# Book Production System

Anders Risager Kristensen*, Kasper Mandal Stehning*, Marco Lambrecht Jacobsen*, Mohamed Gomaa*, Vivek Misra*
University of Southern Denmark, SDU Software Engineering, Odense, Denmark
Email: * {akris19,kaste20,marja22,mogom21,vimis22}@student.sdu.dk
Group Number: 4

***Abstract*—Modern production systems increasingly operate in complex, distributed environments where tasks must be coordinated across multiple production cells while remaining flexible and resilient to failure. Traditional linear production models are insufficient for such settings, as they do not support parallel processing, dynamic reconfiguration, or robust fault handling. This article addresses these challenges in the context of book production, where production cells must execute tasks with both linear and non-linear dependencies.**

**We propose a software-based scheduling architecture that distributes workload among multiple production cells, enforces dependency-aware execution, supports dynamic reordering, and expansion of production steps, and maintains fault tolerance through heartbeat-based monitoring and automatic job re-queuing. The system models book production and is evaluated using 100 batch-test orders in a containerized environment with simulated machines failures. Statistical analysis shows that the system scales linearly with order size, as quantity has no significant effect on completion time. Fault tolerance mechanisms perform efficiently, detecting failures within three seconds and introducing only minimal overhead despite the frequent re-queue of events. However, the evaluation also identifies scheduler-induced waiting time as the prime factor affecting completion time, revealing scheduling inefficiency as the primary performance bottleneck.**

**The results demonstrate that it is possible to build a flexible, scalable, and fault-tolerant book production system that effectively distributes workload across multiple production cells and handles task dependencies. While the architectural design proves robust and extensible, there is a need for further optimization of scheduling logic to significantly reduce waiting time and fully realize the systems performance potential.**

***Index Terms*—Distributed Book Production, Architecture, Containerized Docker Environment, Database, Job Scheduling, Parallel Processing, Dependency-aware, Fault Tolerance, Heartbeat Monitoring, MQTT, Kafka, REST API, Scalability, Performance, Quality Attribute Tactics, UPPAAL**

## I. Introduction and Motivation

Modern production systems are becoming increasingly more complex, requiring coordination of multiple independent components that must operate efficiently, flexibly, and reliably, which represent the shift to Industry 4.0 environments [1]. Traditional linear production models, where tasks are performed strictly in sequence, are often insufficient for such environments. Instead, many new and real-world systems require parallel processing, dynamic task allocation, and the ability to adapt to changes, new features, or failures during operation.

Book production serves as an example of these challenges. Producing a book involves several distinct steps such as printing pages, printing covers, binding, and packaging that must be completed correctly to deliver a finished product. While some of these can be executed independently or in parallel, others depend on the completion of the preceding tasks. coordinating these processes efficiently is essential to minimize waiting times, balance workloads across production cells, and ensure a timely completion of the product.

As the production system grows in scale and complexity, manual coordination or rigid workflows become impractical. There is a clear need for a software-based solution capable of managing task distribution, handling both linear and non-linear dependencies between production tasks, and to adapt to changes in the production process. Furthermore, the system must be robust enough to handle failures in individual production units and flexible enough to integrate new production steps as requirements evolve.

This article focuses on addressing these challenges by exploring the design of a scheduling system for book production. The goal is to define a system that can effectively distribute work/tasks among multiple production cells, enforce the correct execution order, support parallel processing, and remain extensible and fault tolerant. The following chapter formalizes this problem and introduces the key questions that guide the proposed solution.

## II. Problem and Approach

How do we build a Book Production System that distribute workload among production cells and thereby limits waiting time? Our Production consists of four production cells: pages printer, cover printer, binder, and packager. To produce a book, each production cell must contribute to its creation. However, this is not a linear production in which production cells are sequenced one after the other. In our case, the system needs to be able to handle both production cells in sequence and in parallel. Furthermore, the system must also be able to integrate new production cells if extra steps are required to produce a book. *How do we create a software system:*

1) that is able to distribute work among multiple production cells?
2) that can handle both linear and non linear dependencies?
3) that can without changing the core architecture integrate new production cells?
4) that makes it possible to change the order of production cells?

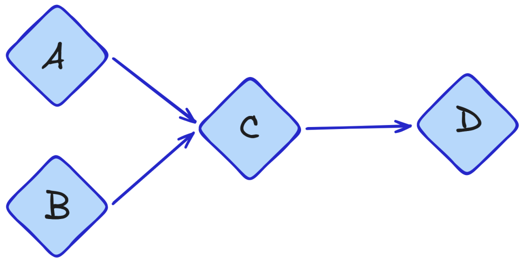


Figure 1. Production Line

5) that can account for failures at production cells, and redistribute work if needed?

### A. Approach

Our approach is to create a scheduling system, that distributes work between production cells and in the right order. As mentioned above, all four production cells must contribute. This creates a recipe for producing a book. The following steps are required: pages printing, cover printing, and binding. Once these three steps have been completed, the book needs to be packaged. So, to produce a book, four jobs must be completed. In our system, each job is processed by a production cell. The book must follow the order seen in figure 1. The scheduling system is responsible for the order of production and that the recipe is fully completed. To account for failures, an observer pattern is added to the scheduling system to monitor production cells. Based on this, the proposed subsystem for the project has been to focus on the production side of the system.

## III. Background and Related Work

### A. Background The shift to Networked Manufacturing

The transition to industry 4.0 (I4.0) represents a fundamental shift in how production environments are organized and integrated. Usually, manufacturing has relied on the hierarchical "automation pyramid" where information flows from field devices to control systems and finally to enterprise planning [2]. However, as shown in the pilot study by Jepsen et al. this hierarchy is being replaced by a "Network Constructed Model" [2].

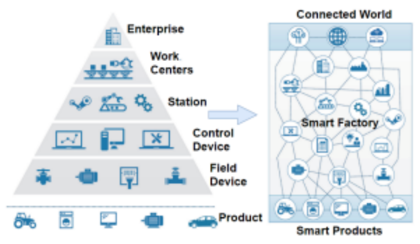


Figure 2. The traditional hierarchical automation pyramid and the I4.0 network constructed model [2]

As seen in the Network Constructed Model, devices, products, and systems must communicate as nodes in a network, enabling flexible and individualized production. This concept is built upon in later research by Jepsen et al. (2021), which emphasizes that flexible production facilities must "adapt to a production order without necessarily changing the layout of the factory" [3].

For the domain of book production, this model is critical. Integrating diverse machines such as page printers, cover printers, binders, and packagers requires moving away from the static hierarchies to a more dynamic, event-driven architecture. This architecture must handle not only sequential steps but also "advanced production processes" such as parallel workflows (e.g. printing covers and pages simultaneously), a requirement that has been identified in modern I4.0 research contexts [3].

### B. Related Work: Challenges and Requirements

While the model of a networked smart factory is well defined, practical implementation faces some issues. Jepsen et al. (2020) examined the readiness of assets like AGVs and robot cells, identifying primary interoperability challenges [2]. Their subsequent work (2021) formalized the requirements for a middleware that can handle complex process flows [3].

These studies highlight three key areas that directly inform our system design:

- **The "Information Backbone" Gap:** To facilitate communication between heterogeneous assets, Jepsen et al. initially proposed the need for an "Information Backbone" (IB) which is a software infrastructure designed to provide a communication layer between assets [2].
- **Asset Maturity and Heterogeneity:** The pilot study found that "interoperability readiness" varies significantly due to missing interfaces or poor documentation [2]. This heterogeneity makes achieving "seamless vertical and horizontal integration" difficult [2].
- **Need for Advanced Process Logic:** Beyond simple connectivity, Jepsen et al. (2021) identify the need for systems that can handle "Complex Processes," including *parallel* and *shared* processes [3]. They argue that to achieve flexibility, facilities must reason about parallelism (two processes running independently) and resource sharing, rather than just linear sequences.

