, it has to tackle issues like managing drivers and file systems. In practice, this is often done by a trusted privileged VM that abstracts all of these challenges away for the other virtual machines. Compared to this, type 2 hypervisors which are also referred to as hosted hypervisors run on top of an operating system. This simplifies the responsibilities of the hypervisor, since the operating system already takes over the abstraction from the hardware and already has required system components like drivers. This approach however comes with the drawback of lower performance compared to type 1 hypervisors and is hence rather used for client systems, where the amount of VMs is very limited and simple UI-based management is usually enough.

When a virtualization technology is used for bare-metal machines in an IIoT platform the type 1 hypervisor should be chosen due to its high performance and API-based management,

both of which allow for large-scale [55].

Now that we understand how virtualization works under the hood and how it should be used in an IIoT system, let us discuss why one would choose virtualization over plain bare-metal machines for provisioning (see Section 4.2). While using the hardware directly offers the best performance, using bare-metal machines still has certain drawbacks compared to a virtualized environment. Implementing multitenancy with robust isolation on bare-metal systems is challenging in contrast to simply using virtual machines, as it requires complex and often custom configurations to ensure both security and resource partitioning. This poses a problem when a single machine should be used by multiple parties that might affect each other, which could cause interference in the manufacturing/production site of an IIoT system (noisy neighbor problem). As already discovered in Section 4.2 the biggest challenge however remains the lack of standardization when it comes to APIs for managing hardware directly. While there exist solutions (see Section 4.2.4) they are often based on outdated tools, dependent on the often very heterogeneous hardware, and are not robust due to many possible failures.

In contrast to this stands a hypervisor-based virtualized infrastructure, where each virtual machine is fully isolated from one another thus allowing for simple multipurpose usage of hardware. An example where this might be used is the deployment of a workload that cannot be containerized on a bare-metal server, which is already running Kubernetes. Using virtualization, this can easily be achieved by creating a separate isolated VM. Also, the fact that each virtual server is software-based offers enhanced infrastructure flexibility. Each server can be relocated to a different physical server effortlessly, created or destroyed on demand ("cattle not pets"), and vertically scaled by allocating more or fewer resources, among other features that expand the infrastructure's adaptability. The largest benefit however is the simple management of virtual servers through APIs. Commonly used virtualization platforms like VMware vSphere or ProxmoxVE offer rich APIs that feel similar to managing virtual machines within a cloud provider. These APIs can be used by ClusterAPI infrastructure providers directly (see Section 4.2.3) which is not only simpler but way more robust compared to the out-of-band management of bare-metal servers. Many multicluster management platforms (see Section 3.6) like Rancher or Gardener that are capable of deploying Kubernetes onto a variety of infrastructure can also provision Kubernetes onto virtualization platforms directly. This way, a multicluster management solution can be used directly rather than building your own setup with ClusterAPI thus again reducing the system's complexity.

The use of virtualization on top of own hardware not only has benefits however. While with modern hardware, virtualization does not introduce much overhead anymore, the physical resources are still used a bit less efficiently due to the overhead of virtualization. Also, running a virtualization platform introduces more components and thus complexity that requires domain experts into the system. Lastly, production-ready virtualization platforms like VMware vSphere introduce additional cost, that needs to be accounted for [56]. If a virtualization infrastructure already exists in the target environment however, it will simplify

and robustify the provisioning mechanism immensely.

Overall, virtualization can significantly simplify the provisioning of bare-metal infrastructure for the edge and fog environments of the reference architecture described in Chapter 3. Instructing ClusterAPI to interact with a virtualization platform can also improve the very crucial robustness of the infrastructure provisioning mechanism compared to using outdated out-of-band management solutions to provision hardware directly. Since drawbacks, i.e. cost, less efficient use of physical resources and additional complexity, exist as well, the decision whether to employ virtualization needs to be made dependent on the requirements of the project in question however.

5 Proof of Concept

This last chapter will deal with a proof-of-concept (PoC) project that was realized during this work. We will fill the abstract components of the reference architecture from Chapter 3 with concrete technologies and apply them to a setting that mimics a real-world scenario, while also exploring and comparing which technologies might be best suited for different use cases. In the project, we focused on the provisioning of infrastructure, since it is the most challenging part of realizing such an IIoT system. The resources and the cloud subscription used in this proof-of-concept were provided by MaibornWolff GmbH.

5.1 Provisioning the Infrastructure

In this PoC we wanted to set up each environment, i.e. edge, fog and cloud, that was introduced in the reference architecture. For the edge environment, a *Supermicro SMC X10 A1SAi-2750F* server, which provides an IPMI and supports PXE boot (see Section 4.2.4), was used, particularly to demonstrate how bare-metal machines can be provisioned. In this project, we have decided against virtualization (see Section 4.3) and sticked to deploying directly to the bare-metal machines. The cloud environment was set up in the Microsoft Azure cloud, mainly revolving around their managed Kubernetes cluster "Azure Kubernetes Service (AKS)". Lastly, the fog environment was created on a pre-provisioned Ubuntu machine, which was already bootstrapped with the Kubernetes distribution "Rancher K3s" and a running instance of ClusterAPI. As discussed in Section 4.2.3, we used ClusterAPI to provision and manage the other Kubernetes instances in combination with infrastructure providers specific to the according infrastructure they should provision to.

For the cloud environment, we installed the official ClusterAPI Provider Azure (CAPZ) [57] as the infrastructure provider into our management cluster which supports managing Kubernetes clusters in the Microsoft Azure cloud. Using custom resource definitions in Kubernetes, we could provision and operate a workload Kubernetes cluster fully managed by the cloud provider by simply applying YAML-manifests into our management cluster. Using the ClusterAPI command line interface "clusterctl" we were able to generate these manifests, and only needed to adjust domain-specific configuration like the name of the cluster or the number of cluster nodes.

