Article

Comparison of Multi-Agent Platform Usability for Industrial-Grade Applications

Zofia Wrona [1,*], Maria Ganzha [1], Marcin Paprzycki [2,*], Wiesław Pawłowski [3], Angelo Ferrando [4], Giacomo Cabri [4] and Costin Bădică [5]

[1] Faculty of Mathematics and Information Science, Warsaw University of Technology, 00-662 Warsaw, Poland; maria.ganzha@pw.edu.pl
[2] Systems Research Institute, Polish Academy of Sciences, 01-447 Warsaw, Poland
[3] Faculty of Mathematics, Physics and Informatics, University of Gdańsk, 80-952 Gdańsk, Poland; wieslaw.pawlowski@ug.edu.pl
[4] Department of Physics, Informatics and Mathematics, University of Modena and Reggio Emilia, 41125 Modena, Italy; angelo.ferrando@unimore.it (A.F.); giacomo.cabri@unimore.it (G.C.)
[5] Department of Computers and Information Technology, University of Craiova, 20776 Craiova, Romania; costin.badica@edu.ucv.ro
* Correspondence: zofia.wrona.dokt@pw.edu.pl (Z.W.); marcin.paprzycki@ibspan.waw.pl (M.P.)

Abstract: Modern systems often employ decentralised and distributed approaches. This can be attributed, among others, to the increasing complexity of system processes, which go beyond the capabilities of singular components. Additionally, with the growth in demand for system automation and high-level coordination, solutions belonging to the decentralised Artificial Intelligence and collaborative decision-making are often applied. It can be observed that these concerns fall within the domain of multi-agent systems. However, even though MAS concepts emerged more than 40 years ago, despite their obvious advantages and continuous efforts of the scientific community, agents remain rarely used in industrial-grade applications. In this context, the goal of this contribution is to analyse the reasons for the lack of adoption of agent solutions in the real world. During the analysis, all pertinent aspects of the modern software development life cycle are examined and compared to what is currently available in the agent system domain. Specifically, the study focuses on identifying gaps that are often overlooked when it comes to scientific applications of MAS, but are critical in terms of potential for large-scale system development in practice.

Keywords: agent systems; industrial-grade systems; agent oriented; software engineering; SDLC

Citation: Wrona, Z.; Ganzha, M.; Paprzycki, M.; Pawłowski, W.; Ferrando, A.; Cabri, G.; Bădică, C. Comparison of Multi-Agent Platform Usability for Industrial-Grade Applications. *Appl. Sci.* **2024**, *14*, 10124. https://doi.org/10.3390/app142210124

Academic Editor: Yutaka Ishibashi

Received: 30 September 2024
Revised: 31 October 2024
Accepted: 1 November 2024
Published: 5 November 2024

Copyright: © 2024 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 (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Software systems are becoming increasingly complex, rendering centralised approaches less feasible. This complexity arises from technological advancements and the dynamic nature of modern system environments. As a result, individual components, often limited in computational capabilities, are unable to efficiently manage entire processing tasks alone (e.g., in edge computing scenarios). Additionally, the competitive software market imposes stringent Quality of Service (QoS) requirements, particularly regarding fault tolerance, security, and responsiveness. These demands are frequently formalised in legally binding Service Level Agreements (SLAs), which can result in significant penalties whenever being violated.

Given the widespread adoption of service provision over the Internet, such as in Cloud computing and, more recently, Cloud-Edge-Continuum (CEC) ecosystems, there is a critical need for applications that remain highly resilient at a large scale. Furthermore, data protection regulations (e.g., GDPR [1]) and/or restrictions on data sharing often preclude the centralisation of all necessary data storage and processing.

These factors underscore the need for decentralised system operation approaches [2]. Within this domain, Distributed Artificial Intelligence (DAI) has gained particular attention [3], addressing challenges related to collaborative decision-making and collective learning. The increasing complexity of modern systems necessitates automation across multiple levels of autonomy, which can significantly reduce the need for costly and error-prone manual operations.

One subdomain of DAI is *agent systems*, first introduced at the Workshop on Distributed Artificial Intelligence at MIT in June 1980 [4]. Although a single definition of an agent has not been universally accepted, several common characteristics have emerged [5], including autonomy, reactivity, social capabilities, proactiveness, goal orientation, and responsiveness. These attributes remain highly relevant in contemporary systems. Over the past four decades, significant contributions have been made toward conceptualising agent systems from both theoretical and practical perspectives [6]. However, despite the alignment of modern system requirements with what Multi-Agent Systems (MASs) promised to deliver, their adoption in the production environments remains limited. Developers and software architects frequently opt for alternative solutions, such as event-driven architectures, service-oriented architectures, microservices, or serverless computing. Although developers may unknowingly implement approaches similar to agent paradigms, core agent-specific terminology and tools are rarely employed.

The contrast between agent systems and industrial applications has been recognised, as evidenced by the multistage AgentLink project [7–9], funded by the European Commission under FP6. During the project's third stage (AgentLink III), a strategic roadmap was developed to analyse the future maturity prospects of MAS [10]. At the time, it was estimated that agent systems would require at least 20 years to achieve industrial maturity, based on the historical evolution of other programming approaches like Object-Oriented Programming (OOP) [11]. It was assumed that progress in MAS would mirror the general trajectory of Software Engineering (SE) [12], where it typically takes 10–20 years for research to transition into widespread practice. Nevertheless, these predictions have not materialised, as there has been little change in the industrial acceptance of MAS since 2005. This raises important questions about the factors preventing MAS from being widely adopted.

A critical argument in this context is the *divergence* between the development trajectories of agent systems and industrial-grade applications. Real-world industrial applications must deliver tangible business value and adhere to stringent robustness requirements. To meet client and market demands, continuous communication with customers is essential, driving the popularity of SE methodologies such as agile development. These methodologies require software that is easily modifiable to accommodate evolving client needs, which, in turn, emphasises modular design, application testing, Continuous Integration and Delivery (CI/CD), and application monitoring.

In contrast, software developed within the scope of scientific research often does not need to address such concerns. Typically, research-oriented software achieves a Technology Readiness Level (TRL) of 4–5 (and sometimes TRL 6), whereas enterprise software must reach TRL 8–9. From a research perspective, achieving TRL 8–9 (even TRL 7) is not required for conducting exploratory experiments, sharing work with other researchers, or publishing papers. Consequently, as will be shown in what follows, the technologies used in MAS development often do not meet the standards of modern Software Development Life Cycle (SDLC) approaches. This, in turn, creates a cyclical issue, where the lack of industrial-grade applications diminishes the need for further advancements in agent-based software. Hence, it can be argued that this is one of the fundamental reasons for the limited adoption of Agent-Oriented Software Engineering (AOSE) by enterprise developers.

To bridge this gap and enable companies to fully leverage agent-related paradigms, it is essential to identify and address the technological limitations of AOSE, which is the primary focus of this study. Such analysis can benefit the scientific community by pinpointing areas in need of refinement within the existing (and possibly future) agent platforms. This type of analysis has to incorporate practical experiences and expertise with

modern production systems, especially given the rapid pace of technological development, which can render tools and standards formulated 5–10 years ago obsolete.

Consequently, the contribution of this work is to contrast relevant areas of production software development with the current state of agent systems. It categorises various aspects of AOSE, from those adequately covered by the existing tools to those requiring significant attention in the future. The goal is to outline future directions for the practical application of MAS. To the best of the authors' knowledge, no previous analysis has focused specifically on the technical limitations of MAS tools from an industrial-grade perspective.

The remainder of this work is structured as follows: Section 2 reviews the state-of-the-art, focusing on industrial-grade applications of AOSE. Section 3 provides an overview of the most popular MAS development frameworks. Section 4 discusses modern SDLC standards, including technological and organisational aspects. The core contribution, detailed in Section 5, examines the technological state of AOSE across various (industrial) software project development areas. Section 6 summarises the findings, categorising areas in need of further refinement. Finally, Section 7 presents the main conclusions and suggests potential future directions for MAS applications.

2. Related Works

Although, thus far, AOSE has *not* been widely adopted in industrial-grade projects, some notable examples can be traced back to the early 2000s [13]. For instance, in the military domain, MAS-based solutions were employed in the simulation and modelling of military training. A prominent example is the Human Variability in Computer Generated Forces (HV-CGF) project [14], funded by the UK Ministry of Defence. This project utilised the JACK agent platform [15] to simulate changes in military personnel behaviour, with JACK agents modelling combatants' beliefs, desires, and intentions. The agents' behaviour was analysed in relation to various drug moderators, such as caffeine. Today, Australian company AOS, the creators of JACK, remains one of the few that openly offers commercial MAS-based solutions. It should be noted that although other companies also continue to offer agent-based solutions, they do not state this information explicitly.