## IV. Use Case

This section describes the primary use cases of the Advanced Book Production system. The system uses a distributed microservices architecture to support customer order submission, backend order management, production coordination, and machine execution through asynchronous communication.

### A. External-Service (Customer Portal)

The External-Service is a web-based portal that serves as the customer-facing user interface. The following use cases describe how customers interact with the system to create and submit book production orders.

- **Customer Creates New Book Production Order:** An external customer submits a new book production order by providing required specifications such as title, author,

number of pages, cover type, page type, and quantity. Upon submission, the system validates the input and forwards the request to the backend. If the request is accepted, an Order ID is returned and the order status is set to `PENDING`.
- **Form Validation and Error Handling:** The system validates user input to ensure that all required fields are provided and values are within acceptable ranges. If validation fails, the user is informed and can correct the input before resubmitting.
- **Order Submission Error Handling:** If a network failure or backend error occurs during order submission, the system informs the user and allows the request to be retried, ensuring robustness against temporary failures.

### *B. API-Gateway*

The `API-Gateway` acts as the single entry point for all external client requests.

- **Request Admission and Protection:** The API-Gateway controls incoming traffic and protects backend services from overload. Requests that exceed allowed limits are rejected.
- **Request Routing and Forwarding:** Requests that pass gateway checks are forwarded to the appropriate backend service for further processing. Invalid requests are rejected before reaching internal services.

### *C. Orchestrator (Order Management Service)*

The `Orchestrator` is responsible for managing production orders and initiating the production workflow.

- **Accept and Process Production Order:** When a valid order request is received, the Orchestrator creates a new production order, assigns a unique Order ID, and prepares the order for production execution.
- **Retrieve Order Details and Status:** Customers can retrieve the current status of an existing order by providing a valid Order ID. The system returns the order information and its current production state.
- **Handle Invalid Requests and Errors:** If an order cannot be found or an error occurs while processing a request, the system returns an appropriate error response.

### *D. Scheduler (Production Coordination)*

The `Scheduler` is responsible for coordinating the execution of production orders across available production cells.

When an order enters production, the Scheduler tracks the progress of individual book units and determines when a unit is ready to proceed to the next production step. Jobs are assigned only when required production dependencies have been satisfied, allowing production steps to execute sequentially or in parallel according to the production recipe.

The Scheduler monitors the availability of production cells and ensures that work is distributed to compatible machines. If a production cell becomes unavailable during execution, the Scheduler redistributes affected work so that production can continue.

### *E. Production Cells (Machine-Level Execution)*

The `Production Cells` represent physical machines responsible for executing individual production steps.

- **Production Cell Executes Assigned Job:** A production cell receives a job assignment from the system and executes the assigned production task. Upon completion, the production cell signals that the task has finished and becomes available for new work.
  If a production cell becomes unavailable, it stops accepting new jobs. Production continues using other available cells, and the unavailable cell may rejoin execution once it becomes operational again.

### *F. Requirements and Non-Functional Requirements*

The described use cases illustrate how the system's central actors and components interact in connection with the creation of order, orchestration and execution of book production. Based on these use cases, a number of requirements can be derived that the system must meet in order to support the project's purpose and defined scope.

In order to create a clear relation between the use cases and the architecture analysis, the most important architecture relevant requirements are summarized in Appendix Figure 5. The table contains both functional and non-functional requirements that are directly derived from the described use cases and the observed interaction patterns between the user interface, backend services, and production machines.

## V. Quality Attribute Scenario

This chapter represents the Quality Attribute Scenarios (QAS) derived from the described use cases in Chapter IV and the architecture-relevant function and non-functional requirements mentioned in Appendix Figure 5. The objective of QAS is to operationalize non-functional requirements, so that they can be used directly to guide architecture design, choice of tactics, and subsequent formal verification and evaluation. While use cases describe the expected behavior of the system in specific workflows, QAS means to make non-functional requirements measurable. This creates a clear link between the projects problem, the defined scope, and the architectural design decisions.

### *A. Relation between Requirements, Scope & QA*

The derived requirements from Appendix Table 5 show that the success of the system does not only depend on the correct functionality of the system. But instead to a large extent on its quality characteristics. The scope of the project focuses on the orchestration, and coordination of book production in a distributed production environment, where machines can be added, removed or failed during operation. This sets special demands on the architecture, which cannot be addressed through our derived functional and non-functional requirements alone.

Based on the analysis of requirements, here are the following Quality Attributes identified as critical for the architecture of the project:

- **Performance** is used to ensure timely handling of machine status and production changes. [4]
- **Scalability** is used to support the expansion of production capacity. [4]
- **Reliability** is used to ensure continued operation in the event of machine failure. [4]
- **Modifiability** is used to enable changes and expansions without extensive rebuilding. [4]
- **Availability** is used to avoid data loss and ensure stable communication. [4]

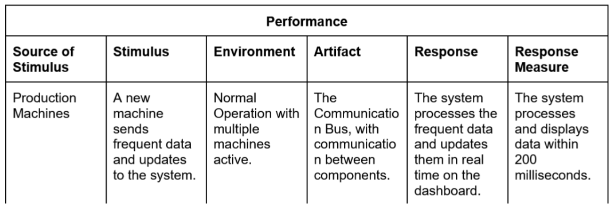
Other Quality Attributes such as reliability and usability have been categorized as secondary in accordance with the prime focus of the project and are therefore not operationalized in the form of QAS in the report.

It is important to note that each Quality Attribute Scenario follows the known 6-step model [5] [6], which is comprised of the following:

- **Source of Stimulus:** The entity that generates the (stimulus/event)
- **Stimulus:** Shows the actual event.
- **Environment:** Is the condition where the stimulus happens.
- **Artifact:** Which part of the system is affected?
- **Response:** How does the system react?
- **Response Measure:** Shows how success is measured.

## *B. Performance*

Performance is an essential quality attribute for the project, since the system continuously receives status updates from multiple machines and has to react quickly on changes during production. Delays in the processing of these events can result in an non-efficient usage of production resources or delayed information of the current system status.



| Performance | | | | | |
|---|---|---|---|---|---|
| **Source of Stimulus** | **Stimulus** | **Environment** | **Artifact** | **Response** | **Response Measure** |
| Production Machines | A new machine sends frequent data and updates to the system. | Normal Operation with multiple machines active. | The Communication Bus, with communication between components. | The system processes the frequent data and updates them in real time on the dashboard. | The system processes and displays data within 200 milliseconds. |

Figure 3. Performance QAS

In order to address the performance quality attribute [7] from the perspective of the project scope, the performance is primarily addressed through tactics within **Control Resource Demand** rather than only increasing resources.

The dominant tactic is to **manage work request** [5] [4], which is achieved through asynchronous (indirect) event-based communication. This reduces the blocking between components and ensures that machines are not waiting for direct responses. The **Reduce Computational Overhead** [5] [4] tactic supports this approach by using MQTT as a lightweight machine level protocol. Tactics from managng resources, such as **Introduce Concurrency** [5] are also used, but as secondary through **parallel event processing** and **horizontal scaling** [8].

This prioritization of **Reduced Load** [5] [4] over aggressive **Resource Increase** [5] [4] reflects on the systems focus on efficient real-time coordination of scheduling production jobs, as shown in figure 3.

## *C. Scalability*

The system is able to manage overload by increasing resources or start up new containers if necessary.

