Platform for Adaptive Integration of Data-Driven Models And Simulations Into Ultra Short Pulse Manufacturing Systems

Moritz Kröger*[1], Corvin Lasogga[2], Martin Kratz[2], Christian Hinke[1], and Carlo Holly[2]

[1] *Chair for Laser Technology LLT, RWTH Aachen University, Aachen, Germany*
[2] *Fraunhofer Institute for Laser Technology ILT, Steinbachstraße 15, 52074 Aachen, Germany*
**Corresponding author's e-mail: moritz.kroeger@llt.rwth-aachen.de*

The field of simulation and machine learning models for ultra-short pulse (USP) ablation manufacturing has seen significant advancements. These models utilize advanced algorithms and mathematical techniques to predict ablation during material interaction or optimize process parameters, resulting in improved precision and efficiency. However, the integration of these systems into production is lacking due to the monolithic design of control software. This design, characterized by tightly coupled components and limited modularity, makes it difficult to incorporate new technologies and updates. Additionally, the computational power required for simulations and machine learning algorithms exceeds the capabilities of traditional shopfloor computers. To address these challenges, a microservice-oriented software design is proposed. Microservices allow for the flexible inclusion of data-driven models by deploying and scaling individual components independently. This enables quick adaptation to changing processes, materials, and business requirements. Moreover, microservices offer the flexibility to use multiple models and algorithms, resulting in a more agile system. However, the infrastructure requirements of a microservice-style architecture are higher compared to a monolithic architecture. The authors will discuss the general architecture, software components, and advantages and disadvantages of controlling USP production machines with a microservice-style architecture.

DOI: 10.2961/jlmn.2024.01.2002
Keywords: ultrashort pulse laser manufacturing, USP, Ablation, Microservices, CNC, spatial light modulator

1. Introduction

Ultrashort pulse (USP) laser manufacturing has emerged as a disruptive technology that enables the fabrication of intricate structures with unmatched precision. However, the realization of USP manufacturing's full potential is hindered by the lack of a comprehensive framework that seamlessly integrates hardware control, measurement sensors, and process simulations [1].

Ideally, the USP manufacturing process would incorporate a feedback loop which enables the assessment of the current state of the produced part using different sensor technologies and afterwards data analytics, simulation or AI to react dynamically in case of unforeseen errors or deviations. Fig. 1 shows a schematic of this ideal feedback loop. The process would therefore consist of a traditional Computerized Numerical Control (CNC) manufacturing loop which is paused to assess the current parameters of the CNC loop. This pause could be used to tweak, for example, laser parameters to adapt the ablation or could incorporate post-processing steps to ensure a homogenous ablation.

However, the traditional monolithic architectures used in manufacturing control systems struggle to accommodate this dynamic and rapidly evolving requirement of the USP manufacturing processes.

To overcome these limitations, we propose the adoption of microservices, a modular architectural style, to empower USP manufacturing. Microservices provide several advantages, including enhanced flexibility, scalability, and modularity [2, 3], which are essential for accommodating the intricacies and interdependencies inherent in USP manufacturing processes. By breaking down the system into loosely coupled microservices, each responsible for specific functionality, we can efficiently control USP manufacturing hardware, seamlessly use measurement sensors like cameras or 3D metrology sensors and integrate simulations and AI models into the running manufacturing process.

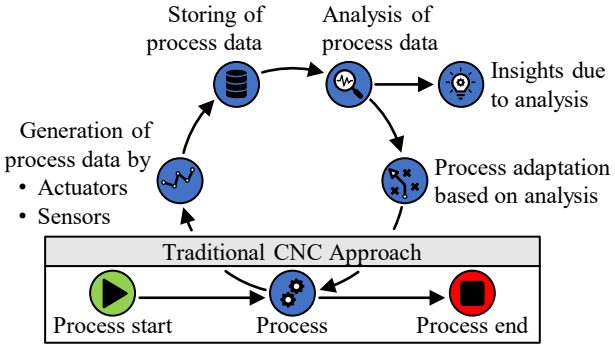

Fig. 1 Ideal in situ process optimization.

2. Traditional Monolithic Approach

Current USP manufacturing systems typically comprise the following components:

- Laser source
- Galvanometric scanner

- Movement system (3D linear axes stage)
- Additional measurement sensors
- Additional support hardware
  (door locks, debris vacuum systems etc.)

The control software the operator controls the USP machine with is typically designed according to a monolithic software approach. Monolithic software is a design pattern where all components of an application are bound together into a single executable system [2]. Fig. 2 shows an example architectural design of a USP control software. The user interface and the controller are bound together into a single software application on the manufacturing machine. The main job of the controller is the coordination of all hardware components of the machine. The controller is also responsible for logging and monitoring the state of the overall system. The controller typically use vendor-specific drivers and interfaces to talk to individual hardware components.

This monolithic style of architecture has the following advantages:

Monolithic systems are simple by design since the development approach is straightforward. All components reside in the same application and the communication between the constituent hardware components is managed by one system, reducing complexity. [4]

The inclusion of all hardware components in a single system also raises the performance since the communication between components is not carried out over a communication bus like Ethernet and less overhead is generated through network calls. However, the capability of the application is limited to the computing power of a single computer. [6]

This design also simplifies the testing of the software since all components and their interaction are tightly coupled together. The workflow of the system is clear which makes the testing process straightforward and easy to implement, resulting in less buggy software and more resilience. [4]

Monolithic software also increases the development speed while remaining under a certain level of complexity [5]. Developers can understand the complete architecture.

Since all software modules are tightly coupled in a single binary the deployment and update of the control software remains simple. Updating to the latest version, therefore, consists of downloading and installing the latest software release.

The single codebase and framework selection also significantly increases team collaboration and the compatibility of functions and classes. All developers and software modules are written in the same language which makes developing the system less complex. Since all team members use the same programming language and software frameworks, the ground for collaboration is set.

