Architecting and Evaluating an Event-Driven Control System for Pasta Production

Aljaz Knific*, Anders S. Nissum*, Dimitrios Dafnis*, Leonardo Zanoni*,
University of Southern Denmark, SDU Software Engineering, Odense, Denmark
Email: * {alkni25,anden21,didaf25,lezan25}@student.sdu.dk

***Abstract*—Introduction: Modern production systems depend on software-intensive control architectures to coordinate heterogeneous machines, react to changing demand, and tolerate faults.**

Gap: While event-driven architectures and availability tactics are well described in the literature, there is limited empirical and formal evidence of how such designs behave in a concrete industrial-like setting under frequent failures and dynamic reconfiguration.

Aim: This paper aims to architect and evaluate an event-driven control system for an automated pasta production line, focusing on the quality attributes of Availability, Modifiability, and Scalability.

Method: The pseudo-layered, microservices architecture centres on a Scheduler and Shadow Scheduler communicating with machines and supporting services via Kafka and MQTT, with Redis providing coordination for leader election and failover. We capture the main behaviours through use cases and quality attribute scenarios, construct a formal model in UPPAAL to verify key properties (e.g. storage threshold compliance, scheduler redundancy, component decoupling), and conduct a controlled experiment with chaos-based fault injection to measure reaction latencies.

Results: The formal analysis confirms correct enforcement of storage bounds and safe failover behaviour, while the empirical study shows that storage alerts are handled within 5 seconds and new machines are integrated into the production plan within 10 seconds on average. Together, these results indicate that the chosen architectural tactics effectively support the targeted quality attributes.

Acknowledgment

Portions of the text have been revised with the assistance of an AI-based grammar and style tool.

I. Introduction and Motivation

Modern production environments increasingly rely on software-intensive systems to coordinate heterogeneous machines, react to fluctuating demand, and tolerate failures. In such settings, architectural decisions strongly influence the ability of the system to remain available under faults, to scale with additional resources, and to be modified safely as requirements evolve. This paper investigates these concerns in the context of an automated pasta production system, where a central Scheduler coordinates multiple cutting and sorting machines based on storage levels and machine health. Our motivation is to design and analyse an architecture that not only fulfils the required functionality, but also provides quantitative evidence that key quality attributes; such as modifiability, availability and scalability; are achieved.

II. Problem and Approach

A. Problem

In developing the initial architectural concept, our team faced the challenge of designing a system that would be both sufficiently comprehensive to represent a realistic industrial production environment and structured enough to satisfy the constraints defined at the outset of the project. We needed an architecture that could meaningfully coordinate heterogeneous machines, support event-driven decision-making, and reflect the operational complexity of a real pasta production process, while still enforcing the quality attributes and domain requirements imposed by the problem context. These considerations led us to articulate the fundamental problem addressed in this work and to derive the research questions that guided the architectural design, formal modeling, and empirical evaluation.

B. Research questions

The intention of this paper is to answer the following research questions:

1) How can architectural tactics and design decisions be applied to achieve the required levels of modifiability, availability, and scalability in the pasta production system?
2) To what extent do formal verification and empirical experiments provide consistent, quantitative evidence that the implemented architecture satisfies its quality attribute scenarios?

C. Approach

The following steps are taken to answer this paper's research questions:

1) We derive the architecture using Attribute-Driven Design, identifying key quality attributes and selecting tactics such as redundancy, heartbeat-based fault detection, and event-based decoupling.
2) We construct a formal model of the critical components in UPPAAL and verify properties related to storage thresholds, scheduler redundancy, and component interaction.
3) We implement the architecture and design an empirical experiment under controlled chaos (random failures) to measure latencies and event frequencies for the main quality attribute scenarios.
4) We compare and interpret the formal and empirical results to assess how well the architecture meets its quality goals

and to derive implications for future design improvements.

III. Related Work

The following subsection outlines the design principles and architectural choices that guided the development of the system in light of the research discussed above.

A. Architectural Design and Tactics

Our design process follows the Attribute-Driven Design method described by Cervantes and Kazman [1]. Rather than focusing solely on functional requirements, we prioritized high-priority Quality Attributes specifically Modifiability, Availability, and Scalability to drive the decomposition of the system. To satisfy these attributes, we selected specific architectural *tactics* as defined by Bass et al. [2].

- To achieve **Availability**, we implemented the *Shadow* tactic via a "Shadow Scheduler" that maintains synchronized state.
- To achieve **Modifiability** and **Scalability**, we employed the *Intermediary* tactic and the *Restrict Dependencies* tactic introducing an event bus to decouple the production scheduler from the physical machines and make the components of the system independent of each other.

These decisions align with Bass's assertion that architectural structures must be chosen specifically to promote desired quality attributes [2].

B. Event-Driven Architecture

To realise the tactics described above, we adopted an Event-Driven Architecture. As noted in recent literature on microservices [3], event-driven architectures enable asynchronous communication and strong decoupling between components, which are essential in distributed systems that must scale dynamically and tolerate partial failures. In our design, Kafka acts as the central event bus between the Java-based core components (Scheduler, Shadow Scheduler, Database and adapter), while MQTT is used for lightweight edge communication with machines. By relying on Kafka's publish-subscribe model instead of synchronous request response interactions, we avoid the tight coupling and cascading failure modes associated with synchronous orchestration [3] a failure in one consumer does not block producers or other consumers, supporting both availability and scalability.

C. Empirical Evaluation

Our empirical evaluation follows the experimentation process defined by Wohlin et al. [4]. The evaluation is organised according to the standard phases:

1) **Definition:** Stating the goal of the experiment (to assess performance and availability of the architecture under faults) and identifying the relevant quality attribute scenarios and response variables.
2) **Planning:** Selecting independent variables (e.g. number of available machines, number of killable components) and dependent variables (e.g. scheduler latency, total response time), designing the chaos-based failure injection, and defining the data collection procedures.
3) **Execution:** Running the implemented system under the defined scenarios, injecting failures into machines and the scheduler, and logging all events and timestamps via Kafka over a fixed observation period.
4) **Analysis:** Applying descriptive statistics and requirement-based analysis to the collected data in order to evaluate whether the measured latencies satisfy the stated requirements and to interpret the architectural implications.

