



Revision of a Smart Factory Software Architecture from Monolith to Microservices

Ronny Seiger¹ and Lukas Malburg^{2,3}

¹ Institute of Computer Science, University of St.Gallen, 9000 St.Gallen, Switzerland
ronny.seiger@unisg.ch

² Artificial Intelligence and Intelligent Information Systems, University of Trier,
54296 Trier, Germany, <https://www.wi2.uni-trier.de>

³ German Research Center for Artificial Intelligence (DFKI)
Branch University of Trier, Behringstraße 21, 54296 Trier, Germany
malburgl@uni-trier.de / lukas.malburg@dfki.de

Abstract. Software architecture plays an important role in the development of modern, complex software systems as it influences a system’s quality attributes and ability to grow with future demand. Designing the software architecture of cyber-physical systems (CPS) becomes even more challenging due to their capability of directly influencing the physical world and thus introducing new non-functional requirements related to fault-tolerance, safety, and resource scarcity. Existing research focuses on systems engineering to achieve the vertical integration of CPS with an organization’s information systems and processes, but not on software architecture to horizontally extend existing systems with new CPS. In this report we describe the process of revising an existing monolithic software architecture for a smart factory towards a microservices-based architecture to meet these new requirements and prepare the factory to be extended with new CPS. For the revision of the existing architecture, we provide an analysis of its code base before and after changes, a description of the refactoring process, and discuss relevant new non-functional requirements and architecture options. We elaborate on the architectural decisions favoring microservices and analyze the new architecture regarding improved quality attributes to evaluate the system.

Keywords: Cyber-physical Systems · Software Architecture · Internet of Things · Microservices · Industry 4.0.

1 Introduction

The architecture of a complex software system has significant influence on its quality attributes and on the feasibility of extending it with new components and functionality in the future [7]. Decisions related to software architecture have a strong impact on the software system and are challenging to revise later [30], which is why they are based on extensive trade-off discussions by software architects [26]. Software architecture has also become of increasing importance when

developing systems that influence and are influenced by the physical world—Cyber-physical Systems (CPS) [16]. Typical systems engineering approaches discuss the *vertical* integration of compounds of sensors and actuators forming CPS (e.g., production machines) with a company’s information systems (e.g., based on the ANSI/ISA-95 pyramid [25]). However, these CPS become increasingly complex and the *horizontal* integration of CPS [42] requires deeper investigations of software architectural aspects. Here, CPS might introduce novel non-functional requirements (NFRs), e.g., related to safety, energy consumption, connectivity, or constraint resources, which are usually not considered when deciding about software architectures in purely digital software systems [30]. On the other hand, common NFRs, e.g., related to performance and elasticity, might not be relevant in CPS as computing resources are constraint. Nevertheless, the CPS software architecture should enable all components to flexibly interact with each other while fulfilling functional and non-functional requirements.

In this paper, we report our experience with revising the software architecture of a model factory as a typical CPS. Starting from a monolithic software system controlling the factory that has originally been developed as a proof-of-concept prototype with focus on vertical integration [34,21,22], we present an analysis of its architecture and code base [6], our experience working with it, and new non-functional requirements. Then, we will discuss breaking down the monolithic architecture into a microservices-based architecture to address the new NFRs. For these changes we elaborate on the architectural decision forces and decisions related to the architectural styles and service sizes, including their implementation, communication, and orchestration. These developments are driven by requirements from Industry 4.0 [11] with the goal of achieving more flexible production scenarios that exhibit high fault-tolerance, extensibility, and maintainability while also considering resource constraints and safety [25].

The paper is structured as follows: We elaborate on experiences with the existing smart factory architecture and problems we have identified in Section 2. Here we also identify new requirements as objectives of a solution in a first design science cycle [28] to improve the operation of the smart factory system. In Section 3 we build and develop a revised version of the software architecture as main artifact to serve as basis for an extension of the CPS. We evaluate and demonstrate this solution in Section 4. Section 5 presents related work. Section 6 concludes the paper and shows potential future work.

2 Existing Smart Factory Software Architecture

2.1 Software and Hardware Components

We use a smart factory model with components provided by Fischertechnik as basic hardware platform for our Business Process Management (BPM) related research [22,34,20,21]. The smart factory simulates a production line that consists of 6 production stations, each representing a different capability that can be executed in a production process (e.g., burning, milling, or transportation). A production station is managed by an embedded controller that executes low-level

commands to control connected sensors and actuators. These commands are sent from a self-developed, monolithic control software running on a dedicated computer. The software architecture of this software system is technically layered with *one* Web Service (cf. Fig. 1) exposing the capabilities of all production stations as REST resources [34,22,21]. The encapsulation of the low-level commands as high-level capabilities that view each production station as an entity is achieved via an object-oriented Domain Model (cf. Fig. 1) following domain-driven design [4]. As depicted in Fig. 1, we use a Workflow Management System (WfMS) as an orchestrator of the production that facilitates the modeling of processes in BPMN 2.0 [27], their automation, adaptation [20], and mining [34].

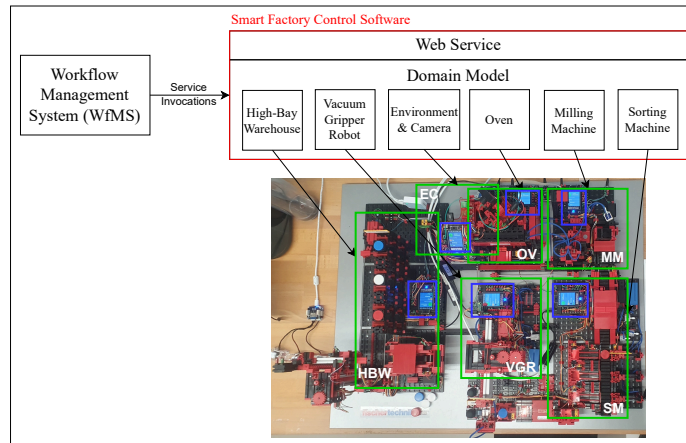


Fig. 1. Original architecture of smart factory control software.