In the eyes of the authors, the monolithic software design is the de facto standard in the USP manufacturing domain as well as the machine control domain in general. Reasons for that could be the simplicity of the design paired with very high performance, which is often needed when producing on mass, as well as a very robust system which simply works. Especially in manufacturing, the focus on high reliability is often the dominant aspect of a control system which makes a monolithic system a perfect match for systems which have low variability and high throughput.

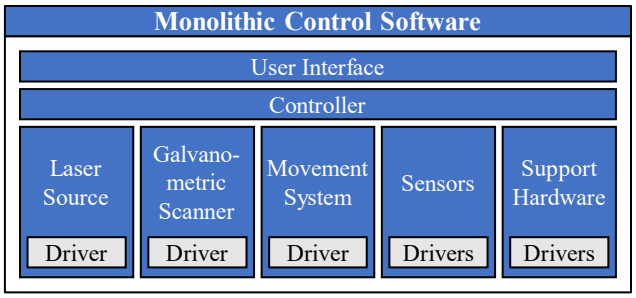

Fig. 2 Example of monolithic software architecture for USP machine control.

But considering that the USP ablation process holds, in theory, the potential for adapting the ablation strategy on demand, the monolithic software design shows significant disadvantages. Especially in research, these shortcomings actively hinder the adoption of simulations and analytics.

Limitation 1: Limited Adaptability

The tight coupling of hardware components and their software modules leads to a control system that is difficult to be changed [2,6]. Changes must be introduced by the control system manufacturer. Especially in research, this leads to complicated procedures where new sensor technologies, analytics or simulations cannot be integrated easily into an existing control system.

The integration of new components into an existing USP machine without the knowledge of the machine manufacturer leads to very complicated systems where information is scattered across multiple subsystems making the system very complex to use. Also, the monitoring and logging architecture is often closed and not extensible.

Limitation 2: Limited Computing Resources

Monolithic systems run on a single computer and are therefore bound to the resources of a single computer [2,6]. Current trends in the USP ablation domain include the integration of high-speed FPGA sensors [7], metrology sensors, etc. which put significant stress on the underlying computing resources. Some systems already need dedicated computers to operate on. The size of the data these sensors generate can also be quite significant which again puts stress on the storage of the underlying computer. Databases or object stores to save measurement data are not implemented up to this date which actively hinders the reuse of data between machining jobs. But especially, this big data is required for the training of AI models. Furthermore, AI and simulations must often run on multiple computing cores or graphics cards. Running multiple simulations or analytic models in parallel, which are in theory required for an adaptive USP process, on a single machine, therefore, becomes quite challenging from a computing perspective.

Limitation 3: Dependency Hell

Monolithic control software put all software dependencies on a single computing device. Installing the drivers of all hardware components on a single computing device can lead to inconsistencies and dependency conflicts [2,6]. The integration of new measurement sensors or algorithms can therefore lead to a complete failure of a manufacturing system just by installing its software dependencies. Again, this hinders the use and especially the dynamic update and deployment of simulations and other analytical systems in a

production environment where reliability and uptime are the keys.

3. Microservice Concept

Microservices have emerged as an architectural software style which allows the building of large distributed applications. The pattern was introduced by several web industry players like Amazon, Netflix, and Uber to conquer the rising complexity and demand of their software systems. A microservice is a single server system that exposes a limited set of functions to its clients, enabling other systems or users to remotely connect to this system and trigger this functionality. This design leads to a decoupling of functionality since the overall application is split into multiple smaller services which communicate over a predefined interface implemented by a lightweight and open messaging protocol. This leads to several advantages:

The modularity and independence of the system increase since each microservice is responsible for a single function. Every service is deployed separately, which enables individual updates as well as the integration of new services into the system on demand [2,3]. In USP manufacturing, the software style ensures that new measurement sensors, functions, or models can be integrated into an existing USP machine on demand.

The use of a well-defined interface description language which defines the API of the services enables a loose coupling of the services. This loose coupling ensures that the individual services can operate independently of each other. It also ensures that the services do not have to run on the same computing device but can be distributed across multiple computing nodes[3]. For usage in USP manufacturing, this leads to the benefit of not being bound to a single computing device but that computing resources can be scattered across the edge and the datacenter.

Microservices enlarge the technology diversity since the usage of open data exchange protocols like HTTP does not lock the user into a specific programming language. Therefore, a developer can choose the most suitable technology stack for a given task without influencing other subsystems. The only applied constraint is the used communication technology [2,3]. In USP manufacturing this has the advantage of being able to adapt the used programming libraries and language to the task at hand. Simulations are often written in C++, while data analytics are typically written in Python. Control software for USP machines, however, is often written in C#. The microservice approach enables a simple data transfer between all these subsystems independently of the used programming languages.

Microservices can be deployed independently on different computing nodes. This leads to fine-grained control over the system resources as well as the underlying dependencies [2,3]. In USP manufacturing, this is especially useful since simulations, data processing and data analytics can have different requirements on the underlying computing hardware. The usage of this architectural pattern ensures that the subsystems do not influence each other.

Due to the decoupling effect of microservices, team autonomy and the possibility for parallel development are enhanced [3]. Multiple actors can work on individual services at the same time. In the domain of USP manufacturing, these actors could even be split across different partners or companies updating the system on demand without disturbing other subsystems.

Every microservice runs in an isolated environment independently. This leads to greater fault isolation since errors in one subsystem do not leak into the other subsystems.

Due to the distributed nature of microservices, monitoring, logging and automated deployment becomes significantly more complex [3]. Therefore, frameworks are required which can aggregate logs and monitor metrics across multiple services to allow operators of the application to assess the system's state [8]. Especially in this infrastructure area, open-source projects like Prometheus, Jaeger, Fluentd or Kubernetes have emerged, enabling the creation of open and connected systems which prevent vendor lock-ins.