The primary tactic considered here is **Increase Resources** [5] [4] from Performance, which is realized through **horizontal scaling** [8] of central services from Scaling Tactics. Along with this performance-tactics such as **Maintain Multiple Copies of Computations** [5] enable parallel processing of production task. In order to ensure stability, it has been combined with **Bond Queue Sizes** [5] and **Schedule Resources** [5] from Performance, which ensure a controlled overload. By using these tactic-forms, it supports containerized architecture of the project, as shown in figure 6.

## *D. Reliability*

Reliability is important in order to ensure continuous production in the event of machine failures. The focus is mainly to detect faults and recover from faults.

Faults are detected through heartbeat and condition monitoring, enabling a rapid response. In the event of a fault, tactics for recovering from faults such as **Retry, Reconfiguration, and Redundant Spare, and Fault Detection** [5] [4] [9] from the availability quality attribute are activated, where the Scheduler redistributes tasks to functioning machines. Please see figure 7 in the Appendix Section.

## *E. Modifiability*

Unlike Reliability & Scalability, Modifiability is also important since the system needs to be scaled and changed over time. The addressed tactics in this context is **Reduce Coupling** [5], which has been supported by **Increase Cohesion and Defer Binding** [5].

Whereas tactics from **Integribility** such as **Use an Intermediary** [5] [7] has been implemented through Kafka as an event bus, reducing direct dependencies between services. **Encapsulate and Abstract Common Services** [5] ensure stable contracts, while adapter-based integration supports Polymorphism. **Defer Binding** [5] from Modifiability is used through configuration-based and dynamic machine connection, which enables changes without conducting big refactoring to the system. Please see figure 8 in the Appendix Section.

## *F. Availability*

Availability focuses on the systems ability to remain in operation even under partial errors and network problems. Tactics like **Recover from Faults** [5] [9] are used as a primary, which is supported by **Detect Faults** [5] [9] from Availability.

Persistent events in Kafka enable **State Resynchronization** [5] and **Non-stop Forwarding** [5], so that data is not lost in the event of temporary outages. **Removal from Service and Graceful Degradation** [5] [4] are used to isolate failed

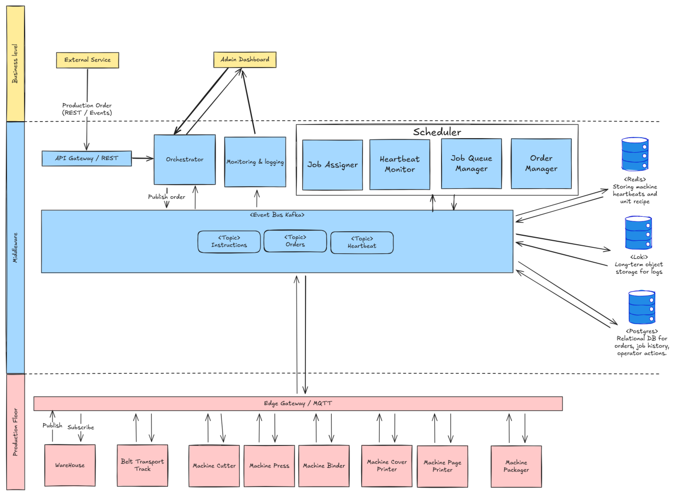


Figure 4. Structural Architecture Modeling

components and maintain reduced functionality rather than complete shutdown. This strategy prioritizes continued operation over complete failure-freedom, as shown in figure 11 in the Appendix Section.

## VI. Design and Analysis Modelling

### A. Architectural Overview

The Advanced Book Production System is designed as a distributed, event-driven microservice architecture intended to coordinate book production across multiple independent production cells. The architecture separates user interaction, order management, scheduling, and machine execution, enabling non-linear and parallel production flows while keeping machine execution decoupled from coordination logic.

This work follows the middleware-oriented approach described by Jepsen et al. [2], [3] by realizing an Information Backbone for service-to-service coordination and an edge-facing communication layer for production assets. Kafka is used as the Information Backbone for durable, asynchronous system events, while machine-level control and monitoring are handled via MQTT through an adapter-based integration approach. The overall architecture is designed at system level; the Scheduler and the MQTT-based machine communication are implemented and evaluated as the main scope of the prototype.

### B. Rationale for Architectural Tactic Selection

The architectural tactics applied in the Advanced Book Production System are selected to satisfy the Quality Attribute Scenarios (QAS) defined in Section V and to realize the architectural concepts of an Information Backbone and middleware-based coordination identified in prior work. The focus is on decoupling, fault isolation, and controlled coordination in a distributed production environment.

*1) Event-Driven Information Backbone:* Kafka is chosen as the system's Information Backbone to address performance, scalability, and availability requirements. The QAS require the system to react to production state changes without blocking and to support non-linear and parallel production flows. An event-driven backbone enables asynchronous communication between services, removing tight temporal coupling between order management and production coordination.

By persisting events, Kafka ensures that production state changes are not lost during temporary service outages, supporting recovery and continued operation in accordance with the availability scenarios.

*2) Separation of Backbone and Machine Communication:* The separation between backbone communication and machine-level communication is a deliberate design decision driven by modifiability and interoperability requirements. The QAS require that new production cells and production steps can be introduced without extensive restructuring of the system.

Treating machine communication as protocol-specific and separate from the backbone allows heterogeneous and legacy production cells to be integrated through adapters. This preserves a stable core architecture while enabling future expansion and evolution of production assets.

*3) Centralized Scheduling with Decentralized Execution:* A centralized Scheduler is selected to satisfy correctness and reliability scenarios that require consistent enforcement of production dependencies and coordinated recovery from failures. The QAS require that production steps with linear and non-linear dependencies execute in the correct order without loss or duplication of work.

Centralized scheduling provides a global view of production state, while execution remains decentralized across independent production cells. This combination supports parallel production and horizontal scalability without coupling machines to global coordination logic.

*4) Dependency-Aware Job Queuing:* Dependency-aware job queuing is used to satisfy performance and correctness requirements. The QAS require that production steps execute only when their prerequisites are fulfilled, while still allowing parallel execution where possible.

Maintaining separate job queues per production step simplifies scheduling decisions and prevents invalid execution sequences, ensuring predictable behavior under concurrent load.

*5) Heartbeat-Based Fault Detection and Recovery:* Heartbeat-based monitoring and automatic job redistribution are selected to satisfy reliability and availability scenarios. The QAS require that production continues despite failures of individual production cells.

Periodic status updates allow the Scheduler to detect failures without synchronous communication or handshakes. When a failure is detected, affected jobs are requeued and redistributed, ensuring continued production without manual intervention. This passive monitoring approach also supports scalability, as additional production cells do not increase coordination overhead. Additionally, queues operates with priorities: low, medium, and high to make sure that units that have been requeued are prioritized. Units that awaits a parallel steps gets priority medium so they will more quicker through the production line.

*6) Periodic Scheduling and Design Trade-offs:* Periodic scheduling represents a pragmatic trade-off between simplicity, correctness, and responsiveness. While, the QAS emphasize timely reaction to production state changes, they do not require strictly event-driven scheduling decisions.
This approach simplifies concurrency handling and state consistency but introduces scheduler-induced waiting time under load. The trade-off is explicitly acknowledged and later validated through empirical evaluation, where waiting time is identified as the dominant performance bottleneck.

*7) Motivation for Formal Verification:* The combination of asynchronous communication, centralized coordination, failure recovery, and timing constraints introduces complex execution paths that cannot be exhaustively validated through testing alone. Several QAS require guarantees related to correctness, availability, and recovery across all possible execution interleavings.
For this reason, the Scheduler, production flow, and failure handling mechanisms are selected for formal verification using UPPAAL, as presented in the following section.

