FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Agent based Scheduling of Production Line

Carlos Filipe Ramos de Sousa



Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Prof. Mário de Sousa

July 31, 2026

Agent based Scheduling of Production Line

Carlos Filipe Ramos de Sousa

Mestrado em Engenharia Eletrotécnica e de Computadores

July 31, 2026

Resumo

Os sistemas de fabrico modernos requerem mecanismos de coordenação flexíveis e interoperáveis, capazes de se adaptar a requisitos de produção variáveis. As Asset Administration Shells (AASs) fornecem uma forma normalizada de representar "assets" industriais em ambientes de Indústria 4.0, enquanto os sistemas multi-agente oferecem uma abordagem distribuída para a tomada de decisão e coordenação de recursos. Esta dissertação investiga a utilização de agentes baseados em AAS para o escalonamento da produção e coordenação de recursos.

Foi concebida e implementada uma arquitetura multi-agente baseada em AAS para coordenar um cenário de fabrico simplificado, envolvendo produtos, máquinas de processamento, veículos guiados automaticamente (AGV) e uma estação de carregamento. Na arquitetura proposta, estes "assets" foram associados às suas próprias instâncias AAS e aos respetivos agentes de software. Os agentes dos produtos obtêm informação sobre a rota do produto, descobrem recursos compatíveis, reservam máquinas e AGVs, solicitam a execução de tarefas e atualizam os estados dos recursos através de interações REST com a infraestrutura AAS.

O sistema foi implementado como um protótipo baseado em Python, utilizando Eclipse BaSyx, serviços Flask e uma camada de recolha de métricas. Foi também implementada uma abordagem centralizada de escalonamento com recurso ao PyJobShop, de forma a fornecer uma referência para comparação. A avaliação considerou três dimensões experimentais: aumento da carga de produtos, aumento dos tempos de processamento e escalamento proporcional dos "assets". O "makespan" foi utilizado como a principal métrica de desempenho.

Os resultados mostram que a abordagem centralizada baseada em PyJobShop obteve consistentemente valores de makespan inferiores, como esperado devido à sua formulação completa do problema e à ausência de sobrecarga de comunicação em tempo de execução. No entanto, a arquitetura multi-agente baseada em AAS completou com sucesso todos os cenários avaliados e apresentou um comportamento previsível perante o aumento das durações de processamento e o escalamento proporcional. O trabalho demonstra a viabilidade da utilização de agentes baseados em AAS para o escalonamento distribuído da produção, destacando também os custos de desempenho associados à comunicação, gestão de estados e coordenação descentralizada.

Abstract

Modern manufacturing systems require flexible and interoperable coordination mechanisms capable of adapting to changing production requirements. Asset Administration Shells (AASs) provide a standardized way to represent industrial assets in Industry 4.0 environments, while multi-agent systems offer a distributed approach for decision-making and resource coordination. This dissertation investigates the use of AAS-based agents for manufacturing scheduling and resource coordination.

An AAS-based multi-agent architecture was designed and implemented to coordinate a simplified manufacturing scenario involving products, processing machines, automated guided vehicles (AGVs), and a charging station. In the proposed architecture, these assets were associated with their own AAS instances and corresponding software agents. Product Agents retrieve route information, discover compatible resources, claim machines and AGVs, request task execution, and update resource states through REST-based interactions with the AAS infrastructure.

The system was implemented as a Python-based prototype using Eclipse BaSyx, Flask-based services, and a metrics collection layer. A centralized scheduling baseline was also implemented using PyJobShop to provide a reference for comparison. The evaluation considered three experimental dimensions: increasing product workload, increasing processing time requirements, and proportional asset scaling. Makespan was used as the primary performance metric.

The results show that the centralized PyJobShop baseline consistently achieved lower makespan values, as expected due to its complete problem formulation and absence of runtime communication overhead. Nevertheless, the proposed AAS-based multi-agent architecture successfully completed all evaluated scenarios and exhibited predictable behaviour under increasing processing durations and proportional scaling. The work demonstrates the feasibility of using AAS-based agents for distributed manufacturing scheduling, while also highlighting the performance costs associated with communication, state management, and decentralized coordination.

UN Sustainable Development Goals

This dissertation is mainly aligned with Sustainable Development Goal 9: Industry, Innovation and Infrastructure, as it investigates digital technologies for improving manufacturing coordination in the context of Industry 4.0. The proposed architecture uses Asset Administration Shells and multi-agent systems to support interoperable and distributed coordination between products, machines, automated guided vehicles, and support infrastructure. By exploring how standardized digital representations can be used in manufacturing scheduling, this work contributes to the development of more flexible and digitally integrated industrial systems.

The work is also indirectly related to Sustainable Development Goal 12: Responsible Consumption and Production. More effective scheduling and resource coordination can contribute to improved use of manufacturing resources by reducing unnecessary waiting times, supporting better allocation of machines and transport resources, and enabling more informed production execution. Although this dissertation does not directly evaluate environmental indicators such as energy consumption, material waste, or emissions, the proposed architecture provides a foundation that could be extended in future work to include sustainability-oriented decision criteria.

Overall, the contribution of this dissertation to the Sustainable Development Goals is primarily technological and methodological. It supports the development of interoperable and adaptive manufacturing systems, while creating opportunities for future extensions involving resource efficiency, energy-aware scheduling, and more sustainable production planning.

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation.

SDG 12 Ensure sustainable consumption and production patterns.

SDG	Target	Contribution	Performance Indicators and Metrics
9	9.4	The proposed architecture supports the digitalization and modernization of manufacturing systems by using Asset Administration Shells to represent industrial assets and enable interoperable coordination between products, machines, AGVs, and support infrastructure.	Qualitative assessment of AAS-based interoperability and successful execution of coordinated manufacturing scenarios.
	9.5	The dissertation contributes to research and innovation in Industry 4.0 by exploring the integration of AASs and multi-agent systems for distributed manufacturing scheduling and resource coordination.	Implementation of a working prototype; comparison with a centralized PyJobShop scheduling baseline; makespan analysis across experimental scenarios.
	9.b	The work supports the development of technological capabilities for smart manufacturing through the use of open and modular digital technologies such as Eclipse BaSyx, Python, Flask, and REST-based services.	Prototype implementation using open software tools and modular agent-based components.
12	12.2	Improved scheduling and resource coordination can contribute indirectly to more efficient use of manufacturing resources by reducing unnecessary waiting times and supporting better allocation of machines and transport resources.	Makespan comparison under workload scaling, processing time scaling, and asset scaling scenarios.
	12.6	The architecture provides a foundation for future extensions in which sustainability-related information, such as energy consumption or resource usage, could be integrated into AAS submodels and used during scheduling decisions.	Not directly evaluated in this dissertation; identified as a possible direction for future work.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Mário de Sousa, for the opportunity to develop this dissertation under his supervision. His guidance, valuable feedback, and continuous motivation throughout this work were greatly appreciated and contributed significantly to its completion.

I would also like to thank my father for his unwavering financial support during my studies. His assistance made it possible for me to pursue and complete this stage of my academic journey.

I am grateful to the psychologists whose support and guidance during challenging periods helped me persevere throughout my studies.

Finally, I would like to acknowledge the Faculty of Engineering of the University of Porto for providing the academic environment and resources that supported the development of this dissertation.

I am also grateful to everyone who, directly or indirectly, contributed to the completion of this work.

Carlos Filipe Ramos de Sousa

Contents

1 Introduction **1**
- 1.1 Context and motivation . . . 2
- 1.2 Problem Statement . . . 2
- 1.3 Research Questions . . . 3
- 1.4 Methodology Overview . . . 3
- 1.5 Contributions . . . 4
- 1.6 Dissertation Structure . . . 4

2 Background and Related Work **6**
- 2.1 Manufacturing Scheduling . . . 6
- 2.2 Centralized and Algorithmic Scheduling Approaches . . . 7
- 2.3 Multi-Agent Systems in Manufacturing . . . 8
- 2.4 Asset Administration Shells . . . 9
- 2.5 Asset Administration Shell (AAS) Types and Active Asset Behaviour . . . 10
- 2.6 AAS-Based Scheduling and Agent Integration . . . 10
- 2.7 Research Gap . . . 11
- 2.8 Chapter Summary . . . 12

3 Proposed AAS-Based Multi-Agent Architecture **13**
- 3.1 Architecture Overview . . . 13
- 3.2 AAS Infrastructure as the Information Layer . . . 14
- 3.3 Agent Types and Asset Representation . . . 16
- 3.4 Communication Mechanism . . . 17
- 3.5 Scheduling and Coordination Mechanism . . . 19
- 3.6 AAS Type 2 and Type 3 Design Considerations . . . 19
- 3.7 State Consistency and Race Condition Handling . . . 21
- 3.8 Architecture Limitations . . . 21
- 3.9 Chapter Summary . . . 22

4 Case Study Definition and Implementation **23**
- 4.1 Development Environment and Tools . . . 23
- 4.2 Eclipse BaSyx AAS Infrastructure . . . 25
- 4.3 Baseline Scenario . . . 26
- 4.4 AAS Infrastructure and Asset Model Creation . . . 27
- 4.5 Agent Implementation . . . 28
  - 4.5.1 Prototype Code Organization . . . 31
- 4.6 Metrics Collection . . . 32
- 4.7 Centralized Scheduling Baseline . . . 33

4.8 Implementation Challenges . . . 35
4.9 Chapter Summary . . . 36

5 Experimental Evaluation **37**
5.1 Baseline Configuration . . . 37
5.2 Experimental Objectives . . . 38
5.3 Experimental Scenarios . . . 38
5.3.1 Workload scaling . . . 39
5.3.2 Processing time scaling . . . 39
5.3.3 Asset scaling . . . 39
5.4 Benchmark Execution Procedure . . . 40
5.5 Evaluation Metrics and Data Processing . . . 42
5.6 Results and Analysis . . . 42
5.6.1 Product Workload Scaling . . . 43
5.6.2 Processing Time Scaling . . . 45
5.6.3 Asset Scaling . . . 46
5.7 Overall Discussion . . . 47
5.8 Chapter Summary . . . 48

6 Conclusions and Future Work **49**
6.1 Conclusions . . . 49
6.2 Main Contributions . . . 50
6.3 Limitations . . . 50
6.4 Future Work . . . 51
6.5 Final Remarks . . . 52

References **53**

A benchmark_runner.py source code **55**

List of Figures

3.1 Interaction pattern of the proposed AAS-based multi-agent architecture. 14
3.2 Conceptual representation of the AAS as a structured digital representation of an asset. 15
3.3 Communication flow between agents, the AAS infrastructure, and the metrics collector. 18
3.4 Simplified scheduling and coordination workflow followed by the Product Agent. 20

4.1 Detailed execution sequence of a Product Agent during product processing. . . . 30

5.1 Makespan comparison under product workload scaling. 44
5.2 Makespan comparison under processing time scaling. 45
5.3 Makespan comparison under asset scaling. 46

List of Tables

3.1 Agent types and main responsibilities. 16

4.1 Main tools and technologies used in the implementation. 24
4.2 Hardware and software environment used for the experiments. 24
4.3 Manufacturing resources considered in the baseline scenario. 26
4.4 Baseline product route and task durations. 26
4.5 Main AAS submodels used in the implementation. 28
4.6 Main implementation files and responsibilities. 31
4.7 Main event categories recorded by the Metrics Collector. 33
4.8 Representation of the product route in the centralized scheduling baseline. 34

5.1 Baseline resource configuration used in the experiments. 38
5.2 Product route used in the experimental scenarios. 38
5.3 Product workload scaling scenarios. 39
5.4 Processing time scaling scenarios. 39
5.5 Asset scaling scenarios. 40
5.6 Summary of makespan results for the evaluated scenarios. 43

List of Acronyms

AAS Asset Administration Shell
AGV Automated Guided Vehicle
API Application Programming Interface
HTTP Hypertext Transfer Protocol
JSON JavaScript Object Notation
MAS Multi-Agent System
REST Representational State Transfer
SDK Software Development Kit
UTC Coordinated Universal Time

Chapter 1

Introduction

Modern manufacturing systems are increasingly required to operate in dynamic and flexible environments. Production systems must be able to respond to changes in product demand, resource availability, processing requirements, and operational constraints. As a result, traditional centralized control and scheduling approaches may become difficult to adapt when manufacturing systems become more distributed, modular, and interconnected.

Industry 4.0 has introduced several technologies and concepts aimed at supporting digitalization, interoperability, and adaptability in industrial environments. Among these concepts, the Asset Administration Shell (AAS) has gained relevance as a standardized digital representation of industrial assets. An AAS can describe relevant information about an asset, such as its identity, capabilities, state, and interfaces, allowing different software components to access and exchange information in a structured way.

At the same time, Multi-Agent Systems (MASs) provide a distributed approach to decision-making and coordination. In manufacturing contexts, agents can represent products, machines, transport resources, or other production entities. These agents can interact with each other and with digital asset representations to coordinate production execution. This makes MASs particularly relevant for scenarios where decisions are distributed across multiple entities instead of being made exclusively by a central controller.

Manufacturing scheduling is one of the areas where these concepts can be explored. Scheduling involves assigning operations to resources while respecting constraints such as precedence, resource capacity, and resource compatibility. In many cases, centralized scheduling approaches can generate efficient schedules when the full problem is known in advance. However, distributed approaches may offer advantages in terms of modularity, adaptability, and integration with digital asset representations.

This dissertation investigates the use of AAS-based agents for manufacturing scheduling and resource coordination. The work focuses on the design, implementation, and evaluation of an AAS-based multi-agent architecture in which products, machines, automated guided vehicles, and support infrastructure are represented through AAS instances and corresponding software agents.

1.1 Context and motivation

The digital transformation of manufacturing requires systems that can exchange information reliably and coordinate resources in a flexible way. In this context, interoperability is a central requirement. Machines, products, transport systems, and software services must be able to share information about their capabilities and states so that production decisions can be made effectively.

The AAS provides a promising foundation for this type of interoperability. By representing assets through standardized digital structures, AASs can support communication between heterogeneous industrial components. This is particularly relevant in modular manufacturing systems, where resources may be added, removed, reconfigured, or reused across different production scenarios.

MASs complement this idea by distributing decision-making among autonomous entities. Instead of relying on a single centralized component to coordinate all resources, agents can make local decisions based on the information available through the AAS infrastructure. For example, a product agent can retrieve its required process route, discover compatible machines, reserve an available resource, request task execution, and update the state of the system during execution.

The motivation for this dissertation comes from the possibility of combining AAS-based digital representations with agent-based coordination. This combination may support manufacturing systems where scheduling and resource allocation are performed through distributed interactions, while asset information remains accessible through a structured and interoperable information layer.

1.2 Problem Statement

The main objective of this dissertation is to investigate the feasibility of using AASs as a shared information layer for a multi-agent manufacturing scheduling architecture.

To achieve this objective, the following specific objectives were defined:

- Define a simplified manufacturing scheduling scenario involving products, processing machines, Automated Guided Vehicles (AGVs), and support infrastructure.
- Design an AAS-based multi-agent architecture for distributed resource coordination.
- Represent products, machines, AGVs, and the charging station through AAS instances and scheduling-related submodels.
- Implement a Python-based prototype using Eclipse BaSyx, Flask-based services, and Representational State Transfer (REST) communication.
- Implement a metrics collection mechanism to record execution events and calculate makespan.
- Develop a centralized scheduling baseline using PyJobShop for comparison.

- Evaluate both approaches under different experimental scenarios, including workload scaling, processing time scaling, and asset scaling.

1.3 Research Questions

Based on the problem statement and objectives, this dissertation is guided by the following research questions:

- How can AASs be used to represent the information required for distributed manufacturing scheduling?
- How can product, machine, and AGV agents use AAS-based information to coordinate production execution?
- Can an AAS-based multi-agent architecture successfully coordinate manufacturing and transportation resources in the considered scenario?
- How does the makespan of the proposed AAS-based agent implementation compare with a centralized scheduling baseline?
- How does the proposed architecture behave under increasing workload, longer processing durations, and proportional asset scaling?

These questions are addressed through the design, implementation, and experimental evaluation of the proposed architecture.

1.4 Methodology Overview

The methodology followed in this dissertation consists of four main stages.

First, a simplified manufacturing scheduling scenario was defined. The scenario includes products following a predefined route composed of Operation A, transport, and Operation B. The resources include machines capable of performing the required operations, AGVs responsible for transport tasks, and a charging station representing support infrastructure.

Second, an AAS-based multi-agent architecture was designed. In this architecture, each relevant asset is associated with an AAS instance, and software agents use the information exposed by the AAS infrastructure to coordinate execution. Each product is associated with its own Product Agent instance, while machines and AGVs are represented by corresponding resource agents.

Third, the architecture was implemented as a Python-based prototype. Eclipse BaSyx was used as the AAS infrastructure, Flask was used to implement REST-based services, and a metrics collection layer was developed to record runtime events. A centralized PyJobShop baseline was also implemented using the same logical scenario definition.

