*Article*

# Action-Oriented Programming and Automatic Agent Generation for Adaptive Data Collection in Decentralized Data Ecosystems

**Mustafa Tayyip Bayram [1,2,*], Houssam Razouk [2,3] and Kyandoghere Kyamakya [1]**

[1] Institut für Intelligente Systemtechnologien, Alpen-Adria-Universität Klagenfurt, 9020 Klagenfurt am Wörthersee, Austria; kyandoghere.kyamakya@aau.at
[2] Infineon Technologies Austria AG, 9500 Villach, Austria; houssam.razouk@infineon.com
[3] Institute of Machine Learning and Neural Computation, Graz University of Technology, 8010 Graz, Austria
* Correspondence: mustafatayyip.bayram@infineon.com; Tel.: +43-676-962-6508

**Abstract**

The semiconductor manufacturing industry depends on effective data collection and analysis for critical processes such as Root Cause Analysis (RCA) and Risk Assessment (RA). Both processes involve software-driven data collection and subsequent analysis by domain experts to support informed decision-making. However, the increasing complexity, volume, and decentralized nature of manufacturing data pose significant challenges for effective data collection. Data is distributed across multiple systems with varying formats and ownership, making conventional programming paradigms and manual data collection scripts inadequate for handling this decentralized data landscape. To address these challenges, this study proposes integrating Action-Oriented Programming (AcOP) with Automatic Agent Generation (AAG) as a novel solution. AcOP emphasizes actions as fundamental execution units, separating system behavior and data. Complementing this, AAG uses large language models (LLMs) to autonomously generate intelligent agents, which manage these actions and perform preliminary data analysis with domain-specific knowledge. Our experimental setup compares three microservice applications supporting RCA and RA: Object-Oriented Programming (OOP), AcOP, and AcOP integrated with AAG. Evaluation results indicate that AcOP improves modularity, adaptability, and error handling in decentralized systems. Integrating AAG enhances automation, provides a flexible, low-maintenance solution for data collection and analysis pipelines, and promotes autonomous microservice architectures in data-intensive environments.

**Keywords:** action-oriented programming; automatic agent generation; generative ai; agentic systems; distributed micro-services; artificial intelligence; decentralized data; decision-making; data collection frameworks; root-cause analysis; risk assessment

Academic Editors: Xiaojun Jin, Jian Zhang, Dong-Qing Yuan and José Alfredo Hernández Pérez

Received: 5 March 2026
Revised: 9 May 2026
Accepted: 18 May 2026
Published: 21 May 2026

**Copyright:** © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

## 1. Introduction

The semiconductor manufacturing industry is a dynamic and intricate field that relies heavily on data and domain knowledge to oversee and optimize manufacturing processes [1]. Even minor inefficiencies can result in significant financial losses and reduced product quality [2]. Thus, semiconductor manufacturing companies continuously improve their operational effectiveness and quality by systematically managing key processes such as Root Cause Analysis (RCA) and Risk Assessment (RA). RCA identifies root causes of manufacturing issues, whereas RA evaluates potential risks and ensures mitigation strategies [3]. Both processes depend critically on timely, accurate, and comprehensive

 https://doi.org/10.3390/pr14101669

data collected [4] from various sources throughout manufacturing environments to make informed decisions. Data collection is supported by software applications that facilitate the extraction, aggregation and processing of the underlying data.

However, the data landscape encompasses diverse data sources, structures, and formats [1]. Data is distributed across multiple systems and units such as sensors, quality assurance (QA) systems, logistic tracking. Each makes independent decisions based on resource availability and objectives [5–7]. This decentralized structure, combined with increasing data volumes and complexity, poses significant challenges to software-driven data collection and analysis. Manual data collection scripts and software development practices are time-consuming, error-prone, and poorly suited for handling decentralized data systems. This limits their effectiveness in supporting RCA and RA efforts [8].

With the growing importance of managing decentralized data systems, this study explores the integration of two innovative frameworks: Action-Oriented Programming (AcOP) and Automatic Agent Generation (AAG). AcOP centers around actions as fundamental units to manage decentralized environments efficiently [9,10]. Complementing AcOP, AAG leverages large language models (LLMs) to autonomously generate intelligent agents to coordinate specialized tasks within these environments and perform preliminary data analysis [11]. The integration of these approaches provides a more efficient, flexible, adaptable and automated method for handling decentralized data collection challenges. It is particularly suitable for semiconductor manufacturing and other data-intensive domains.

The primary objective of this research is to enhance data collection software solutions tailored for decentralized environments. Specifically, this research investigates how integrating AcOP and AAG can improve software application design, modularity, scalability, and automation to effectively support RCA and RA processes. To achieve this, the following research questions are addressed:

1. What challenges does the semiconductor industry face in data collection for RCA and RA?
2. How does the application of AcOP compare to conventional data collection software development techniques currently used in the semiconductor industry?
3. What potential advantages could the AAG offer to improve data collection and analysis phases in RCA and RA?
4. How could the combination of AcOP and AAG, along with a microservice architecture, potentially benefit the semiconductor industry for data collection?
5. What broader impacts might AcOP and AAG have on workforce skills, organizational structure, and technology strategies?

To address these questions, a microservice application prototype was developed to serve as a software-driven data collection tool for RCA and RA. Three distinct implementation approaches were evaluated: conventional Object-Oriented Programming (OOP), AcOP, and AcOP integrated with AAG. Each implementation was assessed under identical simulated workloads, with a focus on key metrics including modularity, scalability, and error-fault management and implementation complexity.

This research contributes both theoretically and practically to software development in decentralized environments. It extends the theoretical foundations of AcOP, originally introduced by Kurki-Suonio and Kankaanpää [12], by applying it to real-world data collection challenges beyond cloud computing [13]. Additionally, this research explores AAG, a relatively new approach that uses AI agents to support coordination in distributed systems [11]. On the practical side, the study demonstrates how agentic systems with a suitable programming paradigm can improve development practices and maintainability in software systems, addressing current limitations in RCA and RA workflows. The

https://doi.org/10.3390/pr14101669

proposed methods are applicable not only in semiconductor manufacturing but also in other domains such as IoT and smart manufacturing, offering scalable and adaptable solutions for decentralized data handling.

The remainder of this paper is structured as follows: Section 2 reviews relevant literature for RCA and RA. Sections 3 and 4 review relevant literature and theoretical background for the proposed frameworks. Section 5 outlines the experimental methodology. Section 6 presents and analyzes our results, while Section 7 discusses their implications and suggests directions for future research. Finally, Section 8 summarizes the key findings and contributions.

## 2. Root Cause Analysis and Risk Assessment

RCA and RA are two essential frameworks widely adopted in high-reliability domains such as semiconductor manufacturing [4].

RCA and RA methodologies, while distinct in their objectives, are complementary in practice. Together, they provide a structured foundation for process optimization, quality improvement, and strategic decision-making [3] (see Figure 1). In semiconductor manufacturing, defective components not only compromise product integrity but also pose risks to entire systems, leading to significant financial losses and reputation damage [14]. RCA and RA are thus fundamental in addressing these challenges through proactive measures and continuous improvement initiatives, which enhance process capabilities, product quality, and yield [15].

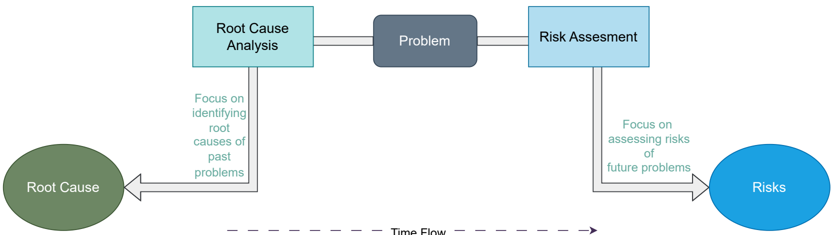


**Figure 1.** Relationship among a problem, RCA, and RA. RCA focuses on identifying the underlying causes of existing problems, whereas RA places more emphasis on preventing and mitigating potential risks in the future.

### 2.1. Root Cause Analysis (RCA)

The essence of RCA lies in identifying the root cause of a problem and eliminating it to prevent recurrence [16]. RCA differentiates between symptoms (observable effects), first-level causes (immediate causes), and higher-level causes (underlying contributors). Effective RCA involves a structured approach comprising several steps: problem identification, cause brainstorming, data collection, data analysis, root cause identification, root cause elimination, and solution implementation [16]. In semiconductor manufacturing, RCA often utilizes structured methodologies such as the Eight Disciplines (8D), which systematically identify root causes and facilitate corrective and preventive actions [17].

### 2.2. Risk Assessment (RA)

RA systematically identifies and evaluates potential risks that could impact organizational objectives [18]. This involves multiple stages: risk identification, risk analysis, risk evaluation, risk treatment, risk monitoring, and reporting [19]. Performance metrics such as risk exposure, mitigation effectiveness, frequency of occurrence, mitigation timeframe, cost, and impact on business objectives are employed to measure the effectiveness of RA processes [18,19].

### 2.3. Challenges in Decentralized Data Collection for RCA and RA

The semiconductor industry's data ecosystem is highly decentralized and fragmented, making data collection challenging. Issues arise from data volume, velocity, and variety across multiple systems such as sensors, QA processes, and logistic systems [1]. Therefore, engineers are compelled to collect data from various places and integrate it promptly, efficiently, and accurately [20]. Conventional data collection approaches, such as manual scripts and conventional software methods, struggle to manage these complexities efficiently, leading to increased human errors and delayed decision-making [8] in RCA and RA. Inefficiencies in the data collection process significantly increase human resource costs as analysts spend more time searching for root causes or potential risks. At the same time, computational resources have become less costly over time. This trend, combined with the increased availability of data, has prompted researchers and practitioners to develop solutions that leverage computational power to enhance data collection and analysis, ultimately accelerating RCA and RA processes [21].

### 2.4. Conventional Data Collection Methods

Data collection refers to gathering data based on specific variables within a defined system and context, such as a product identifier, with the goal of answering targeted questions and enabling informed decision-making. A closely related concept, data extraction, refers to the technical process of retrieving specific data points from structured or unstructured sources, such as databases, spreadsheets, or reports. In short, data collection focuses on answering the question from the data; data extraction ensures that the data is accessible and ready [22,23].