## VII. Formal verification and validation

### *A. Why Verification & Validation?*

The purpose with the formal verification and validation in this project is to reach a systematical and exact analysis of the behavior of the system [10], which cannot be observed through a traditional JUnit-test or simulation alone. Especially in distributed microservices like this project, with error management can result in sudden occurrence of hindrances, which can result in manual inspection as inefficient.

Therefore by using formal models based on Timed Automata [11] [12], the systems behaviour can be analyzed through transitions and locations. This enables us to verify, if the system follows the key properties such as correct error handling, in context to time-constraint without deadlocks across all possible executable paths.

The formal verification is used in the project as supplement to the architectural design process. Please note, that the verification is not used as replacement of an implementation JUnit test. The focus is therefore directed towards the most critical part of the book architectural system, which hereby constitutes of production-flow, machine-distribution and recovery-mechanisms where error do have consequences for the reliability and stability of the system and its performance.

Considering the validation of the system is done through formulation of safety and availability of properties that are related to the identified functional and non-functional requirements from section IV. In that way, the formal verification supports increased confidence that the architecture meets its design goals [10] and that key assumptions about the systems behavior hold under relevant conditions.

### *B. Scope and Focus Area of UPPAAL*

As mentioned in the beginning of the report. The scope area of the project is the Scheduler and MQTT part of the system, where the machines distributes the production order. Therefore in order to match our purpose, we will in the validation & verification section focus on the distributed book production, machine's job-distribution and fail-down and recovery.

### *C. UPPAAL & Timed Automata*

In order to understand Timed Automata, we need to understand two concepts which are "Formal" and "Informal" [13]. Informal Descriptions are used to explain the system in a more general level, whereas Formal Descriptions use exact mathematical models to verify correctness of the system.

In this project Timed Automata has been used with model-checking in UPPAAL, as a formal model to explain the behaviour of the system with time-constraints. The Model expands the classical known state machine diagrams with clocks, guards and invariants, which ensures specification of sequence and duration of events in the system.

### *D. Overall Model Structure in UPPAAL*

The formal method has been built with the network of Timed Automata [10], where each automation represents a central architectural component in the system. The model reflects the overall architecture through seperations of concerns between order, production, machine behaviour and monitoring & recovery. Our UPPAAL Model Checking consists of the following templates.

- **Order Template:** This template models the lifecycle of a production order and the key order states, including pending, orchestrated, completed, and rejected. It is used to analyze proper transition between states and handling retries and timeouts.
- **Scheduler Template:** This template model represents planning, and distribution of production jobs to available machines. It works as a component, that coordinates between orders and production pipelines and is used to analyze correct resource-allocation and job distribution.
- **Pipeline Template:** This template models the productions-flow, where an order goes through different production phases. The time-constraint clocks are used to analyze progress and maximum production time along verification of established time-frames.
- **Machine Template:** This template shows the behavior for individual machines, under different conditions such as normal, error, and recovery. It is used to analyze the availability of machines, management of shutdown and constraint of error.
- **Healthmonitor Template:** This last template models the monitoring of the state of the system with the use of heartbeats. It is used to detect lack of response and to analyze the system's response to errors through alarming situations.

### *E. Time-Constraints & its use.*

Time-Constraints has been modeled with the use of clocks, guards and invariants in more templates [14]. Each clock is used to measure the duration of specific activities and conditions, which enables the analyzing of both response-time, deadlines and recovery-time in the system.

The timing as demonstrated in UPPAAL is critical for distributed production system, since delays can result in long queues of orders, unavailable machines and lack of error-detection. Therefore by modeling time explicitly, the reaction of the system on delays, errors and recovery can be analyzed. The selected time-modeling supports therefore verification of performance-related requirements along with stability of the system. The book production UPPAAL-model uses more clocks, which have a defined responsibility. These are the following:

- **healthclock**: Is used to measure the time, from last heartbeat recieved.
- **machineclock:** Is used to measure work time and recovery-time for a machine.
- **orchestratorclock:** Is used to measure time, especially when an order is being orchestrated or is in a waiting-state.
- **schedulerclock:** Controls the sending of a heartbeat.
- **pipelineclock:** This measures the production time between the pipeline-phases.

As seen above, we have given the clock specified names in order to reduce confusion of time-measurements and which part of the system they are related to.

### *F. Guards & Invariants*

Invariants are used on locations to set an upper time limit for how long the system can stay in a state. In our Pipeline Template, `PRINT_MAX`, `BINDING_MAX` and `PACK_MAX` ensure that each production phase cannot take longer than allowed.

Guards are used on transitions and ensure that changes can only occur when a condition is met. For example, a machine can only leave `Recovering` when $\texttt{machineclock} \geq \texttt{RECOVERY\_TIME}$, and an order can only retry when $\texttt{orchestratorclock} > \texttt{ORCH\_MAX}$. Combined with input/output synchronization [10], this ensures that both timing requirements and correct ordering between templates are enforced.

### *G. Modeling of Deadlines*

Deadlines are modeled with clocks and guards: `orchestratorclock` controls whether an order manages to be orchestrated on time or ends in retry/failure, and the Pipeline limits the entire process with `MAX_PRODUCTION_TIME`.

### *H. Modeling of Recovery Time*

Recovery time is modeled in the Machine template by resetting `machineclock` on `Down` and only allowing a return to `Idle` when the recovery time has been met.

### *I. Modelling of Maximum Production Time*

In the Pipeline-Template, the pipeline clock attribute is used to measure the overall production time through the different production phases. The invariants for each phase limits the allowed time [13], and exceeding of these limitation that lead to an error-state. This modeling ensures, that it is possible to verify that the production is either executed correctly within the time-limit or if it fails in a controlled manner. Please see Appendix figure 9.

### *J. Examining Properties*

In the formal model, the systems correctness is analyzed with the use of both safety- and liveness properties. These properties are used to verify, that the system does not reach a unexpected state in terms of safety.

The **Safety Properties** in the system are used to ensure that the system never finds itself in a shutdown-state [10]. A clear example can be taken from the machine template from Appendix figure 10, where the following verification query defines that a machine in a shutdown-state may never be marked as available.

$$\texttt{A[]}\big(\texttt{Machine0.Down} \Rightarrow \texttt{machineUp[0]} = \texttt{false}\big) \tag{1}$$

Afterwards it has been verified, through the retry's that the orders does not exceed its maximum limits through the following verification query:

$$\texttt{A[]}\big(\texttt{OrderProc.retryCount} > \texttt{MAX\_RETRIES}\big) \tag{2}$$

**Liveness-properties** is used to verify that the system is not halted [10], but things like orchestration of order can happen. This is what the following query does:

$$\texttt{E<>}\big(\texttt{OrderProc.Orchestrated}\big) \tag{3}$$

Whereas it also verifies, that the machines can also recover after error.

$$\texttt{E<>}\big(\texttt{Machine0.Recovering}\big) \tag{4}$$

These properties support the analysis of the execution of the system and ensure that error does not lead to a permanent blockage of the production.

**Deadlock Analysis** in our UPPAAL-model is important, since it is used to avoid deadlocks that can halt the book production system [14]. In this context, we have used the following query to verify that our system design is deadlock free:

$$\texttt{A[]}\big(\texttt{not deadlock}\big) \tag{5}$$

This query ensures, that at least one future transition exists from each state of the model. The result indicates, that there is a flow between the templates of scheduler, pipeline, machine and order which cannot result to circular dependencies or waiting state without an exit.

The deadlock analysis thereby supports the stability of the system, and confirms that the components of the architecture are modeled correctly.

### K. *Validation of Quality Attributes & Requirements*

The verification of the system is not only used to analyze the behavior of the system, but it is also used to validate the architecture in accordance with the functional and non-functional requirements.