The more interesting environments in this PoC were those that required provisioning directly onto bare-metal infrastructure. The hardware used in this project supported PXE

and IPMI, so using an according ClusterAPI infrastructure provider had to be chosen. For this purpose, we evaluated the following options:

- Canonical MAAS | Metal as a Service
- Tinkerbell
- Metal^3 (pronounced "metal kubed")
- Sidero Metal

Canonical MAAS is a tool that provides an abstraction over bare-metal provisioning and offers a rich API to the outside, which can be consumed by the "Cluster API Provider for MAAS". In practice, a dedicated machine hosts the MAAS instance and is responsible for the out-of-band management (see Section 4.2.4) of the target infrastructure. It achieves this by mainly building upon DHCP, IPMI, PXE and TFTP and offers a nice abstraction layer over these low-level protocols. MAAS also ships a user interface that helps in managing large-scale bare-metal clouds, custom OS images, disks and network configuration, authentication, RBAC configuration and many more useful features. The main drawback of using MAAS for provisioning the bare-metal infrastructure in this work however is that it necessitates the provisioning and operation of an additional machine specifically to host MAAS itself. This requirement adds complexity and overhead since it involves not only setting up another server but also ensuring its reliable operation and maintenance, which includes configuration and ongoing management tasks. Since our idea was to use Kubernetes as a uniform technology across all environments in order to only have to manage one technology, we decided not to use MAAS in this PoC and kept looking for a Kubernetes-native solution [58].

A very promising project is Metal^3, which provides a set of tools for managing bare-metal infrastructure using Kubernetes. It consists of three components, that make the provisioning of bare-metal machines possible: the "ClusterAPI Provider Metal^3", the so-called "Bare Metal Operator" and the external tool "OpenStack Ironic". The ClusterAPI provider serves as the infrastructure provider and enables the creation and management of physical servers using custom resource definitions in Kubernetes. The Bare Metal Operator works in tandem with the ClusterAPI provider and is responsible for provisioning and managing the lifecycle of physical servers. It interacts with the underlying bare-metal management tool Ironic commonly known from the OpenStack project to perform tasks like provisioning, deprovisioning, and updating the hardware. The Bare Metal Operator receives instructions from the ClusterAPI provider and acts upon these instructions to manage the physical hardware through Ironic. The robust tool Ironic again relies on technologies like IPMI and Redfish to manage the bare-metal machines. The Metal^3 project appears to be one of the most robust and reliable projects for managing bare-metal infrastructure from Kubernetes. Especially through its reliance on the battle-tested OpenStack component "Ironic" the project is capable of deploying to a broad variety of infrastructure and is a solid choice for large

projects. The downside of Metal^3 however is, that it adds much complexity. Developers have to maintain their own instance of OpenStack Ironic and the Metal^3 Bare Metal Operator, which are both large and complex projects with setups that are far from trivial. It has to be evaluated on a project basis, whether the benefit of employing one of the most reliable projects for bare-metal management outweighs the downside of introducing high complexity into the system [59, 60]. Due to the simplicity of the PoC setup, we chose against adding this level of complexity into our system.

Another Kubernetes-native tool for provisioning and managing bare-metal infrastructure is "Tinkerbell". Apart from being a microservice-driven application, Tinkerbell operates similarly to Metal^3 with the difference of not relying on external tools but shipping its own, e.g. "Boots" as a DHCP server or "PBnJ" for out-of-bands management via the BMC (see Section 4.2.4) of bare-metal machines. However, since at the time of writing this work the ClusterAPI Provider Tinkerbell was still not ready for production use and only supported Kubernetes versions up to v1.22, which was marked as end-of-life for over 15 months now, we decided against the use of Tinkerbell for this PoC project [61].

Finally, Sidero Metal was the last tool we evaluated. Similar to Metal^3 and Tinkerbell, it allows for Kubernetes-native bare-metal management using ClusterAPI. The project includes a metadata service, PXE and TFTP servers, as well as BMC and IPMI management for automation, which all come in handy in practice. Sidero Metal not only consists of a ClusterAPI infrastructure provider but also a bootstrap provider and a control plane provider. This means that Sidero Metal is not only managing the infrastructure itself but also provides an end-to-end ecosystem for deploying Kubernetes. Sidero Metal chooses "Talos Linux" as a secure, immutable, and minimal base operating system for Kubernetes nodes, which ensures a secure and robust basis for running Kubernetes in production. Overall the project shines through its simplicity and a feature set, that does exactly what was required in this PoC, which is the reason why Sidero Metal was chosen as the ClusterAPI provider for bare-metal management in this project [62, 63].

Generally, if a bare-metal management solution based on ClusterAPI is required, we can recommend using Canonical MAAS or if a Kubernetes-native solution is preferred Sidero Metal for smaller projects. In large-scale and productive systems, where reliability and robustness are key requirements, Metal^3 is a solid choice but requires more domain knowledge and setup.

Figure 5.1 illustrates how the provisioning of infrastructure was set up in the project. The management cluster (center), which in this setup was placed in the fog environment, provisioned a managed "Azure Kubernetes Service" (AKS) in the Microsoft Azure cloud using the ClusterAPI Provider Azure, which is all there was to the infrastructure provisioning in the cloud environment. Within the local network section, which resembled the on-premises setup in a production site, the Sidero Metal ClusterAPI provider started to provision an edge