Several techniques are commonly used for data extraction in semiconductor manufacturing [24], including database querying, web scraping and file parsing. Database queries allow for retrieving structured data, such as specific records or combined datasets. However, querying across multiple decentralized systems, often managed by separate teams or organizations, introduces challenges like accessibility, concurrency, and delays in integrating the data [25]. File parsing is another commonly used technique, where data is exported into spreadsheets for visualization or preliminary analysis. While convenient for small-scale tasks, this method is prone to errors, synchronization issues, and inefficiencies when dealing with large datasets [26,27]. Web scraping, which extracts data from web pages by analyzing their HTML structure, is also used but often faces issues such as unpredictable webpage layouts, security restrictions, and the need for extensive post-processing [28]. More structured methods, such as Application Programming Interfaces (APIs), allow for efficient data retrieval from distributed systems using standardized formats like JSON or XML. APIs are particularly beneficial for integrating data from diverse sources but require careful implementation to ensure compatibility and performance [29].

Despite the variety of techniques, a significant portion of collected data remains archived rather than actively utilized due to difficulties in extraction, integration, and interpretation. The decentralized nature of data across multiple systems and locations complicates the process, making it challenging to extract and convert this data into actionable information [24]. As a result, data collection is time-consuming and labor-intensive, requiring significant manual effort to extract, clean, and integrate data. This delays the ability to identify root causes and risks. Furthermore, ensuring the availability and accessibility of data from various sources is critical yet difficult due to frequent interruptions, changes, or compatibility issues [1,30].

The current objective in semiconductor manufacturing is to transform raw data into insights that are easy to access, interpret, and use for monitoring and improving production performance. However, addressing this goal requires overcoming the limitations of existing

https://doi.org/10.3390/pr14101669

methods and adopting more advanced approaches [24]. Rather than focusing solely on improving individual techniques, it is crucial to address the broader challenges of how these methods are applied and managed. While some data extraction methods can function without programming, programming is often required to integrate, manipulate, and process the data effectively. Advancing the programming perspective is important for addressing the complexities of decentralized and distributed data landscapes. This includes adopting new paradigms that can optimize data collection processes, improve accuracy, and enhance scalability. Solutions tailored to the fragmented and distributed nature of semiconductor manufacturing data could significantly increase the efficiency of RCA and RA processes by reducing manual effort and automating repetitive, time-consuming tasks.

### 2.5. Conventional Programming Paradigms

Programming paradigms are structured approaches to organizing code. They provide specific rules and concepts whose determination shapes how programs are developed, how logic is represented, and how behaviors are controlled within the software [31]. Existing programming paradigms, including Object-Oriented Programming (OOP), Functional Programming (FP), Event-Driven Programming (EDP), and Aspect-Oriented Programming (AOP), exhibit significant drawbacks when managing decentralized data and distributed systems [31]. For instance, object-oriented programming (OOP) has limitations in handling decentralized data because of structural rigidity and complexity in synchronization/coordination tasks [31]. Additional limitations include explicit concurrency management complexity, inefficient data and error handling, reduced scalability, and rigid structures that diminish modularity and adaptability while increasing development effort [31–34]. While programming languages provide support mechanisms such as monitors, threads, message passing, and locks to address these challenges, they often require manual intervention, which increases system complexity and the likelihood of design errors. Since the design phase is an essential part of software development, it is important to address architectural challenges at the paradigm level rather than relying solely on language-specific features and implementation details [31].

## 3. Action-Oriented Programming (AcOP)

AcOP shifts computational focus from traditional objects or processes to actions, improving modularity, scalability, concurrency management, and synchronization through guards and atomic execution [9,12,35,36]. Unlike conventional models such as OOP, where processes actively control execution, AcOP treats processes as passive resources. This approach simplifies coordination between distributed components by relying on predefined guards (conditions) instead of complex communication protocols [37,38]. AcOP separates data from behavior, simplifies concurrency through implicit management, and offers enhanced reliability and scalability for decentralized environments [9].

### 3.1. Structure of the AcOP Paradigm

The AcOP paradigm consists of several core components which form the foundation of AcOP, as defined in studies by Järvinen et al. (1990, 2013) [9,35,39].

**Action**: Actions are the execution units in which all system functions are contained. An action has an optional list of parameters, a set of participants that are engaged in the action in specific roles, a guard and a body (see Appendix A.1 for details) [9].

**Guard (Precondition)**: The guard defines the conditions under which an action can be executed. It ensures that actions only take place when the system is in a suitable state, thus controlling when an action is triggered by checking system conditions [35]. The action guards in actions, TLA [36], and DisCo [35] can be divided into two parts:

- Common part that can refer to the contents of several participants, which is in an Action.
- Local part that can refer to the contents of a single participant only.

Action guards are syntactically split to local and common parts to make the scheduler implementation more efficient. For example, the local part of the guard is evaluated only when the contents of the corresponding participant are changed. Additionally, if any local part evaluates to false, the common part does not need to be evaluated [9].

**Roles**: Each action defines specific roles that objects must fulfill to participate. Roles describe the responsibilities of objects in the action and dictate how they interact with other components during the execution of the action [10,39].

**Body**: The body contains the executable logic that defines the interaction between objects assigned to different roles. It ensures that the system performs the desired operation based on the capabilities of the objects involved [10].

**Participants-Objects**: An action has participants (objects) that participate in the action and store data state that can include a variety of data types, structures, and containers. From a programmer's perspective, objects can be seen as record types. These objects are typically created when the system is initialized, but the model also allows for the creation and destruction of objects during runtime. In this model, actions involve participants, which are the objects themselves. Each object can participate in an action only in a single, distinct role (see Appendix A.2 for details) [9,39].

**Scheduler**: The scheduler is responsible for selecting a set of actions that are connected to the objects for execution. It ensures mutual exclusion and synchronization by managing which actions are enabled and ready to run. The scheduler selects and executes actions from the action store based on their guards and system state (see Appendix A.3 for details) [9].

Additional components can be developed by extending the core components of AcOP or through the integration of other paradigms. Nonetheless, the fundamental template of AcOP remains rooted in the foundational studies by Järvinen et al. (1990, 2013) [9,35,39].

The general structure of an AcOP system is illustrated in Figure 2. Unlike traditional thread- or class-based designs, AcOP centers around a dynamic coordination loop managed by a *Scheduler*. A practical implementation example demonstrating the interaction between participants, actions, and the scheduler is provided in Appendix A.4.

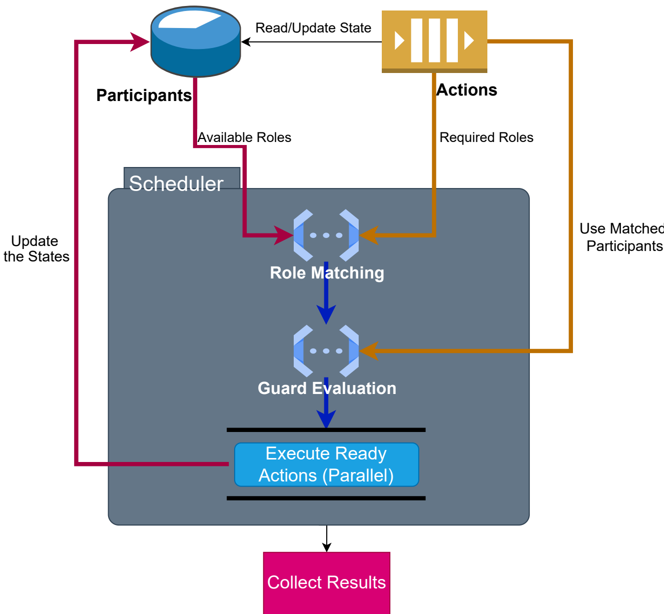


**Figure 2.** General structure of an AcOP-based action system, illustrating the scheduler-driven coordination loop among participants, actions, and guards.

https://doi.org/10.3390/pr14101669

All executable behaviors in the system are defined as *Actions*, each of which specifies a set of required roles and a guard condition. The *Participants* provide these roles and hold the corresponding data states. The scheduler first performs *Role Matching* to assign participants to actions based on available roles. It then evaluates the *Guards* of each action to determine which ones are ready to execute. Once guards are satisfied, eligible actions are executed in parallel with the capability of reading/updating the state of associated participants. After execution, the updated participant states are re-evaluated in the next cycle. This loop continues until no further actions can be executed. Finally, the scheduler collects the resulting data and outcomes for downstream use or analysis.

### *3.2. Objects in AcOP*

In AcOP, the concept of objects differs fundamentally from OOP. Unlike OOP, where objects encapsulate methods, AcOP separates behavior from objects entirely. All functionality is implemented within actions, not within the objects themselves. Actions are executed by a system scheduler rather than through external method calls, promoting modularity and clear separation of system concerns [9].

### *3.3. Execution of Actions*

Actions serve as atomic units of execution in AcOP, representing transitions between system states. Each action is responsible for progressing the system from one state to the next. This step-by-step execution ensures predictable state transitions while maintaining system consistency [40].

### *3.4. Non-Determinism in AcOP*

AcOP supports non-deterministic execution, where any enabled action can be selected for the next state transition. This feature is particularly useful for concurrent and distributed systems, as it allows multiple actions to execute simultaneously when they do not share conflicting variables or objects. A formal condition ensures conflict-free parallelism, providing a foundation for safe and efficient concurrent execution [40].

### *3.5. Atomicity in AcOP*

Atomicity is a core property in AcOP, ensuring that actions execute completely without interference from other actions. This prevents disruptions caused by intermediate states, maintaining the intended flow of execution. Atomicity is critical for applying AcOP in distributed and concurrent systems, enabling reliable and modular system behavior [40].

## 4. Automatic Agent Generation (AAG)

The advancements in AI have led to the development of systems capable of performing increasingly complex tasks. Among these, Large Language Models (LLMs) and agentic systems represent two pivotal innovations for AAG.

The foundation of LLMs lies in the Transformer Model, an architecture introduced by Vaswani et al. in 2017 [41]. Leveraging self-attention mechanisms, Transformers significantly improve their ability to identify patterns and contextual relationships within large volumes of data, ultimately providing role-based adaptability. With properly structured prompts, they can simulate domain-specific expertise and act as virtual specialists in areas like project management, QA, or data analysis [42,43].