In this context, the following pipeline related query supports the validation of performance in our architecture and time. Please see Appendix (figure 12.blue).

This property validates, that the production is always executed within the defined time limit. In the same manner, the following two queries from the Appendix section, related to HealthMonitor (figure 12.red) and Machine Template (figure 12.green) checks the reliability and availability quality attributes. These queries show that the proposed system is consistent with the system requirements and the overall architecture.

### L. *Limitations and Overall Evaluation of Verification*

The formal model contains simplifications, since it makes the verification possible to execute within the time-limits in UPPAAL. An example can be seen in figure 13, where the amount of machines have been defined to `N_MACHINES` = 2. This limitation has been imposed, since the model-checking with higher amount of machines resulted in a longer execution time for verification queries with a high usage of RAM.

Although the limitation does not affect the models purpose, since the focus is not to analyze scalability in the amount of machines, but instead understand the behavior of the system in terms of job-scheduling. This behavior can be scaled to more machines if the amount of machines are increased.

Otherwise the interaction between the components is abstracted with the use of synchronization channels [14] [13] instead of a detailed network behavior. Time parameters such as production time, timeout and recovery time have been fixed defined in order to set boundaries. These boundaries would enable us to understand worst-case scenarios, if the system comes to a halt.

Overall the formal verification shows that the system is working correct under the right circumstances designed for the project. The verification shows reliability & stability to the proposed architectural design and shows that the main design decision supports the quality attributes of reliability, availability, and performance. Even though the model is abstracted, it works as a strong analytical tool that can validate the systems main properties early in the design phase.

## VIII. EVALUATION

### A. *Experiment Design*

This section validates on the schedulers ability to distribute jobs efficiently and manage production cell errors in a robust manner. In the experiment, a observational quantitative design [15] with data from 100 batch-test orders were collected from the Unified Scheduler Service. The system builds on the heartbeat-monitoring, where machines sends status every second through MQTT. The lack of heartbeat in 3 seconds triggers an automatic error-detection and job-requeue. The data has been afterwards exported from TimescaleDB after being executed in a Docker-Environment and processed in R-Studio [16].

### B. *Measurements & Hypothesis*

The main research question for our experiment is: What factors affect completion time, and does fault tolerance work effectively? We measured completion-time-minutes as outcome, with 3 factors. These factors are **quantity for order-size, total-requeues with error-management, and wait-time-seconds for scheduler inefficiency**.

Based on the measurement factors, we have created the following hypothesis:

- **Null Hypothesis: [17]** The three predictors (quantity, total-requeues, wait-time-seconds) have no significant effect on completion-time-minutes.
- **Alternative Hypothesis: [17]** At least one predictor significantly affects completion-time-minutes.

A pilot test with 10 orders verified that the system detected production cell errors within 3 seconds and resumed production after restart. This confirmed that the heartbeat mechanism worked, and we continued afterwards with 100 orders for statistical validity.

### C. *Results & Interpretation*

Based on the analysis of the order-size in R [16], we have got the following results.

- **Descriptive Statistics:** The average completion time is 61.86 ms, while the waiting-time is 56 ms, re-queues = 48.4 and the order-size is 203 items.
- **Correlation Analysis:** This analysis shows that the waiting-time had almost a perfect correlation with completion time (r=0.99) [16], while quantity (r=0.10) and re-queues (r=-0.04) had minimal effect on completion-time.
- **Multiple Lineare Regression:** Here we see that we have got a determination coefficient equivalent to 0.985, and F(3.96)=2179, with a p-value below 0.001 explained 98.55 % of the variation.

Looking closer at each factor, we see the following:

- **Wait-time (seconds):** ($\beta = 0.016$, $p < 0.001$). Each second of waiting time adds 0.016 ms to the completion time. Therefore, the alternative hypothesis is accepted, indicating that waiting time is the dominant factor. The scheduler causes orders to wait unnecessarily long before job assignment, which represents the primary bottleneck of the system. Please see Appendix figure 17.
- **Total requeues:** ($\beta = 0.054$, $p < 0.001$). Each requeue adds only a small amount of time to the overall completion time. The alternative hypothesis is accepted, showing that requeuing has a minimal impact. This supports the architectural decision of using heartbeat-based error detection with a 3-second timeout and automatic requeue, which proves to be effective. Please see Appendix figure 17.

- **Quantity:** ($\beta = 0.007$, $p = 0.309$). The result is not statistically significant, and therefore the null hypothesis is accepted. This indicates that the order size does not have a measurable effect on the completion time. Although the system scales perfectly linearly, confirming that parallel processing (jobs A and B running concurrently) prevents production bottlenecks. Please see Appendix figure 17.

*1) Residual Graphs:* Based on the given graphs, it is seen that there is a normal spread of values in the QQ-Graph, Histogram and Residual-Plot as shown in Appendix (figure 14, 15, and 16) with only 2-3 outliers [16], for data amount equal to 100.

### D. Discussion & Recommendations

**Scheduler Optimization:** The regression analysis identifies waiting time as the dominant factor due to it being significant. The waiting time accounts for almost all of the variation in the completion time. With an average waiting-time of 56 ms before production starts. There are three recommendations. The first one is to reduce the job assignment polling interval from 500 ms to 100 ms in the configured appsettings.json file.

The second refactor to make, is the scheduler in order to support concurrent order processing rather than sequential handling. And then we have the third, were we scale horizontally by adding more machines of each type to increase capacity during peak load.

Then we have the maintaining fault tolerance. The fault tolerance mechanisms do not require any changes, since requeue only adds 3 seconds of overhead per event, despite an average of 48 errors per order, showing that the heartbeat interval and error detection timeout are optimally balanced. This validates our architectural decision to use lightweight heartbeat based monitoring rather than explicit acknowledgment messages.

## IX. Conclusion

This work set out to address the problem of how to design a book production system that effectively distributes workload among multiple production cells while minimizing waiting time, supporting flexible production flows, and remaining robust to machine failures. To answer this, we designed and evaluated a scheduler architecture capable of handling parallel production, dependency-aware job execution, dynamic reconfiguration, and fault tolerance through heartbeat-based monitoring and automatic job re-queuing.

The experimental results from the 100 batch-test orders demonstrates that the proposed architecture successfully meets the core research objectives. The system is able to distribute work across multiple production cells, support both linear and non-linear dependencies, and scale horizontally without performance degradation (Research Questions 1-3, see II). This is further proved by the fact that order quantity has no statistically significant effect on completion time when controlling for waiting time and re-queues, confirming linear scalability and efficient parallel processing.

Furthermore, the fault tolerant mechanisms performed very well. Machine failures are detected within three seconds, and requeued jobs introduce only minimal overhead. Despite an average of over 48 requeue events per order, the total impact on completion time remains negligible. This confirms that the system can handle production cell failures and dynamically redistribute work without disrupting overall throughput, directly addressing Research Question 5, see II.

However, the evaluation also reveals a critical limitation, the scheduler induced waiting time dominates total completion time. Statistical analysis shows a correlation between waiting time and completion time, identifying scheduler inefficiency as the primary bottleneck in the current implementation. While the system architecture is fundamentally right, and supports reconfigurable production flows (Research Question 4, see II), improvements in scheduling logic are required to fully realize its performance potential.

Overall, this study demonstrates that it is possible to build a flexible, fault tolerant, and scalable book production system that satisfies the stated research questions 1-5, see II. The findings confirm that architectural decisions, such as particularly dependency-based queuing and heartbeat-driven fault detection are effective, while also highlighting scheduler optimization as the key area for further improvement.