Finally, experiments were conducted to evaluate the behaviour of the proposed system. The experiments considered product workload scaling, processing time scaling, and asset scaling. Makespan was used as the primary evaluation metric.

During the preparation of this dissertation, generative artificial intelligence tools were used as auxiliary support tools. Perplexity AI was used to help identify potentially relevant related work. OpenAI's ChatGPT was used to support text structuring, language revision, LaTeX assistance, and drafting of text based on the author's ideas. GitHub Copilot, through Visual Studio Code, was used as a programming assistance tool during code development. Artificial intelligence tools were not used as a source of scientific information. The architecture, implementation, experimental work, analysis of results, selection of references, and final content remain the responsibility of the author.

1.5 Contributions

The main contributions of this dissertation are:

- The design of an AAS-based multi-agent architecture for manufacturing scheduling and resource coordination.
- The implementation of a working prototype combining Eclipse BaSyx, Python agents, Flask-based REST services, and a metrics collection layer.
- The representation of products, machines, AGVs, and charging infrastructure through AAS instances and scheduling-related submodels.
- The development of a benchmark execution procedure for evaluating distributed AAS-based coordination under different scenario configurations.
- The implementation of a centralized PyJobShop scheduling baseline for comparison with the proposed agent-based approach.
- The experimental analysis of the proposed architecture using makespan across workload scaling, processing time scaling, and asset scaling scenarios.

1.6 Dissertation Structure

The remainder of this dissertation is organized as follows.

Chapter 2 presents the background and related work relevant to this dissertation. It introduces manufacturing scheduling, centralized and alternative scheduling approaches, MAS, AASs, and recent work combining AASs with production scheduling and agent-based architectures.

Chapter 3 presents the proposed AAS-based multi-agent architecture. It describes the architecture at a conceptual level, including the role of the AAS information layer, the agent types and

responsibilities, the communication mechanism, the coordination workflow, and the main architectural design choices and limitations.

Chapter 4 presents the case study definition and implementation. It describes the development environment, the Eclipse BaSyx AAS infrastructure, the baseline manufacturing scenario, the implemented AAS model, the agent services, the metrics collection mechanism, and the centralized PyJobShop baseline used for comparison.

Chapter 5 presents the experimental evaluation of the implemented system. It defines the experimental scenarios, describes the benchmark execution procedure, presents the evaluation metrics, and analyzes the results obtained for product workload scaling, processing time scaling, and asset scaling.

Chapter 6 concludes the dissertation. It summarizes the main findings and contributions, discusses the limitations of the work, and identifies possible directions for future research.

Chapter 2

Background and Related Work

This chapter presents the background and related work relevant to the dissertation. The work developed in this dissertation is positioned at the intersection of manufacturing scheduling, MASs, and AASs. For this reason, the chapter first introduces manufacturing scheduling and centralized scheduling approaches. It then discusses distributed and agent-based approaches in manufacturing, followed by the role of AASs in Industry 4.0. Finally, the chapter reviews recent work combining AASs, production scheduling, and MASs, and identifies the research gap addressed by this dissertation.

2.1 Manufacturing Scheduling

Manufacturing scheduling consists of planning the allocation and timing of production tasks across available resources, subject to technological and operational constraints. In a typical scheduling problem, products or jobs require a sequence of operations, each operation has a processing duration, and resources such as machines or transport systems have limited capacity. The scheduling decision must therefore determine when each task should be executed and which compatible resource should perform it.

A common objective in manufacturing scheduling is the minimization of makespan, which corresponds to the total time required to complete all jobs in a given scenario. Makespan is widely used because it provides a direct measure of the overall completion time of a production workload. Lower makespan values generally indicate that the available resources are coordinated more effectively.

Several scheduling problem variants exist. In the classical job-shop scheduling problem, each job follows a predefined sequence of operations, and each operation is assigned to a specific machine. In the flexible job-shop scheduling problem, operations may be assigned to one of several compatible machines [1]. This additional flexibility makes the problem more representative of many real manufacturing systems, where multiple machines may be capable of executing the same type of operation. However, it also increases the complexity of the scheduling decision,

since the scheduler must decide both the sequence of operations and the allocation of operations to compatible resources.

Transportation constraints further increase this complexity. In practical manufacturing systems, products often need to be transported between workstations by conveyors, mobile robots, or AGVs. If transportation resources are ignored, the resulting schedule may not accurately represent the real execution of the production system. Products may wait not only for machines, but also for transport resources, and these waiting times can affect makespan, resource utilization, intermediate inventories, and operational performance.

[2] emphasizes that traditional job-shop scheduling studies often neglect travel times or assume unlimited transport resources, which may be inconsistent with engineering practice. Their work addresses the Flexible Job Shop Scheduling Problem with transportation constraints and models the problem using a state transition diagram and a multi-objective function considering makespan, average processing waiting time, and average transportation waiting time. This highlights the importance of considering processing and transportation decisions together.

The scenario considered in this dissertation follows the same general motivation, although in a simplified form. Each product requires Operation A, a transport task, and Operation B. Machines and AGVs are treated as limited-capacity resources, and products must coordinate access to both processing and transportation resources. Therefore, the problem considered in this work includes not only machine allocation and operation precedence, but also the coordination of transport resources.

2.2 Centralized and Algorithmic Scheduling Approaches

Centralized scheduling approaches formulate the scheduling problem as a complete model before execution. In this type of approach, information about all jobs, resources, durations, and constraints is assumed to be available to a central scheduler or optimization algorithm. The scheduler then generates a schedule according to a selected objective, commonly makespan minimization.

Centralized approaches can be highly effective when the problem is well defined and all required information is available in advance. Constraint programming, mixed-integer linear programming, and other optimization methods allow precedence constraints, resource capacity, alternative resource assignments, and operation durations to be modeled explicitly. In this dissertation, PyJobShop is used as a centralized scheduling baseline, representing products as jobs and Operation A, transport, and Operation B as ordered tasks assigned to compatible resources.

However, manufacturing scheduling problems, especially flexible job-shop problems with transportation constraints, are computationally challenging. [3] describes this type of problem as multi-objective and NP-hard, noting that exact approaches are generally more suitable for small-sized cases. As problem size and constraint complexity increase, researchers often investigate alternative algorithmic strategies to obtain high-quality schedules within reasonable computation times.

One important class of alternatives is swarm-intelligence and bio-inspired optimization. [3] proposes an artificial Physarum polycephalum colony algorithm for the flexible job-shop scheduling problem with transportation constraints. Their approach is inspired by the expansion and contraction behaviour of Physarum polycephalum colonies and is designed to improve global search ability and preserve high-quality solutions during iteration. Their computational experiments indicate that the proposed method can outperform several state-of-the-art algorithms in terms of scheduling performance.

This is relevant because it shows that the literature does not rely only on conventional centralized exact methods. Instead, complex scheduling problems have motivated the use of genetic algorithms, particle swarm optimization, ant colony optimization, artificial bee colony algorithms, memetic algorithms, grey wolf optimization, and other swarm or bio-inspired methods [3]. These approaches aim to improve schedule quality, often with makespan as one of the main objectives.

Nevertheless, such approaches remain primarily algorithmic scheduling methods. They usually compute a schedule using a model of the problem, even when the optimization technique is inspired by decentralized natural behaviour. This differs from the architecture proposed in this dissertation, where scheduling and coordination decisions occur during execution through Product Agents, resource agents, AAS state updates, and REST-based interactions.

Therefore, the centralized PyJobShop baseline used in this dissertation should be understood as one reference point rather than as the only possible scheduling alternative. Other algorithmic approaches, such as the swarm-intelligence method proposed by [3], could also be used as scheduling baselines in future work. The focus of this dissertation, however, is not the development of a new optimization algorithm, but the implementation and evaluation of an AAS-based distributed coordination architecture.

2.3 Multi-Agent Systems in Manufacturing

MASs are composed of autonomous software entities, called agents, that interact with each other and with their environment. In manufacturing systems, agents can represent products, machines, transport resources, production orders, or control components. Each agent may be responsible for local decision-making, communication, and coordination.

The use of MASs in manufacturing is motivated by the distributed and modular nature of modern production environments. Instead of concentrating all decision-making in a single central controller, agents can represent different entities and coordinate through interaction. This can support modularity, runtime adaptability, and the ability to react to changes in resource availability or production conditions.

[4] proposed a MAS for dynamic just-in-time manufacturing production scheduling. This type of work shows that agent-based approaches can be used to support dynamic scheduling decisions in manufacturing contexts. [5] also compare centralized and distributed control approaches, highlighting that scheduling complexity and system characteristics influence whether centralized or distributed approaches are more appropriate.

In a manufacturing scheduling context, Product Agents may coordinate the route of individual products, Machine Agents may represent processing resources, and transport agents may represent AGVs or other material-handling resources. These agents can exchange information, request task execution, and coordinate resource allocation during runtime. This differs from conventional centralized scheduling, where the schedule is typically generated before execution.

Distributed agent-based scheduling also introduces challenges. Since multiple agents may make decisions concurrently, the system must handle communication overhead, resource contention, synchronization, and possible race conditions. Local decisions may also be less efficient than schedules generated by global optimization methods. Therefore, agent-based approaches involve a trade-off between adaptability, modularity, and scheduling optimality.

The architecture proposed in this dissertation follows an agent-based coordination model. Each product is associated with its own Product Agent instance. This agent reads route information, discovers compatible resources, claims machines and AGVs, requests operation or transport execution, and updates resource states through the AAS infrastructure.

2.4 Asset Administration Shells

The AAS is a key concept in Industry 4.0 and provides a standardized digital representation of an industrial asset. An asset may be a physical entity, such as a machine, AGV, or charging station, or a logical entity, such as a product order or process description. The AAS organizes asset-related information in a structured way and makes it accessible through standardized interfaces and submodels.

[6] review ten years of AAS development, identifying research opportunities and adoption challenges. Their work shows that the AAS has become an important concept for industrial digitalization, but also that practical adoption still involves challenges related to implementation, interoperability, and integration with existing systems.

An AAS typically contains asset identification information and one or more submodels. Submodels describe specific aspects of the asset, such as technical data, capabilities, operational state, documentation, interfaces, or process-related information. This modular structure allows different categories of information to be represented in a consistent and extensible way.

The AAS is also closely related to the concept of the digital twin. [7] discusses AAS applications in manufacturing and the relationship between AASs and digital twins. In this context, the AAS can be understood as a structured digital representation that supports access to asset information and interaction with other software components.

Interoperability is one of the main motivations for using AASs. [8] presents an AAS-based method for data exchange between manufacturing software applications, showing how AASs can support information exchange between heterogeneous systems. This is relevant for manufacturing environments where production resources, software services, and control components must share information reliably.

In this dissertation, AASs are used as a shared information layer for scheduling and resource coordination. Product AASs store route and execution-related information, Machine AASs expose operation capabilities and state information, AGV AASs expose transport-related information, and the Charging Station AAS represents support infrastructure. Agents use this AAS-based information to coordinate execution during runtime.

2.5 AAS Types and Active Asset Behaviour

Different levels of AAS interaction and behaviour can be considered. In simpler implementations, the AAS mainly acts as a structured information model accessed through Application Programming Interfaces (APIs). More advanced concepts aim to associate the AAS with active behaviour, autonomy, and interaction capabilities.

This distinction is relevant to the work developed in this dissertation. The initial intention of the project was to explore an implementation closer to an AAS Type 3 concept, in which the AAS would not only expose information but also support more active behaviour and interaction. However, due to limited practical documentation and implementation support for this type of AAS, the prototype was developed using a Type 2-oriented AAS API approach.

Sakurada et al. [9] discuss the role of MASs in realizing AAS Type 3. Their work is particularly relevant because it connects the active and autonomous behaviour expected from advanced AASs with the capabilities of MASs. This supports the idea that agents can provide the behavioural layer needed to move beyond passive asset representation.

The implementation developed in this dissertation can therefore be interpreted as a hybrid approach. The AAS infrastructure is used mainly as a structured information layer, which is consistent with a Type 2-oriented implementation. However, active behaviour is provided externally by software agents. These agents read and update AAS information, make coordination decisions, request task execution, and manage resource states during runtime.

This hybrid interpretation is important because it clarifies the role of the AAS in the proposed architecture. The AAS does not directly perform scheduling decisions. Instead, it provides standardized asset information that agents use during coordination. The active decision-making behaviour is implemented in the agent layer.

2.6 AAS-Based Scheduling and Agent Integration

Recent research has explored the use of AASs in production scheduling and MASs. This is relevant because scheduling requires structured information about products, resources, operations, capabilities, states, and constraints. AASs provide a way to represent this information in a standardized and interoperable form.

Siatras et al. [10] investigate the use of AASs for modeling and deploying production scheduling agents within a MAS. Their work is closely related to this dissertation because it explicitly

connects AAS-based modeling with scheduling agents. It shows that AASs can support the organization and deployment of agent-based scheduling components.

Gannouni et al. [11] propose a blueprint description of production scheduling models using AASs. Their work is relevant because it focuses on representing production scheduling models through AAS submodels. This supports the idea that scheduling-related information can be structured within the AAS framework and reused by scheduling systems or decision-support components.

Together, these works show that AASs are increasingly being considered not only as passive digital representations, but also as enablers for scheduling, decision support, and distributed coordination. However, practical implementation still requires design decisions about where decision-making is located, how agents access AAS information, how resource states are updated, and how execution is monitored.

The architecture proposed in this dissertation contributes to this area by implementing a working AAS-based multi-agent scheduling prototype. AAS instances represent products, machines, AGVs, and a charging station, while software agents provide active coordination behaviour. Product Agents use AAS information to discover resources, claim machines and AGVs, execute product routes, and record runtime events.

2.7 Research Gap

The reviewed literature shows that manufacturing scheduling has been widely studied using centralized optimization approaches, flexible job-shop formulations, and transportation-aware scheduling models. It also shows that, due to the complexity of such problems, researchers often investigate alternatives to conventional exact methods, including swarm-intelligence and bio-inspired algorithms. Cai et al. [3] , for example, propose an artificial Physarum polycephalum colony algorithm and report improved scheduling performance for transport-constrained flexible job-shop instances.

At the same time, MASs have been studied as an approach for distributed and dynamic manufacturing coordination, while AASs have been developed as standardized digital representations for Industry 4.0 assets. Recent works have started to connect AASs with scheduling models and scheduling agents [10], [11], and the role of MASs in realizing more active AAS concepts has also been discussed [9].

Despite these contributions, there is still a need to study executable architectures that combine AAS-based asset representation with runtime agent-based manufacturing coordination and compare their behaviour with centralized scheduling baselines. In particular, it is important to understand how AAS submodels can represent scheduling-relevant information, how Product Agents can use this information during execution, and what performance costs arise from distributed resource discovery, claiming, REST communication, and AAS state updates.

The research gap addressed in this dissertation is therefore the design, implementation, and evaluation of an AAS-based multi-agent architecture for manufacturing scheduling and resource

coordination. Unlike algorithmic scheduling approaches that mainly focus on finding high-quality schedules, this dissertation focuses on how scheduling and coordination can be executed through AAS-based interactions between agents. Unlike purely conceptual AAS scheduling models, the work implements and evaluates a working prototype.

The objective is not to demonstrate that the proposed distributed architecture outperforms centralized or swarm-intelligence scheduling algorithms in terms of makespan. Instead, the objective is to evaluate the feasibility of AAS-based distributed coordination and identify the trade-offs introduced by communication overhead, state management, and decentralized decision-making.

2.8 Chapter Summary

This chapter presented the main background concepts and related work relevant to the dissertation. Manufacturing scheduling was introduced as the problem of assigning tasks to resources while respecting precedence, capacity, capability, and transportation constraints. The relevance of transportation-aware scheduling was emphasized, since material movement between production stages can significantly affect the performance of manufacturing systems.

Centralized and algorithmic scheduling approaches were then discussed. While centralized optimization can be effective when the full problem is known in advance, complex flexible job-shop scheduling problems with transportation constraints often motivate the use of alternative algorithms, including swarm-intelligence and bio-inspired methods.

The chapter also presented MASs as a distributed approach for manufacturing coordination, and AASs as standardized digital representations of industrial assets. The relationship between AASs and active asset behaviour was discussed, with particular attention to the role of MASs in supporting more advanced AAS concepts.

Finally, the chapter identified the research gap addressed by this dissertation: the implementation and evaluation of an AAS-based multi-agent architecture for manufacturing scheduling and resource coordination, compared with a centralized PyJobShop scheduling baseline. The next chapter defines the proposed AAS-based multi-agent architecture used to develop and evaluate the proposed system.

Chapter 3

Proposed AAS-Based Multi-Agent Architecture

This chapter presents the proposed AAS-based multi-agent architecture developed for manufacturing scheduling and resource coordination.