In a different context, SCA Packaging [16] developed an agent-based simulation model where agents represented entities such as plant managers and customers, allowing the study of interactions and relationships between order patterns and plant resource capabilities. In the transportation sector, Magenta Technology introduced the agent-based i-Scheduler [17] for vessel scheduling tasks, employing a formal knowledge model for ship scheduling based on Magenta's proprietary multi-agent technology. Additionally, Whitestein Technologies developed the Living Systems Adaptive Transportation Network (LS/ATN) [18], a system designed to optimise costs in transport logistics. In this system, agents represented vehicle drivers, who provided the system with their planned routes, enabling the system to assess the feasibility of additional loads. Although initially marketed with a focus on agent-related features, over time, Whitestein's products have gradually moved away from emphasising these aspects.

Energy-related simulations have also benefited from MAS applications. Notably, NuTech Solutions developed the Supply Chain Production Optimizer [19], a system that tailored oil production processes according to factors such as client demand, energy pricing, and weather conditions. In the financial sector, Acklin B.V. introduced the KIR system [20], which automated the exchange of insurance information through agent communication. In all these cases, proprietary agent software was utilised.

Among more recent industrial-grade MAS applications, projects by the company Actoron [21] stand out. Actoron, the primary developer of the Jadex agent platform [22], delivered a commercial agent-based control system for a German waste incineration plant in 2017. They also developed a Belief–Desire–Intention (BDI)-based customer-centred digital assistance system for The Allstate Corporation. In 2019, Actoron collaborated with Huawei to create an agent-based scheduling solution designed to mitigate disruptions in production chains.

In contemporary MAS applications, the Simudyne platform [23] is employed by Barclays for modelling risk assessment and management [24]. Similarly, AnyLogic's solutions [25] are used by companies such as Walmart for simulations and modelling, particularly in verifying automated tools that support online warehouse operations.

Despite these examples, the overall industrial interest in AOSE solutions has declined. A much greater number of applications for agent-based systems have been found in the scientific domain, where agent-based solutions have been employed to study social interactions [26,27], logistics [28], distributed resource management [29], or planning problems [30]. A number of these practical applications have been presented at conferences, including the International Conference on Autonomous Agents and Multiagent Systems (AAMASs) [31] and the International Workshop on Engineering Multi-Agent Systems (EMASs) [32]. However, these conferences have not upheld the forefront emphasis on AOSE. In the latest proceedings of AAMAS [33], agents were introduced mostly as a tool stimulating research into other scientific domains. On the other hand, although the proceedings of EMAS [34] centre around agent-based systems, software engineering aspects are no longer explicitly addressed. It can be argued that rather than being used as an important component for building contemporary software systems, agents have evolved into a tool to study scientific problems from various domains, which may also be one of the reasons for the lack of AOSE's industrial acceptance.

In this context, some studies have attempted to analyse the progress and barriers to the popularisation of MAS. For instance, a survey [35] examined AOSE-based systems deployed across various industries, summarising data from 152 applications. However, only 89 applications were confirmed as active, suggesting that many developed in the 2000s are no longer maintained. The survey also revealed that 69% of these applications were developed in collaboration with academic institutions, and only 31% were implemented solely by industrial or government entities. Additionally, 61% of the applications were initiated by academia, highlighting a lack of interest, awareness, and/or trust in MAS solutions among the enterprises. The survey further noted that of the 46 applications that reached TRL 8–9, 30 were implemented in collaboration with industry partners. Interestingly, (1) over 45% of these applications utilised Java-based agent platforms, and (2) the majority of them were built on existing AOSE tools, underscoring their importance. However, this survey was conducted nearly a decade ago, making its relevance today questionable. Moreover, it focused on the distribution of MAS applications in the market without delving into specific factors influencing presented outcomes.

In another study [36], key factors influencing the industrial acceptance of MAS were analysed. The authors categorised these factors into design, technology, algorithms, standardisation, application, cost, and the ability to address field-specific challenges. Despite including input from various industrial experts, most respondents were from academic backgrounds, with 80% having at least four years of experience with agent systems. This reliance on academic respondents may have skewed the results, as they were already inclined to believe in the viability of AOSE for system development. This could explain why 68% of respondents agreed that industrial agents were sufficiently mature for production use. The study, however, lacked input from non-agent-related enterprise companies, limiting its scope. Additionally, while it identified general areas for improvement, it did not specify which aspects of AOSE were mature and which required further development. It is also worth noting that this study was conducted in 2015, potentially rendering some of its conclusions outdated.

A more recent analysis of the barriers to MAS implementation and adoption was summarised in [37]. Unlike previous surveys, the study focused on specific challenges limiting the practical use of MAS, such as a lack of general understanding of MAS concepts, insufficient standardisation and specification, limited flexibility of MAS development technologies, and difficulties in benchmarking and validation. The study also outlined prospects for MAS development in industries such as aerospace, smart grids, mobility, and healthcare. Among all reviewed works, this study provided the most comprehensive

commercial perspective, though it did not delve into technological limitations. Notably, it highlighted the persistence of benchmarking and validation issues, echoing concerns raised by earlier works [38]. Although there have been efforts to introduce benchmarking in agent systems [39], these initiatives have not been followed by further discussion and development(s).

As these publications demonstrate, few have evaluated the status of AOSE in commercial use, and most focus on general concerns rather than addressing specific technological details. This lack of detailed analysis is particularly evident in discussions surrounding MAS's technological infrastructure. Research papers describing current MAS tools [6,40] often rely on white papers and documentation that do not address the practical aspects. Furthermore, the continuous development of new agent platforms, such as Agent Assembly [41] and PEAK [42], which are built on top of existing platforms to overcome their limitations (e.g., scalability), indicates an ongoing lack of appropriate solutions to meet industrial needs.

To address this gap, this contribution aims at providing a new, practical perspective on MAS technologies, by emphasising the standards of modern SDLC that should be incorporated into AOSE. The claims presented here are further supported by the authors' expertise in SE and their experience with commercial systems.

To systematise the study, the following section will briefly outline the main features of selected agent platforms, which will serve as the foundation for further discussion.

3. Selected Sample Agent Platforms

The range of available agent platforms spans numerous areas, from general-purpose solutions to those tailored for specific applications. Recent reviews, covering the majority of agent platforms available as of 2023, are presented in [40,43]. Specifically, the study in [40] identified a total of 112 active platforms. For the purposes of this work, the scope will be narrowed to focus on three selected general-purpose platforms that will serve as the main representative group, although references to other platforms may occasionally appear. The three platforms chosen for this work are JADE [44] (version 4.6.0), SPADE [45] (version 3.3.2), and Jadex [46] (version 5.0.0). In particular, these platforms were selected based on three criteria. The criteria originated from the practical, industrial experience of the authors and were formulated to narrow the scope of available agent platforms to those that, in their current technical state and considering their potential functional/non-functional requirements, appear to be well suited to the needs of the industry. Consequently, it enabled the analysis of the quality of these platforms from the perspective of other existing SE tools that have achieved widespread popularity and may serve as a suitable reference point for industrial acceptance. This process supported the identification of agent platforms' shortcomings that potentially hinder the adoption of MAS tools. However, in order to conduct such an evaluation, especially in terms of platform quality, it was necessary to have access to the platforms' source code, which is a rationale for analyzing only open-source platforms. Detailed description of used criteria is as follows:

1. *Source Availability*: Only open-source platforms were considered. This criterion was crucial because it allowed for the evaluation of key factors such as (1) the quality of the source code, (2) the project's maintenance status, and (3) the modularity and interoperability of the platform's architecture. In this context, all of the selected platforms are fully open-source.
2. *Platform Maturity and Market Presence*: The platforms were chosen based on their maturity and the number of use cases within the community. Maturity and a strong presence in the market are significant because the adoption of a tool in industrial applications often depends on the developer's trust in the framework. A platform with a longer maintenance history and a larger user community is more likely to be adopted due to its (1) extensive testing coverage (including user acceptance testing), which enhances reliability, (2) likelihood of continued stability and future maintenance, and (3) availability of resources to support developers. Among the platforms mentioned

in [40], JADE, SPADE, and Jadex are the ones that appear most frequently when searching Google Scholar with the keywords “agents” or “multi-agent systems”. In particular, since 2020, JADE has been featured in 3270 works, SPADE in 572, and Jadex in 347. There were also a substantial number of references reported for JaCaMo [47] (502) and Janus [48] (707); however, these platforms were excluded from the considerations due to their utilisation of Agent-Oriented Programming Languages (AOPLs) rather than common SE programming languages. Furthermore, when considering maturity, it should be noted that the selection of Jadex was further strengthened by the fact that it is supported by the company Actoron and is already being used in commercial projects. In consequence, its technical requirements are perhaps the closest to reflecting the needs of the industry, in comparison to other open-source platforms.