4. Microservice Architecture for USP

Microservices inherently solve some of the limitations discussed in Chapter 2. However, to map a USP control system into microservices, the following questions must be answered:

- What functionality must be included in a microservice architecture for manufacturing machines?
- Which infrastructure is needed to run the microservice system on the shop floor?

To answer the question about service functionality, a predefinition of service categories is crucial since it defines the purpose of the services.

Following the functional viewpoints of the Industrial Internet Reference Architecture [9], we define the following service categories:

- Control
- Information
- Application
- Operations

The control category comprises hardware and control services and represents the connection of the USP manufacturing software to the physical system.

Hardware services build a direct bridge to hardware devices like galvanometric scanners, lasers, or movement systems. The services include the vendor-dependent drivers of the underlying hardware devices and expose their functionality as simple service functions. For a movement system, these functions can be, for example, *MoveToPosition* or *GetCurrentPosition*. The hardware services can be split further down into actuator services and sensor services. Hardware services must not communicate with other services since their only purpose is to provide a standardized, vendor-neutral interface to the underlying hardware devices.

Control services compose functionality across multiple hardware Services or other control services. In the context of USP, an example of a control service might be a *Distance Measurement Service* which calculates and realizes the required movement calls for a movement service to place a specific workpiece underneath a distance sensor. Control services, therefore, enable adding smart automation functionality to the system, aggregating the functionality of multiple hardware services and offering it as a simple service call to the user. Control services can also be used to connect to data services to fulfil their function or save data points. An example of such a setup would be a control service that controls the laser power of a given laser by the usage of a

data service that holds experimental data from former ablation experiments.

A data service, therefore, provides the possibility to introduce experimental or simulation data into the system. They are used to hold information in databases and provide unified, predefined, API-based access to the data. A data service, therefore, provides the functionality to store and access data in a predefined schema. An example in the USP domain would be a service that holds laser calibration data from power measurements or an experiment service which stores metadata of laser ablation experiments. Data services are accessible by control services and analytics services.

Analytics services hold specific algorithms like simulations, data analytics or machine learning algorithms. In general, they are abstract services that provide specific data-driven functionality to other clients in the system. Examples of this in the USP domain are ablation simulations or providers of laser parameters that define the needed laser fluence for a target depth or target roughness based on experimental data. The specific split of this functionality into its own service group makes the algorithms exchangeable and manageable by different stakeholders. Also, the services can be located on different computing nodes which works especially well for simulations and machine learning algorithms since these can have significant computing requirements. Data services and analytics services are located in the information category.

The user interface that allows the control of all these services is the job controller. The job controller is located in the application category and coordinates the information and execution flow of the application. This could be for example the feedback loop discussed in Fig 1.

Since the job controller and other services connect remotely to services all systems must be reachable by a remote API. This allows all services to be placed on individual nodes. Hardware services which provide access to computing-intensive sensing devices like 2D metrology sensors or high-speed FPGA Systems for camera or photodiode acquisition [7] could be installed on individual nodes resulting in less coupled systems that can be installed and changed easily. Data storage systems can be placed in a datacenter providing easier and cheaper access to storage. Also, data services could be shared across multiple computing nodes leading to decreased costs and the possibility of building large-scale datasets. Similar effects could take place for computing-intensive analytics services since they have high demands on computer resources. Scaling and sharing these kinds of services across multiple computing nodes through edge or cloud server systems can have significant impacts on the computing cost of the overall system.

The operations category represents the frameworks needed to operate all the other services. An orchestration and networking tool must be in place to organize services across multiple computing nodes. This system must manage the deployment and observe the state of the services as well as the running computing systems of the hardware devices. It needs to be capable of managing installed drivers as well as allowing connected hardware devices to be accessed by the running services. Also, the system must enable the management of networking rules to implement security mechanisms for microservice access.

The other component of the operations category is monitoring and logging. Since microservices are a distributed system a single place for logging, tracing, and monitoring of service calls must be in place. Logging enables the understanding of the current state of a service, while tracing enables following a service call through the complete system to identify the root of an error. Monitoring enables gathering metrics about each service to improve its performance.

An overview of the complete architecture as well as its service categories is shown in Fig 3.

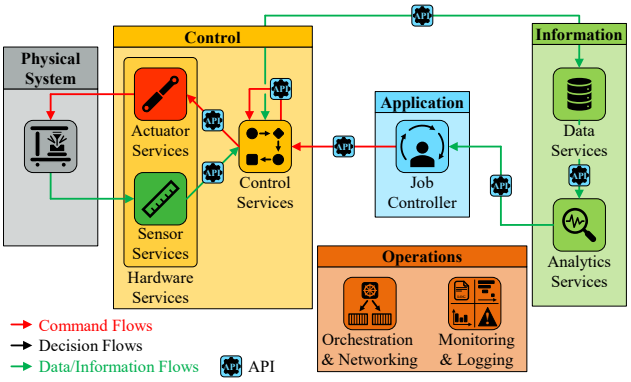

Fig. 3 Overview of the microservice categories.

5. Required Infrastructure

To build a microservice system several technologies must be combined. Especially in research, all these technologies should be open-source to prevent vendor lock-ins or to enforce researchers in reusing system components from colleagues.

The following requirements for the infrastructure arise:

Microservice Requirement 1: Every microservice needs an API and Remote Execution System to allow other services and applications to connect to the system and execute functions.

Microservice Requirement 2: Microservices need to be deployed on computing nodes without the need for the installation of drivers. The technology needs to allow the services and all their dependencies to start automatically. The result would be a flexible system where hardware and analytics services can be placed flexibly on individual computing nodes without the need for a special system setup.