### A. Discussion

The findings demonstrate that robustness and scalability can be achieved in distributed production systems without significant performance reduction. The heartbeat-based fault tolerance mechanism provides rapid failure detection and recovery with minimal overhead. The evaluation also reveals that scheduler-induced waiting time dominates total completion time, accounting for nearly all observed performance variation. This highlights that, in distributed production environments, scheduling strategy has a greater impact on throughput than fault tolerance or dependency management. While the system supports flexible workflows and horizontal scaling, its current scheduler implementation does not fully exploit available production capacity under load, making the scheduler optimization the primary opportunity for performance improvement. **Please note, that Generative AI has been used to support and ask help for this project.** [18].

### B. Future work

Based on the findings of this study, several directions for future work are identified:

1) **Integrate the rest of the components:**
   Even if the core architecture is in place, there are still some components missing that needs to be integrated into the whole and should communicate with other different components such as the orchestrator, admin console, and monitoring tools.
2) **Advanced Scheduler Optimization:**
   Future implementations should focus on reducing waiting time through more responsive and intelligent scheduling strategies. this includes decreasing job assignment polling intervals, enabling true parallel order scheduling, and

exploring event-driven scheduling rather than periodic polling.

3) **Dynamic Load-Aware Scheduling:**
Incorporating real-time machine load, queue lengths, and job priorities into scheduling decisions could further reduce idle time and improve throughput, particularly during peak demand periods.
4) **Large-Scale and Real-World Deployment:**
While the current evaluation demonstrates strong results in a controlled Docker environment, future studies should validate the system under larger-scale deployments and real production conditions, to produce real machine info and failure patterns to correctly address them.

By addressing these areas, we can improve the system to potentially reduce waiting time, improve completion performance, and further strengthen the use of such an architecture to industrial-scale book production and similar distributed manufacturing systems.

## References

[1] IBM, "What is industry 4.0?" accessed: 2025-12-17. [Online]. Available: https://www.ibm.com/think/topics/industry-4-0

[2] S. C. Jepsen, T. I. Mørk, J. Hviid, and T. Worm, "A pilot study of industry 4.0 asset interoperability challenges in an industry 4.0 laboratory," in *Proceedings of the 2020 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*. IEEE, 2020, pp. 1–5.

[3] S. C. Jepsen, T. Worm, A. Johansen, S. Lazarova-Molnar, M. B. Kjærgaard, E.-Y. Kang, J. Friederich, J. E. H. Mena, R. Soltani, S. L. Sørensen, and J. H. Schwee, "A research setup demonstrating flexible industry 4.0 production," in *Proceedings of the 2021 International Symposium ELMAR*. IEEE, 2021, pp. 143–150.

[4] J. S. Hawker and R. Kuehl, "Quality attribute scenarios and tactics (lecture 17)," Lecture slides (PDF), SWEN-440 Requirements and Architecture, Rochester Institute of Technology (RIT), 2017, accessed: 2025-12-18. [Online]. Available: https://www.se.rit.edu/~swen-440/slides/instructor-specific/Kuehl/Lecture%2017%20Quality%20Attributes%20and%20Tactics%20LV.pdf

[5] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 4th ed. Addison-Wesley Professional, 2022.

[6] Ravindra College of Engineering for Women, "It6602 software architecture: Understanding quality attributes (sa unit 3)," Course notes (PDF), 2021, accessed: 2025-12-18. [Online]. Available: https://www.recw.ac.in/v1.8/wp-content/uploads/2021/10/SA_Unit3.pdf

[7] T. Worm, "Lecture 2 and lecture 3 (powerpoint slides)," Course material, University of Southern Denmark (SDU), available via Itslearning, 2025, accessed: 2025-12-18.

[8] S. Modi, "Scalability strategies in system architecture: Exploring horizontal and vertical scaling," Medium (Another Integration Blog), Feb. 2024, accessed: 2025-12-18. [Online]. Available: https://medium.com/another-integration-blog/scalability-strategies-in-system-architecture-exploring-horizontal-and-vertical-scaling-2b2084e9b78f

[9] R. Kazman, "Tactics and patterns for software robustness," Carnegie Mellon University, Software Engineering Institute's Insights (blog), Jul 2022, accessed: 2025-12-18. [Online]. Available: https://www.sei.cmu.edu/blog/tactics-and-patterns-for-software-robustness/

[10] E.-Y. Kang and I. Riaz, "Lectures 8–10: Uppaal (verification & validation and input/output synchronization)," Course lectures/slides, University of Southern Denmark (SDU), available via Itslearning, 2025, accessed: 2025-12-15.

[11] Visual Paradigm, "What is state machine diagram?" Visual Paradigm UML Guide (web page), n.d., accessed: 2025-12-18. [Online]. Available: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-state-machine-diagram/

[12] A. David, T. Amnell, and M. Stigge, "Uppaal 4.0: Small tutorial," Tutorial (PDF) covering UPPAAL v4.0.7, Nov 2009, version history credits updates by the listed authors; distributed via course material (Itslearning).

[13] J. Proenza, "The uppaal model checker," Slides (PDF), Systems, Robotics and Vision Group, UIB (Spain), Oct 2008, distributed via course material (Itslearning).

[14] A. Vörös, "Short uppaal introduction: practical guideline of uppaal usage," Budapest University of Technology and Economics, Dept. of Measurement and Information Systems (PDF), Oct 2013, based on a document by Dániel Darvas and Gergő Horányi; distributed via course material (Itslearning).

[15] M. Alipour, "Lecture 8: Study design (powerpoint slides)," Course slides, University of Southern Denmark (SDU), available via Itslearning, n.d., human-Centered Software Design course material. Accessed: 2025-12-18.

[16] M. Lech Cantuaria and V. Blanes Vidal, "Lecture 12: Rstudio and multiple linear regression (powerpoint slides)," Course slides, University of Southern Denmark (SDU), available via Itslearning, n.d., first-semester Bachelor course material. Accessed: 2025-12-18.

[17] A. Umair, "Lecture 3: Role and use of theory (powerpoint slides)," Scientific Methods course slides, University of Southern Denmark (SDU), available via Itslearning, 2025, includes terminology such as directional vs. non-directional hypotheses. Accessed: 2025-12-18.

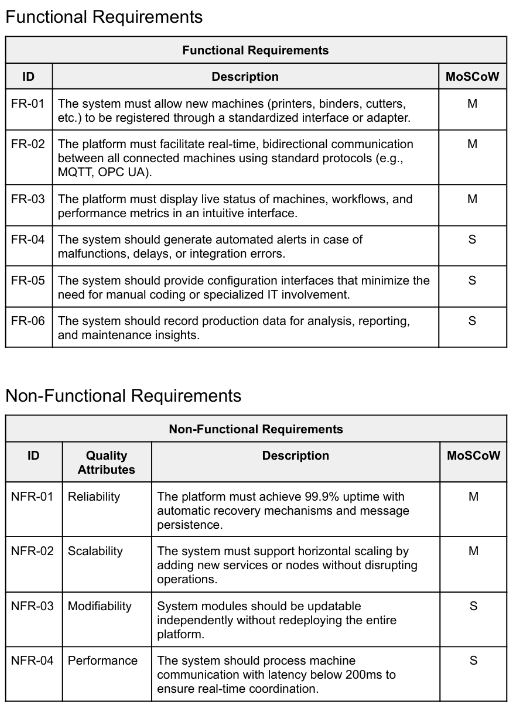
[18] OpenAI, "Chatgpt," Large language model (LLM) accessed via web interface, 2025, used as a supporting tool during the project (e.g., outlining, wording, and clarifying concepts). Accessed: Throughout Semester E-2025.