3. *Programming Language*: Trends in programming languages within SE and AOSE were also considered. According to the analysis in [35], Java was the most widely adopted programming language for MAS development in 2017. This trend remains consistent in 2023–2024, as reflected in the platforms listed in [40], where approximately 33% are Java-based. In second place is Python, with 25% of the platforms using it. This aligns with the findings of Statista (https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/, Accessed on 29 September 2024), which reported that in 2023, Python (about 49%) and Java (about 30%) were the most commonly used programming languages, excluding web targeted languages. Similar results have also been outlined in the yearly “Top Programming Languages” ranking released by IEEE Spectrum (https://spectrum.ieee.org/top-programming-languages-2024, Accessed on 29 September 2024). Therefore, platforms supporting either of these two languages were prioritised. It should also be pointed out that, as previously mentioned, all AOPL-based platforms were excluded from the considerations due to their limited acceptance. The differences between AOPL and OOP were studied among others in [49], where the authors have attempted to determine which aspects of AOPL can be mapped and transformed, making it more akin to OOP. Presented efforts materialised in the creation of platforms such as ASTRA [50]. Nevertheless, although they have been on the market for a similar period of time to other emerging programming languages (e.g., Rust), they have not been widely adopted.

Next, let us outline the general characteristics of each of the three selected platforms. Specific software aspects of these platforms will be further discussed in Section 5.

3.1. JADE Agent Platform

JADE is arguably the most recognised agent platform implemented in Java, known for its strong adherence to standards. It is fully aligned with the FIPA specification [51] and supports distributed agent systems by running agents in containers instantiated on different hosts. JADE agents are mobile, meaning they can move between containers; however, this feature has recently lost some of its appeal. Agent lifecycle management is handled by an Agent Management System (AMS) agent. Additionally, following FIPA standards, JADE includes a built-in Directory Facilitator (DF) agent, which serves as a yellow pages service, allowing agents to register and discover services offered by other agents.

To facilitate testing and debugging, JADE provides a Graphical User Interface (GUI), which displays key information about currently running agents and messages exchanged between them. Furthermore, to ensure secure inter-platform communication, JADE supports plugins such as (1) JADE Security, based on the Java Security Model, which includes authentication for system components (agents and containers), and (2) JADE Public Key Infrastructure (JADE-PKI), which allows for the encryption of agent communication channels. As of late 2024, JADE continues to be actively developed and maintained.

One noticeable limitation of JADE is the lack of built-in support for agent knowledge representation and reasoning methods. To address this, several extended JADE-based platforms have been developed, including Jadex.

3.2. Jadex Agent Platform

Jadex started as an extension of JADE that introduces built-in methods for supporting agent reasoning. The platform is conceptualised around active components (ACs) and service component architecture (SCA). These components can function as passive services or autonomously execute behaviours. Jadex supports BDI [52] reasoning model and also provides workflows for business process modelling. Like JADE, agent systems in Jadex can be distributed across multiple hosts.

New nodes are discovered through dynamic service searching and binding, with Jadex providing automatic awareness mechanisms. In terms of security, the platform supports shared secrets and offers a dedicated Security Tool plug-in for password management. Jadex also features the Jadex Control Center (JCC), a GUI for configuring various aspects of the platform's runtime. The latest version of Jadex, Jadex-V, was released at the beginning of 2024, and the platform continues to undergo incremental development.

3.3. SPADE Agent Platform

In contrast to the Java-based JADE and Jadex, SPADE is implemented in Python and uses XMPP/Jabber as its foundation. SPADE utilises communication channels with message templates for agent communication. While SPADE offers an XML-based communication protocol that supports FIPA-ACL metadata, the platform is not fully FIPA compliant, meaning that communication with other agent platforms may require significant adaptation. SPADE agents are represented as users with unique Jabber identifiers, and the agent platforms (understood as agent's environments, defined by FIPA [51]) correspond to XMPP servers. It is important to note that mapping agent platforms to XMPP should be seen as a conceptual simplification, as SPADE does not adhere to the FIPA-based concept of "platforms".

Currently, SPADE does not offer built-in mobility features. Communication security is handled using XMPP mechanisms, such as certificates and encryption of client-to-server and server-to-server communications. New agent discovery is also supported within the XMPP protocol. SPADE provides a web-based interface for graphical control over the agent system, allowing for insights into agent operations. Although SPADE remains supported and developed, its updates are irregular due to the limited development resources, such as the number of individuals involved in its implementation. This results in SPADE having a narrower feature set compared to JADE or Jadex. Nevertheless, it remains the most widely used general-purpose Python-based platform.

To conclude this section, it should be noted that despite all three platforms remaining under maintenance, they were originally developed several years ago: JADE in 1999, Jadex in 2003, and SPADE in 2006 (with SPADE being re-implemented in 2018 to version 3). Since their inception, approaches to software design and maintenance, including the complete SDLC, have evolved significantly. These changes should have necessitated revisions and adaptations of these platforms to meet modern development standards, which will be outlined in the next section.

4. Standards of Modern Software Development

While the approaches to modern software development vary based on, among others, (1) the specificity of the project and (2) the individual structure and practices of the particular company, there are some commonalities that are integral parts of industrial SE. Typically, development teams follow some form of SDLC, which is a multi-stage process used to streamline the management and control of the project, minimising its potential risks. The most typical key stages of this process are depicted in Figure 1.

The first stage of the SDLC is **Planning**. Here, the project's general objectives, purposes, and outcomes are defined. Typically, this stage involves stakeholders discussing the product and eliciting business requirements. During the **Planning** stage, team leaders estimate the costs and resources needed for development, as well as specify priorities, milestones, and the overall work structure. It also encompasses market research, including benefit

estimation. In most organisations, to streamline **Planning**, dedicated tools for (1) work organisation (e.g., Jira (https://www.atlassian.com/software/jira, Accessed on 29 September 2024), Trello (https://trello.com, Accessed on 29 September 2024)), (2) requirement management (e.g., Confluence (https://www.atlassian.com/software/confluence, Accessed on 29 September 2024), Miro (https://miro.com/, Accessed on 29 September 2024)), and (3) collaborative scheduling (e.g., Google Workspace (https://workspace.google.com, Accessed on 29 September 2024), Smartsheet (https://it.smartsheet.com, Accessed on 29 September 2024)) are used. Moreover, requirements are often illustrated using Unified Modelling Language (UML), such as use case diagrams or user stories. Modern software engineering rarely follows a monolithic structure. Products are often divided into smaller components, each with its own component owner. Therefore, the project's work should be divisible into individual processes that can be carried out concurrently. As a result, projects are usually undertaken simultaneously by multiple teams to effectively leverage their individual expertise.

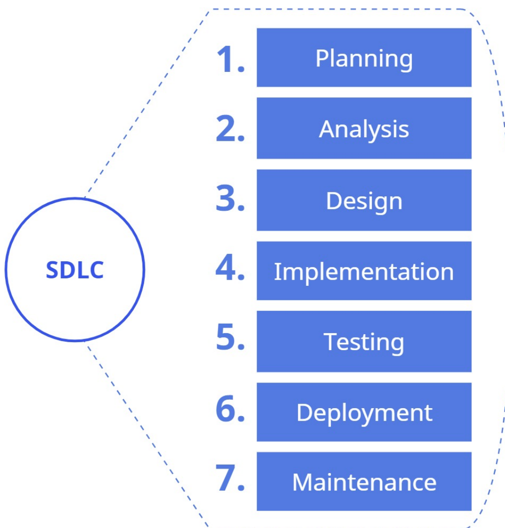


Figure 1. Stages of software development life cycle.

The next stage is the **Analysis** stage, which includes the feasibility study, where software requirements are specified. Developers identify all technological aspects (e.g., hardware or software) needed to build the system. This is the stage where alternative solutions are evaluated (e.g., considering the potential use of AOSE). In terms of software selection, as mentioned in Section 3, aspects such as the maturity of tools, popularisation, licenses, and applicability are often considered. The selected frameworks should also be interoperable and compatible with one another. Moreover, developers often emphasise the importance of documentation. Software that is well-documented and contains illustrative examples is more likely to be chosen.