This systematic approach ensures that our conclusions regarding system latency, failure recovery, and scalability are grounded in a controlled experimental design, in line with the recommendations of Wohlin et al. [4].

IV. Use Cases

The system is designed to handle complex production workflows, focusing on dynamic adaptability and fault tolerance. In our design process, we first identified a set of functional and non-functional requirements, then derived primary use cases from these requirements, and finally refined them into concrete quality attribute scenarios and architectural concerns.

The behaviour of the system is driven by a combination of functional and non-functional requirements. The functional requirements in Table I capture what the system must do in terms of pasta production, monitoring, and reaction to events, while the non-functional requirements in Table II express the desired quality attributes that shape the architecture.

TABLE I: Functional requirements

ID	Description
FR-1	The system shall produce fresh pasta of type A
FR-2	The system shall produce dried pasta of type A
FR-3	The system shall produce fresh pasta of type B
FR-4	The system shall produce dried pasta of type B
FR-6	Each machine shall communicate if it's working correctly using heartbeats
FR-7	The scheduler checks the heartbeats and checks if there are missing or new ones.
FR-11	The system shall have the product packaged in less than 10 seconds
FR-12	The sorting system shall configure whether the product must go to the drying machine or directly to packaging
FR-13	When the production plan gets modified, the machine should react in $\leq$ 10 seconds
FR-14	When storage exceeds the limit for product A or B, the DB should inform the scheduler in $\leq$ 5 seconds
FR-15	After switching blade to a cutting machine, it must ignore repeated threshold triggers for 5 seconds

Together, these requirements define both the operational workflow (e.g. storage monitoring, heartbeats, reaction times) and the expected quality levels (e.g. availability, scalability, security). They provide the basis for the primary use cases that we use to drive architectural decisions and later verification and evaluation activities.

TABLE II: Non-functional requirements

ID	Description
NFR-5	The system shall adjust production based on the current system state
NFR-8	The system shall allow software updates with minimal downtime
NFR-9	The system shall support adding new machines without redesign.
NFR-10	The system must operate continuously 24/7 with an uptime of at least 99.9%
NFR-16	The system shall support modular replacement of devices without downtime.
NFR-17	All external accesses must be secured (only dashboard via Redis cache)
NFR-18	All operator actions in the system shall be logged for auditing.
NFR-19	All messages between devices and the bus must be delivered at least once.

Based on these requirements, we derived a small set of primary use cases that represent the most important interactions with the system. These use cases, summarised in Table III, were then used as drivers for the architectural design, the UPPAAL model, and the empirical experiment.

Use cases UC-1 and UC-2 capture the core dynamic behaviour of the system: reacting to storage-level changes and handling machine status changes through heartbeats and failures. In the next step, we refined these use cases into quality attribute scenarios that specify measurable response times and robustness properties, which are then used as the main evaluation criteria.

V. Quality Attribute Scenarios

While the functional and non-functional requirements state what is needed at a high level, the quality attribute scenarios in Table IV make these requirements operational by binding them to concrete situations, responses, and time bounds. Each scenario is explicitly linked to one or more use cases, ensuring traceability from requirements through design to evaluation. These scenarios also provide the metrics used in our empirical evaluation.

Finally, when deriving the architecture from these scenarios, we also identified a set of explicit constraints that restrict possible design options. Table V summarises the key architectural constraints that were assumed throughout the project, such as centralised scheduling and dependence on specific event-driven technologies. These constraints are important when interpreting both the design decisions and the evaluation results.

TABLE III: Primary use cases

Use case	Description
UC-1	**Storage Level-Based Production Adjustment:** The database monitors storage levels and triggers an alert when a unit exceeds 80% capacity or drops below 20%. The Scheduler receives the alert via the Kafka bus, evaluates machine availability and demand, and updates the production plan (e.g. blade switch, reroute belt) to avoid overfilling or understocking.
UC-2	**Change in Machine Status (Heartbeat & Failure):** Machines periodically send heartbeat messages via Kafka. The Monitoring Subsystem detects missing or new heartbeats and flags status changes. The Scheduler reacts by redistributing workload from failed machines or integrating new machines into the production plan, enabling self-healing and scalable operation.

TABLE IV: Primary quality attributes

ID	Quality attribute	Scenario (informal)	Allocated use cases
QA-1	Availability	The system continues to execute a valid production plan when machines or the Scheduler fail, recovering within an acceptable time bound.	UC2
QA-2	Modifiability	The production plan and machine configurations can be adjusted (e.g. due to storage-level changes) without code changes and without interrupting ongoing production.	UC1, UC2
QA-3	Scalability	The system can integrate additional machines at runtime and redistribute workload so that performance and responsiveness remain within specified limits.	UC2

TABLE V: Key architectural constraints

ID	Constraint
C-1	**Centralised scheduling responsibility:** the Scheduler is the main coordination point for computing and distributing production plans. Mitigation: High availability is ensured through the shadow Scheduler, there is no bottleneck since it only reacts to events, and the system continues to operate during computations.
C-2	**Fixed heartbeat and threshold settings:** failure detection and storage-limit enforcement depend on configured intervals and thresholds that are not yet managed dynamically. Mitigation: In practice these values should be dynamic to reflect an industry 4.0 system, and the architecture already includes an Update Manager to adjust them during production even though this has not yet been implemented in UPPAAL or in the experiment.

VI. Design and Analysis Modeling

This section presents the high-level architecture of the pasta production control system and explains how the main components collaborate to realize the intended behavior. The focus is on the overall structure of the system rather than low-level implementation details.

A. System Architecture

The architecture is organized into multiple layers to ensure modularity and clear separation of concerns.