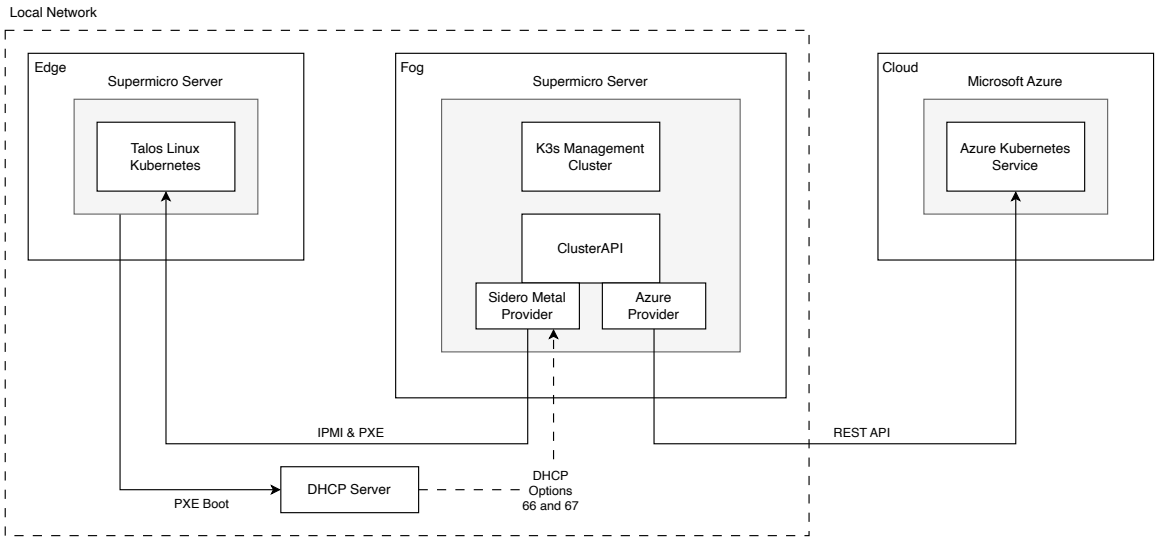


Figure 5.1: Infrastructure Provisioning with ClusterAPI

device, which in practice was a Supermicro bare-metal server, by instructing it to boot via the IPMI. During the boot, the Supermicro server started to initiate the PXE boot process (see Section 4.2.4) during which a broadcast request was sent to the local network and finally received by the DHCP server. The DHCP server had the DHCP options 66 (IP address) and 67 (boot file name) pointed at the Sidero Metal ClusterAPI provider component that held the PXE-relevant files and answered with a boot offer. With these options, the edge device could boot into the operating system and could then finally be provisioned with Kubernetes. We now had a system that was capable of provisioning Kubernetes instances both in cloud providers and onto bare-metal machines.

5.2 GitOps Setup

Since we discovered how GitOps is essential for a modern IIoT system of scale when following the reference architecture described in this work in Section 3.4, we needed to embed the framework into this multicluster setup. The most common GitOps controllers in the market are FluxCD and ArgoCD and are both fully capable of fulfilling the requirements for this system. In our proof-of-concept setup, we chose FluxCD due to its lightweight and straightforward setup, along with its seamless integration with Mozilla SOPS for secret management (see Section 5.3.1).

To be able to sustain the GitOps setup at scale we started by setting up the structure of the Git repository for the GitOps-relevant folders. Inspired by the open-source project “Das Schiff” by the “Deutsche Telekom Technik”, which is a GitOps-based Kubernetes Cluster-

as-a-Service platform, we created the directory "cluster-definitions", which held all GitOps-related configurations like which paths from which repository to reconcile, and the directory "cluster-components" which contained the components that should be deployed onto the clusters [64]. Both of these folders contained one subfolder per cluster in our setup, but any kind of folder structure would have been possible here. In the cluster-components directory, we also added a "shared" folder that held manifests that could be reused by different clusters like the deployment of a dashboard for the FluxCD GitOps controllers.

Once the structure was set up, we continued by setting up FluxCD on the management cluster. To begin with, we created a Kubernetes secret containing a GitHub token, that enabled FluxCD to access our private GitHub repository, and stored it in Git as an encrypted file using the secret management tool Mozilla SOPS (see Section 5.3.1). As a one-time manual step, we now had to create this secret within the Kubernetes cluster and then install FluxCD manually in order to reach the now fully automated state. Both the secret and the FluxCD installation on the management cluster were now managed by Flux itself through manifests stored in Git. The GitOps controller on the management cluster was now configured to continuously reconcile the specified resources from Git. These resources also contained the ClusterAPI manifests, which then triggered the provisioning of the workload clusters in all environments (see Section 4.2.3). Once the clusters were provisioned, FluxCD was pushed onto all workload clusters automatically using the FluxCD-specific custom resource "HelmChartProxy" which provides the option to install and manage a helm release in a different Kubernetes cluster. An alternative would have been to use the custom resource "HelmRelease" which allows to create and manage a helm release and also provides the option to specify a Kubeconfig that determines the target Kubernetes cluster for the deployment of the helm release. This however would have resulted in duplication, since the HelmRelease custom resource has to be created for each workload cluster separately, compared to the HelmChartProxy custom resource that only had to be created once. From then on the GitOps controllers running on the workload clusters continuously reconciled the desired state for the according environment from Git. Together with the secret distribution of secrets like the GitHub token described in the upcoming Section 5.3.2, this resulted in a fully automated infrastructure that resembled the desired infrastructure state of the reference architecture in Chapter 3.

5.3 Secret Management and Distribution

Proper secret management techniques are required in practically every software project, and IIoT platform projects are no different. For this project, we needed to choose a strategy that fits nicely into both the GitOps-driven continuous delivery approach and the multicluster setup due to the high amount of environments.