[19] vimis22, "Advanced-architecture," GitHub repository, 2025, public repository ("This is a Group Collaboration."). Accessed: 2025-12-18. [Online]. Available: https://github.com/vimis22/Advanced-Architecture

## X. Appendix

Table I
Group member contributions.

| Group Member | Sections Contributed | Code Contribution Part |
|---|---|---|
| Anders | Problem and Approach, Use-Cases & Design and Analysis & Experiment | Scheduler, Edge Gateway MQTT, Kafka Event bus, Databases |
| Kasper | Related Work & Design and Analysis | Scheduler |
| Marco | Abstract, Introduction, Conclusion, Discussion & Future Work | Kafka Event Bus |
| Mohamed | Use-Cases | External-Service |
| Vivek | Use-Cases, Quality Attribute Scenarios, Formal Verification & Experiment-Evaluation | External-Service, API-Gateway, & Orchestrator |



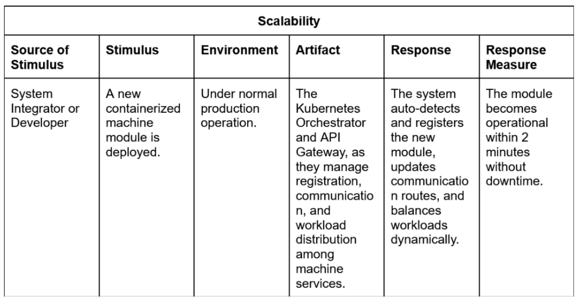
Functional Requirements

| Functional Requirements | | |
|---|---|---|
| ID | Description | MoSCoW |
| FR-01 | The system must allow new machines (printers, binders, cutters, etc.) to be registered through a standardized interface or adapter. | M |
| FR-02 | The platform must facilitate real-time, bidirectional communication between all connected machines using standard protocols (e.g., MQTT, OPC UA). | M |
| FR-03 | The platform must display live status of machines, workflows, and performance metrics in an intuitive interface. | M |
| FR-04 | The system should generate automated alerts in case of malfunctions, delays, or integration errors. | S |
| FR-05 | The system should provide configuration interfaces that minimize the need for manual coding or specialized IT involvement. | S |
| FR-06 | The system should record production data for analysis, reporting, and maintenance insights. | S |

Non-Functional Requirements

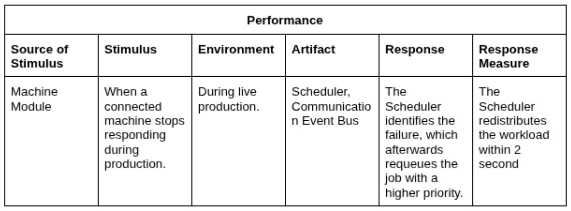
| Non-Functional Requirements | | | |
|---|---|---|---|
| ID | Quality Attributes | Description | MoSCoW |
| NFR-01 | Reliability | The platform must achieve 99.9% uptime with automatic recovery mechanisms and message persistence. | M |
| NFR-02 | Scalability | The system must support horizontal scaling by adding new services or nodes without disrupting operations. | M |
| NFR-03 | Modifiability | System modules should be updatable independently without redeploying the entire platform. | S |
| NFR-04 | Performance | The system should process machine communication with latency below 200ms to ensure real-time coordination. | S |

Figure 5. Requirements Table



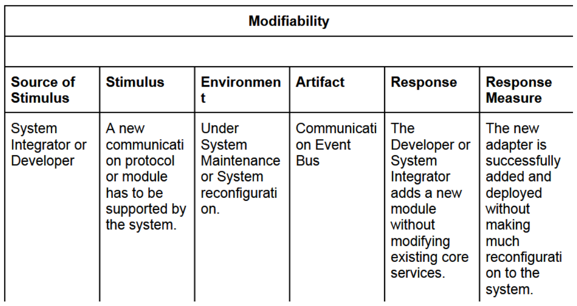
| Scalability | | | | | |
|---|---|---|---|---|---|
| Source of Stimulus | Stimulus | Environment | Artifact | Response | Response Measure |
| System Integrator or Developer | A new containerized machine module is deployed. | Under normal production operation. | The Kubernetes Orchestrator and API Gateway, as they manage registration, communication, and workload distribution among machine services. | The system auto-detects and registers the new module, updates communication routes, and balances workloads dynamically. | The module becomes operational within 2 minutes without downtime. |

Figure 6. Scalability QAS



| Performance | | | | | |
|---|---|---|---|---|---|
| Source of Stimulus | Stimulus | Environment | Artifact | Response | Response Measure |
| Machine Module | When a connected machine stops responding during production. | During live production. | Scheduler, Communication Event Bus | The Scheduler identifies the failure, which afterwards requeues the job with a higher priority. | The Scheduler redistributes the workload within 2 second |

Figure 7. Reliability QAS



| Modifiability | | | | | |
|---|---|---|---|---|---|
| Source of Stimulus | Stimulus | Environment | Artifact | Response | Response Measure |
| System Integrator or Developer | A new communication protocol or module has to be supported by the system. | Under System Maintenance or System reconfiguration. | Communication Event Bus | The Developer or System Integrator adds a new module without modifying existing core services. | The new adapter is successfully added and deployed without making much reconfiguration to the system. |

Figure 8. Modifiability QAS

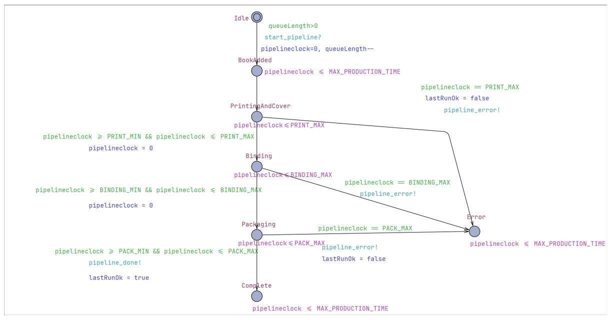

Idle
queueLength>0
start_pipeline?
pipelineclock=0, queueLength--
BookAdded
pipelineclock ⩽ MAX_PRODUCTION_TIME
pipelineclock == PRINT_MAX
lastRunOk = false
pipeline_error!
PrintingAndCover
pipelineclock ⩽ PRINT_MAX
pipelineclock ⩾ PRINT_MIN && pipelineclock ⩽ PRINT_MAX
pipelineclock = 0
Binding
pipelineclock ⩽ BINDING_MAX
pipelineclock == BINDING_MAX
pipeline_error!
pipelineclock ⩾ BINDING_MIN && pipelineclock ⩽ BINDING_MAX
pipelineclock = 0
Error
Packaging
pipelineclock == PACK_MAX
pipelineclock ⩽ PACK_MAX
pipeline_error!
lastRunOk = false
pipelineclock ⩽ MAX_PRODUCTION_TIME
pipelineclock ⩾ PACK_MIN && pipelineclock ⩽ PACK_MAX
pipeline_done!
lastRunOk = true
Complete
pipelineclock ⩽ MAX_PRODUCTION_TIME

Figure 9. Pipeline Template and Excution of States

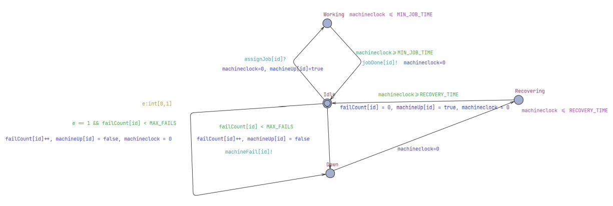
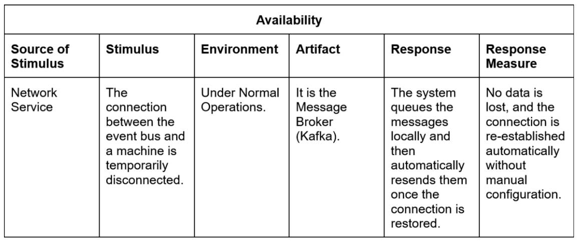
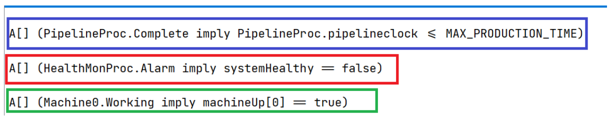