Operation Requirement 1: An orchestration or cluster management tool is needed that enables the scheduling of the microservices on different computing nodes. This tool should be open-source to allow a vendor-independent extension of the system.

Operation Requirement 2: In a microservice architecture, network calls are the standard way of transporting information between the different services. Since services are spread across multiple computing nodes, a network and firewall management system is required for cyber security.

Operation Requirement 3: Microservices are a distributed system. Individual services can fail, or network calls can be routed wrong. Therefore, systems must be in place which aggregate logs of each microservice, collect metrics about the state of every microservice, and enable the tracing of individual calls through several microservices. The system needs to be adaptable to all kinds of services.

There exist several API tools for service-based communication on the market like OpenAPI, gRPC, or OPC/UA,

amongst others. These systems allow designing the remote API of the service in a domain-specific language which is translated into programming classes. These classes can afterwards be implemented. Especially in USP manufacturing, the data that must be transported between the different microservices can be quite large. A message for a galvanometric scanner can comprise several million lines that need to be scanned. Therefore, an API system which transports binary messages and not text messages is required. OpenAPI typically sends JSON-formatted ASCII messages. OPC/UA allows sending ASCII-based JSON or XML messages as well as binary messages. gRPC uses the binary format Protobuf to send messages. Microservice requirement 1 is therefore fulfilled using either OpenAPI, gRPC, or OPC/UA. In this prototype, gRPC is chosen since it supports binary message formats and is open-source. The response time from gRPC, which is for all supported programming languages below 0.5 ms is considered to be sufficient [1].

Microservice requirement 2 focuses on the packaging of the microservices with all their dependencies. For hardware services, this involves especially the inclusion of the hardware drivers. In the web industry, containerization technologies have become the de facto standard for this task. Containers are lightweight, isolated software environments that package applications and their dependencies, enabling consistent and efficient deployment across different computing environments. This technology, therefore, is ideal for use in USP manufacturing systems where sensors, databases, and data analytics need to be changed on demand.

Open-source technologies for containerization are Docker, Podman or containerd. All these tools are conformant to the industry standard of the Open Container Initiative and are therefore interchangeable. Because of the large open-source community and its ease of use, Docker is used for this prototype.

To fulfil Operation Requirement 1, a container scheduling system for computer clusters is needed. In this area, several open-source projects exist like Apache Mesos, Docker Swarm, Nomad, and Kubernetes. Most of these frameworks can be used interchangeably to manage containers in a computer cluster. Also, running an in-depth analysis of all frameworks is not in the scope of this paper. In this prototype, Kubernetes is used due to its large open-source community.

Operation Requirement 2 is a general problem for web-based application development. Therefore, several projects exist that focus on the implementation of a service mesh. A service mesh is a dedicated infrastructure component in a computing cluster that helps to manage communication between microservices as well as implement security controls between services. Typically, a service mesh works by starting a second sidecar container next to the microservice which intercepts the incoming and outgoing traffic of the microservice. The service mesh can control the incoming and outgoing traffic by providing specific filtering or routing rules to the sidecar container. Examples of open-source service mesh projects are Istio, Linkerd and Consul. Again, the complete analysis of all projects would be out of the scope of this paper. However, when designing such a system, a service mesh must be in place to ensure authorized and secure communication between the microservices. In this prototype, Istio was chosen for this task.

Operation Requirement 3 is already partially fulfilled by Kubernetes since it aggregates the logs that are produced by the microservices.

To monitor the metrics of the application, the cloud native computing foundation recommends the use of Prometheus. Prometheus is a system that exposes metrics of a microservice (for example the current laser state or the current laser power consumption) on a specific web server. A Prometheus crawler can collect these metrics in predefined intervals to collect metrics about the running system and aggregate them in a time series database. This database can afterwards be centrally visualized using open-source tools like Grafana or Kibana.

The last requirement is the need for tracing. Since a call to a microservice can make this microservice call another service, a tracing tool enables following the call through all the microservices. Thus, a tracing tool provides valuable inside when working with a microservice system especially when errors occur. At the point of writing, multiple tracing tools like Jaeger or Zipkin exist. Comparing all the alternatives would be out of the scope of this paper. In this prototype, OpenTelemetry is used to create the traces in the individual services, while Jaeger serves for collecting and visualizing the traces.

In summary, we can conclude the required tools for a prototype microservice-based USP manufacturing system as follows:

- gRPC for API design and execution of remote procedure calls,
- Docker for containerization of all services,
- Kubernetes for deploying and orchestrating containers across multiple computing nodes in a cluster,
- Istio for service mesh functionality and network security,
- Kubernetes for log aggregation,
- Prometheus for the collection of microservice metrics and Grafana for their visualization, and
- OpenTelemetry for creating traces and Jaeger for their collection and visualization.

All these tools are used in professional web development and are stable and open-source, resulting in an open and extensible setup for USP research that is based on stable and proven software frameworks.

6. Example Implementation

To demonstrate the flexibility of the system in this paper, multiple use cases are discussed. The use cases are prototypes to showcase the desired feedback loop as shown in Fig. 1. The manufacturing system for the prototype is the laser machine RDX 800 of Pulsar Photonics. The hardware components of the system are:

- Galvanometric scanner:
  Scanlab ExcelliScan with RTC6-Ethernet Controller
- Movement system:
  Aerotech - A3200 with X/Y/Z-movement stages
- Laser source:
  Edgewave FX600 Femtosecond laser
- 3D White light interferometer (WIM):
  GBS smartWLI Extended

- Spatial light modulator (SLM):
  Hamamatsu LCOS-SLM X15223 (self-integrated)

Fig. 4 shows an overview of the used hardware components inside the RDX 800.