5.3.1 Management Tools

To begin with, proper tooling for managing secrets in the project had to be chosen. The following tools are some that allow to declaratively and securely store secrets in an encrypted manner in Git thus allowing to follow the GitOps approach:

- Sealed Secrets
- External Secrets Operator
- Secrets Store CSI Driver
- Mozilla Secrets OPerationS (SOPS)

The tools "Sealed Secrets", "External Secrets Operator" and "Secrets Store CSI Driver" all operate similarly. All of them require the installation of the application and custom resource definitions in the Kubernetes cluster. Instead of then storing the Kubernetes-manifests for secrets in Git directly, developers will create and store instances of the according custom resource definitions. With Sealed Secrets, a developer will encrypt a secret manifest into a "SealedSecret" custom resource using a command line interface that interacts with a custom controller running in Kubernetes, which is then safe to store in Git. The custom controller running in the Kubernetes cluster will from then on be the only entity that can decrypt the SealedSecret and will create a Kubernetes secret from it [65]. When using the External Secrets Operator, developers will create manifests that reference secrets in external sources rather than storing encrypted secrets. The custom controller running in the target Kubernetes cluster will fetch the secrets from external sources like AWS Secrets Manager, HashiCorp Vault, Google Secrets Manager, Azure Key Vault and many more and synchronize them into Kubernetes secrets again [66]. Similar to that, the Secrets Store CSI Driver references secrets from external sources. While also being capable of syncing the external secrets to Kubernetes secrets, its main intent is to directly mount secrets from an external source into the file system of Kubernetes pods as volumes. This method can improve security by directly mounting secrets as volumes instead of storing them in etcd. Since secrets in Kubernetes are merely encoded using base64 and thus remain readable to anybody with access. In contrast with mounting the secret directly as a volume, the exposure of sensitive data and its potential attack surface is minimized [67].

Being able to remain operational during network disruptions is essential for IIoT systems, especially in on-premises environments in manufacturing sites. This is why we ruled out the External Secrets Operator and the Secrets Store CSI Driver, since they both mainly rely on external sources that require internet access. While integrations to self-hosted secret management systems like HashiCorp Vault, which would enable the tools for offline use, exist, we did not have such a system in use. The final decision between using Sealed Secrets and Mozilla SOPS mainly revolved around flexibility. While Sealed Secrets is easy to use and integrates seamlessly into Kubernetes, it is limited to secrets in Kubernetes. Mozilla SOPS

on the other hand is a tool that only encrypts and decrypts files using a variety of encryption mechanisms like Azure Key Vault, age or PGP making them safe to store in Git. Given its versatility beyond Kubernetes, SOPS can serve as a unified secret management tool for all project-related secrets, accommodating not only Kubernetes deployments but also those used directly in scripts and other non-Kubernetes applications [68]. Mozilla SOPS also provides simple integrations with common GitOps controllers like ArgoCD and Flux, as already mentioned in Section 5.2. While we chose to use Mozilla SOPS in this project, a solution for different projects must be chosen based on the requirements of the project in question.

5.3.2 Multicluster Strategy

With SOPS as the secret management solution in place, we needed a strategy for managing and distributing secrets in all environments across the IIoT system. Since such a system, especially at scale, involves many different parties or even tenants, it is essential that access to confidential secrets is scoped according to the principle of least privilege which might for example mean that each team working on the project is only able to decrypt its own secrets. In this PoC project, we decided that each instance of each environment should have a designated encryption key to demonstrate this. When creating a new environment, in particular the Kubernetes cluster, we created a new GPG key. While this was only required a single time per new environment instance, it was still manual work that might become tedious at scale. How this can be automated will be discussed in the further work in Section 6.1. Using this key, developers could now encrypt (and decrypt) secrets relevant to this environment and store them in a Git repository safely, without others with access to the repository being able to decrypt it again. The platform team that managed the whole IIoT system had its own SOPS key, which in practice was a key in the Azure Key Vault. Using this key, all GPG keys of the teams were encrypted again, in order to be able to safely store these in Git as well.

Using a "ClusterResourceSet", which is a custom resource of the ClusterAPI project (see Section 4.2.3) that allows to apply a set of resources defined by users to the workload clusters, the GPG keys were distributed to the according environments as Kubernetes secrets. An alternative to the ClusterResourceSet, which is still marked as an alpha version, would have been using the feature set of the GitOps controller. Both FluxCD and ArgoCD support specifying a custom Kubeconfig ("destination" in ArgoCD) for reconciling resources, thus enabling the platform team to push the GPG key secrets from the management cluster to the workload clusters. In this project, we used the approach with the ClusterResourceSet custom resource, since at the time of setting up the PoC we were not aware of the option to use the GitOps controller directly.

Once the GPG keys were created as secrets in the environments, teams could now retrieve their GPG keys for encrypting and decrypting their secrets. Also, the GitOps controllers running in each environment could use this GPG key for decrypting secrets relevant to that environment. In this setup, each environment had a folder called "secrets" in its respec-

tive cluster-components folder (see Section 5.2). The GitOps controller was then instructed to reconcile this folder by decoding the manifests using the environment-specific GPG key which was now present as a Kubernetes secret. This setup was not only useful for the distribution of GPG keys or the use of environment-specific secrets however. By using the option to push secrets from the central management cluster to practically all workload clusters, we could easily distribute secrets required in all environments like the credentials to our observability components as further discussed in Section 3.5, which simplified operational tasks regarding secrets and credentials immensely.

This approach worked nicely in practice and offered a flexible, scalable and secure strategy for managing secrets in a multicluster setup. For real projects, using the GitOps controller for distributing secrets seems like the better and more robust choice compared to using the ClusterResourceSet custom resource from ClusterAPI which is still in alpha. Since SOPS allows to encrypt a file with multiple keys, it also proved to be useful to add the key of the platform team to critical secrets of teams to be able to support them with their secret management during development.

5.4 Observability

As we discovered the importance of having proper observability across a software system in Section 3.5, we needed to set up tooling for metrics and logs within the system.