2.2 Architectural Decisions

The decisions regarding the design and implementation of the smart factory control system have mostly been driven by functional requirements to control the production line for BPM-based research, and focused on the vertical integration of all hardware components from sensors to services [25] in a proof-of-concept prototype. The goal was to encapsulate the sensors and actuators belonging to the production stations at a reasonable level of abstraction and to provide service-based access to this high-level functionality [34]. NFRs related to agility, elasticity, or scalability did not play an essential role as the operation of the smart factory is limited by the physical resources (i.e., sensors, actuators, materials) available to execute requests and activities in the production processes [34,21]. A detailed description of the existing software system and driving functional requirements can be found in [34] and [21]. We decided to base the initial implementation of the smart factory control system on a **monolithic** service-based architecture for the following reasons (NFRs) [30]:

- *Costs of Prototype*: To serve as basis for several research projects [22], we aimed to have a quick and relatively low-cost proof-of-concept implementation of the smart factory control system. Priorities here were not on the software architecture, but on the prototypical implementation to have a basis for advanced research on flexible and adaptive processes in CPS [20,19].
- *Simplicity*: We aimed for a rather simple solution to achieve a quick implementation of the factory control system that is relatively easy to maintain.
- *Functionality*: The functionality of the individual stations is not too complex. Each station usually offers 1 to 3 different types of capabilities [34,21].
- *Configurability and Deployability*: The control software should be easy to configure, to deploy, and to run on a standard desktop computer.

2.3 Experience and New Requirements

We informally collected anecdotal experience from three research groups working with the monolithic implementation of the smart factory control system. All agree that the low operational costs and easy deployability of the monolithic architecture lead to a positive experience when using the software system to control the smart factory. In normal operation mode, a high degree of fulfillment of the *functional* requirements related to the production stations can be observed. However, over time all groups identified new NFRs related to *fault-tolerance*, *recoverability*, and *extensibility* [32] that are not fulfilled by the system. Moreover, all groups experienced an increased complexity of debugging and maintaining the software system, in particular during experimental evaluations [20,19,34]. New requirements originated from the interactions of the system with the physical world and the rather unreliable hardware/software components of the smart factory model, which often lead to exceptions, unexpected behavior, and network disconnects of its embedded controllers [20]. Furthermore, the smart factory control system is intended to serve as basis for our BPM-related research [22,13], which entails extending its functionality and features to fulfill new requirements from the implementation of research prototypes on a regular basis. Here, the software system does not perform well either. The non-functional characteristics that became more prominent include:

- *Fault-tolerance*: The operations of the smart factory are frequently interrupted by unexpected events at the physical production stations or by errors within the embedded controllers (e.g., loss of network connection) and software. These errors regularly lead to the entire system not working properly anymore and thus, to reduced availability and need for human interventions.
- *Recoverability*: Although errors typically relate to only one production station or one hardware component, the entire control system has to be stopped and restarted to recover from the errors, which leads to unnecessary downtimes for all stations. Recoverability of the system is low.
- *Extensibility and Maintainability*: The smart factory will be extended with additional CPS (robots) to simulate more complex production scenarios [32]. However, the extensibility of the existing software system is rather low as

adding new devices and functionality requires many changes in the existing code base. This symptom also leads to low *maintainability* of the software system as fixing errors, modifying code, and adding new features often result in performing many changes that may also break the system [12,23].

These insights motivate us to investigate the following two research questions:

RQ1 What are reasons for a CPS software system to show symptoms of poor fault-tolerance, recoverability, extensibility and maintainability?

RQ2 How does the architecture of a CPS software system need to be designed to address these non-functional requirements?

2.4 Code Analysis of the CPS Software System

In Section 2.3 we already provided indications to answer RQ1 related to the smart factory’s hardware. Learning factories offer a suitable, low-cost playground for CPS and BPM research as discussed in [22]. However, the hardware components are less reliable than in real production settings, and the hardware-software controllers (PLCs) do not implement any safety or fault-tolerance mechanisms. Thus, issues related to hardware have to be addressed by the software system controlling the smart factory. To further investigate RQ1, we performed a static code analysis of the Python-based code base of the existing factory control system⁴ using Sonargraph Architect⁵. Our main focus here was on code entanglement, dependency cycles, large files/classes, complexity, and code duplication, as issues in any one of these may lead to the symptoms described in Section 2.3 [2].

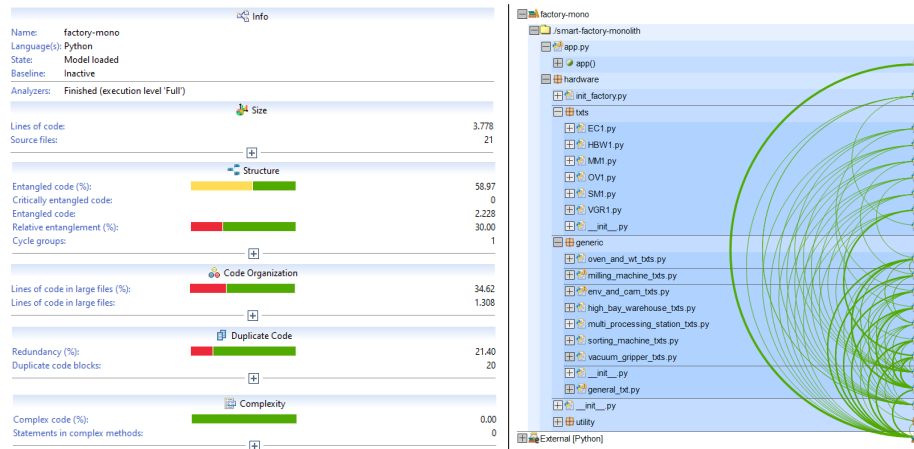


Fig. 2. Code analysis of existing smart factory control software.