The main objective of the architecture is to enable distributed coordination between manufacturing entities without relying on a single centralized scheduling component. Products, machines, AGVs, and support stations are represented by software agents associated with corresponding AAS instances. These agents interact with the AAS infrastructure to retrieve process information, check resource availability, reserve resources, and update execution states.

The chapter first provides an overview of the proposed architecture. It describes the main layers, roles, interactions, and design choices of the proposed approach without focusing on the specific implementation details of the experimental prototype.

3.1 Architecture Overview

The proposed architecture combines AASs with a MAS to support distributed manufacturing scheduling and resource coordination. Each relevant manufacturing entity is represented digitally through an AAS and behaviourally through a software agent. The AAS provides structured information about the asset, while the associated agent interprets this information and performs coordination or execution actions during runtime.

At a high level, the architecture is composed of two main layers:

- **AAS layer:** provides structured digital representations of the assets and exposes information related to capabilities, routes, interfaces, and states.
- **Agent layer:** contains the software agents responsible for coordination, decision-making, task requests, and state updates during execution.

Figure 3.1 presents the core interaction pattern of the proposed AAS-based multi-agent architecture.

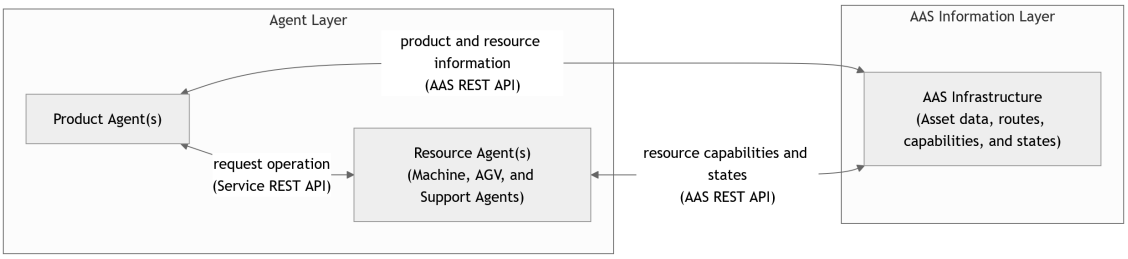


Figure 3.1: Interaction pattern of the proposed AAS-based multi-agent architecture.

The AAS infrastructure acts as a shared information layer between agents. Product Agents use it to retrieve product-related information, identify the next required task, discover compatible resources, and inspect resource states. Resource Agents, such as Machine Agents, AGV Agents, and Support Agents, use the same infrastructure to expose and update information about their corresponding assets.

Task execution is performed through direct service requests between agents. After selecting and reserving a suitable resource, a Product Agent sends an operation or transport request to the corresponding Resource Agent. This separates information access from task execution: the AAS infrastructure stores and exposes asset information, while the agents provide the active behaviour required to coordinate and execute production tasks.

In this architecture, scheduling decisions are not generated by a centralized scheduler. Instead, coordination emerges during execution from interactions between Product Agents, Resource Agents, and the shared AAS-based information layer. The Product Agent plays the main coordination role for each production order by following the product route, selecting compatible resources, reserving them, and requesting task execution.

An evaluation component can also be included to record execution events such as product start, operation start, operation end, transport start, transport end, and product completion. This component does not participate in scheduling decisions; it only records data that can later be used to calculate performance metrics such as makespan.

This separation between asset representation, agent behaviour, task execution, and evaluation logic makes the architecture modular and extensible. Additional products, machines, AGVs, support resources, or submodels can be introduced without changing the general coordination principle of the system.

3.2 AAS Infrastructure as the Information Layer

The AAS infrastructure provides the shared information layer used by the agents during coordination. In the proposed architecture, AASs are used to represent relevant information about manufacturing entities such as products, machines, AGVs, and support infrastructure. This information is accessed and updated by agents during execution.

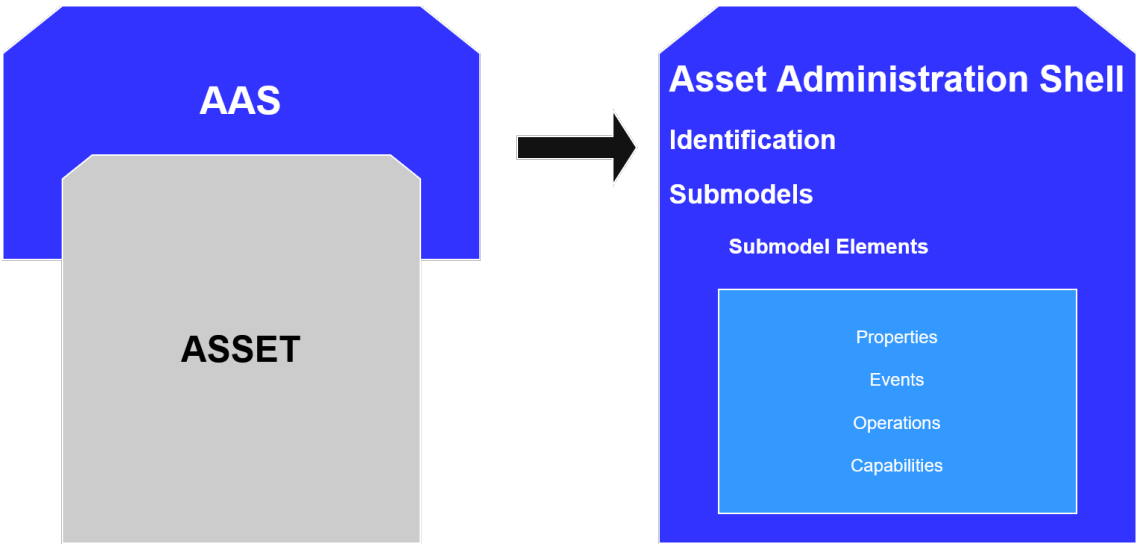


Figure 3.2: Conceptual representation of the AAS as a structured digital representation of an asset.

The purpose of the AAS layer is not to perform scheduling decisions directly. Instead, it provides structured asset information that agents can use to make coordination decisions. Product Agents access the AAS infrastructure to retrieve product routes, discover compatible resources, inspect resource states, and update execution-related information. Resource Agents use the same infrastructure to expose and update information about their corresponding assets.

Figure 3.2 illustrates the conceptual role of the AAS as a structured digital representation associated with an asset. Each AAS contains identification information and submodels that describe specific aspects of the represented asset. In the context of this dissertation, these submodels are used to represent scheduling-related information, such as product routes, resource capabilities, service interfaces, and operational states.

The figure is intentionally abstract. It does not aim to reproduce the complete AAS metamodel, but rather to show how the AAS structure is used in this work as an information layer for distributed scheduling and resource coordination. The concrete AAS instances and submodels implemented in the prototype are described in Chapter 4.

In generic AAS terminology, the asset represented by an AAS can be a physical object, a logical entity, a software component, or another relevant element of an industrial system. In the context of this dissertation, the represented assets correspond to the manufacturing entities considered in the scheduling architecture, namely products, machines, AGVs, and support infrastructure. Since the prototype is implemented in a simulated software environment, these assets are not connected to physical equipment. Instead, their behaviour is represented by Python-based agents and services associated with the corresponding AAS instances.

In the proposed architecture, the AAS layer provides several categories of information required for coordination:

- **Product information:** identifies the product and describes its route and execution-related data. This allows Product Agents to determine which task must be performed next.

Table 3.1: Agent types and main responsibilities.

Agent type	Main responsibilities
Product Agent	Reads product route, selects resources, coordinates execution
Machine Agent	Executes processing tasks and updates machine state
AGV Agent	Executes transport tasks and updates AGV state/location
AGV Station Agent	Represents AGV support/charging infrastructure

- **Resource capabilities:** describe which operations or services a resource can provide. This information is used to identify compatible machines, AGVs, or support resources.
- **Service interfaces:** provide the information required to contact the corresponding resource agents. This allows Product Agents to request operation or transport execution.
- **Resource states:** represent the current availability or assignment state of each resource. These states are used during resource selection, reservation, and release.
- **Support information:** represents auxiliary infrastructure, such as charging or support stations, allowing the architecture to include additional resource types beyond processing machines and AGVs.

This organization allows the agent logic to depend on information exposed through the AAS infrastructure rather than on hard-coded knowledge about the production system. Resource capabilities are used to identify compatible assets, service interfaces are used to request execution, and resource states are used to coordinate reservations during runtime.

By using the AAS infrastructure as a shared information layer, the architecture separates asset representation from agent behaviour. The AAS layer provides structured and accessible information, while the agents provide the active coordination logic required to execute the production workflow.

A design decision in the proposed architecture is the use of an AAS server as a shared information layer, instead of relying only on direct communication between agents. If agents communicated exclusively with each other, information about routes, capabilities, interfaces, and resource states would need to be exchanged and maintained directly between multiple software components. This would increase coupling between agents and make the system more difficult to extend, since each agent would need to know where and how to obtain information from other agents.

3.3 Agent Types and Asset Representation

The proposed architecture relies on software agents associated with the main manufacturing entities represented by AAS instances. Each agent is responsible for monitoring, interpreting, and updating information related to its corresponding asset. Together, these agents coordinate the execution of production orders without relying on a centralized scheduling component.

The main agent types considered in the proposed architecture are summarized in Table 3.1.

The Product Agent is responsible for coordinating the execution of a product order. It retrieves the product route from the Product AAS and determines which task must be executed next. For each step of the route, the Product Agent identifies suitable resources, checks their availability, and requests execution from the corresponding resource agent. In this architecture, the Product Agent plays the main coordination role, since it follows the product through the production process and initiates interactions with machines and AGVs.

The Machine Agent represents a processing resource. It is responsible for executing operation requests received from Product Agents and updating the state of the corresponding Machine AAS. Each Machine Agent is associated with a specific capability, such as Operation A or Operation B. When an operation is requested, the Machine Agent simulates the processing task and returns the result after completion.

The AGV Agent represents a transportation resource. It is responsible for executing transport requests between production stages. When requested by a Product Agent, the AGV Agent performs the transport task, updates its location and state information, and reports completion. The AGV Agent also maintains information related to battery level, which provides a basis for future extensions involving charging decisions.

The AGV Station Agent represents support infrastructure associated with AGV operation. In the current implementation, its role is limited, but it is included in the architecture to support future extensions related to charging station availability and AGV battery management.

Overall, the agent roles are designed to separate coordination, processing, transportation, and evaluation responsibilities. This separation supports modularity and makes it possible to extend the system with additional resource types or more advanced coordination mechanisms.

3.4 Communication Mechanism

Communication in the proposed architecture is based on REST interactions between the agents, the AAS infrastructure, and the execution services associated with manufacturing resources. The AAS server acts as the main information layer, exposing asset data in a structured form and allowing agents to read and update relevant information during production execution.

The agents interact with the Eclipse BaSyx AAS server to access AAS instances and their corresponding submodels. Product Agents retrieve process and route information from Product AAS instances, while resource-related information such as machine capability, machine state, AGV state, location, and battery level is obtained from the AAS instances associated with machines and AGVs.

In addition to accessing the AAS infrastructure, agents may also communicate directly with resource agents when execution of a task is required. For example, after selecting and reserving an available machine, the Product Agent sends an operation request to the corresponding Machine Agent. Similarly, when transportation is required, the Product Agent sends a transport request to an AGV Agent.

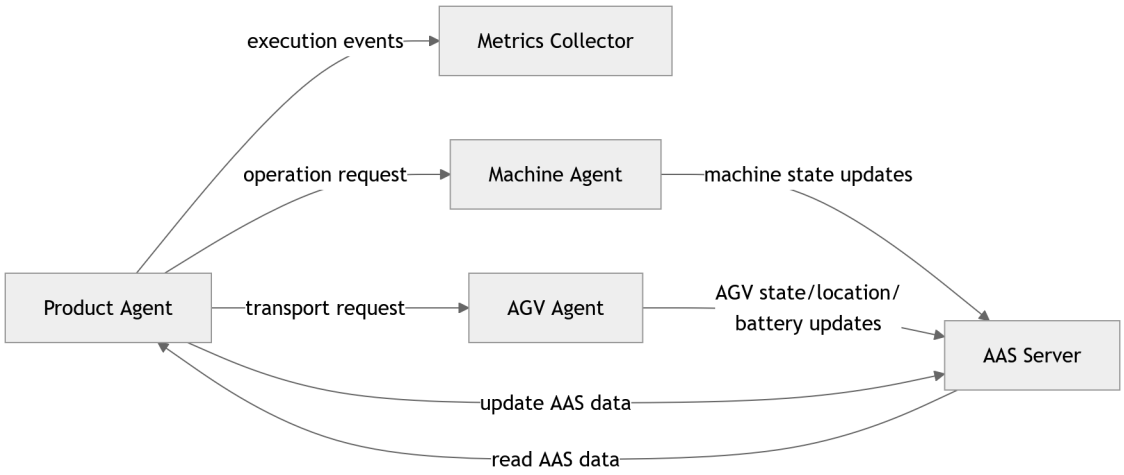


Figure 3.3: Communication flow between agents, the AAS infrastructure, and the metrics collector.

Figure 3.3 summarizes the main communication paths in the proposed architecture. The Product Agent interacts with the AAS server to access and update asset information, sends execution requests to Machine and AGV Agents, and reports execution events to the Metrics Collector.

This communication structure separates information access from task execution. The AAS infrastructure provides a common representation of asset information, while the agents are responsible for interpreting this information and executing the corresponding production or transport actions. This design allows the system to combine standardized asset representation with distributed decision-making.

The communication mechanism used in the proposed architecture can therefore be summarized as follows:

- Agents read process, route, capability, and state information from AAS submodels.
- Agents update AAS submodels when resource states or asset information change.
- Product Agents send execution requests to Machine Agents and AGV Agents.
- Execution events are sent to the Metrics Collector for later analysis.

The communication mechanism therefore follows a hybrid pattern. The AAS infrastructure is used for discovery, shared information access, and state updates, while direct agent-to-agent communication is used for execution requests. This avoids embedding all resource knowledge directly inside Product Agents, while still allowing Resource Agents to provide active task execution services.

3.5 Scheduling and Coordination Mechanism

The scheduling mechanism is driven primarily by the Product Agent. Each Product Agent is responsible for coordinating the execution of the route associated with a product order. The route defines the sequence of tasks required to complete the product, while the AAS infrastructure provides the information needed to identify and reserve suitable resources.

At the beginning of execution, the Product Agent retrieves the product route from the corresponding Product AAS. In the baseline scenario, the route consists of Operation A, a transport task, and Operation B. The Product Agent then discovers the available resources and reads the relevant interface and status submodels exposed through the AAS infrastructure.

For each route step, the Product Agent determines whether the required task is a processing operation or a transport operation. When a processing operation is required, the Product Agent searches for machines capable of executing the corresponding operation. After selecting an available machine, the Product Agent updates the machine state in the AAS to reserve the resource, confirms the reservation, and sends an execution request to the corresponding Machine Agent.

Transportation tasks follow the same general logic. When transport is required, the Product Agent searches for an available AGV, updates the AGV state to reserve it, confirms the reservation, and sends a transport request to the AGV Agent. The AGV Agent then performs the transport task and updates relevant information such as location and battery level.

During execution, relevant events are sent to the Metrics Collector. These events include product start, operation start, operation end, transport start, transport end, and product completion. The collected data is later used to calculate makespan and analyze the behaviour of the proposed scheduling approach.

Figure 3.4 presents the scheduling and coordination workflow followed by the Product Agent during the execution of a production order.

3.6 AAS Type 2 and Type 3 Design Considerations

The architecture developed in this dissertation was initially motivated by the idea of exploring an implementation closer to an AAS Type 3 concept. In such a concept, the AAS would not only expose information about an asset but also support more active interaction and behaviour.

In practice, the implemented prototype adopts a Type 2-oriented AAS interaction model, since the AAS infrastructure is used mainly to expose and update structured asset information through APIs. This choice was influenced by the availability of implementation tools, documentation, and practical support for this type of interaction model.

Nevertheless, the architecture is not a purely passive AAS-based system. Active behaviour is provided by the software agents, which use AAS information to coordinate execution. Product Agents make resource-selection decisions, reserve resources, request operations and transport tasks, and update execution states during runtime. Machine and Transport Agents provide execution capabilities and update asset-related information.