6.1 Data acquisition usecase

The first use case focuses on the introduction of a 3D WIM sensor into a USP process to enable the fast generation of surface data of ablated cavities. Compared to the schematic in Fig. 1, we, therefore, demonstrate the acquisition and the storing of process data to allow other researchers to utilize this data.

The use case comprises a material characterization experiment. This type of experiment is typically executed by ablating small cavities on a material target which is afterwards measured by a metrology sensor. The aim of this use case is the automatization of the experiment. It therefore demonstrates the integration of aggregation and database systems into the USP process which is a crucial step for data-driven analytics to generate large amounts of data, for example, for AI algorithms.

The following services are implemented to realize the use case:

- **Hardware Services**
  - Axes Movement Service (represents A3200 controller)
  - Scanner Service (represents RTC6 controller)
  - Laser Parameter Service (represents FX600 Controller)
  - WIM Service (represents GBS controller)
- **Control Services**
  - Coordinate System Management Service
  - Metrology Acquisition Service
- **Information Services**
  - WIM Storage Service
  - Cavity Experiment Service
- **Application Service**
  - Workflow Execution System (JupyterHub)

As discussed in Chapter 4, the task of the hardware services is to expose the functionality of the underlying hardware components to other services. Examples of this are the Axes Movement Service which offers functionality like *MoveToPosition* and the Scanner Service which offers the function *ScanWorkplane* where the next work plane containing hatches and line patterns is streamed to the scanner.

The job of the Coordinate System Management Service is to define a spatial relation between services. The service provides the possibility to move a specific workpiece point into the center of the galvanometric scanner or the WIM. The service is designed in such a way that the system can be extended on demand and coordinate systems for new sensors can be added.

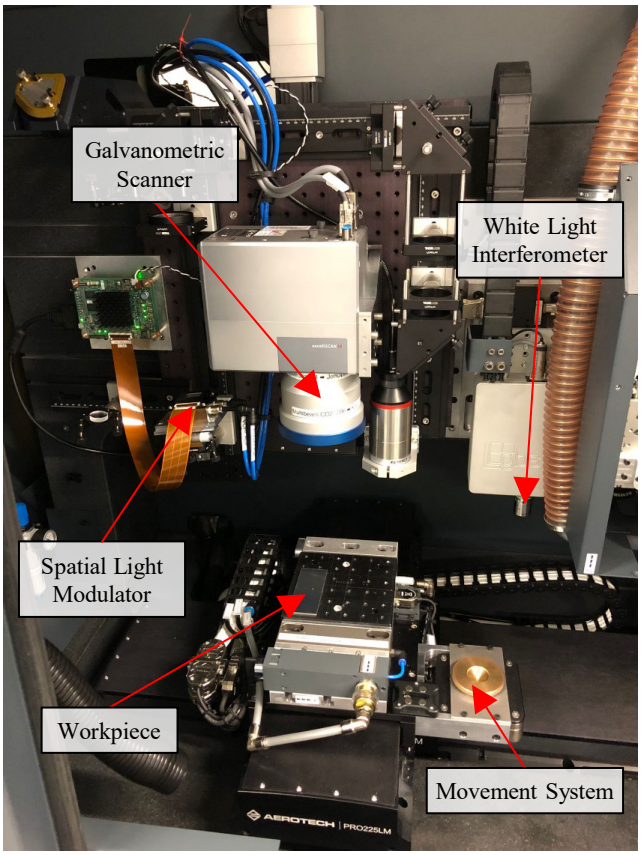

Fig. 4 Overview of the used hardware components.

The purpose of the WIM Storage Service is to provide a standardized endpoint for saving and retrieving WIM measurements in a database. The Metrology Acquisition Service combines the WIM Service and the WIM Storage Service by triggering a metrology measurement through the WIM service and saving it afterwards automatically in the WIM Storage Service.

The Cavity Experiment Service can save an experiment in a database and link it to the corresponding WIM measurements. An experiment entry also provides further metadata about the cavity itself like the used laser power, the scan vectors, the position on the workpiece or the time of execution, amongst others, which might be useful for further analytics of the data.

The whole system is controlled by a Workflow Execution System, called JupyterHub, which can connect to all services remotely via Python and orchestrates the dataflow and the experiment execution. JupyterHub is a browser-based Python execution system. The system runs on a datacenter-node which reduces the loading times of datasets from the WIM storage service and cavity experiment service. Since the system is browser-based, researchers and operators can work on scripts on the shopfloor machine and afterwards switch seamlessly to a laptop in the office.

Overall, the automated experiment execution speeds up the data generation significantly. The setting of laser parameters programmatically via Python requires less than 450 µs. The scanning of a work plane depends on the vector parameters like pulse overlap, jump speed and the number of geometrical primitives in the layer. The cavity experiments under consideration take up to a few seconds. The generation of 3D metrology samples of an ablated cavity depends on the cavity depth. In our experiment run, the maximum

amount of time taken was 108 seconds for a WIM measurement. Combined with the scan time of a cavity, a single ablation experiment takes a maximum of 120 seconds, resulting in at least 720 experiments per 24 hours. Depending on the ablation depth, the cavity size and the scan area of the WIM, the number of experiments per day could rise however significantly.

The integration of the information services into the datacenter leads to less time spent in copying and documenting the experiment. Measured cavities can be analyzed as soon as they are measured. No time is spent copying data on external storage devices resulting in less waiting time for data analysts and researchers.

Overall, the authors experienced a homogenization of tooling in experiment execution since most systems behave similarly and data flows easily from one service to the next. The execution of experiments is sped up significantly and the analysis of the experiment data is homogenized since data is already prepared and stored in a searchable and consumable form, giving data analysts and simulation researchers a platform to build and validate their models. Since data generation is sped up more data samples are generated which provides AI researchers with large enough datasets that have statistical significance leading to more stable and robust models.