⁴ <https://github.com/ics-unisg/smart-factory-monolith>

⁵ <https://www.hello2morrow.com/products/sonargraph/architect9>

A summary of the analysis results is shown in Fig. 2 (left); a simplified dependency graph comprising all Python files organized in packages in Fig. 2 (right). Even though the complexity of individual classes, methods, and code fragments is low, a relatively high percentage of code is entangled, indicating a rather rigid software design [23] and technical debt resulting from the quick prototype implementations, which lead to a high effort in maintainability and low extensibility. The same holds for the identified code redundancy indicating issues in the object-oriented design. However, almost no cyclic dependencies exist, which is a good sign of clean architecture. The main issue influencing fault-tolerance and recoverability is related to the implementations of the web service on top of the domain model (cf. Fig. 1) and the single class initializing the entire functionality of the domain model (file: `init_factory.py`). While the rest of the domain is already structured into individual, functionally cohesive modules per production station, the web service (file: `app.py`) contains all REST end points and logic to call the methods of all classes in the domain model. This one file with >1.3k lines of code (LoC) and the initializer class make the software system a monolith—violating the single-responsibility principle [23], among others—and impose the rather poor performance regarding fault-tolerance and recoverability on it [30]. As indicated in Section 2.3, an issue with the hardware of one production station (e.g., loss of network connection to a controller), requires a restart of the entire software system to recover from the error. With these insights, we can confirm in response to RQ1 that a monolithic architecture and the identified issues (smells) in the code base lead to a decrease of fault-tolerance, recoverability, extensibility and maintainability, also for software systems controlling CPS [36,7].

3 Revision of the Smart Factory Architecture

The new non-functional characteristics discussed in Section 2.3 became the driving forces influencing the operation of the existing software system. To prevent the accumulation of more technical debt and architecture erosion with future extensions [7], and to improve the fulfillment of the new NFRs (cf. RQ2), we decided to redesign and refactor the software system. Thereby, the fulfillment of the functional requirements (cf. Section 2.1) should not be affected.

3.1 Architecture Options

The new NFRs (cf. Section 2.3) motivated the revision of the monolithic software system controlling the smart factory. Based on these new driving forces we evaluated typical software architectural styles summarized by Richards and Ford in [30] regarding their suitability. In addition to the NFRs, the authors discuss the type of partitioning of an architectural style—technical partitioning or domain partitioning—and the number of possible standalone software components (*Architectural Quanta*) the specific style might lead to (one or many) [30].

With maintainability, fault-tolerance, and extensibility being among the main driving forces, we decided that the main strategy of revising the architecture and

implementation is to focus on *decoupling* of components. Thus, we discarded all options that involve monolithic styles resulting in only one architectural quantum (e.g., layered architecture, modular monoliths, microkernel architectures, and service-oriented architectures) as these are likely to not improve fault-tolerance and availability [30]. The software system controlling the production stations of the factory does not require a sophisticated database infrastructure or shared access to data from the stations [3]. For this reason, we discarded the options of service-based and space-based architectures as non-monolithic architectural styles. Finally, we decided that a **microservices** architecture is the best fit regarding fault-tolerance and extensibility as driving forces addressing RQ2 [30].

3.2 Architecture Decisions and Implementation

As discussed, we found the best trade-offs in a *microservices architecture* [30]. While keeping the functionality intact, the main question regarding the redesign concerns the *integrators* and *disintegrators* that determine the size of one microservice, i.e., *how* the monolith should be split up. Typical integrators and disintegrators in digital services include different levels of: 1) performance and throughput, 2) code volatility, 3) data security, and 4) transactional boundaries in specific parts of a software system [30]. If parts of a system show similar characteristics regarding one or more of these aspects, then they might be put together (integrated) in one service. If they exhibit different characteristics, then these parts are likely to be disintegrated into different services. However, the aforementioned four aspects do not apply in the given CPS setting. In addition to the new NFRs (cf. Section 2.3), we determined the following aspects to influence the decisions regarding the size of a microservice in the smart factory:

- *Functionality*: Despite the low amount of functionality associated with each production station (cf. Section 2.1), the one web service controlling the factory depicted in Fig. 1 implements in total 15+ resources that expose the high-level functionality of all stations [34]. As stated in Section 2.4, this is a sign of low functional cohesion [23]. We see a need for smaller web services.
- *Physical setup*: The physical layout and configuration of the production line naturally groups all sensors and actuators belonging to one production station (or production cell). These sensors and actuators are wired to the embedded controllers on a per-production-station basis (cf. Fig. 1), i.e., one embedded controller is responsible for one production station.

Service Sizes: The physical setup as a novel aspect influencing architectural decisions in CPS became the main force to decide about the service granularities. The physical grouping of components related to one embedded controller and thus to one production station—one responsibility—is a natural disintegrator and fits well to the size of one microservice [3]. In addition, we often encounter issues in the factory’s operation on the level of individual controllers, which supports this service granularity as it isolates failures on the level of a production station.