Once the requirements are identified, the **Design** stage begins. This stage focuses on creating a blueprint for the software solution, involving the definition of the architecture, data models, user interface design, and overall system structure. The goal of the **Design** process is to ensure that the software is scalable, maintainable, and meets the client's requirements. As mentioned in the **Planning** stage, systems are often divided into individual components, meaning that, to ensure proper integration, the system architecture must account for the fact that each component is likely to be deployed separately. The envisioned design is often documented in the Design Document Specification (DDS), where the system structure is represented using UML diagrams, such as class diagrams, component diagrams, and package diagrams. These can be created using dedicated tools, such as Miro (https://miro.com/, Accessed on 29 September 2024) or Lucidchart (https://www.lucidchart.com/pages/, Accessed on 29 September 2024).

The **Implementation** stage follows, which is the longest stage and focuses on the development of the actual product. Before implementing individual features, all necessary environments must be initiated. It should be noted that contrary to scientific projects, industrial-grade systems are often integrated with a larger number of external services, supporting their long-term CI/CD. Specifically, for each system component, (1) the repository in the internal development environment (e.g., GitLab (https://www.gitlab.com, Accessed on 29 September 2024)), (2) the documentation page(s) in the content management workspace (e.g., Confluence (https://www.atlassian.com/software/confluence, Accessed on 29 September 2024)), and (3) project configuration in the automation servers (e.g., Jenkins (https://www.jenkins.io, Accessed on 29 September 2024)) are created. Additionally, most companies have their own internal, technology-dependent codebase configuration setup, which allows for the maintenance of standardisation and consistency. All of these aspects need to be pre-configured during the **Implementation** stage and integrated into the actual codebase using appropriate package management tools (e.g., Gradle (https://gradle.org, Accessed on 29 September 2024), NPM (https://www.npmjs.com, Accessed on 29 September 2024), and PIP (https://pypi.org/project/pip/, Accessed on 29 September 2024)). Sometimes, components may need to be integrated with other technologies (e.g., databases such as Oracle or PostgreSQL, or data streams implemented using Kafka (https://kafka.apache.org, Accessed on 29 September 2024)) or with other internal components. The development of the system is often facilitated by dedicated Integrated Development Environments (IDEs), such as IntelliJ (https://www.jetbrains.com/idea/, Accessed on 29 September 2024) or Visual Studio Code (https://visualstudio.microsoft.com/, Accessed on 29 September 2024). Modern approaches to code development prioritise collaboration and early-stage application testing, influenced by the widespread adoption of agile methodologies [53]. Techniques such as pair programming [54,55] or Test-Driven Development (TDD) [56,57] may be particularly favoured.

Next is the **Testing** stage, which aims to verify whether the software functions according to both functional and non-functional requirements. As mentioned earlier, following TDD, some tests (e.g., unit tests or integration tests) can be written by developers alongside the implementation of features. On the other hand, acceptance and automated tests are often conducted by dedicated Quality Assurance (QA) testers. These tests are performed using dedicated technologies, such as Selenium for browser-based tests, JUnit (https://junit.org/, Accessed on 29 September 2024) in Java, or unittest (https://docs.python.org/3/library/unittest.html, Accessed on 29 September 2024) in Python. It is important to note that several libraries, including those mentioned above, form part of the standard technological stack used by multiple companies. This means that the frameworks selected for software development must be compatible with these tools. Additionally, testing involves test coverage assessment, where companies specify a baseline that must be reached. Coverage is measured by frameworks like JaCoCo (https://www.jacoco.org/jacoco/, Accessed on 29 September 2024), which are often integrated with automation tools (e.g., Jenkins).

After testing, the SDLC proceeds to the **Deployment** stage. It is important to note that major organisations often operate in multiple environments, dedicated separately to development (*Dev*), testing (*QA*), and production (*Prod*) use. This is yet another significant difference from the typical implementation of scientific software. During development, the latest versions of components are available in *Dev*. Pre-release testing is often conducted in the *QA* environment, while products released to end-users are deployed to *Prod*. The purpose of this hierarchy is to prevent disruption of live products and to facilitate collaboration among multiple teams. Therefore, the deployment process may vary depending on the selected technologies (e.g., ArgoCD (https://argo-cd.readthedocs.io/en/stable/, Accessed on 29 September 2024), Puppet (https://www.puppet.com, Accessed on 29 September 2024)), and the targeted environment. Automation plays an essential role in this process, particularly in distributed systems, where pre-deployment configurations need to be distributed across multiple components.

The final stage, but not the least important, is **Maintenance**. To maintain product continuity and align it with changing market demands, developers are responsible for continuously monitoring the software (e.g., using tools such as Icinga (https://icinga.com, Accessed on 29 September 2024), Grafana (https://grafana.com, Accessed on 29 September 2024), or Prometheus (https://prometheus.io/docs/introduction/overview/, Accessed on 29 September 2024)) and providing necessary regular updates. This marks another contrast between research and industrial SDLC. In most research projects, after final software delivery, little attention is given to long-term maintenance (e.g., over a 10+ year perspective). In industrial-grade applications, such a short-term perspective is not feasible since end products deliver business value to the organisation.

By analysing the standards of the modern software development life cycle, one can easily observe the differences between approaches used in research communities and enterprises. These differences impact both (1) the requirements formulated for the software and (2) the specificity of delivered products. Therefore, in some cases, frameworks provided by researchers may not (need to) meet the needs of industrial-grade applications, which (as we will argue in what follows) is precisely what has occurred with platforms for MAS development. In this context, the following section describes the specific areas that influence the technological acceptance of AOSE.

5. AOSE in Contrast to Modern SDLC

In order to emphasise the critical differences between the current state of AOSE and the standards of modern SDLC, the discussion is structured as follows. For each selected area, a general description of the problem is provided, along with a summary of the state-of-the-art within the corresponding agent platform's domain. Afterwards, the condition of each of the representative platforms (i.e., JADE, Jadex, and SPADE) is assessed and summarised based on the practical experience with the platform usage, as well as the automated evaluations of the platform's code quality. It should be noted that the focus of this study is framed around the practical impact of different phases of SDLC, described in Section 4. With this in mind, let us begin the discussion with the possibilities of integrating agent platforms into modern software systems.

5.1. Package Management Tools

Typically, companies have a standard base technological stack that is widely known among developers and used across most of the components within the organisation. Package management tools are the most efficient method of handling versioning and injection of external and internal modules. As mentioned in Section 4, some companies have their own internal components for maintaining libraries' versioning. With such an approach, the dependency management is simplified, and potential security vulnerabilities are addressed globally rather than in separate instances. Moreover, package management tools allow for connecting internal components with one another.

One deprecated and no longer used practice, which still appears quite often in legacy systems, is the manual injection of the source code (e.g., in the case of Java–JARs). This approach has several disadvantages, such as (1) a high possibility of inconsistencies and conflicts in dependencies, (2) time-consuming version control, and (3) lack of support for modularity and reusability. For industrial-grade products with long-term perspectives, this approach is highly inefficient and thus inappropriate for robust software maintenance.

In this context, the majority of agent platforms have still not been deployed into global repositories (e.g., Maven Central). Instead, they rely on manual injection or have been deployed in ways that make their management challenging. For instance, a legacy platform, WADE [58] (a supporting tool for JADE), is only injectable by downloading a specific JAR. Additionally, it requires integrating external Eclipse plugins to run the GUI. Due to this, it exposes itself to a high risk of conflicting versions, especially since it relies on older distributions of libraries. In terms of newer platforms, the same applies to Jadescript [59], CellNet [60], or MASON [61]. In each case, a JAR has to be manually added. This problem

seems to be less severe for platforms based on Python (e.g., piaf [62]), JavaScript (e.g., JS-son [63]), or some AOPL-based platforms (e.g., ASTRA [50] and SARL [48]). These are deployed using PyPi, NPM, or Maven Central.

The compatibility of the representative platforms with package management tools is summarised in Table 1.

The JADE agent platform, after the release of version 4.6 in 2022, announced its publication to Maven Central, and its support for injection through common Java-based package management tools. However, the latest version currently available through Maven is JADE 4.5, while JADE 4.6 can only be obtained in the form of a JAR. Moreover, even though JADE is an open-source platform, there is no official JADE repository, and the source code can only be downloaded from the website. Neither of these aspects allows the community to monitor the progress made in framework implementation. Due to this, JADE is currently inapplicable to typical production systems.

Table 1. Compatibility with package management in JADE, Jadex, and SPADE.

Platform Name	Support for Package Management
JADE	Claimed support for Maven Central, but the latest version has not been released.
Jadex	Supports Maven Central but releases the framework in multiple packages.
SPADE	Supports Python's pip.