6.2 Cavity auto depth use case

The second use case focuses on closing the feedback loop as shown in Fig. 1. It demonstrates an adaptive USP ablation process integrating sensor systems for process state analysis.

This use case focuses on the single-beam postprocessing of multi-beam cavities. Spatial light modulators which are often used for dynamic multi-beam shaping can lead to inhomogeneous energy distributions between spots. While this is not a problem for drilling holes into foils certain applications may require a precise depth of cavities. The used microservices in this use case are:

- **Hardware Services**
  - Axis Movement Service (represents A3200 controller)
  - Scanner Service (represents RTC6 controller)
  - Laser Parameter Service (represents FX600 Controller)
  - WIM Service (represents GBS controller)
  - SLM Service (represents Hamamatsu Controller)
- **Control Service**
  - Coordinate System Management Service
- **Application Service**
  - Workflow Execution System (JupyterHub)

Compared to the first use case, the system also introduces the SLM Service which enables a dynamic switching of the SLM phase mask on the fly. The Axis Movement Service, the Scanner Service, the Laser Parameter Service and the SLM Service are scheduled via Kubernetes on a single-edge device connected to the manufacturing machine. The WIM Service is located on another edge device with larger GPU resources to enable efficient analytics of the WIM camera pictures. The Coordinate System Management Service and the JupyterHub are scheduled on an arbitrary datacenter node. It must be pointed out that no information service is used in this use case. The storing and evaluation of the WIM measurements are executed directly in the Workflow Execution System. However, in the future, the evaluation can be externalized and moved into the information domain.

The Workflow Execution System is therefore responsible for accessing, evaluating, and generating sensor data and the scanner move path.

The experiment is split into multiple phases:

1. Single-beam characterisation

1.1. Initial topology measurement of the process area
1.2. Single-beam cavity ablation
1.3. Topology measurement of ablated cavity
1.4. Characterisation of average ablation depth per repetition

2. Multi-beam ablation

2.1. Initial topology measurement of each process area
2.2. 5 x 3 multi-beam cavity ablation
2.3. Topology measurement of each ablated cavity
2.4. Determination of ablation depth of each cavity

3. Single-beam post-processing, harmonizing ablation depth

3.1. Calculation of single-beam post-processing strategy
3.2. Single-beam post-processing of each cavity
3.3. Topology measurement of each post-processed cavity
3.4. Determination of error and standard deviation

The phases can be mapped on the feedback loop in Fig. 1. The ablations of cavities (steps 1.2, 2.2, and 3.2) represent classic CNC-processing steps. The topology measurements (steps 1.1, 1.3, 2.1, 2.3, and 3.3) correspond to the acquisition of process data. The analysis of the topology measurements (steps 1.4, 2.4, and 3.4) represents the analysis of process data and gives insights into the process. These insights can be used in step 3.1 to adapt the process in situ, which closes the feedback loop as presented in Fig 1.

In phase 1, the single-beam ablation system is characterized for post-processing. The Workflow Execution System moves the process area underneath the WIM with the help of the Coordinate System Management Service and triggers an initial topology measurement of the unprocessed area by a call to the WIM service (step 1.1). Afterwards, the Workflow Execution System generates the scan hatches to produce a squared cavity with a side length of 300 µm, bidirectional hatches and a pulse and line overlap of 70%. Using, the Laser Parameter Service, the repetition rate of the laser is set to 300 kHz and the laser power to 2.86 W. All these process parameters are saved in preparation for the post-processing phase 3. The Workflow Execution System uses the Coordinate System Management Service to move the process area underneath the galvanometric scanner and sends the scan hatches 50 times to the Scanner Service, resulting in the ablation of a single cavity in the process area (step 1.2). Hereafter, step 1.1 is repeated, the process area is moved back underneath the WIM and a topology measurement of the ablated cavity is triggered. The topology measurements are saved as a calibration file for the specific workpiece material and laser parameters (step 1.3). Comparing the mean height of a patch of both topology measurements, the Workflow Execution System calculates the ablation depth of the cavity respectively the average ablation depth per repetition

$(z/n_{rep})_{avg}$ as characterizing process variable of the single beam (step 1.4).

In phase 2, the same procedure is repeated for the multi-beam ablation, in which 15 cavities are ablated simultaneously. In step 2.1, the topology of the unprocessed workpiece area is measured. Here, the Workflow Execution system uses the Coordinate System Management Service to place each of the 15 cavity positions underneath the WIM and calls each time the WIM Service to trigger a topology measurement. The positions and the topology measurements are stored together as a file in the datacenter. Hereafter, the Workflow Execution System calls the SLM Service to switch the phase mask of the SLM. In this use case, the phase mask is calibrated to generate a specific beam path that produces 3 x 5 Gaussian laser spots with a diameter of 40 µm on the workpiece surface. Afterwards, the Workflow Execution System reuses the scan hatches of phase 1 and sends it 100 times to the Scanner Service, resulting in the simultaneous ablation of the 15 cavities (step 2.2). Figure 5 shows a picture of this process.

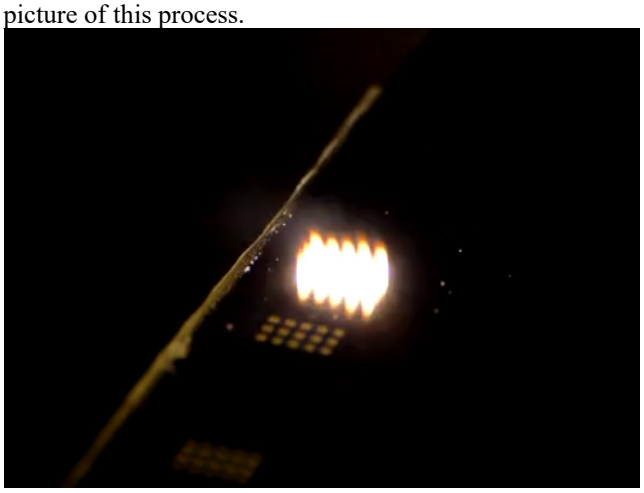

Fig. 5 Picture of the running multi-beam ablation process.