Figure 1 provides an overview of the architecture. The system is organised around a small set of core services that coordinate production based on events:

- **Management Layer**: Comprises core components such as the *Scheduler*, *Shadow Scheduler*, and *Database Subsystem*. The Scheduler acts as the central orchestrator, responsible for generating and updating the production

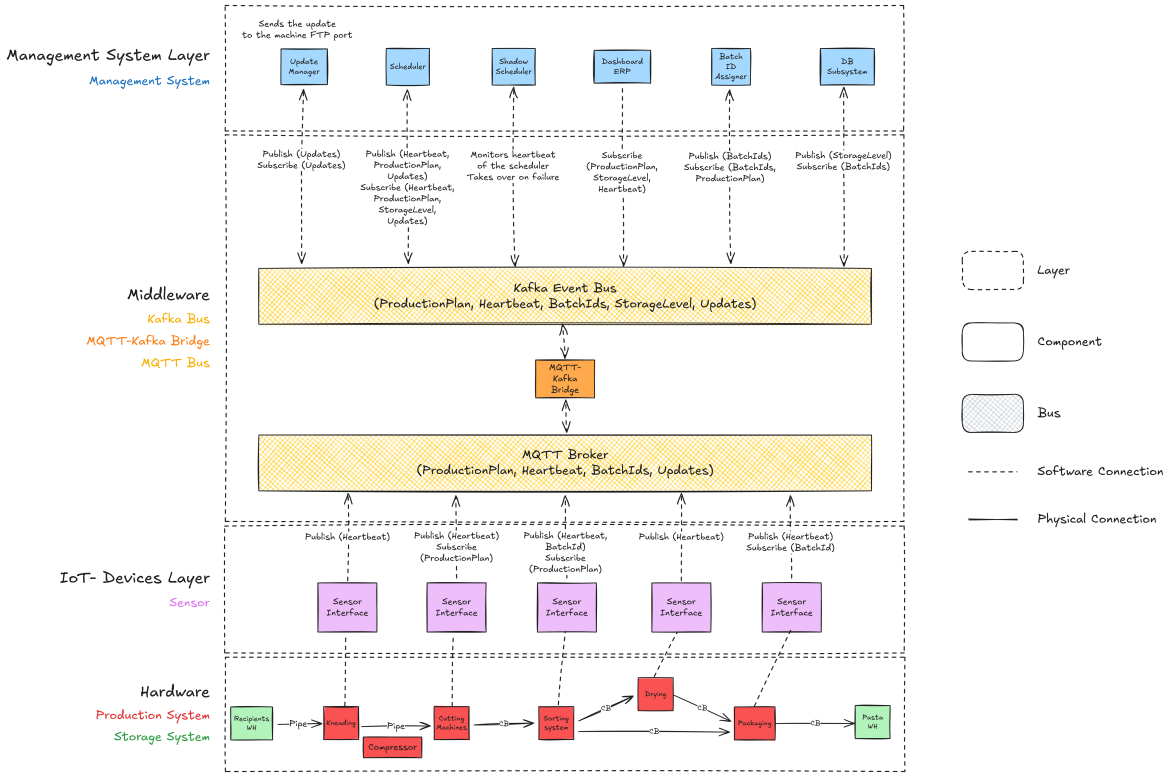

Fig. 1: High-level architecture of the pasta production control system.

plan, monitoring machine health through heartbeat signals, and coordinating production activities. The Shadow Scheduler ensures system availability by taking over seamlessly in case of Scheduler failure.

- **Middleware Layer**: Provides event-driven communication through *Kafka*, which serves as the backbone for asynchronous message exchange between services. This design supports decoupling and enhances modifiability and scalability. Additionally, *MQTT* is used for lightweight communication with IoT devices at the edge.
- **Hardware Layer**: Includes cutting and sorting machines that execute production commands. These machines periodically send heartbeat signals to indicate operational status. If a machine fails (e.g., heartbeat is missing), the Scheduler detects the fault and updates the production plan accordingly.
- **IoT Devices Layer**: Consists of sensors and actuators that provide real-time data about machine states and storage levels.

The *Database Subsystem* publishes storage-level alerts when thresholds are crossed (e.g., below 20% or above 80%). Upon receiving such alerts, the Scheduler adjusts the production plan dynamically to prevent overstocking or shortages. This mechanism illustrates the system's ability to adapt without interrupting ongoing operations and integrate new machines at runtime.

Communication between components follows an **event-driven architecture**, which avoids tight coupling and supports asynchronous interactions. This design choice improves resilience by preventing cascading failures and enables the system to remain responsive under varying load conditions.

B. Trade-Offs

The architectural design entails several inherent trade-offs arising from the need to balance the targeted quality attributes. Centralising coordination within the Scheduler strengthens consistency and enables formal verification; however, it introduces a potential performance bottleneck that must be compensated through redundancy mechanisms. The adoption of an event-driven architecture improves modifiability, scalability, and fault isolation, yet increases operational complexity and complicates system monitoring due to asynchronous message flows. Similarly, employing hot-spare redundancy via a Shadow Scheduler and Redis enhances availability while incurring additional resource consumption and coordination overhead. The use of formal verification provides rigorous early assurance of critical behaviours, although this benefit is achieved by abstracting system details and relying on statistical queries. Collectively, these trade-offs produce an architecture that favours reliability and adaptability while accepting reasonable increases in system complexity.

VII. Formal Verification and Validation

Formal verification and validation activities provide a crucial layer of software assurance, ensuring that the system is engineered correctly to meet its specifications and intended

purpose. Our overall approach to architectural evaluation and formal analysis follows the frameworks described by Cervantes and Kazman [1] and by Bass, Clements, and Kazman [2].

We employed UPPAAL, a model checking tool, to formally test our system against critical non-functional requirements, including modifiability and availability Formal analysis of the architecture enables the early prediction of system qualities and is recognized as a valid research method in software engineering. Both modifiability and availability are core quality attributes that the system architecture enables.

The architectural components implemented for formal analysis included the cutting machine, sorting subsystems, scheduler, shadow scheduler, and database. These components were necessary to observe and verify the complete system flow, as structures comprising software elements and their relations are essential for rigorous reasoning about the system.

A. Why Formal Methods?

Formal analysis was an essential verification step to ensure the architecture was fundamentally sound and to enable early prediction of system qualities. This allowed us to rigorously confirm the system design and identify necessary refinements before moving on to the complex and resource-intensive empirical experiment. Our approach to architectural evaluation and formal analysis is grounded in the methods described in [1], [2], [4].

Due to UPPAAL limitations in handling the verification of highly complex systems, we employed statistical queries rather than symbolic ones in order to avoid the state space explosion.

Model checking symbolically explores possible execution paths, providing objective and quantifiable verification across a wide range of system states. This significantly increases confidence that the design satisfies critical requirements under diverse scenarios. Due to UPPAAL's limitations in handling very large state spaces, we employed statistical queries rather than purely symbolic ones to mitigate state space explosion.