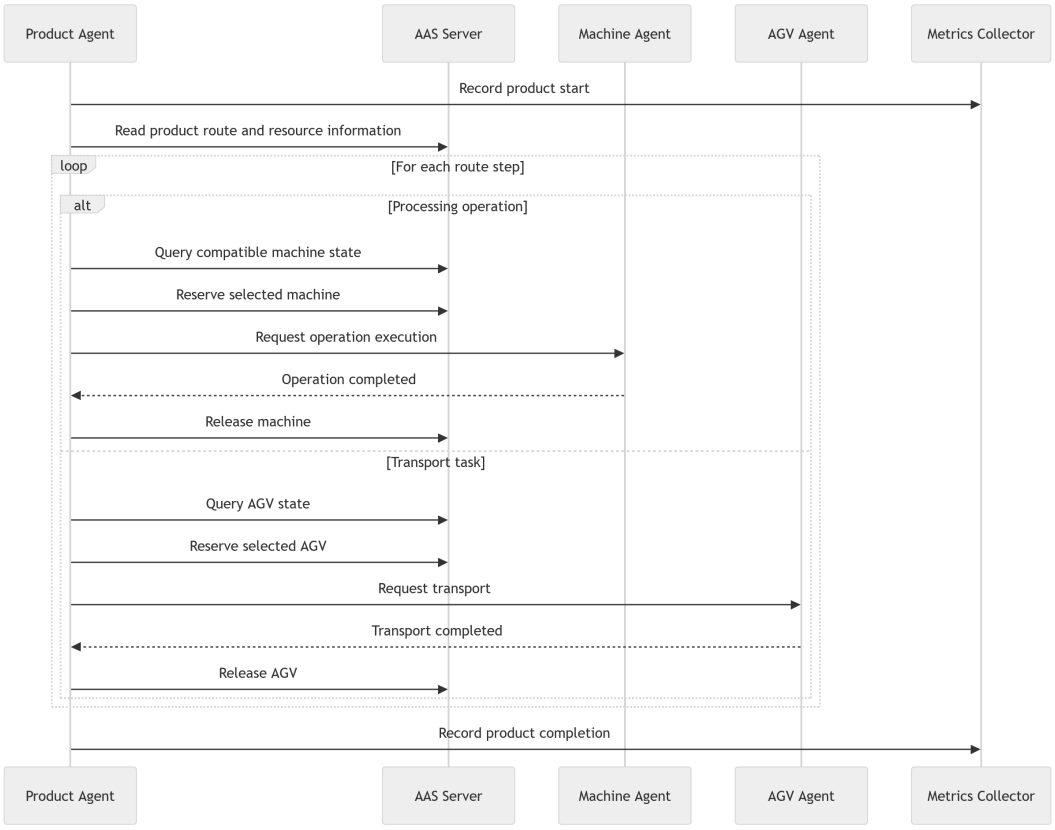


Figure 3.4: Simplified scheduling and coordination workflow followed by the Product Agent.

For this reason, the proposed system can be interpreted as a hybrid between a Type 2-oriented implementation and the behavioural expectations associated with more advanced AAS concepts. The AAS layer provides standardized information access, while the agent layer provides active behaviour and distributed coordination.

This design choice preserves the main objective of the work: investigating how AAS-based asset representations can support distributed manufacturing scheduling. At the same time, it avoids depending on immature or insufficiently documented Type 3 implementation mechanisms.

3.7 State Consistency and Race Condition Handling

Since the proposed architecture relies on distributed agents, state consistency is an important aspect of the coordination mechanism. Multiple Product Agents may attempt to access the same manufacturing resource at similar times, particularly when several products are being processed concurrently. Without a control mechanism, this could lead to conflicting assignments, such as two products being allocated to the same machine or AGV simultaneously.

To reduce this risk, the architecture uses state information stored in the AAS submodels associated with each resource. Before assigning a task, the Product Agent reads the current state of the required resource. If the resource is available, the Product Agent updates the state to indicate that the resource is busy and associated with a specific product. The state is then checked again before execution proceeds.

This read-update-check sequence acts as a simple reservation mechanism. A resource is only used after its state has been updated and confirmed. When the requested operation or transport task is completed, the resource state is updated again to indicate that it is available for future assignments.

Although this mechanism reduces the likelihood of conflicting assignments, it does not provide the same guarantees as a transactional locking mechanism. Since the implementation is based on REST communication and a custom Python-based agent system, race conditions may still occur under certain timing conditions if multiple agents attempt to update the same resource state simultaneously. This limitation is important when interpreting the experimental results, since synchronization and communication overhead can influence the makespan obtained by the agent-based approach.

In this dissertation, the implemented mechanism is considered sufficient for evaluating the feasibility of AAS-based distributed scheduling in the proposed scenarios. However, future versions of the architecture should investigate more robust consistency mechanisms, such as explicit locks, transactional updates, resource reservation services, or negotiation protocols between agents.

3.8 Architecture Limitations

The proposed architecture provides a modular approach for distributed manufacturing scheduling using AAS-based agents. However, several limitations should be acknowledged.

First, the architecture relies on repeated REST communication with the AAS infrastructure. This introduces communication overhead, especially when the number of products increases and more agents interact with shared resources. As a result, the execution time of the agent-based approach may be influenced not only by processing and transport durations, but also by the frequency of state reads, updates, and confirmations.

Second, the reservation mechanism used to coordinate resource access is based on resource state updates rather than on a formal transactional locking mechanism. Although the read-update-check sequence reduces the probability of conflicting assignments, it does not provide the same guarantees as a dedicated synchronization or reservation service.

Third, the architecture was evaluated using simplified manufacturing scenarios with deterministic processing and transport times. More complex scenarios, involving heterogeneous products, dynamic events, machine failures, or more detailed AGV behaviour, would be required to fully assess the applicability of the architecture in more realistic manufacturing environments.

Another limitation concerns the AAS interaction model adopted in the implementation. The original intention was to explore a more active AAS Type 3-oriented approach. However, due to limited practical documentation and implementation support, the prototype was developed using a Type 2-oriented AAS API combined with external software agents. Therefore, the active behaviour of the system is not embedded inside the AAS itself, but implemented in the agent layer. This makes the architecture closer to a hybrid approach, where AASs provide standardized asset information and agents provide the runtime behaviour and coordination logic.

These limitations do not invalidate the proposed approach, but they define important directions for future improvements and provide context for the interpretation of the experimental results.

3.9 Chapter Summary

This chapter presented the proposed AAS-based multi-agent architecture for manufacturing scheduling and resource coordination. The architecture combines AASs, software agents, REST-based communication, and a metrics collection component to support distributed production execution.

The AAS infrastructure provides structured information about products, machines, AGVs, and stations, while the software agents use this information to coordinate operations and transport tasks. The Product Agent plays the central coordination role by following the product route, selecting compatible resources, reserving them, and requesting task execution.

The chapter also described the communication mechanism and the state consistency approach used to reduce conflicting resource assignments. These architectural decisions form the basis for the implementation described in the next chapter.

Chapter 4

Case Study Definition and Implementation

This chapter presents the concrete case study and implementation used to instantiate the architecture described in Chapter 3. While the previous chapter introduced the proposed AAS-based multi-agent architecture at a conceptual level, this chapter describes how that architecture was implemented and evaluated in a simplified manufacturing scheduling scenario.

The chapter first presents the development environment and tools used in the prototype. It then explains the role of the Eclipse BaSyx AAS infrastructure, defines the baseline manufacturing scenario, describes the implemented AAS model, and presents the agent services developed for product coordination, machine execution, and transport execution. Finally, the metrics collection mechanism, centralized PyJobShop baseline, and main implementation challenges are discussed.

4.1 Development Environment and Tools

The prototype was implemented using a set of tools selected according to availability, interoperability, and suitability for rapid development of an AAS-based MAS. Since the objective of the dissertation was to evaluate the feasibility of the proposed architecture rather than to develop a new AAS infrastructure from scratch, preference was given to tools that were open, documented, and compatible with REST-based integration.

Python was selected as the main programming language due to its simplicity, extensive library ecosystem, and suitability for rapid prototyping. It was used to implement the Product Agents, supporting scripts, benchmark execution logic, and data processing components. Flask was used to implement lightweight REST services for resource agents and the Metrics Collector, allowing the different components of the prototype to communicate through Hypertext Transfer Protocol (HTTP) requests.

Eclipse BaSyx was selected as the AAS infrastructure. This choice was motivated by its availability as an open-source implementation, its support for creating and exposing AAS instances and submodels, and its suitability for server-side AAS deployment. In addition, BaSyx is commonly

Table 4.1: Main tools and technologies used in the implementation.

Tool or technology	**Purpose**
Python	Implementation of agents, scripts, experiment logic, and data processing.
Flask	Development of REST services for Machine Agents, AGV Agents, and the Metrics Collector.
Eclipse BaSyx	AAS server and infrastructure for storing and exposing AAS instances and submodels.
REST API	Communication between agents, services, and the AAS infrastructure.
PyJobShop	Implementation of the centralized scheduling baseline.
Podman / Podman Desktop	Container management for running the AAS infrastructure.
Visual Studio Code	Development environment used during implementation.

used in research and practical examples related to AAS-based systems, making it an appropriate choice for implementing the information layer of the proposed architecture. Its REST-based access mechanisms allowed the Python agents to retrieve product routes, inspect resource capabilities, read resource states, and update execution-related information during runtime.

Container-based execution was used to run the AAS infrastructure locally. Docker Desktop was initially considered and tested, but Podman Desktop was adopted for the final implementation as an open-source alternative for managing the required container services. Visual Studio Code was used as the development environment for writing, testing, and organizing the prototype code.

Finally, PyJobShop was selected to implement the centralized scheduling baseline. This tool made it possible to define the same logical scheduling scenario in a centralized form and solve it using a constraint-programming-based scheduling approach. The resulting baseline provides a reference for comparing the makespan obtained by the proposed distributed AAS-based multi-agent implementation.

The main tools and technologies used in the implementation are summarized in Table 4.1.

The experiments were executed locally on a personal computer. Table 4.2 summarizes the hardware and software environment used for the benchmark runs. All experimental scenarios were executed using the same environment, allowing the observed results of the proposed implementation and the centralized baseline to be compared under consistent conditions.

Table 4.2: Hardware and software environment used for the experiments.

Component	**Specification**
Processor	Intel Core I5 11600k
Memory	16 GB DDR4
Operating system	Windows 10 Pro
Container environment	Podman / Podman Desktop
Python version	Python 3.11.9
AAS infrastructure	Eclipse BaSyx

4.2 Eclipse BaSyx AAS Infrastructure

Eclipse BaSyx was used as the AAS infrastructure for the prototype. In this work, the BaSyx AAS server provides the runtime environment where AAS instances and submodels are created, stored, accessed, and updated. The server exposes REST-based interfaces that allow the agents to retrieve product routes, inspect resource capabilities, read resource states, and update execution-related information.

The choice of Eclipse BaSyx was motivated by both practical and literature-based considerations. From a practical perspective, the prototype required an AAS infrastructure capable of hosting AAS instances and submodels locally, exposing them through an API, and allowing external software agents to read and update asset information during execution. BaSyx satisfied these requirements and provided an open-source implementation suitable for integration with the Python-based agents developed in this work.

The selection was also supported by the comparative analysis presented in [12], which evaluates publicly available AAS Software Development Kits (SDKs) and code generation tools for working with the AAS v3 metamodel and AAS API. In that comparison, BaSyx Java V2 was identified as one of the most complete options regarding support for both client-side and server-side functionality. This made it a suitable choice for the present work, where the AAS infrastructure had to provide server-side access to asset information while remaining compatible with external client applications.

The BaSyx server does not perform scheduling decisions. Instead, it acts as a shared information layer between the agents. Scheduling and coordination behaviour is implemented in the agent layer, while BaSyx provides the AAS-based representation of products, machines, AGVs, and support infrastructure. Product Agents use the AAS infrastructure to discover resource capabilities and states, while Resource Agents use it to expose and update information about their corresponding assets.

This design choice follows the architecture described in Chapter 3. Rather than hard-coding all resource knowledge inside the Product Agents or relying only on direct communication between agents, the AAS server provides a common structured access point for asset information. This supports modularity and interoperability, since new resources can be represented by adding or updating their corresponding AAS instances and submodels.

In the implementation, BaSyx was deployed locally using container-based infrastructure. The agents and supporting services interacted with the AAS server through REST requests. This allowed the prototype to reproduce the intended architectural separation between the information layer and the behavioural layer: BaSyx provided the structured AAS representation, while the software agents provided the active coordination and execution logic.

Table 4.3: Manufacturing resources considered in the baseline scenario.

Resource type	Quantity	Capability
Machine A	2	Operation A
Machine B	1	Operation B
AGV	2	Product transportation
AGV station	1	AGV support/charging

4.3 Baseline Scenario

The baseline scenario considered in this dissertation represents a simplified manufacturing environment composed of processing machines, transportation resources, and product orders. The scenario is intentionally abstracted in order to focus on the coordination and scheduling aspects of the system rather than on the physical details of a specific industrial process.

The definition of the baseline scenario was informed by [1]. However, the scenario used in this dissertation does not directly reproduce those instances. Instead, a simple configuration was adopted as a starting point and adapted to include transportation resources and the AAS-based agent architecture developed in this work.

The baseline configuration consists of three processing machines and two automated guided vehicles. Two machines are capable of performing Operation A, while one machine is capable of performing Operation B. The operation names are kept abstract, since the objective is not to model a specific manufacturing process, but rather to evaluate how resources can be coordinated through an AAS-based multi-agent architecture.

The system processes products that follow a predefined route. In the baseline case, each product requires the sequential execution of Operation A followed by Operation B. After completing Operation A, the product must be transported by an available AGV to the resource responsible for Operation B. This introduces both processing and transportation constraints into the scheduling problem.

The manufacturing resources considered in the baseline scenario are summarized in Table 4.3. The product route and the baseline durations associated with each step of the product route considered in the baseline scenario is presented in Table 4.4.

Each machine can process only one product at a time, and each AGV can transport only one product at a time. Therefore, products compete for shared resources during execution. In addition, the precedence relation between operations must be respected, meaning that Operation B cannot start before Operation A has been completed and the product has been transported to the

Table 4.4: Baseline product route and task durations.

Step	Task	Required resource	Duration (s)
1	Operation A	Machine A1 or Machine A2	3
2	Transport	AGV	4
3	Operation B	Machine B1	2

appropriate resource.

Transport operations are abstracted as time-based tasks executed by AGV agents. Although the physical layout of the factory is simplified, the inclusion of AGVs allows the scheduling problem to represent both production and material-handling decisions.

The baseline scenario is later extended in the experimental evaluation by varying the number of products, processing durations, and system size. These variations allow the behaviour of the proposed architecture to be analyzed under increasing production demand, longer processing requirements, and proportional scaling of manufacturing assets.

4.4 AAS Infrastructure and Asset Model Creation

Before executing the scheduling experiments, the AAS infrastructure was populated with the digital representations of the assets involved in the manufacturing scenario. These assets included products, machines, AGVs, and the AGV station. Each asset was represented by an AAS instance containing information relevant to the scheduling and coordination process.

The creation of AAS instances was automated ustoing Python scripts. These scripts defined the required AAS shells and associated submodels, and then submitted them to the BaSyx AAS server through REST requests. This allowed the experimental scenarios to be initialized consistently before each execution.

For products, the AAS representation included information about the product identity, process plan, and route plan. The process plan defined the operations required to complete the product, while the route plan specified the sequence of tasks to be executed. In the baseline scenario, this sequence consisted of Operation A, a transport task, and Operation B.

For machines, the AAS representation included capability and state information. The capability information described which operation type each machine could perform. The state information indicated whether the machine was available or busy and, when busy, which product was assigned to it. This information was used by Product Agents when selecting and reserving processing resources.

For AGVs, the AAS representation included state, location, and battery-related information. The state indicated whether the AGV was available or busy, while the location and battery level were used to represent the condition of the transportation resource during execution. Although AGV movement was simplified as a time-based transport task, these fields provide a basis for extending the system with more detailed transport and charging behaviour.

The AGV Station AAS was included to represent support infrastructure related to AGV operation. In the current implementation, its role was limited, but it was maintained in the architecture to support future extensions involving AGV charging, station availability, or transportation resource management.

The AAS creation process ensured that each experiment started from a known initial state. Machines and AGVs were initialized as available, products were initialized with their predefined routes, and the required submodels were made accessible to the corresponding agents.

Table 4.5: Main AAS submodels used in the implementation.

AAS instance	Submodel	Main information
Product AAS	ProcessPlan	Route plan, status, and assigned machine
Machine AAS	Interface	Operation capability and service endpoint
Machine AAS	Status	Machine state and assigned product
AGV AAS	Interface	Transport capability and service endpoint
AGV AAS	Status	AGV state, location, and battery level
Charging Station AAS	Interface	Charging capability and service endpoint
Charging Station AAS	Status	Station state and location

Table 4.5 summarizes the main submodels used in the implementation. Interface submodels describe the capability offered by each resource and the endpoint used to request execution, while status submodels store state information used by Product Agents during resource selection and reservation.

The Product AAS differs from the resource AASs because it stores the route and execution-related information required by the Product Agent. In the implemented scenarios, the product process plan includes the ordered tasks that must be executed and metadata updated during execution.

During the initial planning of the work, the objective was to explore an implementation closer to an AAS Type 3 concept, in which the AAS would not only expose asset information but also support more active interaction and behaviour. However, due to the limited availability of practical documentation, examples, and implementation support for this type of AAS, the prototype was developed using a Type 2-oriented approach based on the AAS API provided by Eclipse BaSyx.