To measure the topology of each ablated cavity, step 2.1 is repeated and the measurements are appended to the file on the datacenter node (step 2.3). Afterwards, the Workflow Execution System compares the corresponding topology measurements and calculates the current depth of each cavity $z_{cur,i}$, applying the same method as in the single-beam ablation (step 2.4).

In phase 3, the cavities, created by the multi-beam setup, are postprocessed, using a single beam. Therefore, the deepest of the 15 cavities is determined and its depth is set as the target depth z_{target} of all other 14 cavities. Thus, the number of required post-processing repetitions can be calculated for each cavity by:

$$(n_{rep})_{post,i} = \frac{z_{target} - z_{cur,i}}{(z/n_{rep})_{avg}} \quad (1)$$

With the help of the Coordinate System Manager Service, the Workflow Execution Service can transform all cavity center positions into the coordinate system of the scanner and can generate shifted post-processing scan hatches for every individual cavity. The shifted post-processing scan hatches are sent to the Scanner Service by the number of repetitions, calculated with Equation 1. The result is a post-processing procedure that reablates every cavity layer by layer. The number of repetitions depends on the current depth of the cavity and the maximum depth of all cavities. Figure 6 shows two pictures of the post-processing of two individual cavities. To analyze the achieved homogenisation of the ablation depth, topology measurements of the re-ablated cavities are executed like in steps 2.1 and 2.3.

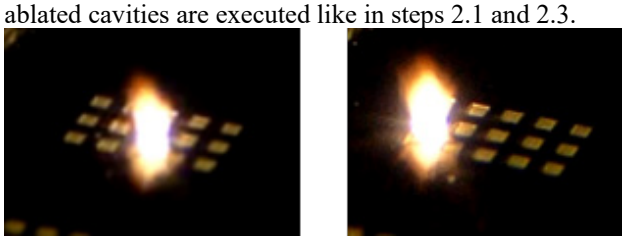

Fig. 6 Pictures of the single beam postprocessing.

The complete evaluation of this experiment is not in scope of this paper.

However, the build use case demonstrates the ability to formulate a closed feedback loop across multiple nodes connected via network and managed in a manufacturing cluster. The architecture can therefore be seen as a platform for further research in reinforcement learning or other feedback driven analytical methods.

The demonstration of the two use cases show that the limitations formulated in chapter 2 can be mitigated.

Limitation 1: The use of open API's like GRPC open up the system. Components can be introduced and rearranged on demand to fulfill multiple purposes (data acquisition or feedback driven manufacturing) with the same set of basic microservices. The introduction of information components which could consist of databases, artificial intelligence models or simulations has been demonstrated. The usage of Kubernetes as the standard deployment method reduces cognitive overhead and homogenizes IT and OT infrastructure systems.

Limitation 2: All use cases where spread across multiple machines ranging from an on-premise datacenter to the shopfloor. Sensor systems with significant higher resource requirements like the WIM Sensor can be placed on individual computing nodes. Databases in the datacenter can be included directly in the process and manufacturing logic can be precalculated on datacenter nodes and afterwards send as streamed commands to the machine on the shopfloor. Theoretically simulations and data driven methods can be scheduled on datacenter nodes as well giving the underlying algorithms enough resources to operate efficiently.

Limitation 3: The usage of containerization tools like Docker isolates the dependencies for individual services. Drivers and other dependencies only influence the microservice that is build for its usage. This allows for plug and play functionality where new hardware or services can be integrated on demand into the system without risking a system crash. The use cases were implemented in docker and show that the containerization does not interfere with the execution of manufacturing steps. Integrating drivers into containers and scheduling them onto nodes with this hardware has been shown and work without interfering other systems.

7. Further use cases and integration of simulation and AI

The proposed system in this paper has been designed to integrate more sophisticated algorithms, data analysis and simulations into the production process. While current use cases show case the flexibility and possible distribution of the system it has not yet been used to integrate these algorithms.

However, simulations can benefit greatly from automatic data acquisition. Especially in this domain it would be possible to run specific calibration experiments automatically on the fly to match and calibrate current calculations. Another integration point for simulations into the system could be during the second use case. Here an initial system calibration has been performed to generate an ablation curve for postprocessing. This step is a perfect example for the inclusion of dynamic on demand simulation during production. The Workflow Execution System could for example propose a postprocessing procedure which is validated on the fly. A simulation service would therefore return the theoretically ablated material based on the current metrology of the system giving the Workflow Execution System more information if the proposed parameters are valid or would destroy the process.

Also, AI systems can easily be included into the architecture. First, the modelling and creation of AI models is supported by the integration of the system into the IIRA Information domain. Data acquisition is streamlined, and data analysis becomes therefore significantly faster since most of the time is normally spend in the data preparation. Secondly models can be included as services. Examples would be an AI model service which could suggest certain laser parameters to ablate a specific depth or create a certain surface roughness.

These ideas mostly influence supervised and unsupervised learning algorithms, where data is generated in batches and afterwards an algorithm is trained on this dataset. Another integration possibility is the introduction of reinforcement learning into the system. Here the Workflow Execution System would be exchanged by a reinforcement learning agent that is allowed to change certain process parameters like laser power, pulse overlap or repeats per layer automatically. By the introduction of certain rule sets (Agent needs to converge to a given target depth as fast as possible, Agent needs to target a specific roughness) the system could decide on a parameter set, execute this parameter set and after wards would receive feedback through the WIM Sensor system. After repeating this process, the machine could in theory teach itself to apply to this ruleset. However typical reinforcement learning experiments often require more than a few thousand episodes to converge. Mapped to the production process which takes around 120 seconds per produced and evaluated layer the training time for 15.000 episodes would already take 500 hours. Again, on demand simulation microservices could be extremely helpful in this situation since the agent could first train in a simulated environment and afterwards switch to a real production system. Similar tactics have been used in the automotive industry for autonomous driving. Also switching from a WIM sensor to a confocal sensor which allows for a much faster determination of processing depth could speed up this process.