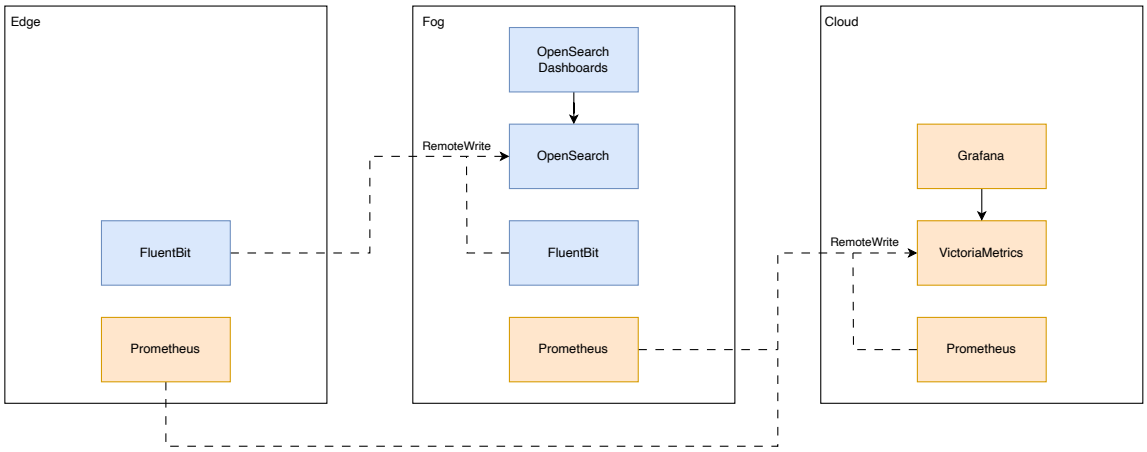


Figure 5.2: Multicluster Observability

To record metrics, we started by deploying a Prometheus instance onto each cluster in the system. Prometheus is an open-source database optimized for time series data that supports scraping and storing metrics from various sources. These Prometheus instances then

gathered metrics for the according environment and acted as buffers by storing them locally. These buffers reduced the risk of losing metrics due to downtimes or network disruptions. The reference architecture suggests having a centralized cloud-based metric store across the whole IIoT system, which is why we deployed an instance of VictoriaMetrics in the cloud environment. VictoriaMetrics is an open-source project that offers similar functionality and identical APIs to Prometheus but solves common problems of Prometheus like the struggle with long-term storage and the lack of support for remote writes from remote Prometheus instances [69]. Suitable alternatives include Grafana Mimir, Thanos or InfluxDB. To accumulate metrics across the whole system, each Prometheus "buffer" instance was configured to remotely write its data into the VictoriaMetrics instance in the cloud for long-term storage as Figure 5.2 illustrates with the dotted lines between the orange components. The open-source analytics and monitoring solution Grafana was used to visualize the metrics across the whole system in one centralized location.

For logging, the reference architecture recommends only aggregating logs per production site due to the volume of the data. To achieve this, the open-source search engine "OpenSearch" was deployed once per production site. OpenSearch is scalable and supports high availability, making it a well-suited storage solution for centralized logging. Alternatives encompass ElasticSearch, Grafana Loki or Apache Solr. For shipping logs from all servers in the production site into the search engine, the log shipper "FluentBit" was used. It was deployed on each Kubernetes node using a Kubernetes DaemonSet and ingested the logs from the according nodes into OpenSearch for persistent storage, as the dotted lines between the blue components in Figure 5.2 display. FluentBit also enriched the logs by adding metadata like the name of the pod or even the cluster in which the logs occurred to the logs, to increase their value even further. Lastly the open-source tool "OpenSearch Dashboards" was used as a graphical user interface for visualizing and querying the log data stored in OpenSearch.

5.5 Multicluster Management

In Section 3.6 we described how a multicluster management solution can be beneficial. While we already decided to use ClusterAPI for the provisioning of clusters and their operation, tools like the Enterprise Kubernetes Management Platform "Rancher" or the enterprise product "D2iQ Kubernetes Management Platform - DKP" could still add valuable functionality like centralized management of role-based access across all environments or access to all clusters from a single location through network tunnels. In this PoC we deployed the Rancher platform by simply adding a "HelmRelease" using FluxCD to the management cluster. The provided functionality worked nicely and is essential to have once the system grows beyond a few environments.

However, we soon discovered that Rancher did not yet support a fully declarative GitOps setup at the time of writing this work. Onboarding a workload cluster into Rancher man-

agement could be initiated by applying a custom resource "Cluster" into the management cluster. Once the custom object was applied, the "Rancher Agent", which connects to the Rancher platform, had to be applied to the workload cluster. For this, Rancher provided the necessary deployments as YAML manifests via an endpoint, but only after having registered the new cluster. While these manifests could again be stored in Git in the according folders of the workload cluster, this manual step of fetching the dynamically generated manifests broke the GitOps workflow. Also, if an already onboarded workload cluster would ever get recreated from scratch this whole onboarding process would have to be repeated. Since the multicluster management solution still provided valuable features, this manual work was accepted for this PoC project however. A strategy on how to deal with this issue in a real project is described in Section 6.1.

5.6 Unified Namespace Implementation

At this point, the necessary platform infrastructure was in place for the actual IIoT infrastructure services to be deployed. Since this work focuses on the infrastructure, the last thing to do was to set up the unified namespace described in Section 3.3.

As a first step, we deployed the MQTT broker in the cloud environment. While any broker fully compatible with the MQTT 5 standard could have been chosen here, we decided to use the HiveMQ MQTT broker. For deploying a production-grade broker in the cloud on Kubernetes, we used the official HiveMQ Kubernetes Operator that helps orchestrate and manage the lifecycle of a highly available HiveMQ cluster. Not only does the operator deploy HiveMQ, but it also assists in monitoring, maintaining, recovering, and upgrading HiveMQ during the operation [70]. All of this was achieved by simply adding manifests, mainly using the FluxCD's "HelmRelease" custom resource, to the corresponding directories in Git.