As a result, the implemented system can be interpreted as a hybrid between a Type 2 and a Type 3 AAS-based approach. The AAS infrastructure is used mainly as a structured information layer, exposing AAS instances and submodels through REST-based APIs, which is consistent with a Type 2 implementation. However, the active behaviour expected from a more advanced AAS-based system is provided externally by the software agents. These agents read and update AAS information, make coordination decisions, request task execution, and manage resource states during runtime.

This design choice allowed the work to preserve the central idea of AAS-based distributed coordination while relying on available and stable implementation tools. The AASs provide standardized asset representations, while the agent layer supplies the active decision-making and execution behaviour required for the scheduling scenario.

4.5 Agent Implementation

The agent-based system was implemented as a set of software components responsible for coordinating and executing the manufacturing scenario. Each agent type was associated with a specific role in the architecture and interacted with the AAS infrastructure to access or update asset information.

In the implemented system, each product order is associated with its own Product Agent instance. Therefore, there is not a single Product Agent responsible for coordinating all products. Instead, each Product Agent independently follows the route of one product, retrieves the required information from the corresponding Product AAS, and attempts to coordinate the resources required to complete that product.

At the start of execution, each Product Agent retrieves the corresponding Product AAS and reads the process plan and route plan. The route is then executed step by step. For each step, the Product Agent determines whether the task corresponds to a processing operation or a transport operation. Based on this information, it searches for compatible machines or AGVs through the information exposed by the AAS infrastructure.

When a processing operation is required, the Product Agent searches for machines capable of performing the required operation. In the implemented scenario, this corresponds to Operation A or Operation B, depending on the product route. After identifying a compatible resource, the Product Agent reads its current state from the corresponding Machine AAS. If the machine is available, the Product Agent attempts to reserve it by updating the state value to indicate that the machine is busy and assigned to that product.

After reserving the machine, the Product Agent sends an execution request to the corresponding Machine Agent. The Machine Agent was implemented as a Flask-based REST service that receives operation requests, simulates the required processing task, and returns a completion response after the operation finishes. Once the operation is complete, the Product Agent records the corresponding metric event and releases the machine by updating its state in the AAS.

Transport tasks are coordinated in a similar way. When transport is required, the Product Agent searches for an available AGV using the state information stored in the AGV AAS instances. After reserving an AGV, the Product Agent sends a transport request to the corresponding AGV Agent. The AGV Agent, also implemented as a Flask-based REST service, simulates the transport task, updates relevant information such as location and battery level, and returns a completion response.

The AGV Station Agent was included to represent the support infrastructure associated with AGV operation. In the current implementation, its role was limited, since detailed charging behaviour was not the main focus of the baseline scenario. However, the station representation was maintained to support future extensions involving charging requests, station availability, and more advanced AGV management.

Figure 4.1 presents the detailed execution sequence followed by a Product Agent during product processing. The diagram shows the interaction between the Product Agent, the AAS server, the Machine and AGV Agents, and the Metrics Collector.

This implementation allows multiple Product Agents to execute concurrently, each attempting to coordinate the resources required by its own product. As a result, resource coordination emerges from the interaction between product-specific agents and the shared AAS-based resource state.

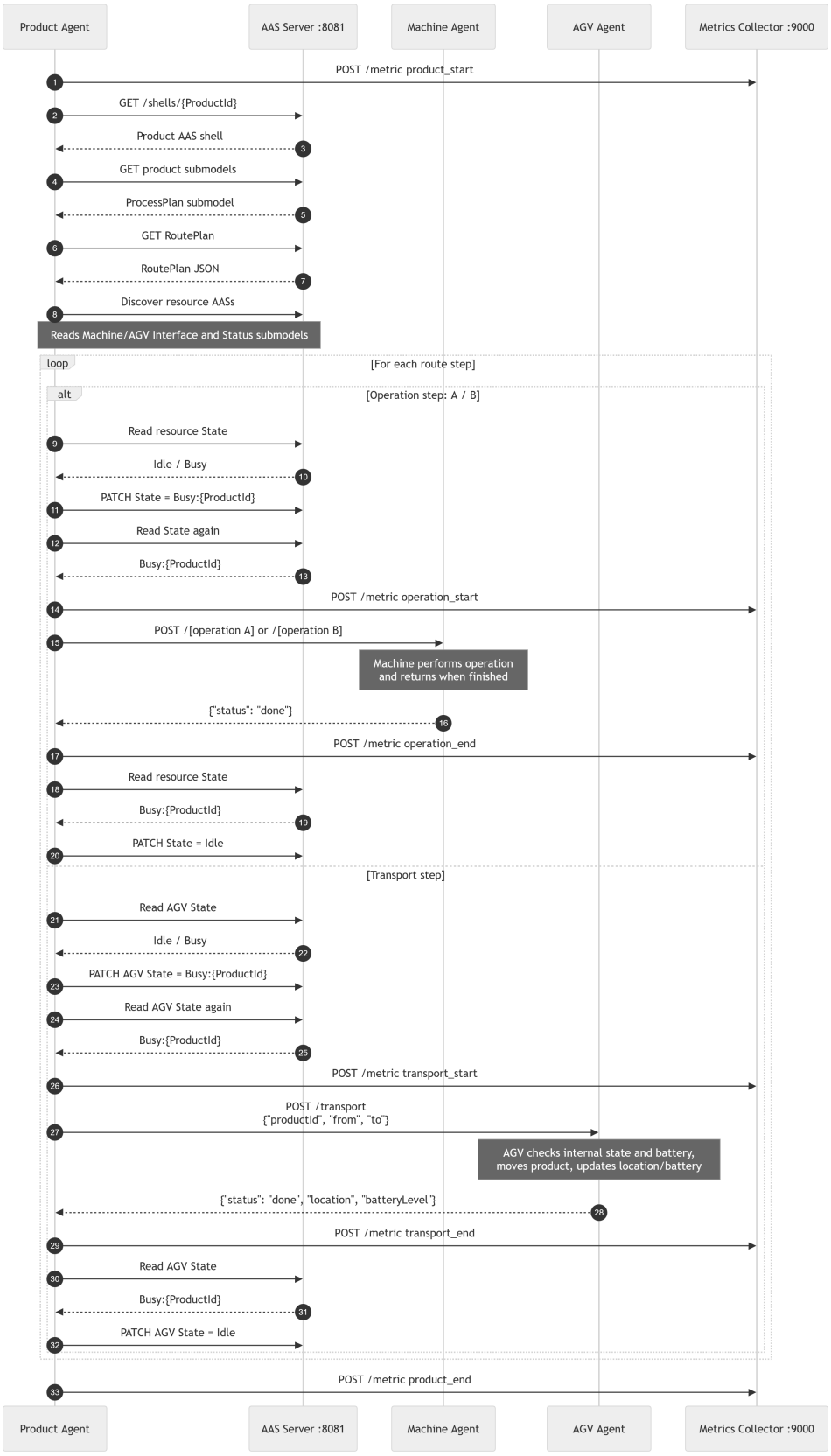


Figure 4.1: Detailed execution sequence of a Product Agent during product processing.

Table 4.6: Main implementation files and responsibilities.

File or component	Main responsibility
`config.json`	Defines runtime parameters, including product count, resource configuration, service ports, and duration multiplier.
`aas_client.py`	Provides helper functions for creating and updating AAS shells and submodels in BaSyx.
`machine_aas.py`	Define the AAS structures used to represent resources.
`product_aas.py`	Define the AAS structures used to represent products.
`createaasdemo.py`	Populates the BaSyx environment with the AAS instances and submodels required for each scenario.
`machine_agent.py`	Implements the Flask service representing a machine resource.
`agv_agent.py`	Implements the Flask service representing an AGV resource.
`charge_station_agent.py`	Implements the Flask service representing the charging station.
`product_agent_multi.py`	Implements the decentralized Product Agent coordination logic.
`launch_all.py`	Start the resource agents and Product Agent processes according to the selected configuration.
`launch_products.py`	Start the Product Agent processes according to the selected configuration.
`metrics_collector.py`	Receives runtime metric events and stores them in a line-delimited JSON event log.
`benchmark_runner.py`	Orchestrates complete benchmark runs, including service startup, AAS refresh, product execution, metrics processing, and shutdown.
`pyjobshop_compare.py`	Implements the centralized PyJobShop scheduling baseline used for comparison.

4.5.1 Prototype Code Organization

At a high level, each experimental run follows four main stages. First, the BaSyx AAS environment is started. Second, the `createaasdemo.py` script creates or refreshes the AAS structure for the machines, AGVs, charging station, and products. Third, the runtime agents are launched as independent Flask services. Finally, Product Agents read their route plans from the AAS, discover available resources, reserve them, execute operation or transport steps, and release the resources after completion.

Table 4.6 summarizes the main implementation files and their responsibilities.

Annex A shows the source code of the `benchmark_runner.py` script, which orchestrates the complete execution of a benchmark run.

Each resource has an AAS shell with submodels describing its interface and current status. For example, a machine AAS contains information about its capability, service endpoint, and state. The capability identifies which operation the resource can perform, the endpoint indicates

where the corresponding service can be accessed, and the state indicates whether the resource is available or currently assigned to a product.

4.6 Metrics Collection

To evaluate the behaviour of the proposed AAS-based multi-agent manufacturing system, a lightweight metrics collection layer was implemented alongside the agents. The purpose of this layer is not to control execution, but to observe the distributed workflow and record the events required to reconstruct each experimental run. The collected data is primarily used to measure makespan, which is used as the main evaluation metric in the experimental evaluation.

The metrics infrastructure follows a push-based model. During execution, agents emit JavaScript Object Notation (JSON) events to a dedicated Flask-based Metrics Collector through a common `POST /metric` endpoint. Each received event is appended to a line-delimited JSON file, `metrics.jsonl`, using one JSON object per line. If an event does not already contain a timestamp, the collector adds a Coordinated Universal Time (UTC) timestamp before writing it. A write lock is used to avoid file corruption when several agents report events concurrently.

This design keeps the measurement layer decoupled from the manufacturing logic. Product Agents, Machine Agents, and AGV Agents can all report events using the same interface, while the Metrics Collector remains independent of the specific resource type. The metrics calls are implemented as best-effort operations: if the collector is unavailable, the agents continue their normal execution. This avoids turning monitoring into a source of failure or artificial blocking during the benchmark execution.

The main event categories recorded by the Metrics Collector are summarized in Table 4.7. These events include product lifecycle events, resource-claiming events, operation events, transport events, and service events.

After a benchmark run, the raw event stream is processed by the benchmark execution script to produce a summary file, `metrics_summary.json`. This summary includes the makespan of the run, calculated from the difference between the earliest product start event and the latest product completion event. Additional indicators, such as throughput, completion count, and estimated resource utilization, can also be derived from the event stream. However, makespan remains the main metric used for the experimental comparison in this dissertation.

This event-based approach is appropriate for the proposed system because execution is decentralized. There is no single scheduler with complete operational knowledge during runtime. Instead, each Product Agent independently discovers, claims, and releases resources through interactions with the AAS infrastructure. Recording distributed events allows the execution trace to be reconstructed after the run without changing the decision-making logic of the agents.

The collected data also supports comparison with the centralized scheduling baseline. The PyJobShop-based model generates schedules for the same scenario definitions and reports the corresponding makespan. This makes it possible to compare the completion time obtained through

Table 4.7: Main event categories recorded by the Metrics Collector.

Category	Events	Purpose
Product lifecycle	`product_start`, `product_end`	Identify the product execution interval.
Resource claiming	`claim_attempt`, `claim_success`, `claim_failed`	Record resource reservation attempts.
Operation execution	`operation_start`, `operation_end`, `operation_error`	Record processing task execution.
Transport execution	`transport_request`, `transport_start`, `transport_end`, `transport_error`	Record AGV transport tasks.
Service events	`service`	Record REST service path, status code, agent name, and request duration.

decentralized AAS-based agent execution with the result obtained using a centralized scheduling approach.

One limitation of the current metrics implementation is that the event stream is intentionally simple and does not enforce a strict schema beyond the JSON structure. Some event names may be emitted by different agents with slightly different meanings. For example, transport-related events may be associated with both the Product Agent coordinating the route and the AGV Agent executing the transport task. For this reason, event interpretation must consider the available fields, such as `productId`, `agv`, `from`, `to`, and response data. Future versions could introduce explicit source fields and correlation identifiers to make the reconstruction of individual workflow stages more robust.

4.7 Centralized Scheduling Baseline

To evaluate the behaviour of the decentralized AAS-based agent implementation, a centralized scheduling baseline was developed using PyJobShop. The purpose of this baseline is not to replace the proposed architecture, but to provide a reference scheduling approach for comparison under the same scenario definitions.

The centralized baseline represents the manufacturing scenario as a scheduling problem in which all products, tasks, resources, durations, and precedence relations are known before solving. In contrast to the agent-based implementation, where each Product Agent coordinates its own product through resource discovery and AAS-based state updates, the centralized model builds a complete representation of the problem and submits it to an optimization solver.

In the PyJobShop model, each product is represented as a job composed of three ordered tasks: Operation A, a transport task, and Operation B. These tasks correspond to the same logical route

Table 4.8: Representation of the product route in the centralized scheduling baseline.

Task	Compatible resources	Constraint
Operation A	Machines capable of Operation A	First task in the route
Transport	AGVs	After Operation A completion
Operation B	Machines capable of Operation B	After transport completion

used in the agent-based implementation. Precedence constraints ensure that Operation A must finish before transport starts, and that transport must finish before Operation B starts.

Each task is assigned to a compatible resource. Operation A may be assigned to any machine capable of performing Operation A, transport tasks may be assigned to any available AGV, and Operation B may be assigned to any machine capable of performing Operation B. Therefore, the centralized baseline can be interpreted as a flexible job-shop scheduling model with transportation resources, since some tasks can be assigned to alternative eligible resources.

The model uses the same logical resource structure and task durations as the agent-based benchmark. This ensures that both approaches are evaluated using equivalent scenario definitions. The main difference is that the centralized baseline solves the scheduling problem as a complete model, while the proposed architecture performs coordination during execution through distributed interactions between Product Agents, resource agents, and the AAS infrastructure.

The optimization objective of the centralized model is to minimize makespan. In this context, makespan corresponds to the completion time of the last scheduled product. This is the same primary metric used to evaluate the AAS-based agent implementation.

The centralized scheduling problem is solved using the OR-Tools backend through PyJobShop. In practical terms, the problem is formulated using interval-based tasks, alternative resource assignments, precedence constraints, and a makespan-minimization objective. The solver then generates a centralized schedule satisfying the modeled constraints.

After solving, the script produces a summary file containing the generated schedule, solver information, and the corresponding makespan. When an agent-based benchmark summary is available, an additional comparison file can be generated to compare the centralized and decentralized results. This comparison is mainly based on makespan, although other indicators may also be derived from the recorded data.

This baseline is useful because it provides a centralized reference point for interpreting the performance of the proposed architecture. The centralized model benefits from a complete problem formulation and does not include the REST communication, AAS state updates, resource-claiming retries, or distributed coordination overhead present in the agent-based implementation. Therefore, differences between the two approaches should not be interpreted only as scheduling inefficiency, but also as a consequence of the architectural differences between centralized optimization and decentralized AAS-based execution.

4.8 Implementation Challenges

Several implementation challenges were encountered during the development of the prototype. These challenges were mainly related to the distributed nature of the architecture, the use of AAS submodels as the shared information layer, and the need to collect comparable execution data.

One of the main challenges was maintaining consistent resource states while multiple Product Agent instances executed concurrently. Since each product is coordinated by its own agent, several Product Agents may attempt to claim the same machine or AGV at similar times. To reduce conflicting assignments, the implementation uses a simple reservation mechanism based on AAS state updates. A Product Agent reads the resource state, updates it to a busy state associated with the product, and then reads the state again to confirm the reservation. Although this approach was sufficient for the evaluated scenarios, it does not provide the same guarantees as a transactional locking mechanism.

Another challenge was the communication overhead introduced by the repeated REST interactions with the AAS server and the Flask-based resource services. Each route step may require multiple operations, including reading AAS submodels, updating resource state, confirming reservations, sending execution requests, recording metrics, and releasing resources. This overhead is part of the implemented decentralized architecture and must be considered when comparing it with the centralized PyJobShop baseline, which does not include the same runtime communication and state-management costs.

The organization of identifiers and submodels was also an important implementation issue. Product Agents needed to reliably locate the correct Product AAS, Machine AAS, AGV AAS, and associated Interface and Status submodels. For this reason, consistent naming conventions were used for AAS instances, submodels, resource identifiers, and service endpoints. These conventions made it possible to initialize different scenarios and execute repeated benchmark runs in a consistent way.