Advancements in LLMs have enabled the development of autonomous agentic systems capable of performing complex tasks [41,42,44]. AAG leverages LLMs to dynamically generate specialized agents capable of role-specific actions and decisions within decentralized systems, significantly enhancing system adaptability and reducing manual configuration requirements [11,45,46].

https://doi.org/10.3390/pr14101669

AAG is particularly effective in handling complex, multidisciplinary tasks and environments requiring intensive domain knowledge [11]. By decomposing tasks into planning and execution phases, it assembles a team of specialized agents and enables them to collaborate iteratively toward solving the task [11]. Figure 3 illustrates this high-level workflow, where the manager agent coordinates the plan and agent team before executing the task cooperatively.

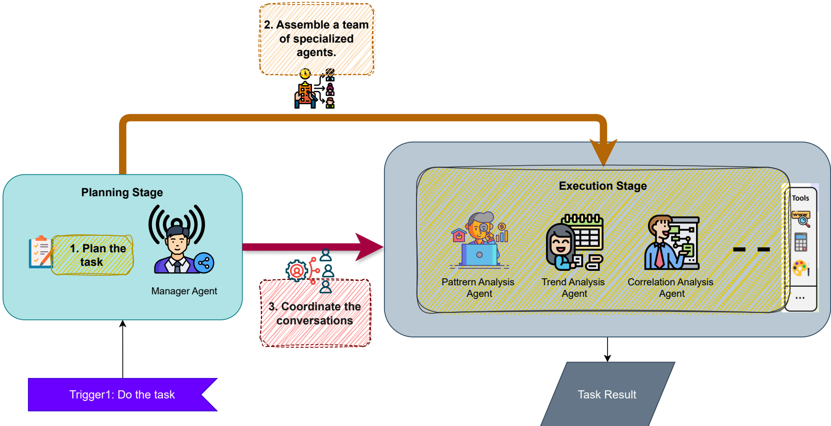


**Figure 3.** Conceptual overview of AAG. The manager agent decomposes a task into a plan and agent team, which then executes the task collaboratively.

Empirical evaluations demonstrate that AAG consistently outperforms traditional methods, delivering coherent and accurate solutions across diverse benchmarks. Its ability to dynamically adapt, assign roles automatically, and coordinate teams makes AAG a robust framework for addressing complex problem-solving in dynamic and knowledge-intensive environments [11,46].

## 5. Methods

This study investigates the research questions by designing and implementing three software prototypes to simulate a data collection application for RCA and RA in a semiconductor manufacturing fab. A modular microservice architecture was used to reflect the decentralized and heterogeneous nature of manufacturing data systems, highlighting the broader applicability and benefits of such approaches.

### 5.1. System Design and Implementation

The software prototypes are based on a simplified version of the Holistic Product Check (HPC) application, a web-based microservice system originally developed to perform ETL (Extract, Transform, Load) tasks in manufacturing environments for data collection. The HPC application interacts with various APIs, databases, and data sources to collect manufacturing data and quickly generate reports based on product IDs, enabling informed decisions in RCA and RA. The HPC application was designed and implemented with the OOP paradigm.

For this study, the application was streamlined to focus on core tasks (as shown in Figure 4), using historical data to facilitate controlled experimentation. The evaluation dataset comprised 700 records sourced from actual manufacturing workflows. The records contained numeric fields (integers and floats) and textual fields, organized as tabular records drawn from relational databases, document-oriented databases, spreadsheet files, and web-scraped pages. To satisfy confidentiality requirements, critical sensitive portions were replaced with structurally equivalent synthetic values, while the remaining records were retained from the original industrial workflows. This configuration preserved the

schema diversity and format inconsistencies characteristic of a decentralized manufacturing environment, providing a realistic basis for evaluating how each implementation handles coordination challenges across heterogeneous data sources. Several key requirements guided the system design. The design must have **concurrency management** through multithreading to parallelize operations such as querying multiple data sources, thereby improving **run time efficiency**. It should ensure **maintainability** through a modular structure that allows individual components and data sources to be modified without affecting the entire system. It should provide **fault tolerance** to prevent errors in one component from disrupting the entire data processing workflow. Finally, the architecture should be **scalable**, capable of efficiently handling a large number of tasks. All three prototypes were developed in Python 3.11. All reported performance metrics were collected from an internal hybrid-cloud deployment, where each service ran as a containerized instance with 4 GB of memory and 400 millicores allocated. To compare different approaches, three variants of the HPC application were designed:

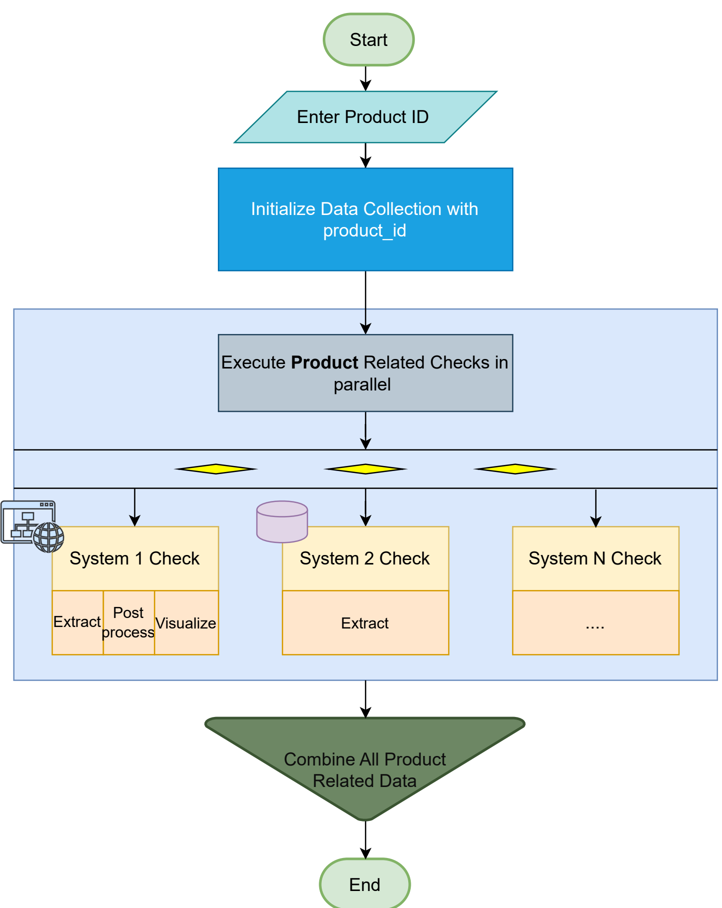


**Figure 4.** Flowchart of the HPC application showing the data collection (extraction, post-processing, and visualization) pipeline.

### 5.1.1. OOP-Based Implementation

Designed and implemented using conventional OOP, where each check process is encapsulated in dedicated classes, directly managing data extraction, processing, and visualization. Class diagrams serve as the blueprints for OOP systems or subsystems, providing visual representations of system structure and illustrating relationships between classes, including inheritance, polymorphism, associations, dependencies, and aggregations [47]. The colors in the class diagram align with those in the flowchart (see Figure 4), visually representing the class hierarchy and its components.

The OOP design shown in Figure 5 prioritizes encapsulation and reusability but relies on explicit synchronization and dependency management, which becomes cumbersome in complex workflows.

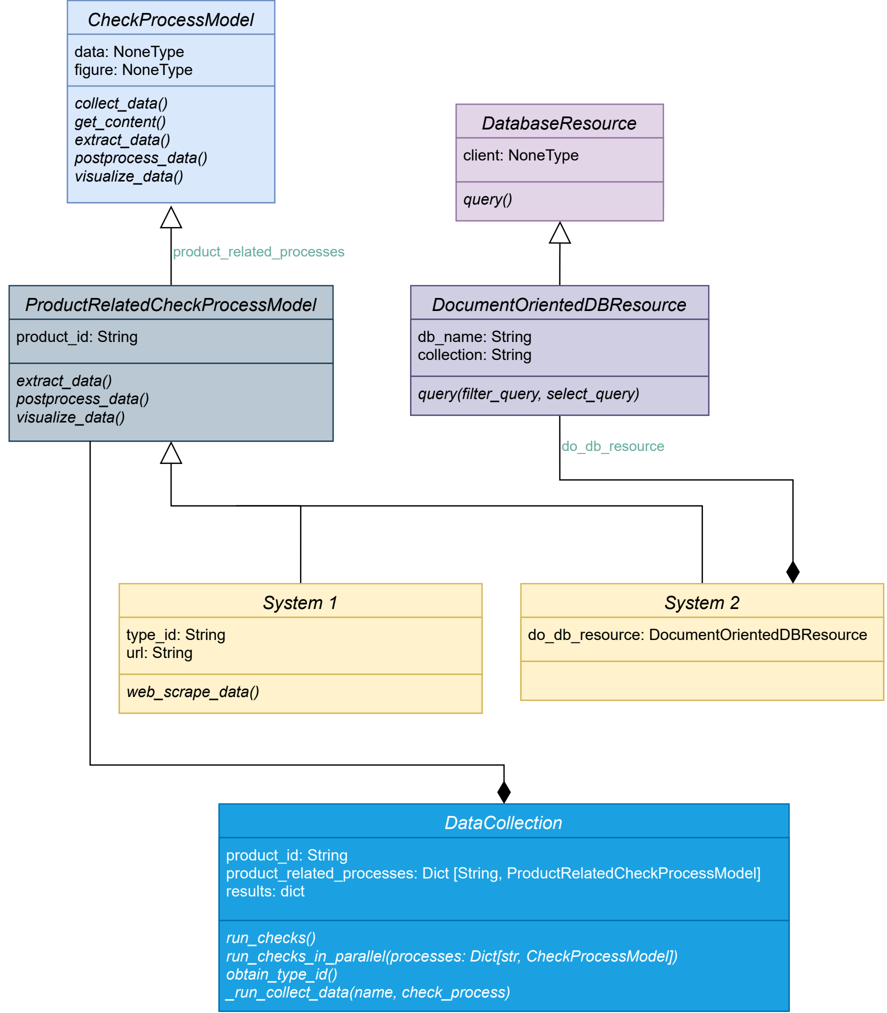


**Figure 5.** Class diagram of the HPC Application with OOP paradigm.

### 5.1.2. AcOP-Based Implementation

The AcOP-based implementation restructured the application using AcOP, clearly separating concerns into discrete actions as illustrated in Figure 6. Each action explicitly defined participant roles, guard conditions (preconditions for execution), and execution logic. The AcOP scheduler dynamically matched actions to participants based on these roles and evaluated guards to manage concurrency implicitly.

This approach significantly simplified coordination, reduced implementation complexity, and improved modularity and scalability by clearly separating behavior from data states, enabling easier modifications and independent component updates.