Moving forward we needed to deploy an MQTT broker in each production site, i.e. the Kubernetes cluster in each fog environment. For this, we used the recently released HiveMQ Edge broker. Not only is this broker optimized for edge environments due to lower resource requirements, but it also helps to set up a unified namespace by providing plug-and-play integrations with common OT protocols like Modbus and OPC-UA. This means that we no longer needed to deploy custom adapter services for these integrations [71]. This broker was also simply deployed by adding manifests to Git and letting the GitOps controller reconcile it eventually. Through custom configuration, the HiveMQ Edge broker was instructed to set up a bidirectional MQTT bridge between the on-premises and cloud brokers by publishing all messages on all topics to and subscribing to all relevant topics from the cloud broker. As a last step, we deployed several NodeRED instances in the edge environment. NodeRED is a browser-based low-code programming tool that is often used by developers to quickly set up IoT applications. In our case, we used NodeRED to simulate edge devices that publish data into the unified namespace via the on-premises MQTT broker. An example topic used for this was "enterprise-1/site-1/area-1/line-1/cell-1" (see Section 3.3). Note that in this setup security concerns like proper IP whitelisting and RBAC were not dealt with since this

project was only meant to be a proof-of-concept.

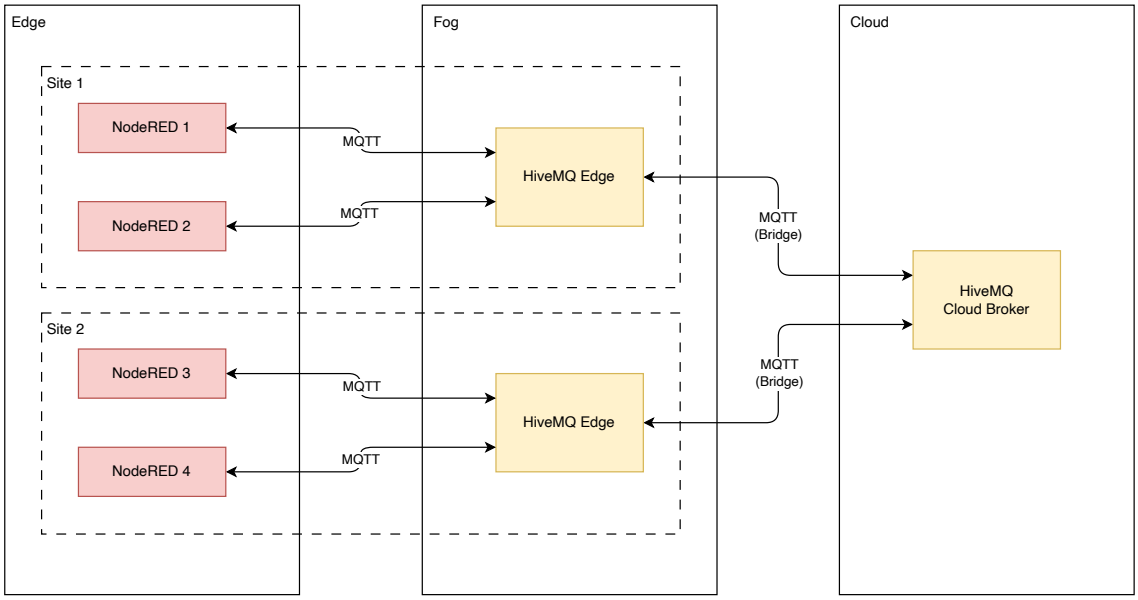


Figure 5.3: Unified Namespace Implementation

Figure 5.3 illustrates how this unified namespace setup could now be scaled to any number of edge devices and production sites. Each IIoT component, in this case, the NodeRED instance, in the edge environments could bidirectionally communicate with the on-premises HiveMQ Edge broker. Workloads in the edge or fog environment were able to access the on-site data with low latency by subscribing to the edge broker directly. Use cases like long-term storage or stream analytics that require high amounts of computational power or storage could be realized in the cloud environment.

5.7 Implementation Review

The implementation of the reference architecture in this proof-of-concept resulted in an extensible, scalable, flexible and robust system. Deploying, decommissioning, or moving workloads around the whole system was simple and the onboarding of (simulated) IIoT devices into the unified namespace in an on-premises environment was a smooth process.

Using ClusterAPI to provision Kubernetes instances onto various kinds of infrastructure proved to be a solid and powerful choice. The bare-metal infrastructure provider Sidero Metal was easy to work with and made provisioning onto plain hardware an easy task. It is possible that using the CNCF project Metal[3] in favor of Sidero Metal would be beneficial for real-world projects since it uses the battle-tested OpenStack Ironic for hardware management and thus is capable of managing a wider variety of infrastructure in a more robust

manner. However since Sidero Metal supports both PXE and IPMI which was required for the hardware in use, it was a reasonable choice for this proof-of-concept. It has to be mentioned that using out-of-band management techniques and network boot for provisioning hardware is a strategy that can fail at many points, and can be simplified immensely by employing virtualization e.g. using VMware vSphere which offers stable and reliable APIs for managing virtual machines at the cost of a small loss in performance (see Section 4.3).

The use of the GitOps framework also showed great results. Having a continuous versioning mechanism in the single source of truth for the whole infrastructure was very useful, especially during development. Also having the option of applying a Git workflow with pull requests and code reviews felt like a great basis for a large-scale project. The GitOps controller "FluxCD" worked nicely and did not show any unexpected behavior. Together with the observability stack, which provided comprehensive monitoring and logging capabilities across all components of the IIoT platform, the integration of FluxCD within the GitOps framework greatly enhanced our overall infrastructure management.

Lastly, the multicluster management solution Rancher proved to be a useful tool. Even though it was not used for provisioning clusters in this setup, the remaining feature set was a meaningful addition to the management capabilities across such a large-scale system. As Rancher does not yet offer full support for the GitOps model, it is necessary to assess each project individually to determine whether developing custom software for integrating Rancher into the GitOps framework is advisable (see Section 6.1). Alternatively, one could consider either manually onboarding clusters into Rancher or opting for a different multicluster management solution.

6 Conclusion

In this work, we found a significant gap in existing standards and reference architectures for modern Industrial Internet of Things (IIoT) systems, particularly in their ability to meet evolving requirements. In Chapter 2, we discovered that traditional and often OT-driven models, such as the automation pyramid, face challenges in terms of scalability and innovation due to point-to-point integrations. Furthermore, while high-level frameworks like RAMI 4.0 and IIRA offer valuable concepts, they fall short in practical implementation guidance. Additionally, architectures from major cloud vendors often lead to a vendor lock-in and limit functionalities like edge computing. This discovery underlines the necessity for new approaches in designing scalable, efficient, and secure IIoT systems that align with current technological methods and industry needs.