Metrics collection introduced additional challenges. The Metrics Collector was designed as a lightweight Flask service receiving JSON events from several agents. Since multiple agents may report events at nearly the same time, the collector appends events to a line-delimited JSON file using a write lock to avoid file corruption. The event stream also had to remain sufficiently flexible to record product lifecycle events, resource-claiming events, operation events, transport events, and service events.

Finally, aligning the agent-based implementation with the centralized scheduling baseline required care. The PyJobShop model had to represent the same logical scenario, including Operation A, transport, Operation B, compatible resources, processing durations, transport durations, and precedence constraints. This alignment was necessary to ensure that differences in the experimental results could be interpreted as differences between coordination approaches, rather than differences in scenario definition.

4.9 Chapter Summary

This chapter described the implementation of the proposed AAS-based multi-agent architecture. The system was implemented as a Python-based prototype using Eclipse BaSyx as the AAS infrastructure and Flask-based REST services for agent communication and metrics collection.

The chapter first presented the development environment and the main tools used in the implementation. It then described how AAS instances and submodels were created for products, machines, AGVs, and the charging station. These AAS representations provided the information required by Product Agent instances to discover resources, check their availability, reserve them, and coordinate product execution.

The implementation of the agents was then described, with particular attention to the fact that each product is associated with its own Product Agent instance. Machine and AGV Agents were implemented as execution services responsible for simulating processing and transport tasks. A Metrics Collector was also implemented to record distributed execution events and support the calculation of makespan.

The chapter also presented the centralized PyJobShop baseline used for comparison. This baseline models the same manufacturing scenario as a centralized scheduling problem with precedence constraints, alternative resource assignments, and a makespan-minimization objective.

Finally, the main implementation challenges were discussed, including state consistency, REST communication overhead, identifier management, metrics collection, and alignment between the decentralized implementation and the centralized baseline. The next chapter presents the experimental setup used to evaluate both approaches.

Chapter 5

Experimental Evaluation

This chapter presents the experimental setup used to evaluate the proposed AAS-based multi-agent architecture and the centralized scheduling baseline. The objective is to define the scenarios, parameters, execution procedure, and evaluation method used to compare both approaches under equivalent conditions.

The experiments were designed to analyze the behaviour of the proposed architecture under different production conditions. In particular, the evaluation considers the effect of increasing the number of products, increasing processing requirements, and scaling the number of products and resources. These dimensions allow the system to be evaluated in terms of completion time, scalability behaviour, and execution stability.

Both the AAS-based agent implementation and the centralized PyJobShop baseline were evaluated using the same logical manufacturing scenario. This ensures that differences in the results are primarily related to the coordination approach rather than differences in scenario definition.

5.1 Baseline Configuration

The baseline configuration is based on the manufacturing scenario defined in Chapter 4. The system contains processing resources, transportation resources, and product orders. Each product follows a predefined route composed of Operation A, a transport task, and Operation B.

In the baseline scenario, Operation A can be performed by two machines, while Operation B can be performed by one machine. Transportation between processing stages is performed by AGVs. Each machine can process only one product at a time, and each AGV can transport only one product at a time.

The baseline resource configuration is summarized in Table 5.1.

The product route used in the baseline configuration is summarized in Table 5.2.

Processing and transportation durations were kept deterministic during each experimental run. This was done to ensure controlled comparison between the AAS-based agent implementation and the centralized PyJobShop baseline. In the processing time scaling experiment, these base

Table 5.1: Baseline resource configuration used in the experiments.

Resource type	Quantity	Capability
Machine	2	Operation A
Machine	1	Operation B
AGV	2	Transport
Charging station	1	AGV support

durations were multiplied by predefined factors in order to evaluate the effect of longer operation requirements.

5.2 Experimental Objectives

The main objective of the experimental evaluation is to assess the behaviour of the proposed AAS-based multi-agent architecture when coordinating manufacturing and transportation resources. The experiments focus on the ability of the system to execute product routes, allocate compatible resources, and complete all products under different scenario configurations.

The centralized PyJobShop baseline is used as a reference approach. Since the baseline solves the scheduling problem using a centralized model, it provides a useful comparison point for evaluating the makespan obtained by the decentralized agent-based implementation.

The experimental evaluation addresses three main aspects. First, workload scaling is used to analyze how the system behaves as the number of products increases. Second, processing time scaling is used to observe how longer operation durations affect the execution of both approaches. Third, asset scaling is used to evaluate the behaviour of the system when products and resources are increased proportionally.

The experiments are not intended to prove that the agent-based approach outperforms centralized scheduling. Instead, they are intended to evaluate whether the proposed architecture can coordinate resources successfully and to quantify the difference between decentralized execution through AAS-based agents and centralized schedule generation.

5.3 Experimental Scenarios

Three groups of experiments were defined to evaluate the behaviour of the proposed architecture under different conditions: product workload scaling, processing time scaling, and asset scaling.

Table 5.2: Product route used in the experimental scenarios.

Step	Task	Required resource
1	Operation A	Machine capable of Operation A
2	Transport	AGV
3	Operation B	Machine capable of Operation B

Table 5.3: Product workload scaling scenarios.

Scenario	Number of products
W1	1
W2	2
W3	3
W4	4
W5	8
W6	16
W7	32

5.3.1 Workload scaling

The first experiment evaluates product workload scaling. In this experiment, the number of products is increased while the number of resources remains constant. This allows the behaviour of the system to be analyzed as more Product Agent instances compete for the same machines and AGVs. The baseline resource configuration is kept unchanged during this experiment.

The product quantities considered in the workload scaling experiment are shown in Table 5.3.

5.3.2 Processing time scaling

The second experiment evaluates processing time scaling. In this experiment, the number of products is kept constant, while the durations of the processing and transportation tasks are multiplied by predefined factors. This allows the evaluation to distinguish between increases caused by additional products and increases caused by longer task durations.

The processing time multipliers considered in this experiment are shown in Table 5.4.

5.3.3 Asset scaling

The third experiment evaluates asset scaling. In this experiment, the number of products and the number of resources are increased proportionally. The objective is to observe whether the architecture maintains stable behaviour when the system size increases while preserving a similar relationship between workload and resource availability.

The asset scaling scenarios are shown in Table 5.5.

Together, these experiments provide a controlled evaluation of the proposed architecture under increasing workload, longer processing requirements, and proportional system growth.

Table 5.4: Processing time scaling scenarios.

Scenario	Number of products	Duration multiplier
P1	3	1
P2	3	2
P3	3	3
P4	3	4
P5	3	5

Table 5.5: Asset scaling scenarios.

Scenario	Asset multiplier	Description
A1	1	Baseline system size
A2	2	Products and resources scaled by factor 2
A3	4	Products and resources scaled by factor 4
A4	8	Products and resources scaled by factor 8

5.4 Benchmark Execution Procedure

The benchmark execution procedure was designed to run the decentralized manufacturing scenario in a controlled and repeatable way. Its purpose is to initialize the required services, execute a configured number of product workflows, collect runtime events, and generate summary data for later analysis.

The procedure is implemented in the `benchmark_runner.py` script, which acts as the orchestration component for each experimental run. Before execution begins, the script reads the runtime configuration from `config.json`. This configuration defines the number of products, the number of AGVs , the machine configuration, the port allocation, and the operation duration multiplier. As a result, the same benchmark procedure can be reused across different scenario sizes and experimental conditions.

At the beginning of each run, previously generated runtime files are removed. These include the metrics event file and the AAS resource cache. This step prevents stale data from previous runs from affecting the current execution. The benchmark then starts the resource agents, including the Machine Agents, AGV Agents, and the charging station service. These components are launched before the Product Agents so that the required resources are available when product execution begins.

After launching the resource agents, the benchmark waits for the expected services to become available by checking their configured network ports. This readiness check reduces the likelihood of Product Agents starting before the resource services are prepared to receive requests. Once the required services are available, the AAS structure is refreshed using the AAS creation script (`create\ac{AAS}demo.py`). This step updates the Asset Administration Shells with the current resource information and service endpoints, ensuring that Product Agents can discover valid machines, AGVs , and support services during execution.

The Metrics Collector is started immediately before launching the Product Agents. This timing limits the recorded event stream to the benchmark execution window while still capturing product lifecycle events. The Product Agents are then launched according to the configured product count. Each Product Agent independently retrieves its route, discovers compatible resources through the AAS infrastructure, claims machines or AGVs when available, executes the required tasks, and emits metrics throughout the process.

During execution, the benchmark runner monitors progress by reading the metrics stream and waiting until the expected number of product completion events has been observed. If all

products complete before the timeout, the run is considered complete. If the timeout is reached, the benchmark proceeds using the data collected up to that point. This allows incomplete or problematic runs to be inspected rather than discarded entirely.

Once execution finishes, the Metrics Collector is stopped and the raw event stream stored in `metrics.jsonl` is processed into a summary file, `metrics_summary.json`. This summary includes the number of completed products and the makespan of the run. Additional values, such as throughput or estimated utilization, may also be produced by the script, but the experimental analysis focuses primarily on makespan.

Finally, unless configured otherwise, the benchmark stops the running agents to return the system to a clean state before the next experiment. The overall procedure can be summarized as follows:

1. Load the runtime configuration.
2. Remove previous runtime files.
3. Start the resource agents.
4. Wait for service readiness.
5. Refresh the AAS structure.
6. Start the Metrics Collector.
7. Launch the Product Agents.
8. Wait for product completion events or timeout.
9. Stop the Metrics Collector.
10. Generate the metrics summary.
11. Stop the running agents.

This procedure supports repeatable experimentation while preserving the decentralized behaviour of the proposed architecture. The benchmark runner coordinates process startup, shutdown, and data processing, but it does not centrally schedule the products during execution. Resource discovery, resource claiming, contention handling, retries, and route execution remain the responsibility of the Product Agents and resource agents.

A key aspect of this setup is the separation between experiment orchestration and agent decision-making. The benchmark runner provides a controlled experimental boundary, while the agents continue to operate through the same AAS-based discovery mechanism and REST interfaces used during normal execution. Therefore, the collected results reflect the behaviour of the decentralized AAS-based architecture rather than a special test-only execution path.

5.5 Evaluation Metrics and Data Processing

The main evaluation metric used in the experiments is makespan. Makespan represents the total elapsed time required to complete all products in a scenario. It was selected because it provides a direct indication of the overall completion time of the manufacturing workload and allows comparison between the decentralized AAS-based execution and the centralized scheduling baseline.

For the AAS-based agent implementation, makespan is calculated from the event stream recorded by the Metrics Collector. Specifically, the start of a run is defined by the earliest product start event, while the end of the run is defined by the latest product completion event. The makespan is therefore calculated as:

$$C_{\max} = \max(t_{end}) - \min(t_{start}) \tag{5.1}$$

where t_{start} represents the timestamps of product start events and t_{end} represents the timestamps of product completion events.

The raw event stream is stored in `metrics.jsonl` and processed after each benchmark run to generate `metrics_summary.json`. This summary contains the main values required for the experimental analysis, including the number of completed products and the calculated makespan. Additional values, such as throughput or estimated utilization, may also be generated by the script, but they are not used as the primary basis for comparison in this dissertation.

For the centralized PyJobShop baseline, makespan is obtained from the generated schedule as the completion time of the last scheduled product. Since the centralized model uses the same product routes, resource capabilities, and task durations as the agent-based implementation, the makespan values obtained from both approaches can be compared directly.

The comparison between both approaches is therefore based primarily on makespan. In addition to the numerical value itself, the analysis considers how makespan changes across the different experimental scenarios: increasing product workload, increasing operation duration, and proportional asset scaling. This allows the behaviour of the proposed architecture to be evaluated not only for a single configuration, but also under varying production conditions.

Although the metrics infrastructure records additional runtime information, the interpretation of secondary indicators is treated with caution. In particular, estimated resource utilization was measured in some runs but is not considered a central evaluation criterion, since the available data was not analyzed in sufficient depth to support detailed conclusions. For this reason, the results chapter focuses on makespan and uses other collected information only as supporting context when relevant.

5.6 Results and Analysis

This section presents and discusses the results obtained from the experimental setup previously described. The objective is to evaluate the behaviour of the proposed AAS-based multi-agent

Table 5.6: Summary of makespan results for the evaluated scenarios.

Experiment	Scenario parameter	Agents makespan (s)	PyJobShop makespan (s)
Workload scaling	1 product	30.93	9.00
Workload scaling	2 products	40.30	13.00
Workload scaling	3 products	50.31	17.00
Workload scaling	4 products	60.18	21.00
Workload scaling	8 products	101.74	37.00
Workload scaling	16 products	173.75	69.00
Workload scaling	32 products	254.91	133.00
Processing scaling	Multiplier 1	50.36	17.00
Processing scaling	Multiplier 2	67.84	34.00
Processing scaling	Multiplier 3	84.16	51.00
Processing scaling	Multiplier 4	102.08	68.00
Processing scaling	Multiplier 5	118.40	85.00
Asset scaling	Multiplier 1	50.36	17.00
Asset scaling	Multiplier 2	54.69	17.00
Asset scaling	Multiplier 4	56.64	19.00
Asset scaling	Multiplier 8	68.64	31.00

architecture and compare it with the centralized PyJobShop scheduling baseline.

The analysis focuses primarily on makespan, which was defined as the main evaluation metric. Makespan provides a direct measure of the time required to complete all products in each experimental scenario. The results are organized according to the three experimental dimensions introduced previously: product workload scaling, processing time scaling, and asset scaling.

The comparison between the two approaches should be interpreted with care. The centralized PyJobShop baseline generates a schedule from a complete model of the problem and does not include the runtime communication, AAS state updates, resource claiming, or distributed coordination overhead present in the proposed architecture. Therefore, the comparison is not intended to show that both approaches operate under identical execution conditions, but rather to provide a reference point for analyzing the behaviour and cost of the decentralized AAS-based implementation.

Table 5.6 summarizes the makespan values obtained for the evaluated scenarios. The corresponding plots are presented in Figures 5.1, 5.2, and 5.3.

5.6.1 Product Workload Scaling

The first experiment evaluates the effect of increasing the number of products while keeping the resource configuration constant. In this scenario, the number of Product Agent instances increases, while the number of machines and AGVs remains fixed. As a result, more products compete for the same shared resources.

Figure 5.1 presents the makespan obtained by the AAS-based agent architecture and by the centralized PyJobShop baseline for increasing product quantities.

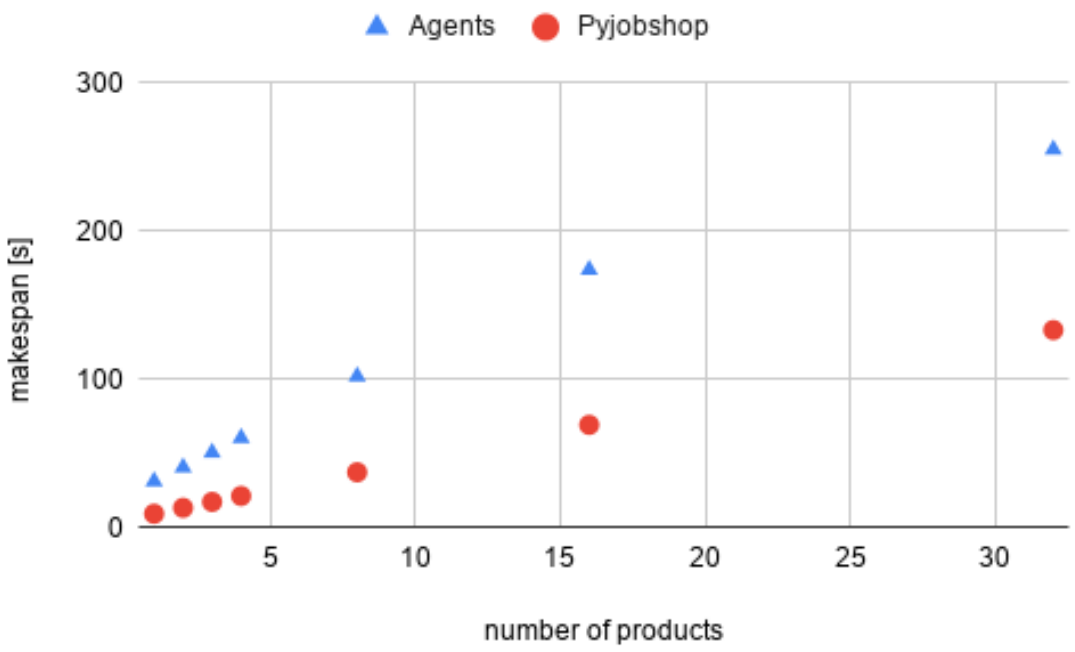


Figure 5.1: Makespan comparison under product workload scaling.

The results show that makespan increases for both approaches as the number of products increases. This behaviour is expected, since more products require more processing and transport tasks to be completed using the same resource configuration.