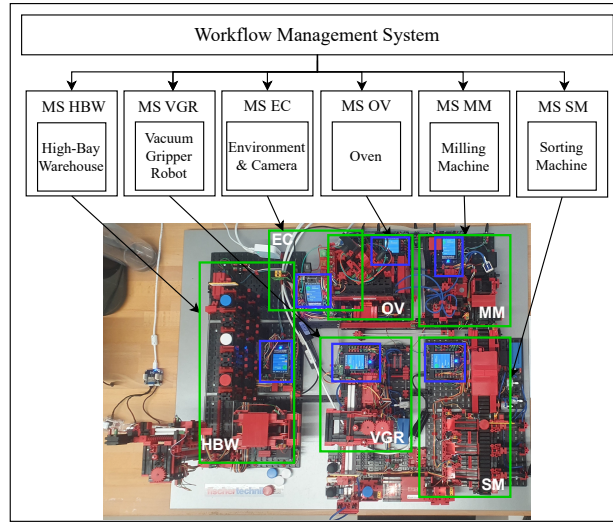


Fig. 3. Revised microservices (MS)-based architecture of the smart factory system.

The decision of having *one microservice per production station* partitions the architecture related to the domain functionality and capabilities of individual production stations, i.e., we will have a bounded context per station (subdomain [3]) and thus a higher functional cohesion per service [4].

Refactoring: The implementation of this new architectural style based on the results from the static code analysis of the original system [6] was straightforward as the domain core of the existing monolithic system was already structured based on the different production stations (cf. Section 2.4). With the original source code and additional metadata (e.g., architectural diagrams) available, we were able to follow the decomposition process described in [6]. In the object-oriented domain model, there was one class per type and instance of a production station [34,21], which contained the low-level logic to control the station’s sensors and actuators via the respective embedded controller. The monolithic web service on top of the domain core also grouped the resources based on the production stations [34,21]. Therefore, it was relatively easy to refactor the monolith and to extract the relevant logic and service implementations into individual microservices. The main refactoring that needed to be done was splitting up the web service (file: `app.py`) and factory initializer class (file: `init_factory.py`) based on the individual production stations. As we were not using a sophisticated database infrastructure to persist data, decomposing the software system into microservices was straightforward [3]. Fig. 3 shows the resulting new architecture. The interactions of the microservices as part of the production processes are still orchestrated by one WfMS [34]. In our setup, all 6 microservices are deployed to and running in a container-based Docker environment on a desktop computer, which is connected to the embedded controllers via Ethernet.

3.3 Code Analysis of the Microservices-based System

Similar to the code analysis for the existing system described in Section 2.4, we analyzed the code base⁶ of the revised, microservices-based smart factory control system using Sonargraph, considering code entanglement, dependency cycles, large files/classes, complexity, and code duplication.

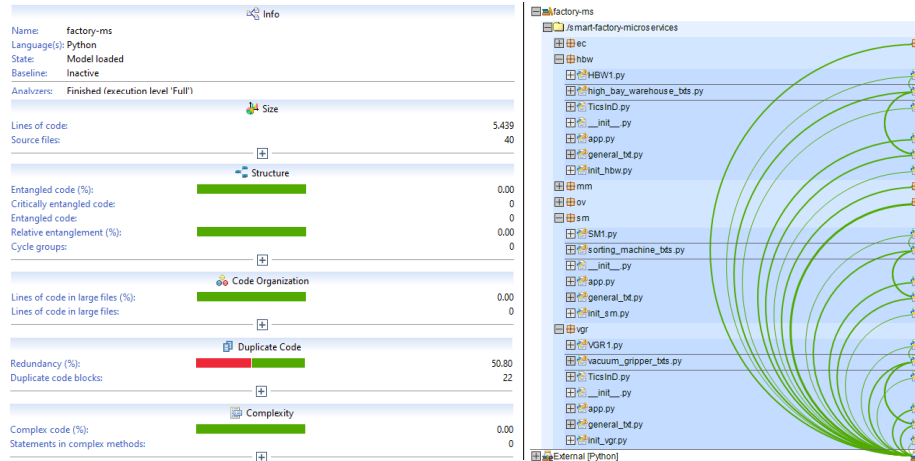


Fig. 4. Code analysis of revised smart factory control software.

A summary of the analysis results can be seen in Fig. 4 (left). A simplified dependency graph showing all microservices in dedicated modules and exemplary Python files is depicted in Fig. 4 (right). Here we see that the code has been reorganized from a more object-oriented structure into domain-partitioned modules (e.g., hbw, sm, vgr, etc.) each representing a microservice for one production station. The redesign and refactoring show a major improvement and reduction regarding code entanglement and code in large files. We achieved this through decomposition, reorganizing the code, and splitting up the web service implementation into the individual microservices. However, we also observe an increase in code redundancy, which is often encountered when migrating from an object-oriented monolith to microservices [5]. In our implementation, each production station has common functionality and attributes implemented in a base class that is then specialized by the production station. When migrating to microservices, we decided to replicate this base class for each microservice, which led to an increase of the code base by 44% and duplicated code by 20%. Several strategies exist to deal with shared code (e.g., shared libraries, shared services, sidecar pattern [5]). In our revised implementation, we decided to use code replication as we do not expect many future changes to the base class [5].

⁶ <https://github.com/ics-unisg/smart-factory-microservices>

4 Evaluation and Discussion

By investigating RQ1 based on the analysis of the original code base and architecture in Section 2.4, we identified various issues in the design and code of the monolithic system that led to a poor performance when fulfilling the new NFRs fault-tolerance, recoverability, extensibility, and maintainability discussed in Section 2.3. In this section we evaluate the effects of the revised, microservices-based architecture on these NFRs based on a qualitative discussion to answer RQ2. These discussions partially refer to the results of analyzing both versions of the software system using Sonargraph. The detailed metrics are contained in the provided repositories for the monolith and microservices-based implementation.