B. Verification of Storage Threshold Compliance (FR-2, UC-1)

A critical objective of our formal testing was to rigorously verify that the production control logic ensures that storage levels always remain within intended thresholds. This verification directly supports the system's primary quality attribute of Production Modifiability (R2), which requires adjusting output based on storage levels.

The key UPPAAL query used to confirm this property was:

$$\Pr[\leq 1000]([\,](\text{freshA} \geq 0 \wedge \text{freshA} \leq 100 \wedge \text{freshB} \geq 0 \wedge \text{freshB} \leq 100 \wedge \text{driedA} \geq 0 \wedge \text{driedA} \leq 100 \wedge \text{driedB} \geq 0 \wedge \text{driedB} \leq 100)) \quad (1)$$

This query verifies that storage levels for all product types (fresh A/B, dried A/B) are maintained between 0 and 100. The successful analysis confirmed that the control logic prevents storage from exceeding upper bounds or falling below lower thresholds across all possible execution paths. This guarantee would have been difficult and risky to validate through empirical testing alone, given the complex interplay of production rates, consumption patterns, and timing constraints.

C. Verification of Scheduler Redundancy (FR-4, UC-2)

To ensure system availability the property that the system remains ready to carry out its task when required we formally verified the redundancy mechanism for the scheduler component. Two crucial availability properties were verified:

1) Mutual Exclusion: Ensuring only one primary scheduler is operational at any given time:

$$\Pr[\leq 1000]([\,]\,!(\text{SchedulerState0.active} \wedge \text{SchedulerState1.active})) \quad (2)$$

This query verifies that only one scheduler is active at any time, preventing both from attempting to respond to the same problem.

2) Failover Time: Assessing the maximum duration the system remains without an active scheduler:

$$\Pr[\leq 1000]([\,]\,!(\text{SchedulerState0.active} \vee \text{SchedulerState1.active}) \implies (\text{SchedulerState0.c} \leq 11 \wedge \text{SchedulerState1.c} \leq 11)) \quad (3)$$

This query checked that if both schedulers are down, the system would recover within at most 11 time units. While the analysis confirmed the failover mechanism works, the successful verification also revealed an unacceptable design constraint: 11 time units is too high for a real-world scenario.

D. Verification of Component Decoupling (FR-2, FR-3)

The formal analysis also examined the system's modifiability the cost and risk of making anticipated changes. We verified that system behavior maintains low coupling by ensuring operational changes affect only the necessary components. Two queries confirmed the correct and timely interactions between the scheduler and the cutting machine:

1) Scheduler Latency: Testing the time required for the scheduler to recognize and integrate a newly connected machine:

$$\Pr[\leq 1000]([\,](\text{CuttingMachine0.waiting} \implies \text{CuttingMachine0.c} < 5)) \quad (4)$$

This checks that if a machine is in a waiting state (having sent a heartbeat without a blade type), it waits less than 5 seconds before being included in the production plan, thereby testing the scheduler latency.

2) Correct Message Sequencing: Ensuring the Scheduler sends the appropriate command based on the machine state:

$$\Pr[\leq 1000]([\,]((\text{CuttingMachine0.workingA} \vee \text{CuttingMachine0.workingB}) \implies (\text{SchedulerMachine0.assignBlade} == \text{false}))) \quad (5)$$

This verifies that the scheduler does not send an `AssignBlade` message while the machine is already working. In that state, the correct message would be `SwapBlade`.

This successful verification confirms that the Scheduler orchestrates components in the correct sequence without creating hidden dependencies.

Note: We also issued the auxiliary query `Pr[`$\leq$ 1000`]([](CuttingMachine0.failed` $\implies$ `CuttingMachine0.c` $\leq$ 30`))`. *Its primary purpose was to validate the UPPAAL model configuration and ensure that a failed machine does not remain indefinitely in the failed state. The outcome of this query did not directly affect the main architectural design conclusions.*

E. Impact on Design Decisions

The formal analysis directly influenced a crucial design refinement concerning system resilience. During formal modeling, we discovered that the heartbeat mechanism for the scheduler lacked sufficient responsiveness. This presented a critical architectural problem: if the scheduler fails and the shadow scheduler response is delayed (as indicated by the 11 time unit failover constraint), heartbeat signals could be lost, causing cascading system failure, and this high delay was revealed to be an unacceptable constraint by the analysis. Heartbeats are a fault detection tactic used to monitor process health, and their failure would compromise the system's ability to recover, presenting a high-risk scenario. Fault detection is a fundamental category of availability tactics.

To address this vulnerability and create a more resilient system, we implemented Redis as the coordination layer for ephemeral state management, leader election, and heartbeat storage. Because the 11 time unit failover measured during formal verification was deemed too slow, Redis provides fast, in-memory access that ensures the shadow scheduler can detect failures and assume control within milliseconds from the expiration of the Redis key. This implementation aligns with active redundancy patterns (hot spare), where the backup component maintains a synchronized state, achieving the fastest possible recovery time and reducing the effective failover time down to 5 seconds. This design decision, driven directly by the insights from formal verification, transformed the system into a resilient architecture capable of transparent failover, reinforcing how architectural evaluation enables the early prediction of system qualities and ensures design decisions are appropriate to address the requirements.

VIII. Evaluation

This section describes the empirical evaluation of the proposed system design. Our overall experimental approach follows the guidelines of Wohlin et al. [4] and is organised into the phases of Definition, Planning, Operation, and Analysis.

A. Experiment design

The planning phase of the experiment determines how the tests (trials) are organized to maximize efficacy and align with the intended statistical analysis.

1) Goal Definition: The goal of the evaluation, defined using an approach similar to the Goal/Question/Metric paradigm, is to test the performance of the reaction to events and assess system availability when subjecting key components to controlled failures.

- **System Drivers:** This study primarily evaluates the critical non-functional requirements, specifically Production Modifiability (FR-2), which relies on timely responses to storage levels, and System Availability (NFR-10), achieved through redundancy and timely fault detection (FR-3, FR-4,UC-2). Architectural evaluation is necessary to ensure that design decisions are appropriate to address critical requirements.
- **Availability Testing:** Availability is tested through controlled fault injection specifically, by randomly killing machines or the scheduler to observe the subsequent recovery and failover mechanisms. A failure in a system occurs when it no longer delivers a service consistent with its specification, and this evaluation focuses on ensuring fault mitigation and recovery.