As described in Section 3, an action in the AcOP paradigm represents a state transition. A state diagram is the most appropriate tool for illustrating an AcOP system. Unlike class diagrams (used in OOP), which can be confusing as AcOP has a different object concept, and activity or sequence diagrams, which fail to capture non-sequential and non-deterministic workflows, state diagrams effectively represent the design of AcOP. For simplicity, the state diagram used here includes special shapes for actions and participants, along with their associated guards.

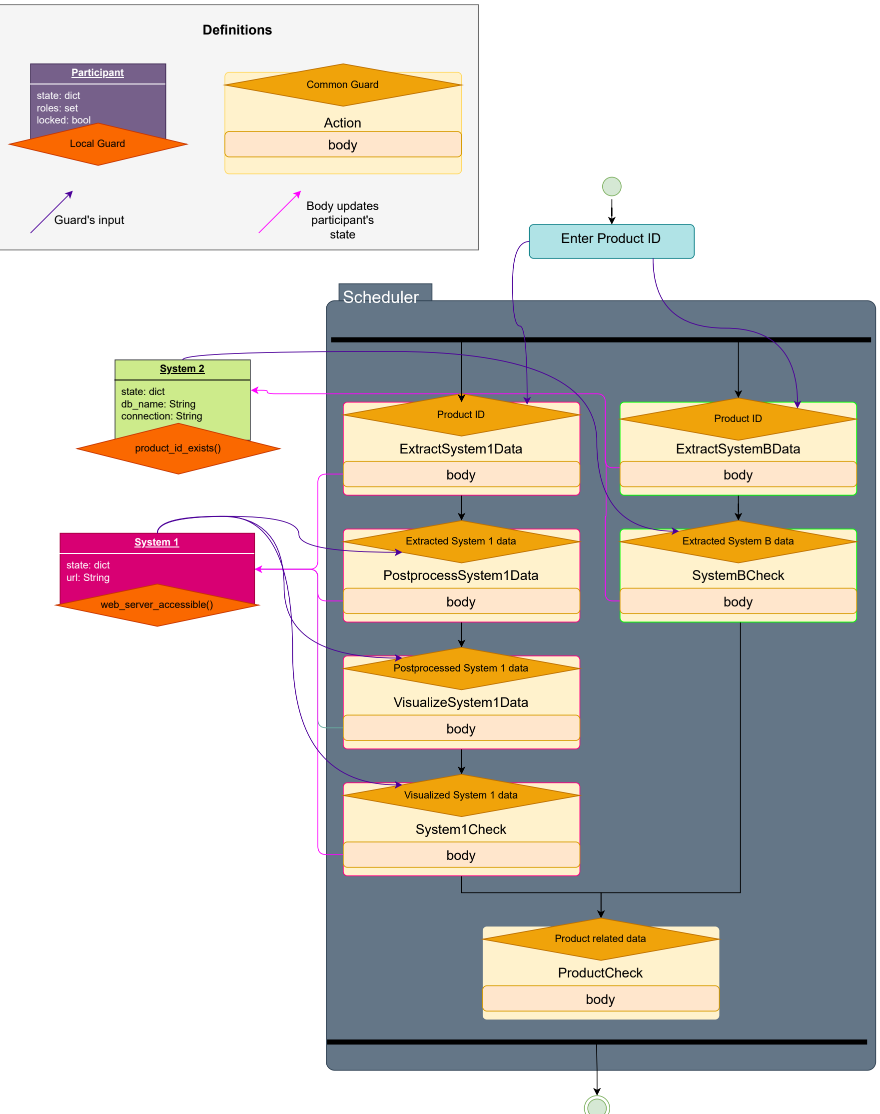


**Figure 6.** State diagram of the AcOP-based HPC application, depicting actions, participants, and guard conditions. Arrows with different colors represent different types of relationships and behaviour.

The state diagram provides a deeper understanding of the components and the workflows of the HPC system. Additional implementation details are available in Appendix A.

### 5.1.3. AAG Implementation

To evaluate the impact of AAG in the AcOP-based HPC application for data collection and analysis (see Section 5.1), two use cases were identified. The original AAG framework was adapted to align with the specific requirements of each experimental use case. While the overall structure of planning and execution remained consistent, additional predefined or dynamically generated phases were incorporated or omitted depending on the specific needs of the application.

- **AcOP + AAG Integrated Implementation (AAG Coordinator)**: Enhances the AcOP scheduler by automating participant-action matching and guard generation, eliminating manual configuration of component relationships and runtime dependencies.
- **AAG Analyzer**: Provides immediate preliminary analysis of collected data through domain-specific agents, generating insights that streamline subsequent RCA and RA processes, independent from the programming perspective.

The AAG components were built using the CrewAI 0.86 orchestration library and an internally deployed Llama 3.3 70B model. No task-specific fine-tuning or manual hyperparameter tuning was applied; the model was used with the default generation settings available through the deployment environment and CrewAI integration. In addition, no fixed random seed or other explicit reproducibility control was enforced.

### 5.1.4. AcOP + AAG Integrated Implementation

AcOP's inherent separation of data states and behaviors through guards and centralized scheduling provides an ideal foundation for AAG integration. Building upon this modular architecture, AAG enhances system capabilities by automating task adaptation and enabling dynamic reconfiguration with minimal manual intervention. The integration leverages AAG's ability to generate intelligent agents that adapt actions to evolving requirements and resource availability.

As demonstrated by Chauhan et al. [48], multi-agent systems can automatically generate microservices and debug workflows based on API definitions. This approach aligns naturally with AcOP's structure, where services correspond to actions and resources to participants, creating synergies that streamline development and prototyping processes.

Through dynamic generation of LLM-based agents, this implementation automates critical configuration tasks while autonomously coordinating system components and defining execution conditions at runtime. This approach significantly enhances system adaptability, reduces manual development effort, and eliminates traditional synchronization complexity, directly contributing to improved modularity and maintainability. To control ambiguity at the integration boundary, agent-generated configurations are validated through a schema enforcement layer that ensures type safety and structural consistency before they are passed to the AcOP scheduler.

As illustrated in Figure 7, AAG helps automate the matching process. Given that behavior and data-state are already separated in AcOP, these matching and guard operations are managed by the AI agents.

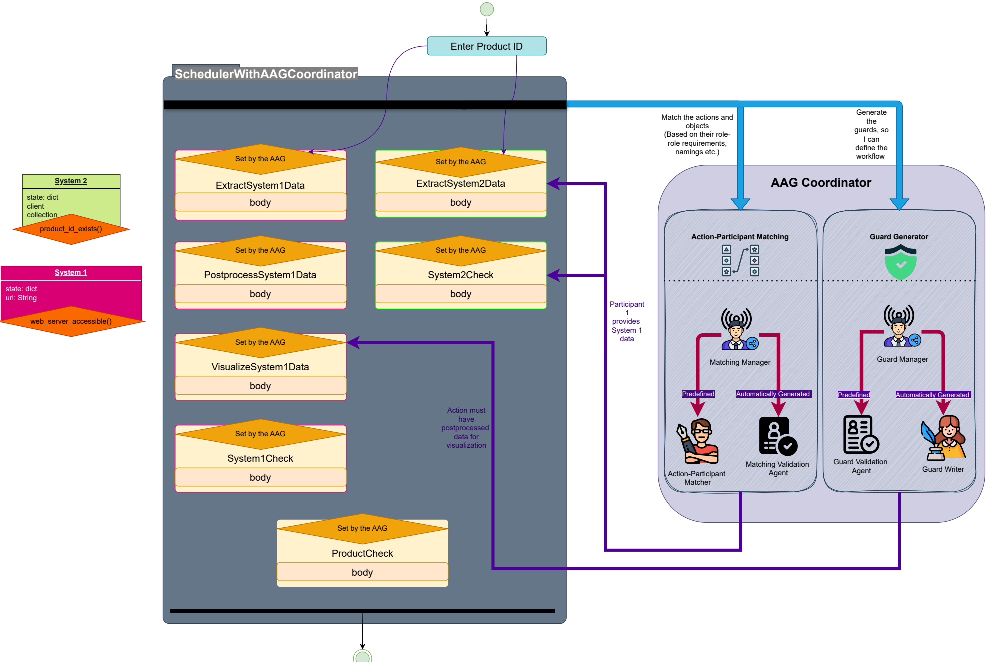


**Figure 7.** Execution stages of the AAG Coordinator within the AcOP-based HPC application. Compared with the manually configured AcOP system (Figure 6), workflow dependencies are substantially reduced through automated participant–action matching.

All three implementations aim to fulfill the same functional objectives and were evaluated under equivalent execution scenarios. The prototype intentionally excludes non-essential interface layers, such as user interface components, in order to isolate core system process behavior under controlled conditions. The underlying HPC application has been actively maintained in an industrial context for two years; the prototype preserves the same operational process logic that is relevant to the paradigm-level comparison conducted in this study.

#### 5.1.5. AAG Analyzer Component

The AAG Analyzer operates as an independent component that processes data collected by the HPC application to generate preliminary insights for RCA and RA workflows. The framework employs a structured four-stage process:

1. **Planning Stage**: The manager agent analyzes input data to identify patterns, determine analytical requirements, and assemble domain-specific agent teams
2. **Execution Stage**: Specialized agents (Process Engineers, Quality Engineers, Data Analysts) independently analyze data within their expertise domains
3. **Validation Stage**: Statistical and domain validators verify analysis accuracy and practical feasibility
4. **Presentation Stage**: Validated results are compiled into structured markdown reports for RCA/RA integration

AAG Analyzer dynamically adapts analysis approaches based on collected data characteristics, generating domain-specific insights that bridge data collection and preliminary analysis phases. Detailed workflow descriptions with concrete stage-level input/output examples for both AAG components are provided in Appendix B.

### *5.2. Evaluation Metrics and Experimental Scenarios*

Based on the literature review [9,31,49,50] and application key aspects (see Section 5.1), the following evaluation metrics (see Table 1) were defined to systematically assess the three implementation approaches.