Across the evaluated scenarios, the centralized PyJobShop baseline achieves lower makespan values than the AAS-based agent implementation. This is also expected, since the centralized baseline solves the scheduling problem using a complete model of the scenario and does not include the runtime overhead associated with distributed execution.

For the AAS-based architecture, the increase in makespan is influenced by both resource contention and coordination overhead. As more Product Agents execute concurrently, more agents attempt to discover, claim, and release machines and AGVs through the AAS infrastructure. This increases the number of REST interactions, state updates, and possible claim retries. Therefore, the observed makespan reflects not only the processing and transport durations, but also the cost of distributed coordination.

The results indicate that the proposed architecture is able to complete all evaluated workload scenarios, demonstrating functional scalability in the tested range. However, the difference between the agent-based approach and the centralized baseline becomes more visible as the number of products increases, showing that communication and synchronization overhead become more relevant under higher workload conditions.

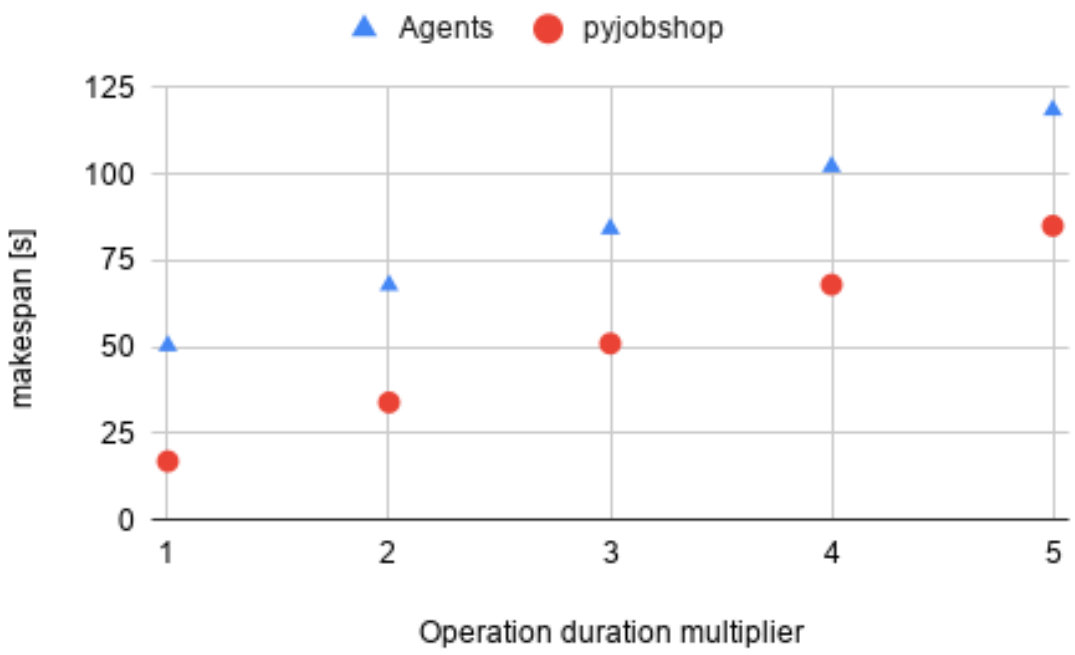


Figure 5.2: Makespan comparison under processing time scaling.

5.6.2 Processing Time Scaling

The second experiment evaluates the effect of increasing processing and transport durations while keeping the number of products constant. In this experiment, three products are executed in each scenario, and the operation durations are multiplied by predefined factors.

Figure 5.2 presents the makespan obtained for each duration multiplier.

The results show an approximately linear increase in makespan for both approaches as the duration multiplier increases. This is expected because the number of products and the number of coordination steps remain constant, while the duration of each task increases.

For the centralized PyJobShop baseline, makespan increases from 17 seconds with multiplier 1 to 85 seconds with multiplier 5. For the AAS-based agent implementation, makespan increases from approximately 50.36 seconds to approximately 118.40 seconds over the same multiplier range.

An important observation is that the difference between the two approaches remains relatively stable across the processing time multipliers. This suggests that, in this experiment, the additional time introduced by the AAS-based agent implementation is mostly associated with the fixed coordination process required to execute the three products. Since the number of products and route steps remains constant, the number of AAS interactions, resource claims, and execution requests also remains relatively constant.

Therefore, this experiment suggests that longer processing durations affect both approaches in a predictable way. The agent-based architecture does not show unstable behaviour as operation

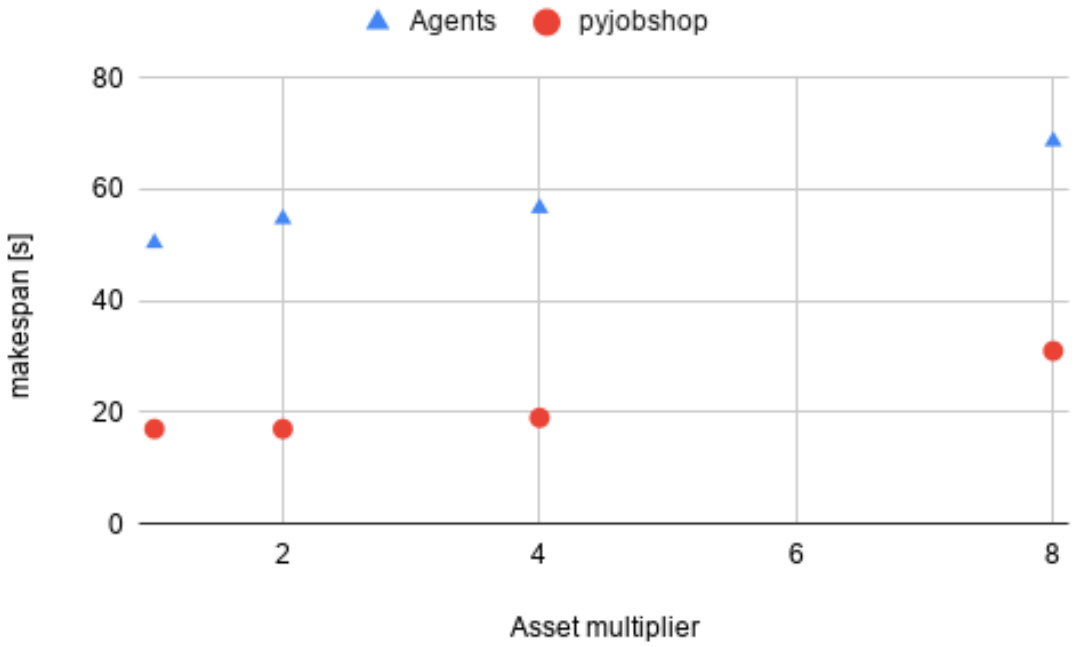


Figure 5.3: Makespan comparison under asset scaling.

durations increase, although it maintains a higher makespan due to the overhead introduced by distributed coordination and runtime communication.

5.6.3 Asset Scaling

The third experiment evaluates the behaviour of the system when products and resources are increased proportionally. Unlike the workload scaling experiment, where more products compete for a fixed set of resources, this experiment increases the system size by scaling both workload and available assets.

Figure 5.3 presents the makespan obtained for each asset scaling factor.

The results show that the AAS-based implementation maintains relatively stable behaviour for the lower scaling factors, with makespan increasing from approximately 50.36 seconds at multiplier 1 to approximately 56.64 seconds at multiplier 4. At multiplier 8, the makespan increases to approximately 68.64 seconds.

The centralized PyJobShop baseline also shows an increase in makespan at higher scaling factors, from 17 seconds at multiplier 1 to 31 seconds at multiplier 8. This indicates that even when resources are scaled proportionally, the larger scenario still introduces additional scheduling complexity and execution length.

For the AAS-based architecture, the increase observed at higher scaling factors can be associated with the larger number of agents and interactions required during execution. Even though more resources are available, the system must still initialize more AAS instances, manage more Product Agents, perform more resource discovery operations, and record more execution events.

This experiment suggests that proportional asset scaling reduces the resource bottleneck effect observed in workload scaling, but does not eliminate the coordination overhead of the distributed architecture. The proposed system remains capable of completing the scaled scenarios, but its execution time is still affected by the number of distributed interactions required.

5.7 Overall Discussion

Across the three experimental groups, the centralized PyJobShop baseline consistently achieves lower makespan values than the AAS-based multi-agent implementation. This result is expected, since the centralized approach solves a complete scheduling model and does not include the communication and coordination overhead introduced by the proposed architecture.

The AAS-based approach, however, successfully completes all evaluated scenarios. This demonstrates that the proposed architecture can coordinate machines and AGVs through AAS-based resource representation, product-specific agents, and REST-based execution services. The system is therefore functional as a distributed scheduling and coordination prototype.

The workload scaling experiment shows that the difference between the two approaches becomes more significant as more products are introduced into a fixed resource configuration. This is mainly because the number of Product Agents increases while the available resources remain constant, leading to more contention and more AAS-based coordination activity.

The processing time scaling experiment shows a more predictable trend. Since the number of products remains constant, the number of coordination interactions also remains relatively constant. As a result, both approaches show an approximately linear increase in makespan as operation durations increase.

The asset scaling experiment shows that increasing products and resources proportionally leads to more stable behaviour than increasing products alone. Nevertheless, the AAS-based implementation still experiences additional overhead due to the larger number of agents, AAS instances, resource states, and runtime interactions.

Overall, the results highlight the trade-off between centralized optimization and decentralized AAS-based coordination. The centralized baseline provides better makespan performance under the evaluated scenarios, while the proposed architecture demonstrates modularity, distributed execution, and the ability to coordinate assets through standardized digital representations.

These results should not be interpreted as evidence that the AAS-based agent approach is more efficient than centralized scheduling in terms of makespan. Instead, they show that the architecture is feasible and that its performance cost is mainly associated with runtime coordination, communication, and state management. This distinction is important because the objective of the proposed system is not only schedule optimality, but also the exploration of AAS-based distributed manufacturing coordination.

5.8 Chapter Summary

This chapter presented the results obtained from the experimental evaluation of the proposed AAS-based multi-agent architecture and the centralized PyJobShop baseline. The analysis focused on makespan across three experimental dimensions: product workload scaling, processing time scaling, and asset scaling.

The results show that the centralized PyJobShop baseline consistently achieves lower makespan values. This is expected due to its centralized optimization model and the absence of runtime communication overhead. In contrast, the AAS-based implementation performs scheduling and coordination during execution through distributed Product Agents, resource agents, AAS state updates, and REST communication.

Despite the higher makespan values, the proposed architecture successfully completed all evaluated scenarios. The results demonstrate that AAS-based agents can coordinate manufacturing and transportation resources in a distributed execution environment. The experiments also show that the architecture behaves predictably under increasing processing durations and proportional asset scaling, while higher product workloads introduce greater coordination overhead.

The following chapter presents the conclusions of the dissertation and discusses possible directions for future work.

Chapter 6

Conclusions and Future Work

This chapter presents the main conclusions of the dissertation, summarizes the contributions of the work, discusses the limitations of the proposed approach, and identifies possible directions for future development.

The work presented in this dissertation focused on the design, implementation, and evaluation of an AAS-based multi-agent architecture for manufacturing scheduling and resource coordination. The proposed system combines AASs, software agents, REST-based communication, and a metrics collection layer to coordinate products, machines, AGVs, and support infrastructure in a distributed manufacturing scenario.

6.1 Conclusions

The main objective of this dissertation was to investigate the feasibility of using AASs as an information layer for a multi-agent manufacturing scheduling architecture. To achieve this objective, a prototype was developed in which each asset (products, machines, etc) is associated with its own corresponding AAS instance and software agent.

The proposed architecture was evaluated using a simplified manufacturing scenario composed of Operation A, transport, and Operation B. The system was tested under three experimental dimensions: increasing product workload, increasing operation duration, and proportional asset scaling. A centralized scheduling baseline implemented with PyJobShop was also developed to provide a reference point for comparison.

The experimental results show that the centralized PyJobShop baseline consistently achieved lower makespan values than the AAS-based agent implementation. This result is expected, since the centralized baseline builds a complete model of the scheduling problem and does not include the runtime communication, AAS state updates, resource claiming, retries, or distributed coordination overhead present in the proposed architecture.

Nevertheless, the AAS-based MAS successfully completed all evaluated scenarios. This demonstrates that the proposed architecture is capable of coordinating manufacturing and transportation resources through distributed agents and AAS-based resource representations. The results also

show that the architecture behaves predictably when operation durations increase and when products and resources are scaled proportionally.

The workload scaling experiment showed that the makespan difference between the agent-based system and the centralized baseline increases as more products compete for a fixed set of resources. This indicates that communication and synchronization overhead become more relevant as the number of Product Agents and resource interactions increases.

Overall, the results suggest that the proposed architecture is feasible as a distributed coordination approach for manufacturing scheduling. However, the implementation also highlights the performance cost associated with decentralization, particularly when compared with a centralized optimization-based scheduler.

6.2 Main Contributions

The main contribution of this dissertation is the development and evaluation of an AAS-based multi-agent architecture for manufacturing scheduling and resource coordination. The architecture demonstrates how Asset Administration Shells can be used to represent products, machines, AGVs, and support infrastructure in a distributed scheduling scenario.

A second contribution is the implementation of a working prototype using Python, Flask-based services, Eclipse BaSyx, and REST communication. The prototype shows how Product Agents can retrieve process information, discover compatible resources, claim machines and AGVs, request task execution, and update resource states through AAS-based interactions.

A third contribution is the development of an experimental benchmark procedure for evaluating the proposed architecture. The benchmark runner supports repeated execution of different scenario configurations, initializes the required services, refreshes the AAS structure, launches Product Agents, collects distributed execution events, and generates summary metrics.

A fourth contribution is the comparison with a centralized scheduling baseline implemented using PyJobShop. This baseline provides a reference for interpreting the makespan obtained by the AAS-based agent implementation and helps identify the performance cost associated with distributed coordination.

6.3 Limitations

The work developed in this dissertation has several limitations that should be considered when interpreting the results.

First, the manufacturing scenario is intentionally simplified. Products follow a predefined route composed of Operation A, transport, and Operation B. Although this scenario captures important scheduling aspects such as precedence, resource capacity, alternative resource selection, and transport coordination, it does not represent the full complexity of real industrial production systems.

Second, processing and transport durations are deterministic. This makes the experiments easier to control and compare, but it does not capture disturbances such as machine failures, variable processing times, delays, maintenance, or unexpected AGV behaviour.

Third, the resource reservation mechanism is based on AAS state updates and confirmation reads. While this approach was sufficient for the evaluated scenarios, it does not provide the same guarantees as a transactional locking mechanism or a dedicated reservation service. Under higher concurrency, more robust synchronization mechanisms may be required.

Fourth, the proposed implementation introduces communication overhead due to repeated REST interactions with the AAS infrastructure and Flask-based resource services. This overhead affects the makespan of the agent-based system and contributes to the performance difference observed when compared with the centralized PyJobShop baseline.

Finally, the evaluation focuses primarily on makespan. Although the metrics infrastructure records additional information, such as service events and resource-related events, these data were not analyzed in sufficient depth to support detailed conclusions about resource utilization or throughput.

6.4 Future Work

Future work should extend the proposed architecture to more complex manufacturing scenarios. This includes products with different routes, heterogeneous operation sequences, additional machine types, multiple transport stages, and more realistic production constraints. It could also investigate a fuller AAS Type 3 implementation, where more active behaviour is integrated into or more directly associated with the AAS components themselves. This would allow the proposed architecture to move beyond using the AAS mainly as an information layer and toward a more active asset representation model.

Another important direction is the improvement of the resource reservation and synchronization mechanism. Future versions of the system could include explicit locking, transactional updates, reservation services, or negotiation protocols between agents. These mechanisms could reduce the probability of race conditions and improve coordination under higher levels of concurrency.

The communication efficiency of the architecture should also be investigated. Since the current implementation relies on repeated REST requests to the AAS infrastructure, future work could explore caching strategies, event-driven updates, message brokers, or more efficient interaction patterns between agents and AAS components.

The AGV behaviour could also be extended. In the current implementation, transport is modeled as a simplified time-based task. Future work could include more detailed AGV routing, battery-aware scheduling, charging decisions, station availability, and path conflicts between AGVs.

Further work could also improve the AAS modelling approach. More detailed and standardized submodels could be introduced to represent capabilities, resource states, process plans, execution history, and scheduling-related information. This would make the architecture more aligned with industrial interoperability goals.

Finally, the experimental evaluation could be expanded. Future experiments should include larger scenarios, repeated runs for statistical analysis, dynamic disruptions, and additional performance metrics. This would allow a more complete assessment of the behaviour, robustness, and scalability of the proposed AAS-based multi-agent architecture.

6.5 Final Remarks

This dissertation demonstrated that Asset Administration Shells can be used as a shared information layer for a distributed multi-agent manufacturing scheduling architecture. Although the centralized scheduling baseline achieved lower makespan values, the proposed system successfully coordinated products, machines, and AGVs through AAS-based interactions.