The situation is significantly better for Jadex, which has both (1) its source available on GitHub and (2) artifacts published to Maven Central. Therefore, in terms of package management, it more closely meets the standards of modern industrial-grade applications. However, Jadex artifacts are published at a very fine level of granularity. Currently, there are 144 artifacts published under the Jadex group, whereas according to the description in the Jadex-V repository, 14 of them need to be added for a minimal project configuration. Consequently, handling each version separately would be extremely inefficient in the production setup, creating concerns regarding Jadex's enterprise usage.

Lastly, SPADE fully meets the standards of modern package management. Its latest version is available on PyPi and can be installed using pip (a widely known package management tool for Python). Furthermore, SPADE has its sources available on GitHub and does not require manual installation of additional dependencies.

Upon analysing the package management support, it becomes evident that the two Java-based platforms are less frequently included in the global repositories than Python or JavaScript. Additionally, the trend of automating package management is progressing at a much slower pace in the AOSE domain than in the industrial SE, where it was fully adopted years ago. Although the situation has recently started to change with respect to the agent platforms, the use of modern techniques for code and dependency management must become the standard for them to gain acceptance in the production environment(s). This is imperative, especially since package management plays a critical role in facilitating the integration of external services, another area that has been summarised in the next section.

5.2. Possibility of Integration with External Services

As mentioned in Section 4, enterprise applications often require integration with a range of external services. In particular, due to the automation needs, internal practices of companies or even multi-environment deployment production systems often require the provision and maintenance of alternate configurations. For instance, this may include separate configurations for (1) multiple internal databases, (2) selected message broker topics, (3) microservice-based architecture dependencies (which may need to be combined with AOSE), (4) CI/CD automation scripts, (5) deployment rules for different environments, (6) access and privacy control, or (7) scopes allowing the use of individual internal components. These are typically defined in property files and then automatically injected by reference, using constructs specific to the framework (e.g., dedicated annotations).

In single-platform AOSE-based systems, where all agents have the same rights, there should be no difficulties in handling integration aspects since the initial configuration does not differ significantly from that in typical industrial SE systems. However, it becomes much more problematic when considering multi-platform systems, or those in which selected agents (e.g., of the same type) may need different rights. In such cases, the configuration must be propagated to individual agents or their federations. It may be compared to a scenario where a component having a single deployment process needs to handle multiple configurations at once. In many platforms, this issue has not been automated and requires significant manual effort. This may be acceptable for a single-time deployment, but since modern SE often relies on iterated or agile SDLC, which is based on frequent releases in multiple environments, such a procedure would be both time-consuming and costly. Therefore, no company is likely to choose to support such a solution in production use.

To gain a more detailed understanding, Table 2 summarises how this issue is addressed in JADE, Jadex, and SPADE.

Table 2. Possibility of integration with external services in JADE, Jadex, and SPADE.

Platform Name	External Services Integrability
JADE	No automated support for general configuration propagation. Limited support for additional configuration through add-ons.
Jadex	No automated support for general configuration propagation. Built-in limited support for security measures and Web Service integration.
SPADE	No automated support for general configuration propagation. Possibility of individualised configuration of XMPP protocol.

In the case of JADE, selected technologies and frameworks can be integrated through supportive add-ons. For instance, the Kafka Messaging add-on allows for the replacement of native JADE messaging with the one based on a Kafka cluster, while the Web Service Integration Gateway (WSIG) add-on enables the exposure of agent services as Web Services. Nevertheless, all these add-ons have a limited scope of possibilities and would still require significant development effort to fully support a given technology. Moreover, several of them were developed in the past and are not continuously maintained. For example, WSIG's last release was in 2017, while Kafka Messaging was published along with the latest JADE version in 2022. As such, some of them are no longer compatible with the latest JVM versions (e.g., IPMS, which supports inter-platform communication, published in 2017, is no longer compatible with Java 17). Furthermore, none of these solutions would allow automated configuration or propagation of company-specific components.

In the case of Jadex, the situation is similar, except the platform does not rely on add-ons but rather has limited built-in functionality to integrate selected technologies. For instance, it allows configuring security measures for agent communication or integrating WSDL-based and RESTful-based web services. It should be noted, however, that these built-in capabilities rely on either the Jadex syntax or specific external libraries, which may be challenging for developers to adapt to. For example, Jadex relies on web services based on Jakarta EE, which would require a company that exclusively uses Spring to switch technology in some of its components. Similarly, the security measures are not flexible enough to accept custom access and privacy restrictions, which may be imposed by the internal architecture of a given system.

Finally, SPADE encounters the same problems. The automated configuration is restricted to what is available with the XMPP protocol, which is used to establish the agent's operational environment and communication. However, it does not provide any other out-of-the-box automation for specific technologies or services in individual agents. Additionally, since SPADE is based on the XMPP communication, extra effort may be needed to integrate, for instance, microservice REST APIs. In particular, a dedicated middleware layer, acting as a bridge between the two communication protocols, would need to be

implemented. While there have been some ideas for SPADE to support a more general message broker in the past, no further official information has been provided on this matter.

In summary, any corporate production system should be able to integrate different technologies and components in a flexible manner. As a result, the agent platform that aims to be used in industrial-grade applications must have a modular design to provide sufficient capability for this automation. According to what was outlined, the major platforms have not taken this into account. To provide such flexibility, however, AOSE tools would need to adapt to modern code standards first. In particular, they need to align their interfaces and methods with what developers encounter daily. Let us now discuss the current state of code quality in different MAS frameworks.

5.3. Adherence to Modern Code Standards

Over the years, code standards have evolved in SE. In this context, the book [64] is probably the most well-known work describing best code practices. Among other things, it outlines (1) how methods, interfaces, and tests should be structured, (2) how the project should be organised, and (3) what the most effective naming conventions are. Modern code standards emphasise aspects such as rich, yet not overly complicated interfaces, modularity of software constructs, and the selection of appropriate libraries. Additionally, the code standards can also depend on the technology and programming language of choice. For instance, in Java, built-in streams with declarative means of collection processing are more favoured than standard loops.

As most AOSE frameworks were developed several years ago, they tend to follow practices that are usually found in legacy code. Within this domain, the lack of intermediate interfaces, which abstract the complexity of built-in methods, is the most severe problem. In particular, available agent platforms tend to directly use fairly complicated agent-related concepts that require substantive knowledge of the agent systems domain. These concepts are common in agent-related communities (e.g., terminology of BDI); however, they do not appear in general SE. Therefore, developers who are not familiar with the theory of agent systems may find them difficult to use. Such frameworks require gaining the necessary know-how, which additionally increases the costs of overall development. This problem could be solved by providing additional layers of abstraction that would hide the complexities associated with agents from the end user. For instance, this has been done in Akka [65], which may be one of the factors contributing to its industrial acceptance. Nevertheless, in most agent platforms, such practices have not been applied.

Moreover, many agent platforms have not paid enough attention to smaller yet essential code details (e.g., Jadex). Their official codebases contain "workarounds" or features that are internally marked as unfinished. Additionally, several methods have also been named ambiguously, which may lead to misinterpretation of their implementation specifics. When using such code, developers may find themselves in situations where certain functionalities do not work as intended. Furthermore, if one were to examine the source code of an application, it would be extremely difficult to identify errors.

Specific examples have been determined based on JADE, Jadex, and SPADE platforms, as summarised in Table 3.

Among all agent platforms, JADE probably has the widest built-in support for FIPA-specific concepts (see Section 3). As a result, developers working with JADE can take advantage of a rich set of predefined functionalities without having to implement them on their own. However, all of these use strictly agent-specific terminology, which may not be familiar to most industrial programmers. Moreover, even though the design of JADE is sufficiently extendable (as evidenced by the number of JADE-based extensions/add-ons), overriding its advanced features can be difficult due to (1) high cognitive complexity [66], often reaching scores above 20 at the method level, where 15 is the recommended threshold, (2) large classes, sometimes exceeding 700–2000 lines, and (3) code repetitions, with a total of 442 cases found in the source code. To outline the scale of existing issues, running the IntelliJ code inspection on the latest sources of JADE results in 364 errors and 17,431 severe

warnings. Among these, 164 errors are attributed to the usage of deprecated APIs, which reflects the maintenance state of JADE (see Section 5.5). Furthermore, there are a large number of lines with comments that are not part of the documentation. This deviates significantly from the standards of high-quality code, which should be self-explanatory, with occasional comments referring only to the most complex functions. Although source code details may not always impact the framework's usage, they become problematic whenever the original code does not work as intended (as discussed further in Section 5.5). In such cases, with a low level of readability of the source code and high process complexity, identifying problems and reporting them becomes extremely challenging. In this regard, the ambiguous naming occurring in JADE is yet another factor leading to potential errors. For example, *ParallelBehaviour* in JADE is not actually "parallel" in the sense of multi-threading; it simply selects the underlying behaviours in a round-robin fashion. Lastly, as the platform's authors have mentioned in one of the forums, JADE does not allow access to the agent constructed within the agent container. The reason for this is to minimise developer impact on the agent, which, by nature, is autonomous. While this approach works well for development, it limits the possibilities for testing, as described in detail in Section 5.6.