**Table 1.** Evaluation Metrics for Systematic Assessment.

| Metric | Description |
| --- | --- |
| Runtime Efficiency | Measures the total execution time required to complete all tasks, capturing overhead introduced by the programming paradigm. |
| Computational Complexity | Characterizes the theoretical time complexity of core algorithmic components using asymptotic analysis where applicable. |
| Concurrency Management | Examines the effectiveness of parallel task execution, thread utilization, and overhead associated with explicit or implicit synchronization mechanisms. |
| Error and Fault Management | Evaluates each approach's capability to detect, isolate, and recover from errors while ensuring workflow robustness. |
| Modularity and Adaptability | Measures how easily the system can adapt to changes, such as adding or modifying components, without affecting the overall architecture. |
| Scalability | Assesses how well the architecture manages increased workloads, evaluating the ease of integrating additional tasks or data sources. |
| Implementation Complexity | Assesses the complexity of implementing and maintaining each paradigm, factoring in the learning curve, manual coding effort, and complexity of managing task coordination explicitly or implicitly. |
| Explainability and Transparency | Evaluates how clearly the system communicates internal operations and decision-making processes to developers, emphasizing system clarity and ease of understanding. |

https://doi.org/10.3390/pr14101669

Runtime Efficiency and Computational Complexity are evaluated quantitatively through execution time measurements and asymptotic analysis, respectively. The remaining metrics capture architectural and design-level properties—such as coordination strategy, fault isolation, and structural modularity—that are inherently qualitative when comparing fundamentally different programming paradigms. Established quantitative software metrics such as Depth of Inheritance Tree, Number of Children, or Coupling Between Object Classes presuppose an object-oriented design and are therefore inapplicable in cross-paradigm evaluations. Accordingly, these metrics are assessed through profiling data, code-review assessments, and structured experimental observations, following standard practice in comparative software engineering research [31]. Experimental scenarios were designed to assess these metrics comprehensively under identical conditions. These scenarios are outlined in Table 2.

**Table 2.** Experimental Scenarios for Comprehensive Evaluation.

| Scenario | Description |
|---|---|
| Normal Operational Scenario | All implementations executed the complete workflow (data extraction, postprocessing, visualization) with historical datasets representative of realistic manufacturing data, assessing runtime efficiency and concurrency management. |
| Fault Injection Scenario | Simulated errors, such as database unavailability, missing data, and incorrect parameters, were introduced to evaluate the robustness of each approach in error and fault management. |
| Incremental Workload Scaling Scenario | The number of tasks and data sources was incrementally increased to test scalability, modularity, and adaptability under higher workloads and changing requirements. |
| Implementation Effort Scenario | The complexity of initial implementation, maintenance, and modifications was evaluated considering factors like the learning curve, coding effort, and maintainability. |
| Transparency Evaluation Scenario | Each implementation's internal logic, task coordination mechanisms, and decision-making operations were reviewed and evaluated for explainability and transparency. |

Performance and behavior were systematically measured and analyzed using profiling tools, qualitative assessments, and manual code reviews to ensure comprehensive evaluation.

## 6. Results

This section presents the experimental evaluation comparing OOP, AcOP, and AcOP + AAG implementations based on clearly defined metrics derived from the experimental design.

Runtime Efficiency

Multiple experimental runs were conducted across different workload scenarios to ensure statistical validity of performance measurements. The OOP implementation achieved consistent performance with a mean runtime of 3.93 s (standard deviation: 0.3 s, sample size: 12 runs), while AcOP demonstrated stable performance at 8.17 s (standard deviation: 0.5 s, sample size: 12 runs). The AcOP + AAG implementation showed higher variability due to LLM infrastructure dependencies with a mean runtime of 40.2 min (standard deviation: 4.8 min, sample size: 12 runs). Sample sizes were determined based on computational resource constraints and convergence criteria for performance stability.

Computational Complexity

The computational complexity of deterministic algorithmic components is summarized in Table A3 (Appendix C). Each data processing stage (extraction, postprocessing, visualization) runs in $O(n)$ where $n$ is the number of records. In the OOP implementation,

$N$ check systems are orchestrated through threading, yielding $O(N \cdot n)$ total work. In the AcOP implementations, each stage is an individual action; the $N$ check systems decompose into $a$ total actions (e.g., $a = 20$ for $N = 5$ in the experimental prototype). Each action body executes in $O(n)$, contributing $O(a \cdot n)$ for action execution. The scheduler additionally performs participant matching in $O(r \cdot p)$ per action and guard evaluation in $O(r)$ per action, where $r$ is the number of roles and $p$ the registered participants. The main loop iterates $I$ times ($I \leq a$), yielding $O(I \cdot a \cdot r \cdot p)$ scheduler overhead and $O(a \cdot n + I \cdot a \cdot r \cdot p)$ combined. For the AAG components in the AcOP + AAG implementation, classical Big-O analysis is not applicable because the computational cost depends on variable-length LLM input context, the number of internal agent delegations is determined at runtime, and identical inputs can produce different execution paths. The empirical runtime measurements reported above (mean: 40.2 min, SD: 4.8 min, $n = 12$) therefore provide a more meaningful performance characterization for these non-deterministic components.

Concurrency Management

Concurrency management was explicitly handled in the OOP variant through manual thread management and synchronization, leading to increased development effort and complexity. Conversely, the AcOP implementations (AcOP and AcOP + AAG) simplified concurrency significantly through implicit, centralized thread coordination managed by the scheduler and guards, effectively reducing synchronization overhead and concurrency-related development effort.

Error and Fault Management

The OOP implementation required explicit manual handling of error conditions and lacked systematic mechanisms for fault isolation, increasing the likelihood of human errors and oversight. The AcOP-based implementations demonstrated robust error isolation through the atomic execution of actions and guard-based execution conditions, effectively preventing propagation of errors throughout the system. However, the implicit nature of AcOP's error management made debugging somewhat more challenging due to limited tooling and practical support.

Modularity and Adaptability

The modularity of AcOP-based implementations surpassed the OOP implementation. AcOP clearly separated logic and data states within self-contained actions, enabling seamless updates, replacements, and integration of new components. In contrast, the OOP variant's tightly coupled classes limited modularity and adaptability, requiring significant effort for component modifications or additions.

Scalability

Scalability testing involved incremental workload increases from 3 to 18 concurrent tasks across multiple experimental runs. All three implementations demonstrated acceptable scalability, with distinct trade-offs. AcOP-based systems allowed easy addition of new actions due to their modular and additive architecture; however, increasing complexity arose from managing guards explicitly as application complexity scaled. In comparison, the OOP implementation required careful architectural restructuring for scalability, placing higher demands on developer expertise.

Implementation Complexity

Implementation complexity was highest for AcOP-based implementations due to the steep learning curve and the necessity of arranging workflows using guards, resulting in increased initial development complexity. The OOP implementation demonstrated moderate complexity, given developers' familiarity with explicit synchronization methods

https://doi.org/10.3390/pr14101669

and sequential structures. AcOP + AAG notably reduced implementation complexity, automating significant portions of configuration and participant-role matching through generative agents.

Explainability and Transparency

Without AAG, the system is more explainable and transparent, as workflows and guard mechanisms are explicitly defined and traceable. In contrast, AAG introduces opacity due to its reliance on black-box LLM operations for tasks like guard generation and action-participant matching, making validation more difficult even though individual agent logs, outputs and actions are recorded for post-execution analysis, enabling traceability of agent decisions and validation of automated logic flows. This trade-off between automation and explainability requires careful consideration in critical applications.

### *6.1. Summary of Comparative Results*

Table 3 summarizes the evaluation outcomes, clearly derived from the experimental results:

**Table 3.** Comparative evaluation summary of OOP, AcOP, and AcOP + AAG implementations. Qualitative ratings are based on profiling data, code-review assessments, and experimental observations detailed in Section 6.

| **Evaluation Metric** | **OOP** | **AcOP** | **AcOP + AAG** |
|---|---|---|---|
| Runtime Efficiency | High | Medium | Low |
| Computational Complexity | $O(N \cdot n)$ | $O(a \cdot n + I \cdot a \cdot r \cdot p)$ | Non-det. |
| Concurrency Management | Medium | High | High |
| Error and Fault Management | Basic | Robust | Robust |
| Modularity and Adaptability | Low | High | High |
| Scalability | Medium | Low | High |
| Implementation Complexity | Medium | High | Medium |
| Explainability and Transparency | High | High | Low |

Overall, the AcOP paradigm notably enhanced modularity, concurrency management, error handling compared with conventional OOP, despite incurring runtime overhead, guard complexity, and increased initial implementation complexity. The integration of AAG further improved modularity, scalability, and substantially reduced implementation complexity through automated agent-based configuration tasks; however, this introduced significant runtime overhead and reduced explainability due to reliance on opaque LLM processes. These trade-offs highlight the importance of balancing automation benefits against transparency and performance considerations in practical applications. The AAG Analyzer component, operating independently from the programming paradigms, successfully demonstrated automated domain-specific analysis capabilities that enhance RCA and RA workflows beyond the core programming paradigm benefits.

### *6.2. AAG Analyzer*

The AAG Analyzer operated independently of programming paradigm metrics, providing automated preliminary analysis for RCA and RA workflows through its four-stage process (Planning, Execution, Validation, Presentation).

**Key Performance Outcomes:**

- **Dynamic Adaptation**: Successfully generated domain-specific agents based on data characteristics—process optimization agents for yield data, compliance analysts for quality metrics
- **Workflow Integration**: Effectively bridged data collection and analysis phases, reducing time gaps between data availability and actionable insights

- **Validation Accuracy**: Multi-stage validation ensured statistical rigor and domain relevance of generated insights

**Limitations**: Similar to AAG Coordinator, introduced transparency challenges due to black-box LLM operations, requiring additional validation steps in critical applications.

## 7. Discussion

The experimental evaluation demonstrates clear benefits of integrating AcOP and AAG over conventional OOP for decentralized data collection systems, albeit with distinct trade-offs. It is important to note that AcOP remains an under-researched programming paradigm with limited industrial applications for direct comparison. AAG represents an emerging approach where comparative benchmarks are primarily internal to specific application domains. The focus of this research is on demonstrating framework capabilities and workflow improvements rather than performance optimization against established competitors.

### *7.1. Interpretation of Findings*

Our findings directly address the challenges identified in RQ1, revealing four primary issues in semiconductor data collection: data fragmentation across decentralized systems, scalability limitations, synchronization overhead, and adaptability constraints that delay RCA and RA processes.

The AcOP-based design (RQ2) improved modularity and flexibility through loosely coupled actions, each encapsulating its own logic, roles, and conditions. This modular structure enabled easier adaptation to changes and better separation of concerns, especially in distributed environments with varying data sources. However, this came with runtime trade-offs (8.17 s vs. 3.93 s for OOP) and increased initial implementation complexity due to the paradigm's learning curve.