The results highlight both the potential and the challenges of AAS-based distributed scheduling. The approach supports modularity, distributed execution, and standardized asset representation, but introduces communication and synchronization overhead that must be addressed in future developments. Therefore, the work provides a foundation for further research into AAS-based MASs for manufacturing coordination.

References

[1] D. Behnke and M. J. Geiger, "Test Instances for the Flexible Job Shop Scheduling Problem with Work Centers," 2012, ISSN: 2192-0826.

[2] H. Song and P. Liu, "A Study on the Optimal Flexible Job-Shop Scheduling with Sequence-Dependent Setup Time Based on a Hybrid Algorithm of Improved Quantum Cat Swarm Optimization," *Sustainability*, vol. 14, no. 15, p. 9547, Aug. 3, 2022, ISSN: 2071-1050. DOI: 10.3390/su14159547

[3] Z. Cai, Y. Feng, S. Yang, and J. Yang, "A State Transition Diagram and an Artificial Physarum polycephalum Colony Algorithm for the Flexible Job Shop Scheduling Problem with Transportation Constraints," *Processes*, vol. 11, no. 9, p. 2646, Sep. 4, 2023, ISSN: 2227-9717. DOI: 10.3390/pr11092646

[4] A. Boccalatte, A. Gozzi, M. Paolucci, V. Queirolo, and M. Tamoglia, "A multi-agent system for dynamic just-in-time manufacturing production scheduling," in *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No.04CH37583)*, vol. 6, The Hague, Netherlands: IEEE, 2004, pp. 5548–5553, ISBN: 978-0-7803-8567-2. DOI: 10.1109/ICSMC.2004.1401077

[5] O. Antons and J. C. Arlinghaus, "A Manufacturing Scheduling Complexity Framework and Agent-Based Comparison of Centralized and Distributed Control Approaches," *IEEE Journal of Emerging and Selected Topics in Industrial Electronics*, vol. 3, no. 1, pp. 31–38, Jan. 2022, ISSN: 2687-9735, 2687-9743. DOI: 10.1109/JESTIE.2021.3100272

[6] L. Sakurada, F. De La Prieta, and P. Leitao, "Ten Years of Asset Administration Shell: Developments, Research Opportunities, and Adoption Challenges," *IEEE Access*, vol. 13, pp. 127 721–127 741, 2025, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2025.3586716

[7] T. A. Abdel-Aty, E. Negri, and S. Galparoli, "Asset Administration Shell in Manufacturing: Applications and Relationship with Digital Twin," *IFAC-PapersOnLine*, vol. 55, no. 10, pp. 2533–2538, 2022, ISSN: 24058963. DOI: 10.1016/j.ifacol.2022.10.090

[8] X. Ye, M. Yu, W. S. Song, and S. H. Hong, "An Asset Administration Shell Method for Data Exchange Between Manufacturing Software Applications," *IEEE Access*, vol. 9, pp. 144 171–144 178, 2021, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3122175

[9] L. Sakurada, F. De La Prieta, and P. Leitao, "The Role of Multi-Agent Systems in Realizing Asset Administration Shell Type 3," *Future Internet*, vol. 17, no. 7, p. 270, Jun. 20, 2025, ISSN: 1999-5903. DOI: 10.3390/fi17070270

[10] V. Siatras, E. Bakopoulos, P. Mavrothalassitis, N. Nikolakis, and K. Alexopoulos, "On the Use of Asset Administration Shell for Modeling and Deploying Production Scheduling Agents within a Multi-Agent System," *Applied Sciences*, vol. 13, no. 17, p. 9540, Aug. 23, 2023, ISSN: 2076-3417. DOI: 10.3390/app13179540

[11] A. Gannouni, P. Sapel, A. Abdelrazeq, C. Hopmann, and R. H. Schmitt, "A blueprint description of production scheduling models using asset administration shells," *Production & Manufacturing Research*, vol. 12, no. 1, p. 2 324 339, Dec. 31, 2024, ISSN: 2169-3277. DOI: 10.1080/21693277.2024.2324339

[12] T. Gneuß, M. Ristin, and N. Braunisch, "Combining Publicly Available SDKs and Code Generation Tools to Streamline the Implementation of AAS Applications," 2025. DOI: 10.1109/ETFA65518.2025.11205694

Appendix A

benchmark_runner.py source code

```
"""Simple benchmark runner: starts metrics collector, launches
    agents/products, then summarizes metrics."""
import argparse
import subprocess
import sys
import os
import time
import json
import socket
from datetime import datetime

from config_utils import (
    build_indexed_port,
    get_int_config,
    get_machine_specs,
    get_port_config,
    load_runtime_config,
)

BASE_DIR = os.path.dirname(os.path.abspath(__file__))
PROJECT_ROOT = os.path.dirname(BASE_DIR)
VENV_PY = os.path.join(PROJECT_ROOT, ".venv", "Scripts", "python
    .exe")
PY = VENV_PY if os.path.exists(VENV_PY) else sys.executable
METRICS_FILE = os.getenv("METRICS_FILE", os.path.join(BASE_DIR,
    "metrics.jsonl"))
```

```
RESOURCE_CACHE_FILE = os.getenv("AAS_RESOURCE_CACHE", os.path.
    join(BASE_DIR, ".aas_resource_cache.json"))
DEFAULT_PRODUCT_COUNT = 3
DEFAULT_MACHINE_OPERATIONS = {
    "drill": 2,
    "assemble": 1,
}

RUNTIME_CONFIG = load_runtime_config()
DEFAULT_AGV_COUNT = get_int_config(RUNTIME_CONFIG, "agv_count",
    2)
DEFAULT_WAIT_TIMEOUT_SECONDS = 900
AGV_PORT_STEP = 2


def _get_runtime_counts() -> tuple[int, int]:
    config = load_runtime_config()
    product_count = get_int_config(config, "product_count",
        DEFAULT_PRODUCT_COUNT)
    machine_count = len(get_machine_specs(config,
        DEFAULT_MACHINE_OPERATIONS))
    return product_count, machine_count


def parse_args():
    parser = argparse.ArgumentParser(description="Run benchmark
        and summarize metrics.")
    parser.add_argument(
        "--no-stop-agents",
        action="store_true",
        help="Leave agents running after the benchmark completes
            .",
    )
    parser.add_argument(
        "--force-stop",
        action="store_true",
        help="Force-stop agents at the end (Windows only,
            default behavior).",
    )
    parser.add_argument(
```

```
        "--graceful-stop",
        action="store_true",
        help="Attempt a graceful stop without force-kill.",
    )
    return parser.parse_args()


def start_collector():
    config = load_runtime_config()
    metrics_port = get_port_config(config, "metrics", 9000, "
        METRICS_PORT")
    env = os.environ.copy()
    env["METRICS_FILE"] = METRICS_FILE
    env["METRICS_PORT"] = str(metrics_port)
    proc = subprocess.Popen([PY, os.path.join(BASE_DIR, "
        metrics_collector.py")], env=env)
    print("Started metrics collector, pid=", proc.pid)
    return proc


def start_agents(agv_count=DEFAULT_AGV_COUNT):
    config = load_runtime_config()
    metrics_port = get_port_config(config, "metrics", 9000, "
        METRICS_PORT")
    launcher_env = os.environ.copy()
    launcher_env["METRICS_PORT"] = str(metrics_port)
    launcher_env["METRICS_URL"] = launcher_env.get("METRICS_URL
        ", f"http://localhost:{metrics_port}/metric")
    # Launch agents first (without products) in detached mode
    cmd = [
        PY,
        os.path.join(BASE_DIR, "launch_all.py"),
        "--agv-count",
        str(agv_count),
        "--detach",
    ]
    print("Starting agents (without products):", " ".join(cmd))
    ret = subprocess.run(cmd, cwd=PROJECT_ROOT, env=launcher_env
        )
```

```
    print("Agents launcher finished with returncode", ret.
        returncode)
    return ret.returncode


def refresh_aas_structure(agv_count=DEFAULT_AGV_COUNT):
    env = os.environ.copy()
    env["AGV_COUNT"] = str(agv_count)
    cmd = [PY, os.path.join(BASE_DIR, "createaasdemo.py")]
    print("Refreshing AAS structure:", " ".join(cmd))
    ret = subprocess.run(cmd, cwd=BASE_DIR, env=env)
    print("AAS refresh finished with returncode", ret.returncode
        )
    return ret.returncode


def wait_for_agents(machine_count: int, timeout_seconds=30):
    """Wait for agents to be ready by checking if they're
        listening on their ports."""
    config = load_runtime_config()
    base_port = get_port_config(config, "machine_base", 5000, "
        MACHINE_BASE_PORT")
    agv_base_port = get_port_config(config, "agv_base", 5004, "
        AGV_BASE_PORT")
    charge_station_port = get_port_config(config, "
        charge_station", 5005, "CHARGE_STATION_PORT")
    agv_ports = [
        build_indexed_port(index, agv_base_port, AGV_PORT_STEP,
            {charge_station_port})
        for index in range(1, DEFAULT_AGV_COUNT + 1)
    ]
    machine_ports = [base_port + (idx - 1) for idx in range(1,
        machine_count + 1)]
    ports = machine_ports + agv_ports + [charge_station_port]
    deadline = time.time() + timeout_seconds
    all_ready = False

    while time.time() < deadline:
        ready_count = 0
        for port in ports:
```

```
            try:
                sock = socket.socket(socket.AF_INET, socket.
                    SOCK_STREAM)
                sock.settimeout(0.5)
                result = sock.connect_ex(('localhost', port))
                sock.close()
                if result == 0:
                    ready_count += 1
            except Exception:
                pass

        print(f"Agents ready: {ready_count}/{len(ports)}", end
            ="\r", flush=True)
        if ready_count >= len(ports):
            print()
            all_ready = True
            break
        time.sleep(0.5)

    if not all_ready:
        print(f"\nWarning: Not all agents ready after {
            timeout_seconds}s, proceeding anyway")
    return all_ready


def run_demo(product_count=DEFAULT_PRODUCT_COUNT, agv_count=
    DEFAULT_AGV_COUNT):
    config = load_runtime_config()
    metrics_port = get_port_config(config, "metrics", 9000, "
        METRICS_PORT")
    # Launch products in detached mode (agents are already
        running)
    cmd = [
        PY,
        os.path.join(BASE_DIR, "launch_products.py"),
        "--count",
        str(product_count),
    ]
    launcher_env = os.environ.copy()
    launcher_env["AGV_COUNT"] = str(agv_count)
```

```
    launcher_env["METRICS_PORT"] = str(metrics_port)
    launcher_env["METRICS_URL"] = launcher_env.get("METRICS_URL
        ", f"http://localhost:{metrics_port}/metric")
    print("Starting products launcher:", " ".join(cmd))
    ret = subprocess.run(cmd, cwd=PROJECT_ROOT, env=launcher_env
        )
    print("Products launcher finished with returncode", ret.
        returncode)
    return ret.returncode


def wait_for_completion(product_count=DEFAULT_PRODUCT_COUNT,
    timeout_seconds=DEFAULT_WAIT_TIMEOUT_SECONDS):
    print(f"Waiting for {product_count} product_end event(s) in
        {METRICS_FILE}")
    deadline = time.time() + timeout_seconds
    seen_products = set()

    while time.time() < deadline:
        if os.path.exists(METRICS_FILE):
            try:
                with open(METRICS_FILE, "r", encoding="utf-8")
                    as handle:
                    for line in handle:
                        if not line.strip():
                            continue
                        event = json.loads(line)
                        if event.get("event") == "product_end"
                            and event.get("productId"):
                            seen_products.add(event["productId
                                "])
            except json.JSONDecodeError:
                # The collector may still be appending a line;
                    retry on the next poll.
                pass

        print(f"Completed products: {len(seen_products)}/{
            product_count}", end="\r", flush=True)
        if len(seen_products) >= product_count:
            print()
```

```
            return True

        time.sleep(1)

    print()
    print(f"Timeout waiting for products after {timeout_seconds}
       s; seen={sorted(seen_products)}")
    return False


def summarize_metrics():
    if not os.path.exists(METRICS_FILE):
        print("No metrics file found:", METRICS_FILE)
        return

    print(f"Reading metrics from {METRICS_FILE}")
    lines = [json.loads(l) for l in open(METRICS_FILE, "r",
       encoding="utf-8") if l.strip()]

    # product timings
    starts = {}
    ends = {}
    for e in lines:
        ev = e.get("event")
        if ev == "product_start":
            pid = e.get("productId")
            starts[pid] = starts.get(pid) or e.get("ts")
        if ev == "product_end":
            pid = e.get("productId")
            ends[pid] = e.get("ts")

    parsed_starts = [datetime.fromisoformat(s.rstrip("Z")) for s
        in starts.values()] if starts else []
    parsed_ends = [datetime.fromisoformat(s.rstrip("Z")) for s
       in ends.values()] if ends else []

    summary = {}
    if parsed_starts and parsed_ends:
        makespan = (max(parsed_ends) - min(parsed_starts)).
           total_seconds()
```

```
        completed = len(parsed_ends)
        throughput = completed / makespan if makespan > 0 else
            None
        summary.update({"makespan_s": makespan, "completed":
            completed, "throughput_per_s": throughput})
    else:
        summary.update({"makespan_s": None, "completed": 0, "
            throughput_per_s": None})

    # resource utilization from service events
    service_sum = {}
    for e in lines:
        if e.get("event") == "service":
            agent = e.get("agent")
            dur = float(e.get("duration_ms") or 0)
            service_sum[agent] = service_sum.get(agent, 0.0) +
                dur

    util = {}
    if summary.get("makespan_s"):
        for agent, total_ms in service_sum.items():
            util[agent] = (total_ms / (summary["makespan_s"] *
                1000.0)) * 100.0

    summary["utilization_percent"] = util

    out_path = os.path.abspath(os.path.join(BASE_DIR, "
        metrics_summary.json"))
    try:
        with open(out_path, "w", encoding="utf-8") as f:
            json.dump(summary, f, indent=2)
        print(f"Summary written to {out_path}")
    except Exception as e:
        print(f"Error writing summary to {out_path}: {e}")
        return

    print(json.dumps(summary, indent=2))


def stop_agents(force: bool = False) -> None:
```

```
    script_path = os.path.join(BASE_DIR, "stop_agents.py")
    if not os.path.exists(script_path):
        print("stop_agents.py not found; skipping agent shutdown
            .")
        return

    cmd = [PY, script_path]
    if force:
        cmd.append("--force")
    print("Stopping agents:", " ".join(cmd))
    try:
        subprocess.run(cmd, cwd=PROJECT_ROOT)
    except Exception as exc:
        print(f"Failed to stop agents: {exc}")


def main():
    args = parse_args()
    product_count, machine_count = _get_runtime_counts()
    agv_count = DEFAULT_AGV_COUNT
    # Clean generated runtime files
    try:
        if os.path.exists(METRICS_FILE):
            os.remove(METRICS_FILE)
        if os.path.exists(RESOURCE_CACHE_FILE):
            os.remove(RESOURCE_CACHE_FILE)
    except Exception:
        pass

    # 1. Start agents first (without products)
    code = start_agents(agv_count=agv_count)
    if code != 0:
        print(f"Agent launcher exited with code {code};
            continuing anyway")

    # 2. Wait for agents to be ready
    wait_for_agents(machine_count)
    time.sleep(0.5)
```

```
    # 3. Refresh AAS endpoints after agents are up, so products
       do not discover stale ports.
    code = refresh_aas_structure(agv_count=agv_count)
    if code != 0:
        print(f"AAS refresh exited with code {code}; products
           may discover stale endpoints")
    try:
        if os.path.exists(RESOURCE_CACHE_FILE):
            os.remove(RESOURCE_CACHE_FILE)
    except Exception:
        pass

    # 4. Start metrics collector (just before launching products
       )
    collector = start_collector()
    time.sleep(0.5)

    interrupted = False
    try:
        # 5. Launch products (agents are already running and
           metrics is collecting)
        code = run_demo(product_count=product_count, agv_count=
           agv_count)
        if code != 0:
            print(f"Products launcher exited with code {code};
               continuing to wait for metrics anyway")

        # 6. Wait for products to complete
        wait_for_completion(product_count=product_count)
    except KeyboardInterrupt:
        interrupted = True
        print("Benchmark interrupted; summarizing collected
           metrics so far.")
    finally:
        try:
            collector.terminate()
            collector.wait(timeout=10)
        except Exception:
            pass
```

```
    time.sleep(0.5)
    try:
        summarize_metrics()
    except Exception as e:
        print(f"Error summarizing metrics: {e}")
        import traceback
        traceback.print_exc()

    if interrupted:
        print("Partial summary written after interruption.")

    if not args.no_stop_agents:
        force_stop = not args.graceful_stop
        stop_agents(force=force_stop)


if __name__ == "__main__":
    main()
```