Table 3. Adherence to modern code standards in JADE, Jadex, and SPADE.

Platform Name	Standard of the Platform's Code
JADE	Provides advanced built-in abstracts supporting concepts of FIPA standards but uses strictly agent-related terminology. It does not follow commonly used patterns and has ambiguous naming, contradicting the code implementation. Following the principle of autonomy of the agents restricts accessibility to some functions.
Jadex	Provides agent concepts through Java annotations and offers constructs for agent reasoning. The source code contains many commented-out functionalities, methods marked for fixing, and indicated workarounds. The platform does not reuse commonly applicable libraries for specific software constructs (e.g., collections) and instead implements its own versions.
SPADE	Contains a much narrower scope of base functionalities than JADE and Jadex. The source code adheres to coding standards and includes typing of methods and variables. However, there are known issues of optimisation, indicating inefficient implementation of certain mechanisms.

In contrast to JADE, Jadex offers more modern constructs for agent implementation. Although it also faces the problem of conceptual complexity in agent paradigms, through configurable annotations, it provides a facade that simplifies the definition of agents and their reasoning processes. Moreover, compared with JADE, it exhibits a greater level of modularity with significantly more concise classes, reaching up to 1000 lines of Java code. However, the overall architecture of the source project is so complex that code inspection tools are insufficient to capture all the sub-modules simultaneously. Separately, the essential issue of Jadex, which raises the question of whether Jadex-V is production-ready, is the number of commented-out lines in the code. In particular, there are multiple classes where the number of lines of code left commented-out is greater than the number of lines of actual operational code (e.g., class *BDIAgentFeature* has 264 lines of Java code with 960 lines of comments). Moreover, there is a substantial amount of "TODO" comments, with notes indicating potential bugs or parts of functionalities that are yet to be implemented. In the eyes of potential (outside) developers, this seriously decreases the perceived reliability of the framework's application. Additionally, Jadex depends on its own constructs instead of using libraries that are commonly applicable in the industry. For instance, it provides its own implementations of complex collections (e.g., *MultiCollection*, allowing the storage of multiple values under the same key) instead of using libraries such as Apache Commons Collections (e.g., where *MultiValuedMap* does the same thing as *MultiCollection*). On the one hand, this can be seen as an advantage, as Jadex reduces the size of its artifacts through

a minimal number of external dependencies. However, on the other hand, development necessitates the creation of additional methods that accept Jadex-specific objects or dedicated mappers. It also raises concerns about compliance between data structures originating from different sources.

Finally, in the case of SPADE, it should be noted that it has a considerably smaller number of included features. In comparison, JADE contains around 700 classes, and Jadex has more than 900, while SPADE, excluding its extensions, has only about 20. Consequently, it is less functionally advanced than the other platforms and requires additional effort to develop more complex agent systems. Nevertheless, in its simplicity, it adheres to what can be considered a code of good quality. Among its advantages are (1) the implementation of agents in SPADE is much simpler than in JADE and Jadex, (2) it is type-based in terms of classes and methods, and (3) it does not require manual addition of dependencies to run (which can be the case in some legacy systems). Moreover, upon its code inspection (conducted using PyCharm code inspection tools), the platform produces only 1 error and 152 severe warnings, which is much fewer than the other two platforms. Additionally, the SPADE codebase provides a significant number of tests, contributing to its overall reliability. However, according to the information in the official GitHub repository, several tests are failing in the latest version of SPADE (for Windows distributions). This raises questions about whether all functionalities work as intended or if some code fragments are obsolete. The primary issue with SPADE pertains to optimisation and memory management. While the problem was identified three years ago, it has still not been officially resolved (more in Section 5.5). Moreover, these memory issues are not visible to outside developers at first sight and require extra debugging efforts to resolve. This raises concerns regarding the accuracy of built-in error propagation mechanisms.

All of these factors adversely impact the technological maturity of agent platforms, potentially causing serious distrust among software developers. The majority of issues consist of implementation details and workarounds, which can affect the correctness of their work. Due to the overall quality of the code, these details are difficult to track for programmers who did not directly participate in the development of these platforms. This is particularly evident when implementation details differ from the general platform documentation, which is treated as a source of truth when using a specific framework. Let us now outline the state of the code's documentation among agent platforms.

5.4. Documentation

Documentation plays a significant role in building trust among developers towards a given framework and, therefore, can affect their decision to adopt a particular technology [67]. Well-documented code streamlines development, reduces costs, and improves the quality of the final product. It serves as a knowledge transfer tool, becoming particularly essential when the technology being considered is not widely used within the industry and when there are not many specialists available to share their expertise. This is precisely what makes comprehensive documentation imperative in the context of AOSE.

The lack of adequate documentation has been identified as one of the main problems faced by beginner programmers seeking to use agent systems. As of now, it is common for MAS platforms to provide one-pagers that outline only their general features and concepts (e.g., Cresco [68], CloneMap [69]). Further, it is often found that agent platforms provide only simplistic examples without sufficient information needed to construct moderately complex applications (e.g., osBrain [70]). Most often, however, agent platforms provide incomplete or outdated documentation, which does not reflect their most recent functionality.

In this context, Table 4 summarises the state of documentation in each of the selected representative agent platforms.

The documentation of the JADE 4.6 API, user guides, and examples can be accessed on the JADE website. The main issue here is the limited JavaDoc coverage of public methods and fields. Specifically, an analysis conducted using the MetricsReloaded plugin indicated that the JavaDocs coverage of methods in JADE reaches only around 41%. As a reference,

a similar JavaDoc coverage assessment for the commonly used Apache Commons Lang library yielded results close to 93%. In terms of JADE, the methods included in the latest 4.6 version are especially under-documented. For example, the *TemplateBasedMessageQueue* class has around 14% of its methods documented. This, combined with the lack of up-to-date user guides and illustrative examples (the latest user guides were created around 2010), makes it difficult to fully utilise the most recent features of the platform. Moreover, the incompleteness of some of the descriptions (e.g., information that the method *acquire()* called on an agent locks the agent) necessitates developers to study the source code, which can significantly hinder the software development process.

Table 4. Availability and state of documentation in JADE, Jadex, and SPADE.

Platform Name	State of Documentation
JADE	Provides JavaDoc of the latest version (4.6); however, the page stopped working (April 2024). The API documentation of JADE 4.5 is still available. It contains additional user guides, last updated in 2010, and offers illustrative examples.
Jadex	Provides extensive documentation and guides for Jadex versions 3 and 4. It lacks documentation and guides for the latest version—Jadex-V. Presents illustrative examples.
SPADE	Provides up-to-date documentation with simple use cases of the system usage. Does not offer comprehensive user guides like Jadex or JADE.

Similar to JADE, Jadex documentation also does not cover the changes included in Jadex-V. The materials available on the website refer to Jadex 3, while the latest API documentation includes the release of version 4.0.241. Both of these were uploaded more than two years ago, but as discussed with the platform's authors, Jadex has still been under incremental development since then. This suggests that, as of now, the description of features implemented over the past two years is incomplete. However, in contrast to JADE, the JavaDoc coverage of the Jadex-V API is much greater, reaching 80%, which can be considered sufficient in terms of industrial-grade standards.

Finally, in contrast to JADE and Jadex, SPADE has the narrowest documentation. Specifically, it does not include detailed user guides and focuses primarily on simple examples of the platform's usage. As such, it may not be sufficiently extensive to assist developers in implementing more complex systems. However, among all representative platforms, SPADE documentation is the most frequently updated. As indicated in the history of changes, the documentation is updated, on average, twice per year, with the last update made in mid-2023. Moreover, all new SPADE features are regularly being described in research papers [71]. It should be noted that such papers may serve as an appropriate source of information from the conceptual perspective. However, they may not be appropriate for quick inquiries about implementation details in the case of development concerns. In terms of documentation coverage (measured using the doctor-coverage library), SPADE reaches only around 57%, which is clearly below the sufficient threshold.