By incorporating AAG into the AcOP framework (RQ3), automation was significantly enhanced through dual advantages: automated participant-role matching and guard generation, plus bridging data collection and preliminary analysis phases. Intelligent agents generated using LLMs successfully minimized manual configuration, reducing development effort while improving maintainability and scalability. The AAG Analyzer component demonstrated particular value in automating preliminary data analysis, essential support for RCA and RA workflows in semiconductor manufacturing environments where data variability across production cycles requires adaptive analytical approaches.

### *7.2. Comparison with Conventional Approaches*

The results reinforce that conventional OOP systems, although widely used, are less suitable for dynamic, decentralized scenarios due to their rigid architecture and high coordination overhead. The combined AcOP + AAG approach (RQ4) achieved enhanced scalability, improved fault tolerance through atomic action execution, and reduced operational overhead—directly addressing semiconductor manufacturing's need for rapid response to production issues through microservice architecture benefits.

The AcOP + AAG implementation significantly reduced the manual effort required in managing complex decentralized workflows. In contrast to static implementations, this agent-driven design improved responsiveness to changing system conditions and reduced human workload during development and system expansion. Nevertheless, the black-box nature of the LLM-based agents decreased system explainability and transparency, making debugging and validation challenging. This poses risks in scenarios requiring high transparency, rigorous validation, and trustworthiness, such as critical manufacturing processes or safety-critical applications.

### 7.3. Implications for Research and Practice

From a practical perspective, adopting AcOP and AAG presents clear advantages in flexibility, maintenance, and scalability, especially in complex semiconductor manufacturing environments. However, practical adoption necessitates infrastructure optimization and careful management of transparency risks inherent in generative AI components.

The broader organizational impacts (RQ5) include shifted workforce skill requirements toward declarative programming concepts, reduced inter-team coordination overhead, and strategic advantages through faster adaptation capabilities, though requiring investment in validation frameworks for critical applications.

Theoretically, this research contributes to advancing both AcOP and agent-based software design. While AcOP's formal foundations have existed for decades, its application in modern, distributed computing contexts has remained limited. This work provides an updated use case demonstrating its practical relevance, while LLM-based agents expand the role of AI in programming paradigms.

### 7.4. Limitations

This study was conducted in a controlled simulation environment with historical data and predefined workloads. While these conditions were designed to reflect realistic challenges, real-world deployments may involve additional uncertainties such as data quality issues, real-time data streaming, infrastructure limitations, or integration constraints. Additionally, LLM infrastructure dependencies introduce potential model drift concerns, and agent training requires significant lead times that may limit real-time responsiveness.

### 7.5. Future Work

Agentic systems, especially AAG, are highly adaptable and can be helpful in domain knowledge-intensive areas, workflows, component matching such as resources-services and their management. AAG use-case examples can be examined in broader perspectives including preliminary analysis, workflow optimizations and bottleneck identifications. Future work should examine improving the runtime efficiency of the AAG approach through optimization techniques such as caching, continuous agent learning, real-time role adaptation, periodic pre-computation, using lightweight models for less complex subtasks, and broader integration with external data ecosystems. Furthermore, enhancing the transparency and explainability of agent-generated decisions remains an essential direction for research. Extending evaluations beyond semiconductor manufacturing to other domains, such as healthcare, IoT, and logistics, would further validate the generalizability of these frameworks.

## 8. Conclusions

This study investigated the integration of AcOP and AAG to address challenges in decentralized data collection supporting RCA and RA processes in semiconductor manufacturing. Three software implementations—OOP, AcOP, and AcOP + AAG—were systematically evaluated across several performance and complexity metrics.

The findings indicate that AcOP substantially improves modularity, concurrency management, error isolation, and adaptability compared with conventional OOP designs, despite moderate runtime overhead and higher initial implementation complexity. The additional integration of AAG significantly reduces manual effort and enhances system scalability through automated participant-role matching and guard generation. However, the automation introduced notable runtime inefficiencies due to the iterative processing–evaluation loop of the AcOP paradigm and agent generation, and reduced system transparency due to reliance on black-box LLM components.

The AAG Analyzer component specifically addressed the critical gap between data collection and analysis, providing domain-specific insights that streamline RCA and RA workflows.

The contributions of this work are both theoretical and practical. It extends the applicability of AcOP in modern software design and introduces AAG as a viable tool for automating complex development tasks using LLMs. The findings suggest that such an integrated framework can reduce development complexity while supporting adaptive and autonomous data pipelines.

Future research should further explore and validate the integrated AcOP-AAG framework, optimizing its balance between automation, transparency, and performance across diverse application domains.

**Author Contributions:** Conceptualization, M.T.B., H.R. and K.K.; methodology, M.T.B.; software, M.T.B.; validation, M.T.B., H.R. and K.K.; formal analysis, M.T.B.; investigation, M.T.B.; resources, H.R. and K.K.; data curation, M.T.B.; writing—original draft preparation, M.T.B.; writing—review and editing, M.T.B., H.R. and K.K.; visualization, M.T.B.; supervision, H.R. and K.K.; funding acquisition, H.R. and K.K. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was funded by the European Union's Horizon Europe research and innovation program Chips Joint Undertaking (Chips JU) under grant agreement number 101097300. Additional support was provided by Infineon Technologies Austria AG.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** The datasets used in this study are not publicly available due to confidentiality and proprietary restrictions. The data include sensitive semiconductor manufacturing and quality records provided by industrial partners, and public sharing is restricted to protect confidential operational information.

**Acknowledgments:** The authors deeply thank Infineon Technologies Austria AG for providing resources and support for this research. This work is based on the first author's master's thesis research conducted at both institutions. The project EdgeAI "Edge AI Technologies for Optimised Performance Embedded Processing" is supported by the Chips Joint Undertaking and its members including top-up funding by Austria, Belgium, France, Greece, Italy, Latvia, Netherlands, and Norway under grant agreement No. 101097300.

**Conflicts of Interest:** Authors Mustafa Tayyip Bayram & Houssam Razouk were employed by the company Infineon Technologies Austria AG. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

**Ethics Statement:** This research adheres to established ethical principles for artificial intelligence applications in industrial settings. All experimental procedures were conducted with transparency in automated decision-making processes while maintaining appropriate human oversight. The study utilized only historical manufacturing datasets that were properly anonymized and contained no sensitive personal information. Data handling procedures followed institutional guidelines for responsible research practices, ensuring secure storage and access controls throughout the research period. The integration of large language models in the AAG framework was implemented with consideration for responsible AI deployment, including validation mechanisms and audit trails for agent-generated decisions. The research methodology prioritized reproducibility with all code implementations and experimental protocols documented for verification purposes.

## Abbreviations

The following abbreviations are used in this manuscript:

| | |
|---|---|
| RCA | Root Cause Analysis |
| RA | Risk Assessment |
| QA | Quality Assurance |
| AcOP | Action-Oriented Programming |
| AAG | Automatic Agent Generation |
| LLM | Large Language Model |
| OOP | Object-Oriented Programming |
| 8D | Eight Disciplines |
| API | Application Programming Interface |
| GUI | Graphical User Interface |
| FP | Functional Programming |
| EDP | Event-Driven Programming |
| AOP | Aspect-Oriented Programming |
| DisCo | Distributed Cooperation |
| TLA | Temporal Logic of Actions |
| HPC | Holistic Product Check |
| IoT | Internet of Things |
| ETL | Extract, Transform, Load |
| RAG | Retrieval-Augmented Generation |

## Appendix A. Action-Oriented Programming (AcOP) Generic Implementation

### *Appendix A.1. Actions Generic Implementation*

The following class demonstrates the generic implementation of actions in AcOP. Actions define the system's functionality, containing a set of roles, guards, and a body. Listing A1 presents the Action class implementation.

**Listing A1.** Action Class.

```
class Action:
    """Defines an action in AcOP."""
    def __init__(self, name: str, required_roles: List[str],
                 common_guard: Union[Callable, bool] = False):
        self.name = name
        self.required_roles = required_roles
        self.common_guard = common_guard
        self.participants: Dict[str, Participant] = {}

    def check_guards(self) -> bool:
        """Check if all guards (common and local) are satisfied."""
        if not self.participants:
            return False
        if callable(self.common_guard) and not self.common_guard(self.participants)
    :
            return False
        return all(
            callable(participant.local_guard) and participant.local_guard()
            for participant in self.participants.values()
        )
```

**Listing A1.** *Cont.*

```
    def body(self):
        """Defines the action core logic."""
        raise NotImplementedError("Body method must be implemented.")

    def execute(self):
        """Executes the action if guards are satisfied."""
        if self.check_guards():
            results = self.body()  # Execute logic
            for participant in self.participants.values():
                participant.set_state(self.name, results)
            return results
        else:
            print(f"Action {self.name} skipped due to failed guards.")
        return None
```

*Appendix A.2. Participants-Objects Generic Implementation*

The following class demonstrates the generic implementation of participants-objects that hold data and state in AcOP. Participants-objects store the data state and participate in actions based on their roles. Listing A2 presents the Participant class implementation.

**Listing A2.** Participant Class.

```
@dataclass
class Participant:
    """Represents a participant-object in AcOP."""
    def __init__(self, name: str, data_name: str = None, params: Optional[Dict] =
    None,
                 roles: Optional[List[str]] = None, local_guard: Union[Callable,
    bool] = False):
        self.name = name
        self.data_name = data_name if data_name is not None else name
        self.params = params if params is not None else {}
        self.roles = list(set(roles)) if roles else []  # Ensure unique roles
        self.local_guard = local_guard
        self.state: Dict[str, Any] = {}  # Stores participant states
        self.locked = False
        self.data: Optional[Dict] = None  # Tabular data or equivalent
        self.figure: Optional[plt.Figure] = None  # Optional Matplotlib figure

    def update_roles(self, new_roles: List[str]):
        """Add new roles dynamically."""
        self.roles = list(set(self.roles).union(new_roles))
        print(f"Updated roles for {self.name}: {self.roles}")

    def set_state(self, action_name: str, results: Any):
        """Update the participant's state with action results."""
        self.state[action_name] = results
        if isinstance(results, plt.Figure):
            self.figure = results
        elif isinstance(results, (dict, list)):
            self.data = results
```

https://doi.org/10.3390/pr14101669

*Appendix A.3. Scheduler Generic Implementation*

The following class demonstrates the generic implementation of the scheduler in AcOP. The scheduler manages the execution of actions, ensuring concurrency and synchronization. Listing A3 presents the Scheduler class implementation.