We then introduced a new modern reference architecture encompassing a three-tier architecture and the concept of the unified namespace in Chapter 3. We showed how having an edge, fog and cloud environment with one pub-sub broker per fog environment and a central pub-sub broker in the cloud can provide the means for building a scalable IIoT platform that satisfies modern requirements for such systems. To be able to realize a system of such scale successfully, we discussed GitOps, observability, identity and access management, orchestration and multicluster management.

In Chapter 4, we discussed the topic of provisioning the necessary infrastructure, particularly Kubernetes, for the reference architecture. It became apparent that the Kubernetes-native project "ClusterAPI" can be a solid technology choice since it is capable of provisioning Kubernetes on a large variety of infrastructure ranging from physical machines to virtual machines and even cloud providers while offering a uniform interface to the developers. We also examined how using virtualization tools like VMware vSphere can both simplify and robustify the process of provisioning infrastructure at the cost of a small loss in performance.

Finally in Chapter 5, we implemented the reference architecture in a proof-of-concept. We saw that the strategy for realizing this architecture in the real world, which was developed in this thesis, works effectively, demonstrating the practical feasibility and scalability of our approach. The successful implementation of the architecture across all environments, including the setup of a unified namespace based on MQTT using the HiveMQ broker, together with the integration of the key technologies Kubernetes, ClusterAPI, and FluxCD as the GitOps controller resulted in a flexible and robust setup. This proof-of-concept not only confirms the viability of the proposed architecture and implementation strategy but also

provides a basis for the future development of IIoT systems, allowing for further innovation and efficiency improvements.

6.1 Further Work

While the research and findings presented in this thesis lay the groundwork for building modern and scalable IIoT systems, this work leaves some points open for future work on this topic. One important aspect is the necessity of an in-depth security strategy for IIoT systems. While we had the topic of security in mind at all times, this thesis only scratches the surface of the complex and evolving landscape of cybersecurity. Since this can be seen as one of the most important requirements for an IIoT system in the real world, this is a crucial topic for further research. Another topic is a strategy for the development of domain services operating on the IIoT platform described in this work. The presented strategy mainly deals with infrastructure and leaves designing a strategy for creating workloads within the IIoT system open for future investigation. Lastly, we discovered a lack of support for GitOps in the multicluster management platform Rancher. Subsequent research could solve this by developing custom Kubernetes operators that transform the manual onboarding process of Kubernetes clusters into the management platform into a fully declarative, GitOps-driven approach or by contributing to the open-source Rancher project.

6.2 Acknowledgements

First and foremost, I would like to thank Prof. Dr. Bernhard Bauer and Prof. Dr. Alexander Knapp from the University of Augsburg for supervising this work and assisting me throughout the whole duration. I also want to show my appreciation to Lars Gielsok who supervised this work from the side of the MaibornWolff GmbH and supported this work immensely.

Completing this thesis involved much more than mere supervision, and I would like to extend my gratitude to several individuals whose contributions were invaluable. Thanks to Marc Jäckle (MaibornWolff GmbH) and Sebastian Wöhrl (MaibornWolff GmbH) for sharing incredible amounts of knowledge, providing resources, assisting in the implementation and overall offering me to work on this topic for my master's thesis. Last but not least, I am also grateful to Martin Zehetmayer (MaibornWolff GmbH). Despite having no direct connection to this thesis, he went out of his way to offer essential guidance and support, particularly in the context of bare-metal server management, enriching both the theoretical and practical aspects of this thesis.

Bibliography

[1] United Manufacturing Hub. 2.1 Automation pyramid, February 2023.

[2] Kudzai Manditereza. Smart Manufacturing Using ISA95, MQTT Sparkplug and the Unified Namespace, 2021.

[3] Alexandra Rinke. Was ist OPC UA? Die wichtigsten Begriffe im Überblick, April 2022.

[4] DKE - VDE Verband der Elektrotechnik Elektronik Informationstechnik e.V. RAMI 4.0: Ein Referenzarchitekturmodell als Kommunikationsgrundlage in der Industrie 4.0, 2022.

[5] Gunther Koschnick, Martin Hankel, and Bosch Rexroth. Industrie 4.0: Das Referenzarchitekturmodell Industrie 4.0 (RAMI 4.0), 2015.

[6] Microsoft Azure. Microsoft Azure IoT Reference Architecture, 2018.

[7] Sebastian Wöhrl and Marc Jäckle. MaibornWolff GmbH: Building IIoT 2023: "One size fits all"-Architektur für Industrial-IoT-Plattformen, 2023.

[8] Kudzai Manditereza. HiveMQ: Unified Namespace (UNS) Essentials, 2023.

[9] Kubernetes SIGs. Concepts - The Cluster API Book, 2023.

[10] Fraunhofer IKS. Whitepaper: Industrial Internet of Things: Referenzarchitektur für die Kommunikation, 2016.

[11] Mary Simpkins. Opportunities and Challenges in Edge Computing Under Kubernetes, March 2022.

[12] Claude Baudoin, Erin Bournival, and Ruben Guerrero. The Industrial Internet of Things Vocabulary, 2010.

[13] Jorge Pérez, Jessica Díaz, Javier Berrocal, Ramón López-Viana, and Ángel González-Prieto. Edge computing. *Computing*, 104(12):2711–2747, December 2022.

[14] Atos. A 2021 perspective on edge computing, 2021.

[15] IBM. Why organizations are betting on edge computing Insights from the edge, 2020.

[16] IBM Technology. What is edge computing?, October 2019.