As can be observed, although most platforms include additional resources intended to assist developers during implementation, their quality and relevance remain questionable. In light of the development of complex systems, missing or outdated documentation requires additional efforts to fully leverage the features that a given tool offers. It is important to note that factors such as the time taken to complete a project are closely associated with development costs. Therefore, especially in industrial-grade SE, extensive investment of time and resources into familiarisation with the given tools may not be sufficiently compensated by the profits obtained. As such, the state of code documentation is certainly one of the areas that should be addressed. However, while important, the sparseness of documentation is not the most significant barrier hindering the industrial acceptance of AOSE. Next, let us focus on the maintenance of agent platforms, as described in the following section.

5.5. State of Maintenance

Frameworks used for industrial-grade applications should be under continuous maintenance to accommodate changes in the latest software distributions. In particular, they need to provide constant updates to prevent vulnerabilities in dependencies and compliance issues within (1) internally used libraries and (2) changes introduced within programming languages. Moreover, such frameworks should regularly address the issues reported by their users, either by implementing direct solutions or suggesting temporary workarounds that ensure the stability of production applications. In the context of current AOSE, this is one area that is significantly lacking.

To begin with, many agent platforms rely on outdated software distributions. For instance, MASON [61] uses Java 8 (when Java 23 is the newest version), while NetLogo [72] uses Scala 2.12.18 (with Scala 3.5.1 being the newest). This not only limits the possible features offered by these platforms and impacts the quality of their code (see Section 5.3) but also may cause compatibility issues with the latest programming language versions (e.g., Java 21 introduced modifications in threads, concurrency, and sequenced collections). Moreover, since many agent platforms are maintained by individuals or small research groups, they often lack the necessary resources to provide adequate support for their continuous updates. As such, some platforms are maintained irregularly (e.g., SpaDES [73]), bringing uncertainty regarding their depreciation. Additionally, this factor also affects the efficiency of bug fixing, resulting in some platforms having open issues for extended periods.

In this context, the state of maintenance of JADE, Jadex, and SPADE is outlined in Table 5.

Table 5. State of maintenance in JADE, Jadex, and SPADE.

Platform Name	State of Maintenance
JADE	Builds on Java JDK 1.5, with its compatibility tested with Java 21. Its updates are published irregularly at intervals spanning several years. Some old issues remain unaddressed.
Jadex	Targets Java 17. Updates are made regularly and include frequent bug fixes.
SPADE	Is compatible with Python 3.10. Updates are highly irregular, and some technological issues remain unaddressed.

Unlike most modern frameworks that use Maven or Gradle in their builds, JADE operates on Ant. Notably, it targets Java JDK 1.5, which is extremely outdated compared to the latest Java JDK 23. Although JADE has been used in practical applications based on Java 21 (e.g., JRBA [74]), the existing difference between distributions raises concerns about the long-term functionality of the framework. These concerns become even more emphasized when analyzing the code inspection results, which indicate a total of 168 errors related to the usage of deprecated APIs (e.g., constructors of *Long* and *Integer* classes, which were removed in Java 9) and 2,637 warnings suggesting possible improvements related to consecutive language level migrations (starting from Java 5 up to Java 21). This technical debt is further compounded by the irregular releases of JADE. Specifically, the latest JADE release was made in December 2022, while the prior one appeared in 2017. The low frequency of releases also affects the number of accumulated errors in JADE over the years. For example, a known problem that leads to the loss of messages in inter-platform communication was reported around 2005 but still remains unresolved. Furthermore, JADE does not offer concise means for reporting issues (e.g., a common backlog), which would streamline communication with end users. Consequently, all of these factors contribute to a high risk for JADE's applicability to production-class applications.

In terms of maintenance, Jadex appears to be much more stable than JADE. The latest Jadex-V version relies on Java 17, which is the distribution commonly used by industrial-grade applications. In its core module, there are no errors regarding the usage of deprecated methods, and the code inspection indicates only 103 warnings, with possible suggestions for language-level migration improvements. Moreover, up until Jadex-V, subsequent versions

were released monthly. However, there has been a noticeable gap between the release of version 4.0.267 and Jadex-V. This may be due to the migration of the project, but the consistency of upcoming releases should be closely observed. Due to the frequent code updates, Jadex is better able to manage technological debt (despite the issues outlined in Section 5.3) by including extensive bug fixes in its successive versions. Additionally, by publishing the repository on GitHub, Jadex provides a common space for issue tracking and bug reporting, making both of these concerns transparent to external developers.

Like Jadex, SPADE remains up-to-date in terms of the latest software distributions, supporting the most recent versions of Python. Moreover, during the code analysis, no errors indicated possible migration improvements or usage of deprecated APIs. However, the main issue with the platform lies in the lack of frequent updates. This is primarily due to the fact that SPADE is maintained by a small group of developers. Nonetheless, from the perspective of end users, such an approach does not offer stability sufficient for application in industrial-grade systems. Moreover, due to irregular updates and the limited amount of human resources available for functionality improvements, SPADE contains several unresolved issues. For example, one of the critical bugs involves inefficient memory management. Specifically, it was reported that the agent's trace store, used in SPADE, consumes increasing amounts of memory in simulations with constant memory complexity. Additionally, due to the upfront size limitations, when running applications, it became possible to fill the entire memory capacity. Although several years have passed, the issue has not been addressed. A similar case exists with the other 26 issues reported in the official repository. In such a case, if an industrial-grade large-scale system based on SPADE were to be deployed in a production environment, it would likely result in burdensome errors that could take a substantial amount of developers' time to resolve, generating additional project costs.

The severity of technical debt may differ significantly depending on the agent platform. Nevertheless, it is evident that legacy platforms need to be migrated to modern software distributions commonly used by the industry. Furthermore, some platforms still struggle with recognising and resolving errors. This can be attributed, among other things, to the lack of sufficient test coverage. Most platforms (e.g., JADE and Jadex) contain only a few functionality tests, which prevents automatic bug detection. Lack of testing is evident not only in the internal functionality of these platforms but also in their incompatibility with modern industrial-grade test libraries. Up to this section, the focus was placed on the SDLC stages prior to and including **Implementation**. Let us now address, in more detail, the barriers encountered in terms of software **Testing**.

5.6. Test Support

As indicated in Section 4, **Testing** is one of the most critical stages of modern software development. Through early detection of errors and failures, tests support the mitigation of risks. They provide more control over concurrent incremental development and are particularly helpful when the implementation is carried out by multiple teams. Moreover, tests are conducted frequently when an agile SE process is implemented. Additionally, applications with high test coverage can be considered more reliable and transparent, for example, in terms of satisfying acceptance criteria. Within the domain of distributed systems, such as MAS, integration and scenario tests, which allow for monitoring and verifying the results of cooperation between multiple entities, are especially important.

In current AOSE, software testing has been severely neglected. Even though some works introduce dedicated frameworks for testing [75], they provide custom-made tools that, in the context of industrial applications, are difficult to incorporate into standard technologies and libraries. This particularly pertains to the automation of CI/CD tools. In such cases, industrial-grade developers working on specific technology stacks would need to learn completely new testing approaches. Moreover, the tools introduced hinder the possibility of automated multi-platform testing, which can become essential in the context of stress and load tests.

As in previous sections, Table 6 describes the support for automated and standard-based tests, within each of the reference platforms.

Table 6. Test support in JADE, Jadex, and SPADE.

Platform Name	Provided Test Support
JADE	Provides dedicated GUI for scenario testing. Does not offer any other means of test automation. Moreover, restricts access to the agents, limiting the possibilities of integration tests.
Jadex	Provides dedicated GUI for scenario testing. Does not offer any other means of test automation.
SPADE	Does not provide any dedicated means of test automation.

In order to support running test cases, JADE offers a dedicated JADE Test Suite add-on. However, the framework does not provide any means of performing tests that are common to modern standards and technologies. While modern Java-based systems use libraries, such as Mockito, JUnit, or AssertJ, the description of test cases in JADE Test Suite is definable only through XML files. Similarly, there are no methods for writing tests that can verify the communication between agents and external services. For instance, considering an application that combines JADE agents with a REST-based system, integration tests would need to be executed according to a previously prepared test context, which is not possible in the JADE Test Suite. Furthermore, this dedicated framework cannot be integrated with common tools for assessing test coverage, which is considered necessary by some large industrial companies to verify whether components are production-ready. Similarly, the JADE Test Suite is incompatible with automation tools such as Jenkins. Therefore, writing integration tests in JADE requires adding extensions to the standard test runners. However, here, developers encounter yet another issue. As briefly mentioned in Section 5.3, initiated agents cannot be retrieved from containers by default. While this can be achieved using some workarounds (e.g., using reflections to change the accessibility to private fields), these solutions do not match production system standards. Lastly, JADE does not automate tests to verify communication between agents residing in multiple agent platforms or agent containers. Integration tests of this type will require the preparation of test contexts and may not be easily automated.