**Listing A3.** Scheduler Class.

```
class Scheduler:
    """Manages the scheduling and execution of actions."""
    def __init__(self):
        self.actions = []  # List of actions
        self.participants_pool: Dict[str, Participant] = {}  # Pool of participants
        self.executed_actions = set()  # Tracks completed actions

    def add_participant(self, participant: Participant):
        """Add a participant to the pool."""
        self.participants_pool[participant.name] = participant
        print(f"Added participant {participant.name}")

    def add_action(self, action: Action):
        """Schedules an action."""
        self.actions.append(action)
        print(f"Added action {action.name}")

    def match_participants(self, action: Action):
        """Assigns participants to an action based on required roles."""
        action.participants = {
            role: participant for role, participant in self.participants_pool.items
    ()
            if role in action.required_roles
        }

    def execute(self):
        """Executes all scheduled actions."""
        for action in self.actions:
            self.match_participants(action)
            action.execute()
```

*Appendix A.4. Workflow Example*

The following workflow demonstrates the interaction between participants, actions, and the scheduler in the HPC application. Listing A4 presents a complete workflow example.

**Listing A4.** Workflow Example.

```
# Step 1: Define Participants
participant_a = Participant(name="Database1", roles=["Role1"])
participant_b = Participant(name="Database2", roles=["Role2"])

# Step 2: Define Actions
def common_guard_example(participants):
    return True  # Example always-true guard

action_x = Action(name="ActionX", required_roles=["Role1"], common_guard=
    common_guard_example)
```

https://doi.org/10.3390/pr14101669

**Listing A4.** *Cont.*

```
action_y = Action(name="ActionY", required_roles=["Role2"], common_guard=
    common_guard_example)

# Step 3: Initialize Scheduler
scheduler = Scheduler()
scheduler.add_participant(participant_a)
scheduler.add_participant(participant_b)
scheduler.add_action(action_x)
scheduler.add_action(action_y)

# Step 4: Execute Workflow
scheduler.execute()
```

## Appendix B. AAG Analyzer Implementation

The AAG Analyzer processes HPC-collected data through a structured four-stage workflow to generate preliminary insights for RCA and RA processes.

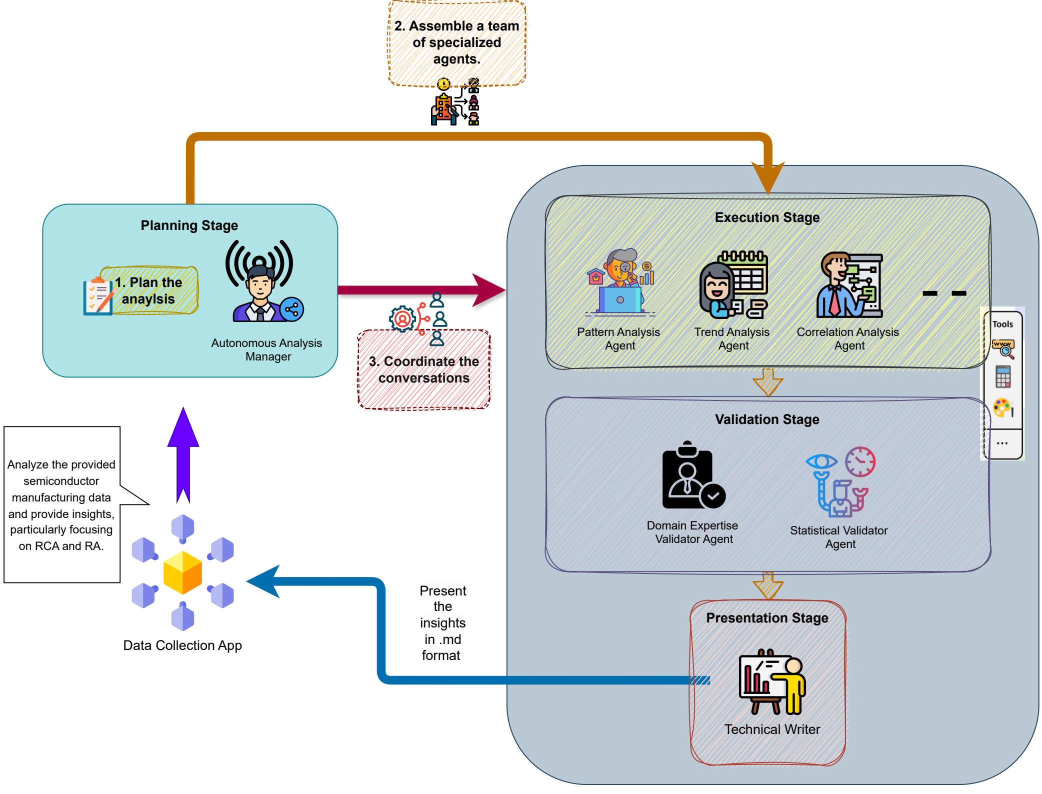


**Figure A1.** Four-stage execution workflow of the AAG Analyzer: Planning, Execution, Validation, and Presentation.

### *Appendix B.1. AAG Component Input/Output Overview*

Table A1 details the input, output, and concrete data examples for each stage of both AAG components (AAG Coordinator and AAG Analyzer), drawn from the HPC application execution with product `PID_123456`.

**Table A1.** Input, output, and concrete examples for each stage of the AAG Coordinator and AAG Analyzer, based on HPC application execution with product `PID_123456`.

| Component | Stage | Input | Output | Example |
|---|---|---|---|---|
| **AAG Coordinator** | Matching | Actions with required roles; participants with assigned roles | Action-to-participant mapping | Sys. 1: `extract_log` → `WebResource` Sys. 2: `extract_quality` → `DBResource` |
| | Guard Gen. | Action–participant pairs, participant states | Boolean guard per action | Sys. 1: `True` (source available) Sys. *N*: `False` (awaiting type ID) |
| | Validation | Generated matchings and guards | Validated matchings with corrections | `extract_quality` needs role `DBReader`; `DBResource` confirmed; no gaps |
| | Execution | Validated guards, matched participants | Results stored in participant states | Iter. 1: Sys. 1, 2 run (guards met) Iter. 2: Sys. *N* runs after dep. resolved |
| **AAG Analyzer** | Planning | Collected data from System $1-N$ (e.g., 700 records, 12 documents, 48 monthly entries) | Analysis types, roles, key metrics, data categories | Types: statistical, optimization Roles: Data Analyst, Process Eng. Metrics: yield rates, defect counts |
| | Execution | Analysis plan; collected data; specialist agents (delegation + code exec.) | Per-agent role-specific insights | Yield: mean 95.2%, SD 2.1 Defect density: 0.05/wafer Yield – cycle time corr. found |
| | Validation | Execution results; Statistical Validator and Insight Verifier agents | Verified and domain-checked results | CIs verified; methodology validated against industry standards |
| | Presentation | Validated results; Technical Writer agent | Structured markdown report | Sections: Executive Summary, Yield Analysis, Defect Density, Recommendations |

*Appendix B.2. Agent Specifications*

Table A2 details key agent roles and operational prompts.

**Table A2.** AAG Analyzer Key Agent Specifications.

| Agent Role | Tasks | Operational Prompt |
|---|---|---|
| Analysis Manager | Orchestrates workflow, identifies data patterns, assembles agent teams | "Analyze the data structure to identify key patterns. Generate specialized agents based on analytical requirements and define their tasks." |
| Automatically Generated Agents | Generated by Analysis Manager | Generated by Analysis Manager |
| Statistical Validator | Validates the statistical rigor of correlations, regression models, and confidence intervals. | "Evaluate statistical models to ensure accuracy and highlight potential improvements." |
| Domain Expertise Validator | Ensures insights align with semiconductor industry standards and best practices. | "Assess the feasibility and practicality of recommendations within the manufacturing domain." |
| Technical Writer | Compiles the findings into a professionally formatted markdown report. | "Structure the report with sections like Executive Summary, Key Findings, and Recommendations." |

## Appendix C. Computational Complexity

**Table A3.** Computational complexity of core algorithmic components. Each check system (System 1 through System $N$) decomposes into multiple actions ($a > N$). Data processing stages are common to all implementations; AcOP adds scheduler overhead. LLM-based AAG components are non-deterministic.

| Component | Complexity | Variables |
|---|---|---|
| Data extraction (per source) | $O(n)$ | $n$ = records in the source |
| Data postprocessing | $O(n)$ | $n$ = records in the dataset |
| Visualization | $O(n)$ | $n$ = data points |
| OOP total | $O(N \cdot n)$ | $N$ = check systems, $n$ = records |
| AcOP action execution | $O(a \cdot n)$ | $a$ = total actions ($a > N$) |
| AcOP participant matching | $O(r \cdot p)$ per action | $r$ = roles per action, $p$ = participants |
| AcOP guard evaluation | $O(r)$ per action | $r$ = roles per action |
| AcOP scheduler (per iteration) | $O(a \cdot r \cdot p)$ | $a$ = remaining actions |
| AcOP scheduler (total) | $O(I \cdot a \cdot r \cdot p)$ | $I$ = iterations ($\leq a$, dependency depth) |
| AcOP combined | $O(a \cdot n + I \cdot a \cdot r \cdot p)$ | Action execution + scheduler overhead |
| AAG matching/guard gen. | Non-deterministic | LLM inference, variable context length |
| AAG Analyzer | Non-deterministic | Dynamic agent count, LLM inference |