4.1 Maintainability

Maintainability improved due to the microservices being less complex regarding their functionality and implementation [15]. A stricter adherence to the single responsibility principle led to smaller microservices with individual sizes between 871 and 2175 LoC (compared to a total of 3778 LoC for the monolith) and cleaner implementations with higher functional cohesion, less coupling, and less dependencies. The average component dependency ACD [2] decreased from 3.71 to 1.55; the propagation cost metric according to MacCormack et al. [2] strongly decreased from 17.69 to 3.88; and the cumulative structural debt index for system decreased from 11 to 0 [2]—all indications of a cleaner implementation and improved structure [23]. Thus, we can assume that the individual microservices are easier to maintain and to debug [2]. Bugs can be more easily located and associated with a specific microservice as there is higher functional cohesion and less source code inside one service [12]. However, in Section 3.3 we have also identified an increase in code redundancy due to a common base class being reused in all microservices, which increased the Average Complexity (based on McCabe’s cyclomatic complexity) [2] of the entire system from 1.36 to 1.59. In general, replicated code leads to an increase in maintainability of a software system as all changes at one location need to be tracked and consistently changed across all copies, often requiring code versioning [5]. In our implementation, we assume the implementation of the base class to be stable and barely changing—circumstances where replicated code is acceptable in a microservices context [5].

4.2 Fault-tolerance and Recoverability

The NFRs *fault-tolerance* and *recoverability* became the most important driving forces when working with the smart factory system. Among many sources, Richards and Ford attribute an improved fault-tolerance and recoverability to a microservice-based architecture compared to monolithic approaches [30]. We can confirm this observation in our CPS context as decomposing the monolithic architecture into per-production-station microservices increased fault-tolerance and recoverability. By focusing on the decoupling of components, we were able to confine potential errors to the scope of a production station. Hence, hardware

errors (e.g., related to the connectivity of a production station’s controller) do not affect other stations or services. We can easily resolve issues at the granularity of a production station and recover the failed component by restarting the corresponding controller and microservice to restore normal operations. The application of techniques from fault-tolerant programming may further improve the system’s tolerance against software errors. Our focus was on addressing hardware-related issues as primary source for interruptions of the smart factory.

The deployment and operation of the now 6 microservices has become more complex. However, the platform used for running the containerized microservices greatly reduces this complexity as all containers can be started at once and re-deployed and restarted individually to recover from errors [8], without adding significant overhead to the computer’s resource consumption [37]. So far, our experience working with the revised software system, especially for experimental settings, confirms an increased fault-tolerance, better recoverability, and thus a higher overall availability of the production system. Nevertheless, a higher number of involved (micro)services also means an increase of network communication among the services, which may result in higher latency and communication issues [1]. These aspects do not play a significant role in our setup with all services running in the same local, container-based environment. A more systematic analysis of the software system’s availability, ability to recover from errors, and need for human interventions remains subject to future work.

4.3 Extensibility

One of the main goals of this work was to prepare the smart factory software system for an extension with additional CPS. As acknowledged in literature [30], a microservice-based architecture exhibits a better extensibility than a monolithic system, which motivated us to revise the existing software architecture. The results of analyzing the revised implementation (cf. Sections 3.3 and 4.1) show a decrease in dependencies, coupling, and code entanglement, which are indications of improved extensibility. Adding new microservices to the software system is facilitated due to the lower coupling and no shared code and dependencies between services. A new Fischertechnik-based production station of known type can be added non-invasively to the production line by simply instantiating a second container for the respective microservice with a slightly different configuration (e.g., IP address). A new production station of unknown type can be added based on a new microservice where the common base class is reused and extended with the machine specific implementation. We have successfully extended the implementation and instantiated the containers for a different smart factory configuration with multiple instances of Fischertechnik-based machines [21].

To further demonstrate the extensibility, we added four robots—two mobile robots of type TurtleBot 4 Pro and two robotic grippers arms of type Dobot Magician—to the existing CPS (cf. Fig. 5). As the microservices-based style of the CPS dictates the software architecture, it has been straightforward to integrate the control systems for the robots into the overall CPS, also in a (micro)service-based manner with dedicated services for each robot. Putting these systems

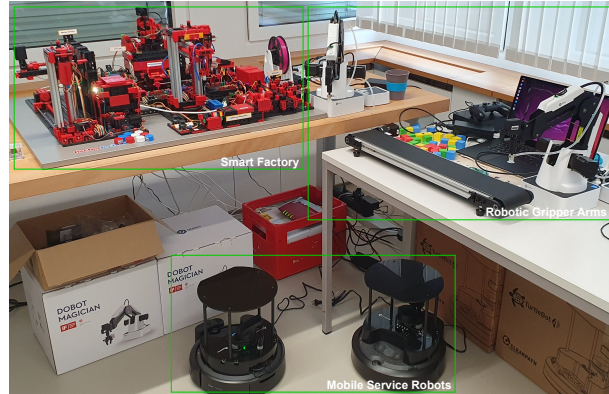


Fig. 5. Extended smart manufacturing setup with new CPS.

together with one or more of the existing microservices of the smart factory would have led to a more monolithic approach negatively impacting the quality attributes. The two robotic gripper arms have been integrated in a service-based manner, sharing a common instance of ROS2 [18] to control both robots and expose their capabilities via web services. The shared ROS2 instance is suitable because of the close physical proximity of both robots and the need to save resources on the available computer running the ROS2 instance. The two mobile robots are regarded as standalone microservices, which are each controlled by a dedicated ROS2 instance on a Raspberry Pi 4. The main driver for having one microservice per robot was to enable both robots to operate autonomously from any other software service and thus not being reliant on a constant network connection [33]. Adding the robots in this (micro)service-based way did not require any modifications to the existing microservices-based software system controlling the smart factory, and it allows us to orchestrate all services of all components uniformly based on REST using a WfMS. In our setup, additional communication among all services is facilitated by using a messaging and stream processing system [34]. Table 1 summarizes all decision forces and decisions we took when revising the smart factory system and adding the robots to the CPS.