2) Technologies and System Summary: The evaluation environment is based on the implemented Pasta Production system architecture. Key technologies involved in the execution environment include: Python (for the Simulator and Adapter), Java (for the core logic components: Scheduler and Shadow Scheduler), and Redis for fast coordination and leader election. The architecture relies on MQTT for lightweight Edge Communication and leverages a robust Kafka ecosystem for reliable communication between the Java components. This infrastructure, especially the use of Redis, was driven by the formal verification process to ensure highly responsive heartbeat detection and immediate scheduler failover.

3) Treatments (Independent Variables): The experiment employs a multivariate design, manipulating several factors (independent variables) to observe their effect on performance and availability. In experimental design, independent variables are deliberately manipulated and controlled, and the term factor refers to those variables being studied by changing their values. A treatment represents one particular value of a factor [4] (Chapter 4,Chapter 6).

The manipulated factors (treatments) are:

1) **Number of available machines (M):** Tested at two levels, $M \in \{3, 5\}$. This allows investigation of System Scalability (NFR-9) under load, determining whether the system maintains performance and availability as resources increase [2] (Chapter 9).
2) **Number of killable components (K):** Tested with one killable scheduler and a variable number of killable cutting machines, where $K \in \{1, 2, \ldots, M-1\}$. This treatment simulates increasing failure load or fault complexity, testing the system's resilience and fault tolerance mechanisms, such as redundant spares and error handling, which are foundational concerns of availability tactics [2] (Chapter 4).

A factorial design is appropriate to capture main effects and

potential interactions across these multiple factors, as increasing the number of factors beyond one necessitates analyzing both the effects of individual factors and the combined effect (interaction) between them.

B. Measurements

The system's response to the imposed stimuli (events and faults) is quantified primarily through latency measures, which fall under the category of performance response measures. The goal is to measure the resources expended to make a change or recover from a fault, typically expressed as elapsed time.

We rely on measurements to accurately capture the elapsed time between critical events and the system's corresponding reaction. Measurements are organized around three key events that trigger necessary architectural responses, reflecting requirements (FR-13, FR-14).

The experiment focuses on three primary event types, each associated with a specific system reaction and corresponding latency measure:

Storage Alert: When the system triggers a storage alert via a database notification as defined in (FR-14), the measurement captures the time until the Production Scheduler generates and publishes a new production plan. This duration represents the *Storage Response Latency*, tied to Production Modifiability (FR-2) and timely storage-limit notifications.

New Machine Connection: When a new machine is detected through its initial heartbeat (NFR-5), we measure the time required for the Scheduler to incorporate it into the Production Plan (NFR-5). This *Integration Latency* evaluates the system's scalability and responsiveness to resource expansion (R5).

Machine Failure: In cases where a heartbeat is missing (FR-3) or a component is deliberately killed, the system must adjust the Production Plan. The measurement captures the time from failure detection until the updated plan is executed by the active Scheduler. This *Failure Recovery Latency* supports evaluating timely scheduler updates (FR-4) and overall system availability (NFR-10).

These latency metrics are measured on a ratio scale, making them suitable for statistically analyses.

C. Pilot test

A pilot test was conducted to validate the instruments and refine the experimental procedures before proceeding to the full execution.

1) Findings and Mitigation: The initial pilot test confirmed that the timestamp data for events and reactions was correctly collected and stored. However, a significant issue was discovered concerning the analysis algorithm. Specifically, if an event occurred without an expected reaction within the system (e.g., a cutting machine failed, but no operational, capable machine was available for a blade swap), the algorithm recorded very high, random latencies.

This problem posed a threat to Conclusion Validity (the ability to draw correct conclusions about the relationship between treatment and outcome) and highlighted an issue with data validation. The recording of random, high latencies distorted the true performance metrics and failed to distinguish between genuine system slowness and expected lack of capacity or inability to execute a specific remediation tactic (like Substitution).

To address this, the analysis approach has been refined to correctly handle scenarios where no immediate countermeasure exists. This means implementing checks for:

1) **System State Integrity:** Recognising when the system logically determines that a reaction is impossible due to current resource constraints, rather than a failure of the scheduler mechanism itself.
2) **Data Reduction:** Applying methods to identify and properly handle these extreme, non-representative latency values during the data validation phase, possibly through outlier analysis or by excluding such trials if they fall outside the defined scope of a verifiable system reaction.

D. Analysis and Interpretation

The empirical evaluation followed the established phases of experimentation: Definition, Planning, Operation, and Analysis and Interpretation. The collected experimental data undergoes quantitative interpretation structured to draw valid conclusions. This analysis was conducted in primary steps that first involved characterizing the data using descriptive statistics. Descriptive statistics aim to give an informal understanding of the data's distribution through measures of central tendency, dispersion, and graphical visualization of the data (e.g., event counts) [4]. Following this characterization, the data underwent systematic analysis, typically through hypothesis testing, to rigorously evaluate the derived latency values against the stated requirements. These analyses confirmed the system's responsiveness and demonstrated that the architectural design successfully addresses critical requirements, ensuring fulfillment of key quality attributes such as System Availability (NFR-10), Production Modifiability (FR-2), and System Scalability (NFR-9).

Each scenario was executed over 200 minutes (12000 seconds). All system messages were captured in Kafka during this period, and analysis focused on the three event types explained previously

1) Event Frequency, Correlation, and the Impact of Chaos: The initial descriptive statistics were displayed via boxplots (Figure 2) and histogram (Figure 3) to visualize the central tendency and dispersion of the collected data. The experiment, which simulates resource fluctuation and failure, necessarily involved high levels of unpredictable input, including the use of a chaos panda (inspired by the Chaos Monkey from Neflix) that randomly kills machines and schedulers. This inherently chaotic environment means there is not a perfect correlation between the number of killable machines (the independent variable) and the resulting event frequency.