## References

1. Moyne, J.; Iskandar, J. Big Data Analytics for Smart Manufacturing: Case Studies in Semiconductor Manufacturing. *Processes* **2017**, *5*, 39. [CrossRef]
2. Choudhary, A.K.; Harding, J.A.; Tiwari, M.K. Data mining in manufacturing: A review based on the kind of knowledge. *J. Intell. Manuf.* **2009**, *20*, 501–521. [CrossRef]
3. Latino, M.A.; Latino, R.J.; Latino, K.C. *Root Cause Analysis: Improving Performance for Bottom-Line Results*; CRC Press: Boca Raton, FL, USA, 2019.
4. Razouk, H.; Kern, R.; Mischitz, M.; Moser, J.; Memic, M.; Liu, L.; Burmer, C.; Safont-Andreu, A. AI-Based Knowledge Management System for Risk Assessment and Root Cause Analysis in Semiconductor Industry. In *Artificial Intelligence for Digitising Industry–Applications*; River Publishers: Gistrup, Denmark, 2022; pp. 113–129.
5. Reed, D.P. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst. (TOCS)* **1983**, *1*, 3–23. [CrossRef]
6. Leca, C.; Fitts, E.P.; Kempf, K.; Uzsoy, R. Towards Decentralized Decisions for Managing Product Transitions in Semiconductor Manufacturing. In *Proceedings of the 2022 Winter Simulation Conference (WSC)*; IEEE: New York, NY, USA, 2022; pp. 3442–3452.
7. Thoke, V.D. Theory of Distributed Computing and Parallel Processing with Its Applications, Advantages and Disadvantages. *Int. J. Innov. Eng. Res. Technol.* **2014**, 1–11. Available online: https://www.academia.edu/download/54559645/1452798652_ICITDCEME-15.pdf (accessed on 17 May 2026).
8. Chandrasekaran, N. Intelligent, data-driven approach to sustainable semiconductor manufacturing. In *Proceedings of the 2022 6th IEEE Electron Devices Technology & Manufacturing Conference (EDTM)*; IEEE: New York, NY, USA, 2022; pp. 1–5.
9. Järvinen, H.M. Actions, objects, and subjects. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*; The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp): Athens, Greece, 2013; p. 277.
10. Mäkitalo, N.; Aaltonen, T.; Raatikainen, M.; Ometov, A.; Andreev, S.; Koucheryavy, Y.; Mikkonen, T. Action-Oriented Programming Model: Collective Executions and Interactions in the Fog. *J. Syst. Softw.* **2019**, *157*, 110391. [CrossRef]
11. Chen, G.; Dong, S.; Shu, Y.; Zhang, G.; Sesay, J.; Karlsson, B.F.; Fu, J.; Shi, Y. Autoagents: A framework for automatic agent generation. *arXiv* **2024**, arXiv:2309.17288. [CrossRef]
12. Kurki-Suonio, R.; Kankaanpää, T. On the design of reactive systems. *BIT Numer. Math.* **1988**, *28*, 581–604. [CrossRef]
13. Aaltonen, T.; Myllarniemi, V.; Raatikainen, M.; Makitalo, N.; Paakko, J. An Action-Oriented Programming Model for Pervasive Computing in a Device Cloud. In Proceedings of the 2013 20th Asia-Pacific Software Engineering Conference (APSEC), Los Alamitos, CA, USA, 2–5 December 2013; Volume 2, pp. 467–475. [CrossRef]
14. Queisser, H.J.; Haller, E.E. Defects in semiconductors: Some fatal, some vital. *Science* **1998**, *281*, 945–950. [CrossRef]
15. Patil, C.A. The Role of Root Cause Analysis in Semiconductor Manufacturing. 14 March 2021. Available online: https://www.chetanpatil.in/the-role-of-root-cause-analysis-in-semiconductor-manufacturing/ (accessed on 7 October 2024).
16. Andersen, B.; Fagerhaug, T. *Root Cause Analysis*; Quality Press: Seattle, WA, USA, 2006.
17. Prasetyo, Y.T.; Cagubcob, A.M.A.; Persada, S.F.; Redi, A.P. Application of 8D methodology for minimizing test mixing event in semiconductor test manufacturing. In *Proceedings of the 2021 IEEE 8th International Conference on Industrial Engineering and Applications (ICIEA)*; IEEE: New York, NY, USA, 2021; pp. 360–367.

18. Hopkin, P. *Fundamentals of Risk Management: Understanding, Evaluating and Implementing Effective Risk Management*; Kogan Page Publishers: London, UK, 2018.
19. Hutchins, G. *ISO 31000: 2018 Enterprise Risk Management*; Greg Hutchins: Portland, OR, USA, 2018.
20. Barar, P.; McCormick, E. Leveraging the Data Continuum for Advanced Root Cause Analysis. In *Proceedings of the 2024 35th Annual SEMI Advanced Semiconductor Manufacturing Conference (ASMC)*; IEEE: New York, NY, USA, 2024; pp. 1–5.
21. e Oliveira, E.; Miguéis, V.L.; Borges, J.L. Automatic root cause analysis in manufacturing: An overview & conceptualization. *J. Intell. Manuf.* **2023**, *34*, 2061–2078.
22. John, R. Data Extraction vs. Data Collection: A Comprehensive Guide for Professionals. 2024. Available online: https://www.docsumo.com/blogs/data-extraction/vs-data-collection (accessed on 7 October 2024).
23. KlearStack. Data Extraction vs. Data Collection: Understanding the Difference. 2024. Available online: https://medium.com/@klear-stack/data-extraction-vs-data-collection-understanding-the-difference-af72c1a9a034 (accessed on 7 October 2024).
24. Dabbas, R.M.; Chen, H.N. Mining semiconductor manufacturing data for productivity improvement—An integrated relational database approach. *Comput. Ind.* **2001**, *45*, 29–44. [CrossRef]
25. Embley, D.W.; Campbell, D.M.; Jiang, Y.S.; Liddle, S.W.; Lonsdale, D.W.; Ng, Y.K.; Smith, R.D. Conceptual-model-based data extraction from multiple-record web pages. *Data Knowl. Eng.* **1999**, *31*, 227–251. [CrossRef]
26. Webb, J.W.; Webb, J. A method for storing semiconductor test data to simplify data analysis. In *Proceedings of the 2016 IEEE AUTOTESTCON*; IEEE: New York, NY, USA, 2016; pp. 1–10.
27. Silva, V.; Leite, J.; Camata, J.J.; De Oliveira, D.; Coutinho, A.L.; Valduriez, P.; Mattoso, M. Raw data queries during data-intensive parallel workflow execution. *Future Gener. Comput. Syst.* **2017**, *75*, 402–422. [CrossRef]
28. Singrodia, V.; Mitra, A.; Paul, S. A review on web scrapping and its applications. In *Proceedings of the 2019 International Conference on Computer Communication and Informatics (ICCCI)*; IEEE: New York, NY, USA, 2019; pp. 1–6.
29. Glez-Peña, D.; Lourenço, A.; López-Fernández, H.; Reboiro-Jato, M.; Fdez-Riverola, F. Web scraping technologies in an API world. *Brief. Bioinform.* **2014**, *15*, 788–797.
30. Chien, C.F.; Liu, C.W.; Chuang, S.C. Analysing semiconductor manufacturing big data for root cause detection of excursion for yield enhancement. *Int. J. Prod. Res.* **2017**, *55*, 5095–5107. [CrossRef]
31. Bal, H.E.; Steiner, J.G.; Tanenbaum, A.S. Programming languages for distributed computing systems. *ACM Comput. Surv. (CSUR)* **1989**, *21*, 261–322. [CrossRef]
32. Hughes, J. Why functional programming matters. *Comput. J.* **1989**, *32*, 98–107. [CrossRef]
33. Paykin, J.; Krishnaswami, N.R.; Zdancewic, S. The essence of event-driven programming. In *Leibniz, Leibniz International Proceedings in Informatics*; Schloss Dagstuhl–Leibniz-Zentrum für Informatik: Wadern, Germany, 2016.
34. Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.M.; Irwin, J. Aspect-oriented programming. In *Proceedings of the ECOOP'97—Object-Oriented Programming: 11th European Conference, Jyväskylä, Finland, 9–13 June 1997*; Proceedings 11; Springer: Cham, Switzerland; pp. 220–242.
35. Järvinen, H.M.; Kurki-Suonio, R. DisCo specification language: Marriage of actions and objects. In *Proceedings of the ICDCS*; Citeseer: University Park, PA, USA; pp. 142–151.
36. Lamport, L. The temporal logic of actions. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **1994**, *16*, 872–923. [CrossRef]
37. Chandy, K.M. Parallel program design. In *Opportunities and Constraints of Parallel Computing*; Springer: Cham, Switzerland, 1989; pp. 21–24.
38. Järvinen, H.M.; Kurki-Suonio, R. *The DisCo Language and Temporal Logic of Actions*; Tampere University of Technology: Tampere, Finland, 1990.
39. Jarvinen, H.; Kurki-Suonio, R.; Sakkinen, M.; Systa, K. Object-oriented specification of reactive systems. In *Proceedings of the [1990] Proceedings. 12th International Conference on Software Engineering*; IEEE: New York, NY, USA; pp. 63–71.
40. Kurki-Suonio, R. Towards an Action Language. In *A Practical Theory of Reactive Systems: Incremental Modeling of Dynamic Behaviors*; Springer: Berlin, Germany, 2005; pp. 25–56.
41. Ashish, V. Attention is all you need. *Adv. Neural Inf. Process. Syst.* **2017**, *30*.
42. Park, J.S.; O'Brien, J.; Cai, C.J.; Morris, M.R.; Liang, P.; Bernstein, M.S. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*; Association for Computing Machinery: New York, NY, USA, 2023; pp. 1–22.
43. Chen, W.; Su, Y.; Zuo, J.; Yang, C.; Yuan, C.; Qian, C.; Chan, C.M.; Qin, Y.; Lu, Y.; Xie, R.; et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. *arXiv* **2023**, arXiv:2308.10848.
44. Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 1877–1901.
45. White, J. Building Living Software Systems with Generative & Agentic AI. *arXiv* **2024**, arXiv:2408.01768. [CrossRef]
46. Qian, C.; Liu, W.; Liu, H.; Chen, N.; Dang, Y.; Li, J.; Yang, C.; Chen, W.; Su, Y.; Cong, X.; et al. Chatdev: Communicative agents for software development. *arXiv* **2023**, arXiv:2307.07924.

47. Pokkunuri, B.P. Object oriented programming. *ACM Sigplan Not.* **1989**, *24*, 96–101. [CrossRef]
48. Chauhan, S.; Rasheed, Z.; Sami, A.M.; Zhang, Z.; Rasku, J.; Kemell, K.K.; Abrahamsson, P. LLM-Generated Microservice Implementations from RESTful API Definitions. *arXiv* **2025**, arXiv:2502.09766. [CrossRef]
49. Zhang, Y.; Zhang, J.; Chen, Y.; Wang, Q. Resolving synchronization and analyzing based on aspect-oriented programming. In *Proceedings of the 2008 International Symposium on Computer Science and Computational Technology*; IEEE: New York, NY, USA, 2008; Volume 1, pp. 34–37.
50. ELssaedi, M.M. A New Approach to Event-Driven Programming. Ph.D. Thesis, BTU Cottbus-Senftenberg, Senftenberg, Germany, 2008.


**Disclaimer/Publisher's Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

https://doi.org/10.3390/pr14101669