4.4 Research Questions and Limitations

With answering RQ1 in Section 2.4, we identified in our case study the monolithic style and issues in the software design and code base related to code entanglement, high coupling and low cohesion, as main reasons for the existing CPS software system to perform rather poorly regarding the NFRs fault-tolerance, recoverability, extensibility, and maintainability [36]. The results of discussing these NFRs after revising the existing monolithic software system to a microservices-based architecture indicate an improvement of all these NFRs. Thus, as an answer to RQ2, we suggest to focus on decoupling monolithic CPS control systems towards more individual standalone components, preferably *microservices* the

Table 1. Summary of architectural decisions in the extended CPS.

| CPS | Architectural Style | Service Size | Decision Forces |
|-----------------------|---------------------|----------------------|---------------------------------------------------------------|
| Smart Factory | Microservices | 1 production station | Physical setup, fault-tolerance, extensibility, functionality |
| Robotic Gripper Arms | Service-based | 1 robot | Physical setup, computing resources, fault-tolerance |
| Mobile Service Robots | Microservices | 1 robot | Autonomy, energy consumption, connectivity, fault-tolerance |

size of a production station or production cell, to improve fault-tolerance, recoverability, extensibility, and maintainability.

As a limitation we acknowledge that we work with a small-scale learning factory [22], which does not have any hard real-time and safety constraints, and which allows us to interface with the embedded controllers using high-level programming languages from desktop computers. Assuming that these kinds of open interfaces and full control of the hardware exist in a real industrial production environment might not be completely realistic, but with machine interfaces becoming more accessible and standardized (e.g., based on the *Asset Administration Shell* [32]) we can observe a trend towards more openness and interoperability in the future. The learning factories serve as bridges between completely simulated, virtual environments and real-life production settings. They can be used to educate shop floor personnel and to conduct sophisticated low-cost and low-effort research of CPS and BPM in more realistic settings [22]. Therefore, we deem the insights of this work to be also relevant more generally in CPS as existing CPS software systems are usually structured in a more monolithic way and also exhibit typical software design flaws and code smells [36], which negatively impact the quality attributes of the software systems. We hypothesize that our proposal of moving towards more decoupled service-based architectures, where the physical setup of the individual CPS components has a strong influence on the service granularities, can be generally applied to improve non-functional characteristics of distributed CPS. Moreover, we observe an increasing number of research groups using a variant of the learning factory as basis for their research in BPM, CPS/IoT, software engineering, and automation. The insights and artifacts presented in this experience report might facilitate their setup of the factory based on our proposed microservices-based software architecture.

5 Related Work

Literature surveys on using microservices-based software architectures in the context of IoT can be found in [35] and [29]. In [35] the authors discuss about NFRs relevant in IoT and they provide pointers to different approaches that discuss solutions to fulfill these NFRs.

In [15] the authors highlight the flexibility and versatility of using microservices to implement small features bounded within processes in IoT, in contrast to heavy weight inflexible monoliths. A microservice architecture for the industrial IoT (IIoT) is presented by Dobaj et al. in [3]. The authors discuss different types of design patterns for IIoT and relevant decision forces such as dependability, performance, and flexibility. Based on these, a layered microservices-based architecture and the application of the design patterns to address the IIoT-related NFRs is presented. Among others, the aspects of decomposition into subdomains and shared data access in the microservices architecture are discussed, which are well aligned with the decisions and discussions we have presented in our approach. The design of a microservice architecture for a smart city IoT platform that organizes the microservices around business capabilities, similar to the production stations representing the capabilities of the smart factory, is presented in [14]. Deciding about the granularity of individual microservices based on business capabilities is also a common strategy in purely digital, non-CPS systems as pointed out by the authors in [8] discussing their move from monoliths to microservices. NFRs around architectures for IIoT are discussed in [39]. Besides high availability, extensibility, and interoperability, which are discussed in our work, too, the authors emphasize real-time operations and cyber-security as critical aspects in IIoT. We agree that these are highly relevant in real-world deployments, but we found them less significant for the used smart factory model in our laboratory environment. We skipped these non-functional aspects due to their complexity and refer to the discussions in [39].

A literature study on the migration of monoliths to microservices is presented in [1]. In addition, the authors provide a case study on how to benchmark the migration by comparing the performance and consumption of computing resources of the system before and after migration. This study can be very helpful for our future work to conduct a more quantitative analysis of our proposed revision of the existing CPS software architecture. The aspect of migrating existing software architectures in CPS towards microservices is discussed in [17] and [31]. Sarkar et al. showcase their migration of a complex monolith controlling an industry automation system to containerized microservices [31]. Among others, they are faced with strong couplings between components, which increases the difficulty of breaking down the monolith. Liu et al. present migration strategies that consider economic factors to reduce downtimes and costs when moving an active production line to microservices [17], which is only partially relevant in our laboratory environment. In [24], the authors discuss the migration of a software system for autonomous UAV-based infrastructure inspection from monolith to microservices. They nicely show the benefits microservices for scalable data processing in their work, which will become relevant for our smart factory once we put a stronger focus on processing of the data emitted from the factory.