Despite this experimental noise, attributed partly to the simulation of purchases and the nature of the chaotic environment, clear trends emerged:

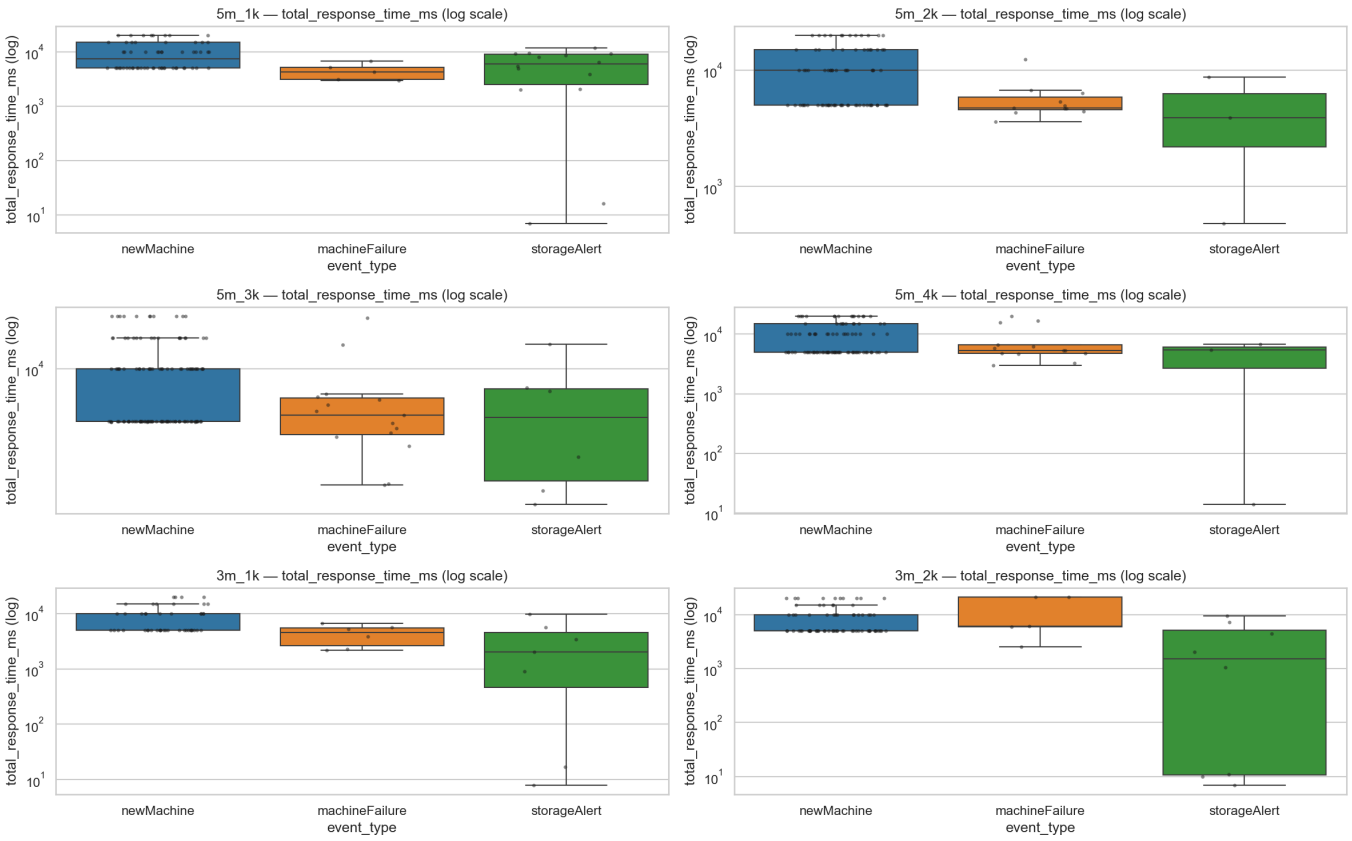

Fig. 2: Experimental data viewed in boxplots with total response time in milliseconds

- **Storage Alert Decrease:** The number of `storageAlert` events decreases as the number of killable machines increases. This counter-intuitive trend is explained by the scheduler sometimes lacking sufficient available resources to execute a corrective `swapBlade` command when multiple machines are down.
- **Machine Failure Increase:** Predictably, the number of `machineFailure` events increases as the number of killable machines increase, directly reflecting the applied fault load imposed during the evaluation.

A notable finding related to failure load involved a saturation phenomenon, can be seen in Figure 2. When 4 out of 5 machines were killed, the number of observed `machineFailure` events was not higher than the 3-kill scenario, indicating a point where the scheduler could no longer react meaningfully to the destruction of the final machine. This situation demonstrates the operational limits of the fault recovery logic when resource availability is near zero.

Crucially, the experiment showed that the total number of `machineFailure` (55) and `storageAlert` (41) events is far lower than the number of `newMachine` events (465). This high ratio of successfully integrated components to failed components suggests that the scheduler efficiently handles problems while minimizing disruptive `swapBlade` commands that slow production. Can be viewed in table 1

2) Latency Analysis Against Architectural Drivers: The analysis of system latency, measured on a ratio scale, directly assesses the system's ability to satisfy its high-priority Quality Attribute Scenarios, providing quantifiable evidence of architectural efficacy.

TABLE VI: Latency measurements per event type

Event	Total	Sched. Lat. (s)	Mach. Lat. (s)	Total Lat. (s)
`storageAlert`	41	3.656	0.933	4.588
`machineFailure`	55	4.655	2.007	6.662
`newMachine`	465	5.627	3.783	9.409

a) Storage Response Latency (`storageAlert`): The `storageAlert` event directly assesses compliance with the performance constraint FR-14: when storage exceeds the limit, the database should inform the scheduler in ≤ 5 seconds. The average total latency recorded was 4.588 seconds. This result confirms that the system meets this strict, high-priority timing requirement and reacts swiftly to inventory feedback.

b) Failure Recovery Latency (`machineFailure`): The `machineFailure` event measures the time required for the system to detect and recover from a fault, supporting System Availability (NFR-10). The average scheduler latency was 4.655 seconds, that means that on average a machine failure is detected before the end of the next heartbeat cycle.