Working machineclock ⩽ MIN_JOB_TIME
assignJob[id]?
machineclock=0, machineUp[id]=true
machineclock ⩾ MIN_JOB_TIME
jobDone[id]! machineclock=0
Idle
machineclock ⩾ RECOVERY_TIME
Recovering
e:int[0,1]
failCount[id] = 0, machineUp[id] = true, machineclock = 0
machineclock ⩽ RECOVERY_TIME
e == 1 && failCount[id] < MAX_FAILS
failCount[id]++, machineUp[id] = false, machineclock = 0
failCount[id] < MAX_FAILS
failCount[id]++, machineUp[id] = false
machineFail[id]!
machineclock=0
Down

Figure 10. Machine Template shows a non-shutdown transition between nodes & query



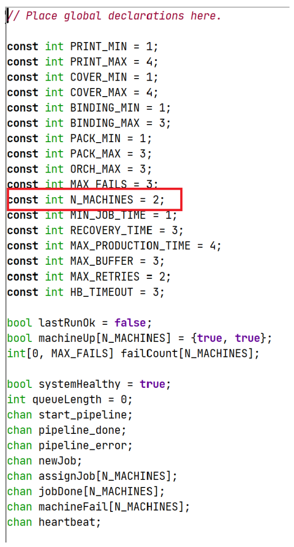
| Availability | | | | | |
|---|---|---|---|---|---|
| Source of Stimulus | Stimulus | Environment | Artifact | Response | Response Measure |
| Network Service | The connection between the event bus and a machine is temporarily disconnected. | Under Normal Operations. | It is the Message Broker (Kafka). | The system queues the messages locally and then automatically resends them once the connection is restored. | No data is lost, and the connection is re-established automatically without manual configuration. |

Figure 11. Availability QAS



```
A[] (PipelineProc.Complete imply PipelineProc.pipelineclock ≤ MAX_PRODUCTION_TIME)
A[] (HealthMonProc.Alarm imply systemHealthy == false)
A[] (Machine0.Working imply machineUp[0] == true)
```

Figure 12. Health and Machine Query



```
// Place global declarations here.

const int PRINT_MIN = 1;
const int PRINT_MAX = 4;
const int COVER_MIN = 1;
const int COVER_MAX = 4;
const int BINDING_MIN = 1;
const int BINDING_MAX = 3;
const int PACK_MIN = 1;
const int PACK_MAX = 3;
const int ORCH_MAX = 3;
const int MAX_FAILS = 3;
const int N_MACHINES = 2;
const int MIN_JOB_TIME = 1;
const int RECOVERY_TIME = 3;
const int MAX_PRODUCTION_TIME = 4;
const int MAX_BUFFER = 3;
const int MAX_RETRIES = 2;
const int HB_TIMEOUT = 3;

bool lastRunOk = false;
bool machineUp[N_MACHINES] = {true, true};
int[0, MAX_FAILS] failCount[N_MACHINES];

bool systemHealthy = true;
int queueLength = 0;
chan start_pipeline;
chan pipeline_done;
chan pipeline_error;
chan newJob;
chan assignJob[N_MACHINES];
chan jobDone[N_MACHINES];
chan machineFail[N_MACHINES];
chan heartbeat;
```

Figure 13. Global Declarations

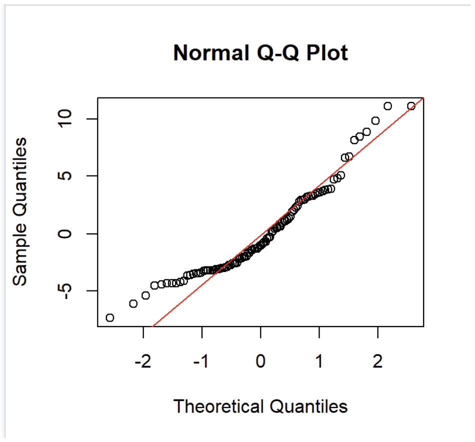


Figure 14. QQ Graph

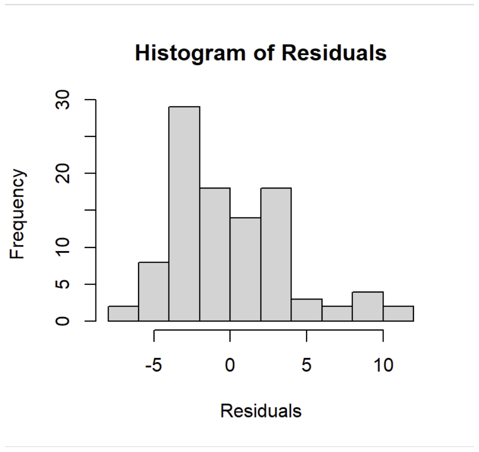


Figure 15. Histogram

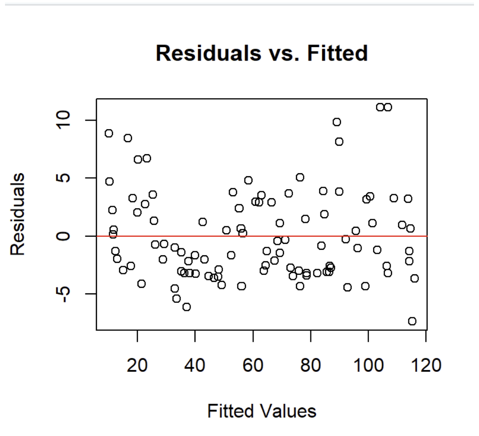
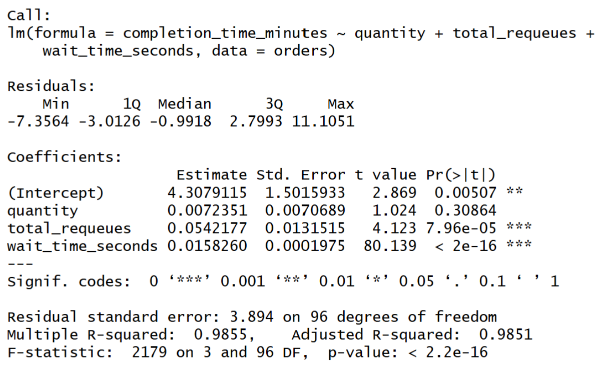


Figure 16. Residual Graph



```
Call:
lm(formula = completion_time_minutes ~ quantity + total_requeues +
    wait_time_seconds, data = orders)

Residuals:
    Min      1Q  Median      3Q     Max
-7.3564 -3.0126 -0.9918  2.7993 11.1051

Coefficients:
                   Estimate Std. Error t value Pr(>|t|)
(Intercept)       4.3079115  1.5015933   2.869  0.00507 **
quantity          0.0072351  0.0070689   1.024  0.30864
total_requeues    0.0542177  0.0131515   4.123 7.96e-05 ***
wait_time_seconds 0.0158260  0.0001975  80.139  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.894 on 96 degrees of freedom
Multiple R-squared:  0.9855,    Adjusted R-squared:  0.9851
F-statistic:  2179 on 3 and 96 DF,  p-value: < 2.2e-16
```

Figure 17. Multiple Lineare Regression Values from R