Several works discuss the design of specific IoT architectures in smart production [38,10,41], which also address the aspect of integrating different types of robots based on ROS [38]. All approaches follow service-based architectures that feature messaging systems for loosely coupled interactions, similar to our

proposed architecture. Additionally, various works discuss the use of workflow engines, as we do, to orchestrate processes [38] and (micro)services [40].

From the discussion of related work we can conclude that we identified and address NFRs with high relevance for IIoT and CPS in our work. The methodological approach, decision forces, and architectural decisions we took when breaking down the smart factory monolith into microservices and extending it with robots are well aligned with existing literature. Moreover, we can confirm that containerization of microservices, messaging systems for communication, and WfMS for orchestration of processes are suitable means for creating flexible and extensible software architectures that promote high fault-tolerance, autonomy, and loose coupling in complex software systems controlling CPS.

6 Summary and Future Work

In this work we discussed the case of revising an existing monolithic software architecture for a smart factory control system towards a microservices-based architecture. The original design and implementation of the monolith was driven by functional requirements and the goal to have a quick, low-cost implementation of a prototype to serve as basis for more advanced research. However, the experience from working with this prototype led to the emergence of new non-functional requirements (NFRs) related to fault-tolerance, recoverability, maintainability and extensibility, which became the main driving forces to decompose the monolith into microservices as it did not fulfill these new NFRs sufficiently. We analyzed the monolith and its code base to identify the reasons for not fulfilling these new NFRs. A discussion of architectural options and ways to mitigate identified flaws in the software design and code base led us to focus on decoupling of the monolithic system's components and breaking the system down into microservices. The physical grouping of sensors and actuators belonging to one production station that is managed by one embedded controller became a natural fit to dictate the size of one microservice representing the station's business capabilities. An analysis of the new system's design and code base showed improvements regarding maintainability, coupling, and cohesion of components. The characteristics of the new microservices-based architecture confirm an improved fault-tolerance, recoverability, and extensibility, which we demonstrated by non-invasively adding new robots controlled by (micro)services to the existing CPS software architecture. All microservices are orchestrated by a workflow management system enabling advanced research on BPM and IoT [9].

In future work we plan to adapt the implementation of the individual microservices to be compatible with the asset administration shell to improve extensibility and interoperability with other systems [32]. Furthermore, we will conduct a more systematic quantitative evaluation of the CPS where we will compare different architectural decisions and their impact on NFRs and resource consumption. In this context, we plan to develop a formal model based on [26] and a case base of experiences regarding architectural decisions in CPS to document their impact on NFRs and provide guidance to software architects.

References

1. Bjørndal, N., Bucchiarone, A., Mazzara, M., Dragoni, N., Dustdar, S., Kessler, F.B., Wien, T.: Migration from monolith to microservices: Benchmarking a case study. Tech. Rep. (2020)
2. Ciceri, C., Farley, D., Ford, N., Harmel-Law, A., Keeling, M., Lilienthal, C., Rosa, J., Von Zitzewitz, A., Weiss, R., Woods, E.: Software Architecture Metrics. " O'Reilly Media, Inc." (2022)
3. Dobaj, J., Iber, J., Krisper, M., Kreiner, C.: A microservice architecture for the industrial internet-of-things. In: Proceedings of the 23rd European Conference on Pattern Languages of Programs. pp. 1–15 (2018)
4. Evans, E.: Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional (2004)
5. Ford, N., Richards, M., Sadalage, P., Dehghani, Z.: Software Architecture: The Hard Parts. " O'Reilly Media, Inc." (2021)
6. Fritsch, J., Bogner, J., Zimmermann, A., Wagner, S.: From monolith to microservices: A classification of refactoring approaches. In: Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment: First International Workshop, DEVOPS 2018, Chateau de Villebrumier, France, March 5-6, 2018, Revised Selected Papers 1. pp. 128–141. Springer (2019)
7. Furrer, F.J.: Future-proof software-systems. Springer (2019)
8. Gouigoux, J.P., Tamzalit, D.: From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW). pp. 62–65 (2017)
9. Janiesch, C., Koschmider, A., Mecella, M., Weber, B., Burattin, A., Di Ciccio, C., Fortino, G., Gal, A., Kannengiesser, U., Leotta, F., et al.: The internet of things meets business process management: a manifesto. IEEE Systems, Man, and Cybernetics Magazine **6**(4), 34–44 (2020)
10. Jepsen, S.C., Worm, T.: Designing and evaluating interoperable industry 4.0 middleware software architecture: Reconfiguration of robotic system. In: European Conference on Software Architecture. pp. 205–220. Springer (2023)
11. Kagermann, H., Wahlster, W.: Ten Years of Industrie 4.0. Sci **4**(3), 26 (2022)
12. Kalske, M., Mäkitalo, N., Mikkonen, T.: Challenges when moving from monolith to microservice architecture. In: Current Trends in Web Engineering: ICWE 2017 International Workshops, Liquid Multi-Device Software and EnWoT, practi-Web, NLPIT, SoWeMine, Rome, Italy, June 5-8, 2017, Revised Selected Papers 17. pp. 32–47. Springer (2018)
13. Kirikkayis, Y., Gallik, F., Seiger, R., Reichert, M.: Integrating iot-driven events into business processes. In: International Conference on Advanced Information Systems Engineering. pp. 86–94. Springer (2023)
14. Krylovskiy, A., Jahn, M., Patti, E.: Designing a smart city internet of things platform with microservice architecture. In: 2015 3rd international conference on future internet of things and cloud. pp. 25–30. IEEE (2015)
15. Lai, C., Boi, F., Buschetti, A., Caboni, R.: Iot and microservice architecture for multimobility in a smart city. In: 2019 7th International Conference on Future Internet of Things and Cloud (FiCloud). pp. 238–242. IEEE (2019)
16. Lee, E.A.: Cyber physical systems: Design challenges. In: 2008 11th IEEE international symposium on object and component-oriented real-time distributed computing (ISORC). pp. 363–369. IEEE (2008)