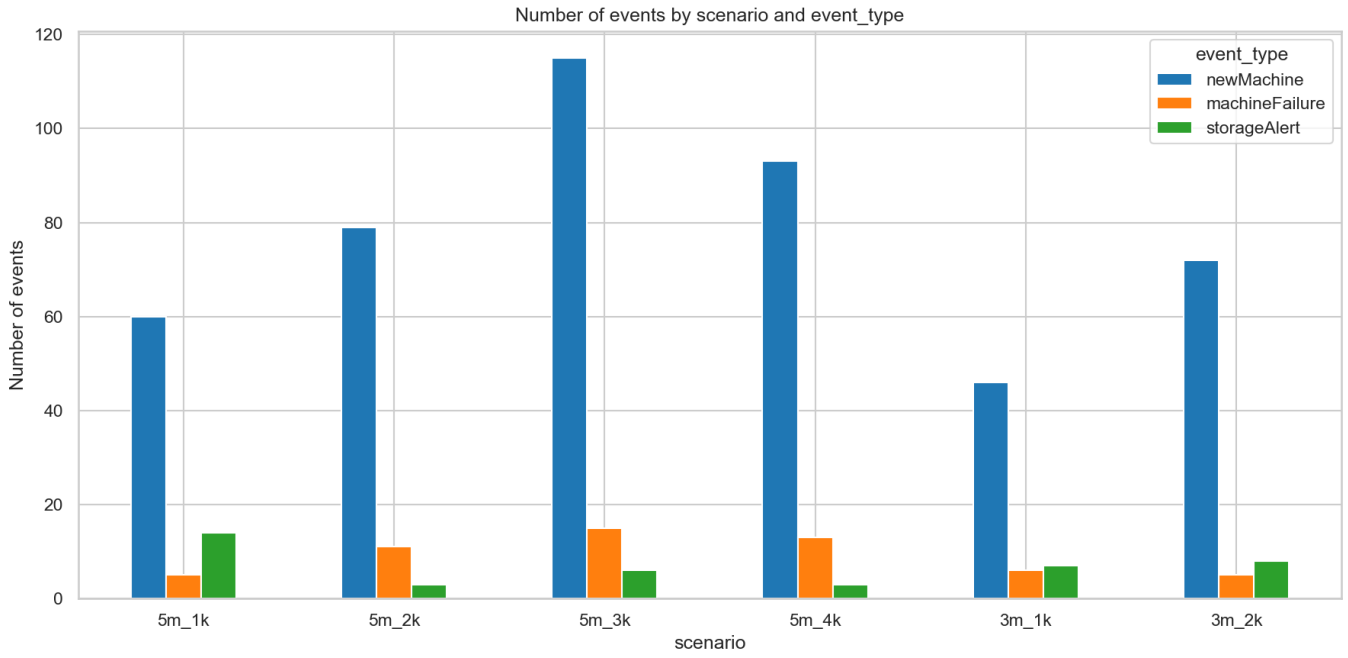

Fig. 3: Experimental data viewed in histogram counted by event type

The measured value demonstrates the effectiveness of the implemented availability tactics.

c) Integration Latency (`newMachine`): The `newMachine` event measures the system's ability to assimilate new resources, addressing System Scalability (NFR-9). This was the most frequent event type (465 occurrences). The average total latency was 9.409 seconds, which is acceptable because resource integration is less time-critical than fault recovery. This value aligns closely with FR-13, which requires machines to react to production plan changes within ≤ 10 seconds.

3) Interpretation of Design Decisions: Across all event types, the average scheduler latency constitutes most of the total latency (from 5.627 seconds for `newMachine` down to 3.656 seconds for `storageAlert`). This indicates that the majority of the reaction time is consumed by the scheduler's internal processing (e.g., generating production plans, coordinating Java components).

Since scheduler computation dominates latency, future performance optimization should primarily target reducing computational overhead within the scheduler.

Overall, the quantitative results provide controlled, empirical evidence that the chosen architectural tactics including Redis for low-latency coordination, Kafka for reliable message delivery, and structured scheduler logic successfully enable the system to meet its most critical quality attribute requirements for speed and resilience.

IX. Conclusion

This paper presented the design, formal verification, and empirical evaluation of a fault-tolerant, event-driven architecture for an automated pasta production system. The architecture was systematically derived using Attribute-Driven Design, with a focus on the quality attributes of System Availability (QA-1), Production Modifiability (QA-2), and System Scalability (QA-3).

Answer to RQ1. RQ1 asked how architectural tactics and design decisions can be applied to achieve the required levels of modifiability, availability, and scalability in the pasta production system. Our architecture operationalises availability through active redundancy with a shadow scheduler, heartbeat-based fault detection, and Redis-based coordination for leader election and failover. Modifiability is supported by event-based decoupling using Kafka and MQTT, configuration-driven production plans, and a clear separation between core services, adapters, and hardware. Scalability is achieved by relying on a publish–subscribe communication style, stateless integration of new machines via heartbeats, and central scheduling logic that redistributes workload at runtime. The refinement from a heartbeat-based failover mechanism to a Redis-backed hot-spare design illustrates how concrete architectural tactics can be selected, implemented, and iteratively improved to realise these quality attributes.

Answer to RQ2. RQ2 asked to what extent formal verification and empirical experiments provide consistent, quantitative evidence that the implemented architecture satisfies its quality attribute scenarios. Formal verification with UPPAAL provided early and rigorous feedback on the architectural design. The analysis confirmed key properties, including storage threshold compliance, safe scheduler redundancy (mutual exclusion and bounded failover), and correct decoupling between scheduler and machines. At the same time, the model checking results

exposed an unacceptable failover time bound, which directly motivated the introduction of Redis to reduce recovery latency and strengthen the resilience of the scheduler subsystem.

The subsequent empirical evaluation, executed under a controlled yet highly chaotic environment with randomized failures, demonstrated that the implemented system satisfies its main quality attribute scenarios in practice. Storage alerts were handled within the required 5 seconds, machine failures were detected and mitigated within one heartbeat cycle, and new machines were integrated into the production plan in under 10 seconds on average. The event distributions and latency measurements confirmed that the architecture remains responsive under varying failure loads, and that the system continues to operate effectively even when resources are heavily constrained. Together, these results show that the formal and empirical evidence are consistent and complementary: the UPPAAL model anticipates critical behaviours and constraints, and the experiment confirms that the implemented system delivers them under realistic operating conditions.

Overall, the work provides an architected, implemented, and quantitatively evaluated event-driven control system for pasta production, and demonstrates how combining Attribute-Driven Design, formal verification, and experimentation can guide architectural refinement. The results indicate that the chosen tactics and technologies successfully realise the intended availability and scalability requirements, and offer supportive but not yet conclusive evidence for modifiability, which is further discussed in Section IX-A.