8. Benefits and Challenges

The proposed microservice system has several benefits compared to monolithic systems.

First, the system is extensible and open. This is especially useful for researchers since it allows them to introduce new components and subsystems into the machine without patching or “hacking” the initial software. This results in an environment for researchers, data analysists and computer scientist where the system embraces the introduction of new sub-components. Using the protobuf modeling language the designed microservices can also be integrated easier into larger data concepts like the digital shadow reference model as discussed by Judith Michaels et al. [11].

Also, the system complexity decreases since Information domain and control domain services are all deployed in the same way. From an operator view it does not matter if a hardware controller or an analytics controller is deployed into the system.

Using open-source software like GRPC, Kubernetes and Docker vendor lock ins are prevented. The system can be deployed on any other system and users of the system do not need any licensing keys to use the system.

Also, using GRPC as a HTTP2 API the system can be operated completely remotely. Not only can researchers connect to the databases and information systems remotely but also the remote control of the machine is in theory possible. This is especially interesting for remote maintenance.

In this system Kubernetes acts a single point of configuration. Especially when considering that Kubernetes is extensible to a few thousand nodes it is imaginable that the computing clusters and their configuration of complete shop floors could be handled by Kubernetes.

One of the largest disadvantages of this system is the default network-based API. While this API provides enormous flexibility and reconfiguration capabilities it also opens the door for sophisticated cyber-attacks on the system. To operate properly and safely the system components need to be security audited regularly. While network tools like Istio provide security and authorization possibilities further investigation needs to be done to bring the system in a fully production ready state. Compared to a classical monolithic architecture which can be hermetically isolated the disadvantage of this system becomes even clearer.

Another disadvantage of the system that it’s Infrastructure is mostly based on Linux components since this is the de facto standard operating system for web servers and applications. The industry standard in USP manufacturing however is Windows. While it is possible to run most open-source infrastructure software on Windows the support for doing so is mostly limited. The other option is to reimplement most components for Linux. The authors decided for the later.

Another disadvantage compared to monolithic systems is the distributed nature of the system. Tracing calls across multiple computing nodes and microservices becomes very complicated and overwhelming and even with tools like Jaeger, Prometheus and Kubernetes distributed state analysis is still complex. This leads to high introduction and lead up times for new personal. At this point in time the system is not usable without at least Python programming knowledge and some sort of distributed state management knowledge.

9. Summary

In this paper a new architectural style paradigm for the control of USP machines was evaluated: the microservice

pattern. First the current limitations of monolithic control systems where discussed. Afterwards the microservice approach was mapped in depth into the industrial internet reference architecture and a theoretical service categorization system has been proposed. This system proposes a categorization into hardware services, control services, information services as well as application services. Afterwards a practical guideline was given to also provide build an operation framework for microservice which in theory would support such an architecture.

The theoretical system has been demonstrated and tested by several use cases. One use case focused on the generation of large amounts of data points for further research while the other use case show cased a WIM controlled feedback loop across multiple computing nodes and systems. The use cases were successful and proved to be a suitable candidate to ease the integration of AI and simulation into the USP ablation process. Also, the limitations of monolithic architectures were evaluated and compared with the help of these use cases. Afterwards the benefits and challenges of the system have been analyzed.

Acknowledgments

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2023 Internet of Production – 390621612.

The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the framework of Research Campus Digital Photonic Production (project number: 13N15423).

We acknowledge and sincerely appreciate the support and hardware provided by Hamamatsu Photonics K.K. for their contribution to our research project.

References

[1] J. Ryll, J. Holtkamp and S. Eifel: PhotonicsViews, 16, (2019) 65.

[2] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina: "Present and Ulterior Software Engineering" ed. by M. Mazzara and B. Meyer (Springer, 2017) p. 195.

[3] M. Fowler: online available at: https://martinfowler.com/articles/microservice-trade-offs.html, (2015).

[4] S. Newman: "Monolith to Microservices. Evolutionary Patterns to Transform Your Monolith", ed. by C. Guzikowski, A. Young, and N. Barber, (O'Reilly Media Inc., Sebastopol, 2019) p. 15.

[5] M.Fowler: online available at: https://martinfowler.com/bliki/MicroservicePremium.html, (2015).

[6] C. Wunck: Epic Ser Comput, 63, (2019), p. 241-250.

[7] M. Zuric and A. Brenner: Proc. SPIE Vol. 11989, (2022) 119890N.

[8] M.Fowler: online available at: https://martinfowler.com/bliki/MicroservicePrerequisites.html, (2014).

[9] Industry IoT Consortium: online available at: https://www.iiconsortium.org/wp-content/uploads/sites/2/2022/11/IIRA-v1.10.pdf , (2022).

[10] D. Perez and L. Lewis: Phys. Rev. B Condens. Matter, 67, (2003) 184102.

[11] J. Michael, I. Koren, I. Dimitriadis, J. Fulterer, A. Gannouni, M. Heithoff, A. Hermann, K. Hornberg, M. Kröger, P. Sapel, N. Schäfer, J. Theissen-Lipp, S. Decker, C. Hopmann, M. Jarke, B. Rumpe, R. H. Schmitt, and G. Schuh; "A Digital Shadow Reference Model for Worldwide Production Labs" in Internet of Production: Fundamentals, Applications and Proceedings (Springer, 2023) p. 1.

(Received: July 10, 2023, Accepted: December 10, 2023)