17. Liu, Y., Yang, B., Ren, X., Liu, Q., Liu, S., Guan, X.: E2ms: An efficient and economical microservice migration strategy for smart manufacturing. *IEEE Transactions on Services Computing* (2024)
18. Macenski, S., Foote, T., Gerkey, B., Lalancette, C., Woodall, W.: Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics* **7**(66), eabm6074 (2022)
19. Malburg, L., Brand, F., Bergmann, R.: Adaptive Management of Cyber-Physical Workflows by Means of Case-Based Reasoning and Automated Planning. In: 26th EDOC Workshops. LNBIP, vol. 466, pp. 79–95. Springer (2023)
20. Malburg, L., Hoffmann, M., Bergmann, R.: Applying MAPE-K control loops for adaptive workflow management in smart factories. *J. Intell. Inf. Syst.* pp. 1–29 (2023)
21. Malburg, L., Klein, P., Bergmann, R.: Semantic Web Services for AI-Research with Physical Factory Simulation Models in Industry 4.0. In: *Int. Conf. on Innovative Intelligent Industrial Production and Logistics*. pp. 32–43. ScitePress (2020)
22. Malburg, L., Seiger, R., Bergmann, R., Weber, B.: Using Physical Factory Simulation Models for Business Process Management Research. In: *BPM Workshops*. LNBIP, vol. 397, pp. 95–107. Springer (2020)
23. Martin, R.C.: *Clean architecture*. Prentice Hall (2017)
24. Matlekovic, L., Schneider-Kamp, P.: From monolith to microservices: Software architecture for autonomous uav infrastructure inspection. *arXiv preprint arXiv:2204.02342* (2022)
25. Monostori, L.: Cyber-physical production systems: Roots, expectations and r&d challenges. *Procedia Cirp* **17**, 9–13 (2014)
26. Nowak, M., Pautasso, C.: Team situational awareness and architectural decision making with the software architecture warehouse. In: *European Conference on Software Architecture*. pp. 146–161. Springer (2013)
27. Object Management Group: BPMN 2.0 specification. <https://www.omg.org/spec/BPMN/2.0/> (2011)
28. Peffers, K., Tuunanen, T., Rothenberger, M.A., Chatterjee, S.: A design science research methodology for information systems research. *Journal of management information systems* **24**(3), 45–77 (2007)
29. Razzaq, A.: A systematic review on software architectures for iot systems and future direction to the adoption of microservices architecture. *SN Computer Science* **1**(6), 350 (2020)
30. Richards, M., Ford, N.: *Fundamentals of software architecture: an engineering approach*. O’Reilly Media (2020)
31. Sarkar, S., Vashi, G., Abdulla, P.: Towards transforming an industrial automation system from monolithic to microservices. In: *23rd Intern. Conf. on Emerging Technologies and Factory Automation (ETFAs)*. vol. 1, pp. 1256–1259. IEEE (2018)
32. Schnicke, F., Kuhn, T., Antonino, P.O.: Enabling industry 4.0 service-oriented architecture through digital twins. In: *Software Architecture: 14th European Conference, ECSA 2020 Tracks and Workshops, L’Aquila, Italy, September 14–18, 2020, Proceedings 14*. pp. 490–503. Springer (2020)
33. Seiger, R., Herrmann, S., Aßmann, U.: Self-healing for distributed workflows in the internet of things. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. pp. 72–79. IEEE (2017)
34. Seiger, R., Malburg, L., Weber, B., Bergmann, R.: Integrating process management and event processing in smart factories: A systems architecture and use cases. *J. Manuf. Syst.* **63**, 575–592 (2022)

35. Siddiqui, H., Khendek, F., Toeroe, M.: Microservices based architectures for iot systems-state-of-the-art review. *Internet of Things* p. 100854 (2023)
36. Sonnleithner, L., Oberlehner, M., Kutsia, E., Zoitl, A., Bácsi, S.: Do you smell it too? towards bad smells in iec 61499 applications. In: 2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFFA). pp. 1–4. IEEE (2021)
37. Sun, X., Liang, Y., Huang, H.: Design and implementation of internet of things platform based on microservice and lightweight container. In: 2020 IEEE 9th Joint International Information Technology and Artificial Intelligence Conference (ITAIC). vol. 9, pp. 1353–1357. IEEE (2020)
38. Traganos, K., Grefen, P., Vanderfeesten, I., Erasmus, J., Boultadakis, G., Bouklis, P.: The horse framework: A reference architecture for cyber-physical systems in hybrid smart manufacturing. *Journ. of Manufacturing Systems* **61**, 461–494 (2021)
39. Urbina, M., Acosta, T., Lázaro, J., Astarloa, A., Bidarte, U.: Smart sensor: Soc architecture for the industrial internet of things. *IEEE Internet of Things Journal* **6**(4), 6567–6577 (2019)
40. Valderas, P., Torres, V., Serral, E.: Modelling and executing iot-enhanced business processes through bpmn and microservices. *Journal of Systems and Software* **184**, 111139 (2022)
41. Xia, C., Zhang, Y., Wang, L., Coleman, S., Liu, Y.: Microservice-based cloud robotics system for intelligent space. *Robotics and Autonomous Systems* **110**, 139–150 (2018)
42. Zuehlke, D.: Smartfactory—towards a factory-of-things. *Annual reviews in control* **34**(1), 129–138 (2010)