A. Discussion

We could not directly test modifiability in UPPAAL, since the verifier focuses on safety and liveness properties rather than explicit change-related metrics. Instead, we complemented the formal analysis with long-running simulation to indirectly assess modifiability-related behaviour. In particular, we observed that no deadlocks occurred even under repeated machine failures and that storage thresholds remained within the intended bounds over extended executions. These results, supported by queries (1), (4), and the auxiliary query noted in Section VII, increase confidence that the architecture can accommodate anticipated changes and faults without violating core constraints.

B. Future work

In future work, we plan to improve maintainability and to incorporate the *Update Manager* into both the UPPAAL model and the experiment.

First, we intend to refactor the implementation of the experiment more systematically, so that specific pieces of code are easier to locate and components are simpler to update. We have already adopted a clearer separation of concerns, which should make further restructuring more straightforward.

Second, extending the UPPAAL model with update-related behaviour (for example, dynamic reconfiguration events triggered by the *Update Manager*) would allow us to reason formally about configuration changes and their interaction with availability and performance properties.

Finally, we aim to introduce a dedicated *Update Manager* component responsible for runtime configuration tasks such as managing storage thresholds, heartbeat intervals, and machine-specific parameters. This component would expose a controlled interface to the operations team and propagate configuration changes to the Scheduler, the Database subsystem, and other services through the existing event-driven infrastructure. A concrete modifiability scenario is the ability to adjust storage thresholds (for example, from 80/20 to 90/10) at runtime under load, without downtime and without violating the storage invariants verified in Section VII. Future experiments could then measure the latency from a configuration request to consistent system behaviour, as well as the effort and impact of such changes on the code base.

REFERENCES

[1] H. Cervantes and R. Kazman, *Designing Software Architectures: A Practical Approach*, 2nd ed. Pearson, 202x.

[2] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 4th ed. Pearson Education, Inc./Addison-Wesley, 2022.

[3] A. Chavan, "Exploring event-driven architecture in microservices: patterns, pitfalls and best practices," *International Journal of Science and Research Archive*, vol. 4, no. 1, pp. 229–249, 2021.

[4] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer, 2000.

APPENDIX A

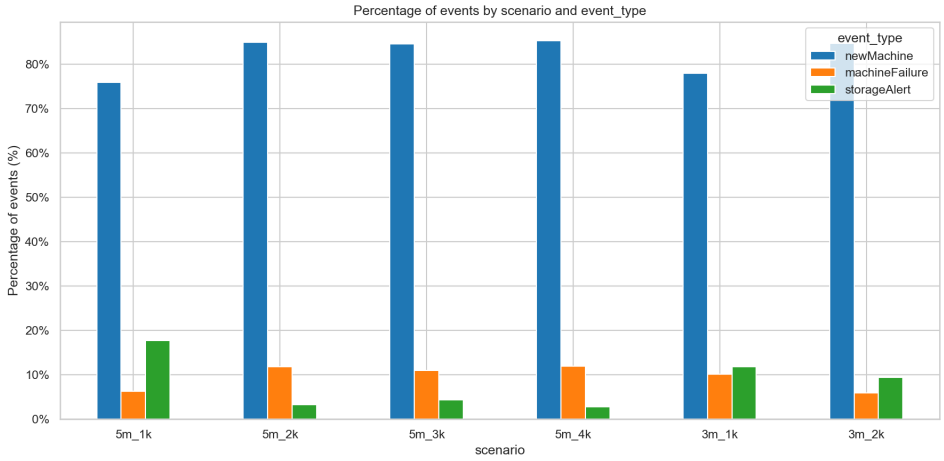

Fig. 4: Experimental data viewed in histogram counted by event type in percentage

TABLE VII: Functional Requirements

ID	Name	Description	Priority
FR-1	Fresh Pasta A Production	The system shall produce fresh pasta of type A	High
FR-2	Dried Pasta A Production	The system shall produce dried pasta of type A	High
FR-3	Fresh Pasta B Production	The system shall produce fresh pasta of type B	High
FR-4	Dried Pasta B Production	The system shall produce dried pasta of type B	High
FR-6	Machine Heartbeats	Each machine shall communicate if it's working correctly using heartbeats	High
FR-7	Scheduler Checks Heartbeats	The scheduler checks the heartbeats and checks if there are missing or new ones.	High
FR-11	Product Packaging Time	The system shall have the product packaged in less than 10 seconds	Medium
FR-12	Product Sorting	The sorting system shall configure whether the product must go to the drying machine or directly to packaging	High
FR-13	Production Plan Modification	When the production plan gets modified, the machine should react in $\leq$ 10 seconds	High
FR-14	Storage Limit Notification	When storage exceeds the limit for product A or B, the DB should inform the scheduler in $\leq$ 5 seconds	High
FR-15	Storage Threshold Cooldown	After switching blade to a cutting machine, it must ignore repeated threshold triggers for 5 seconds	High

TABLE VIII: Non-Functional Requirements

ID	Name	Description	Priority
NFR-5	Production Modifiability	The system shall adjust production based on the current system state	High
NFR-8	Software Deployability	The system shall allow software updates with minimal downtime	High
NFR-9	System Scalability	The system shall support adding new machines without redesign.	High
NFR-10	System Availability	The system must operate continuously 24/7 with an uptime of at least 99.9%	High
NFR-16	System Maintainability	The system shall support modular replacement of devices without downtime.	High
NFR-17	Security	All external accesses must be secured	Medium
NFR-18	Audit Trail	All operator actions in the system shall be logged for auditing.	High
NFR-19	Message Reliability	All messages between devices and the bus must be delivered at least once.	Medium

TABLE IX: Contribution Table

	Aljaz	Anders	Dimitrios	Leonardo
Use Cases & Requirements	✓	✓	✓	✓
Architecture	✓	✓	✓	✓
EAST-ADL	Features Deinition, Feature Diagram	Behaviour Diagram	Feature Definition, Feature Diagram	Behaviour Diagram
Pitch		Main Presenter		Second Presenter
UPPAAL		Helper		Main Developer
Experiment	Dashboard	Experiment Evaluation	DB	Scheduler, CuttingMachines, SortingSubsystem & Experiment Runner
Second Pitch	✓	✓	✓	✓
Report	✓	✓✓	✓	✓