[17] Ramón López-Viana, Jessica Díaz, and Jorge E. Pérez. Continuous Deployment in IoT Edge Computing : A GitOps implementation. In *2022 17th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6, June 2022. ISSN: 2166-0727.

[18] WeaveWorks. GitOps across the Edge and all Clouds, 2021.

[19] WeaveWorks. Deutsche Telekom uses Weave GitOps to deliver edge Kubernetes at scale, 2021.

[20] RedHat. The Path to GitOps, July 2022.

[21] Google. Container-Optimized OS Overview, 2023.

[22] Ahmad Alamoush and Holger Eichelberger. Adapting Kubernetes to IIoT and Industry 4.0 protocols - An initial performance analysis, October 2022. Publisher: Zenodo.

[23] Ivan Čilić, Petar Krivić, Ivana Podnar Žarko, and Mario Kušek. Performance Evaluation of Container Orchestration Tools in Edge Computing Environments. *Sensors*, 23(8):4008, January 2023. Number: 8 Publisher: Multidisciplinary Digital Publishing Institute.

[24] Edwin Mauricio Martinez, Pedro Ponce, Israel Macias, and Arturo Molina. Automation Pyramid as Constructor for a Complete Digital Twin, Case Study: A Didactic Manufacturing System. *Sensors*, 21(14):4656, July 2021.

[25] OPC Foundation. Opc unified architecture - interoperability for industrie 4.0 and the internet of things, 2023.

[26] HiveMQ. A Comparison of OPC UA and MQTT Sparkplug, 2021.

[27] Kudzai Manditereza and HiveMQ. The Key Differences Between OPC UA And MQTT Sparkplug, 2022.

[28] HiveMQ Team. MQTT Topics, Wildcards, & Best Practices – MQTT Essentials: Part 5, 2019.

[29] Industry IIoT Consortium. The Industrial Internet Reference Architecture.

[30] Microsoft. Azure IoT Edge documentation, 2022.

[31] Microsoft. Azure Industrial IoT Overview, November 2022.

[32] MaChecks. AWS IoT Greengrass vs Azure IoT Edge, June 2021.

[33] b.telligent. End of Google IoT core's life – looking for alternatives?, 2022.

[34] Microsoft. Industrial IoT connectivity patterns - Azure Architecture Center, 2023.

[35] LeanIX GmbH. Reference Architecture - The Definitive Guide | LeanIX, 2023.

[36] Marcus Geiger. Zones & Conduits: Das Schutzkonzept aus der IEC 62443 einfach erklärt, March 2018.

[37] Holger Amort. Does MQTT Unified Namespace solve all your data integration issues? - TQS, May 2021. Section: Integration.

[38] OWASP Top Ten | OWASP Foundation, 2023.

[39] Cindy Sridharan. The Need for Observability - Distributed Systems Observability [Book], 2018. ISBN: 9781492033424.

[40] Sematext. What Is Structured Logging and Why You Should Use It, 2023.

[41] Rancher Labs. Rancher - Enterprise Kubernetes Management, 2023.

[42] Efficient Kubernetes Cluster Management: Building Infrastructure-Agnostic Clusters with Cluster API, 2023.

[43] OpenStack. Building the Future on Bare Metal.

[44] Kubernetes SIGs. GitHub: Cluster API, December 2023.

[45] Kubernetes SIGs. Kubernetes Documentation: Operator pattern, 2023.

[46] spectro cloud. CNCF On-Demand Webinar: Cluster API and GitOps: the key to Kubernetes lifecycle management, April 2023.

[47] Geek_Dude. DevOps: What Does Cattle Not Pets Mean?, June 2021.

[48] Tobias Giese and Sean Schneeweiss. Mercedes-Benz: How to Migrate 700 Kubernetes Clusters to ClusterAPI with zero downtime, 2022.

[49] Gardener. GitHub: gardener/docs/concepts/cluster-api.md at master · gardener/gardener, 2023.

[50] Rancher Labs. Launching Kubernetes with Rancher | Rancher, August 2023.

[51] Ulrike Ostler and Martin Hensel. Datacenter insider: Was ist ein Baseboard Management Controller?, March 2022.

[52] Ariane Rüdiger. Datacenter insider: Was ist IPMI?, May 2022.

[53] PARAGON Software. PC Deployment over Network using PXE environment, 2005.

[54] Samer El-Haj-Mahmoud Hewlett Packard. Firmware in the datacenter: Goodbye PXE and IPMI. welcome HTTP Boot and Redfish!, 2015.

[55] Bundesamt für Sicherheit in der Informationstechnik. VS-Anforderungsprofil Hypervisor für Server - Supporting Document, 2020.

[56] Milad Karimyar. Bare Metal vs Hypervisor: Which Is Right For Your Project?, 2023.

[57] Kubernetes SIGs. The Cluster API Provider Azure Book, 2023.

[58] Canonical. Metal as a Service, 2023.

[59] Metal³-Metal Kubed website team. Metal Kubed, 2023.

[60] Metal³-Metal Kubed website team. Baremetal Operator - Metal³ user-guide, 2023.

[61] Tinkerbell Community. Flexible automation for bare metal, 2021.

[62] Sidero Labs. Sidero Metal, 2023.

[63] Sidero Labs. Talos Linux, 2023.

[64] Telekom. GitHub: das-schiff/README.md at main · telekom/das-schiff, 2022.

[65] Bitnami Labs. GitHub: bitnami-labs/sealed-secrets, December 2023.

[66] Introduction - External Secrets Operator, 2023.

[67] GitHub: kubernetes-sigs/secrets-store-csi-driver, December 2023.

[68] Mozilla. GitHub: getsops/sops, December 2023.

[69] VictoriaMetrics. Open Source Time Series Monitoring Tools & Solutions, 2023.

[70] HiveMQ. HiveMQ Kubernetes Operator :: HiveMQ Documentation, 2023.

[71] HiveMQ. HiveMQ Edge: Open-source IIoT gateway & protocol converter to MQTT, 2023.