In this context, Jadex faces similar issues. In particular, it also relies on a dedicated GUI for the scenario tests. Although it provides JUnit-like support for defining test cases, it is still not automatically extendable for providing additional integration configurations (e.g., setting up containers for database testing). Therefore, all of the issues mentioned for JADE, which are related to the custom-based testing component, also apply to Jadex. However, one small difference between these two agent platforms is that Jadex offers easier access to initiated agents than JADE. Nonetheless, custom extensions are still required to support integration testing.

Finally, SPADE does not provide any built-in means to facilitate tests. It is purely based on standard Python libraries for testing (e.g., Pytest). However, since it does not offer any additional support, all necessary setup (e.g., for distributed testing) and its automation require additional efforts from developers.

In summary, none of the prevalent platforms provide adequate tools for automating tests. Therefore, there is a considerable amount of work to be done in the field of testing MAS to ensure their industrial acceptance. Moreover, in addition to the issues raised earlier, tests are not the only problem area that AOSE faces in terms of SDLC automation. The next section addresses the problem of automated deployment.

5.7. Automated Deployment and Monitoring Tools

Due to the impact of agile methodologies, as outlined in Section 4, frequent deployments of small application features have become a standard in modern SDLC. As a result, automation of deployment processes has been set as the minimum requirement. This

includes, among other things, the flexibility of configuration propagation, which not only needs to encompass several technologies (as mentioned in Section 5.2) but also must accommodate differences occurring in the targeted environments. Equally important is the control over the application state after deployment, which can be streamlined using several monitoring tools (e.g., Prometheus) and dedicated dashboards (e.g., Grafana). Because of the architectural design and the complexity of distributed MAS, these aspects are even more challenging to address.

None of the existing agent platforms fully support (1) automated methods for propagating deployment configurations and (2) integration with monitoring tools. In this domain, some potential has emerged with the creation of cloud-based agent platforms (e.g., CloneMap [69]). These could, for instance, leverage the automated possibilities of Kubernetes and compatible technologies. However, as of now, they have not been used to address the challenge directly. Accordingly, at present, deployments in AOSE must be performed manually (or, in the best-case scenario, semi-manually). This also applies to the monitoring tools, which are even more complex due to the necessity of verifying and tracking distributed systems. Such a solution does not meet the standards of production systems. In particular, in industrial-grade applications, this would be unacceptable due to the time and effort required from the developers.

A deeper understanding of the severity of these issues can be seen by considering how they are addressed in JADE, Jadex, and SPADE, as summarised in Table 7.

Table 7. Automated deployment and monitoring in JADE, Jadex, and SPADE.

Platform Name	Methods of Addressing Automated Deployment and Monitoring
JADE	No built-in automation support.
Jadex	No built-in automation support.
SPADE	No built-in automation support.

As can be observed, none of the platforms provide support for distributed automated deployment. This includes both configurations that need to be propagated to run distributed systems and scripts for allocating agents and running them in the company's environment using single deployment pipelines. Regarding JADE, these limitations and disadvantages have been encountered in a practical use case. As part of the scalability tests for the JADE agent platform, the JADE-based system was run on multiple Azure virtual machines located in different world regions [76]. Testing was performed using a variety of internal agent system architectures, where agents were distributed among many agent platforms and containers. In this setting, the pre-configuration of the test environment took up to four hours of the developers' time. Similarly, each reconfiguration of internal architecture required manual changes in initial scripts, taking around half an hour for each run. Moreover, it should be considered that these tests were conducted on a relatively simple infrastructure, which was at TRL 3–4. This emphasises even more that AOSE would not be advantageous in industrial-grade use cases (1) considering the investment necessary for deployment and (2) in contrast to the remaining SDLC technologies, which already provide a high level of automation.

A lack of automation and built-in integrability can also be observed in terms of advanced monitoring. When considering highly distributed systems, none of the platforms provided functionalities that would allow monitoring of each agent's (1) state of resource usage, (2) lifecycle stage, or (3) possible exceptions. Even though platforms such as JADE, Jadex, or SPADE provide dedicated GUI tools that allow tracking the state of an agent system, they are only local and do not integrate with common SDLC dashboards. From the industrial-deployment perspective, it is only possible to control the post-deployed agent system at the level of the entire application's components, which is not sufficient when it is assembled from multiple instances.

Both of these factors outline that the state of AOSE is still far from what is necessary in terms of its industrial acceptance. Let us now summarise and categorise all of the findings.

6. Discussion

To further compare the findings of the study with the actual needs of the industry, a survey was conducted among 30 developers from 5 major IT companies. In order to obtain the most diverse results, the developers were selected from different industries, held different job titles, and varied in terms of the programming languages they use and the years of experience they have. In this context, Figure 2 presents the distribution of surveyed developers among the programming languages that they have most frequently used in their professional experience.

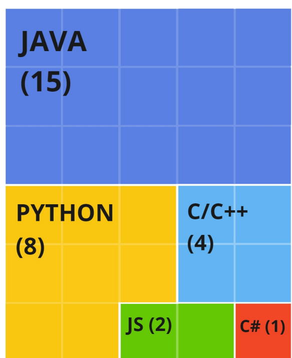


Figure 2. Chart presenting number of developers that use distinct programming languages.

As can be observed, the majority of developers who participated in the survey use Java (50%) and Python (26%) as their primary programming languages. However, the answers were also collected from a small group of developers who work with C or C++ (14%), JavaScript (7%), and C# (3%), which allows viewing the survey's findings in a broader context. In order to minimize bias in the results, the developers also originated from different industrial sectors, as shown in Figure 3.

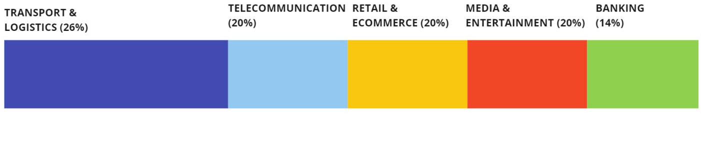
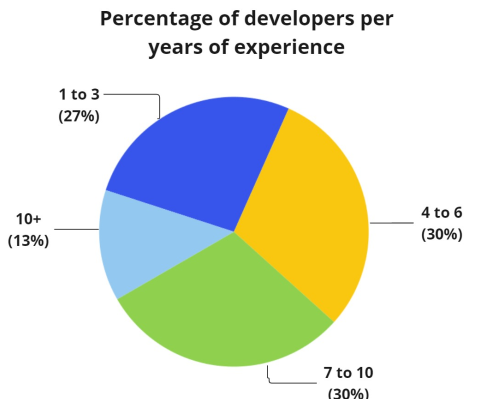


Figure 3. Distribution of developers among different industrial sectors and job types, including development vs. research and development.

The IT industrial sectors, which were included in the survey, encompassed *Transport and Logistics*, *Telecommunication*, *Retail and Ecommerce*, *Media and Entertainment*, and *Banking*. It is observable that even though the group of developers coming from *Transport and Logistics* (26%) was slightly larger than the remaining groups, the distribution is more or

less even. In terms of the job types, two categories were considered, i.e., (1) *development* that includes programmers who are purely responsible for writing code and (2) *Research and Development (R&D)*, where programmers not only write code but are also actively engaged in working on innovation.

Finally, selected developers had varying work experience, which is illustrated in Figure 4. The pie chart presented on the left outlines the distribution of developers among the number of years of experience, including groups *1 to 3*, *4 to 6*, *7 to 10*, and *10+*. On the other hand, the figure presented on the right summarizes the number of developers per their job titles, which encompasses *Junior Developers*, *Mid Developers*, *Senior Developers*, *Quality Assurance Testers*, and *Technical Leads*.



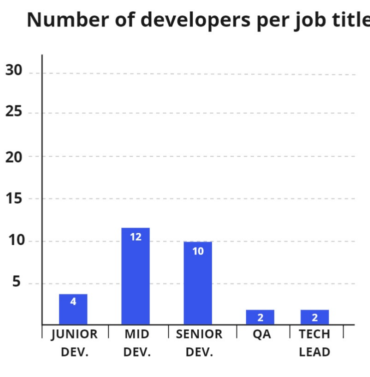


Figure 4. Distribution of developers based on the number of years of experience and job titles.

As indicated, the largest group that participated in the survey had between 4 and 10 years of experience (in total 60%), as well as had either the role of *Mid Developer* (40%) or *Senior Developer* (10%). Nevertheless, developers with extensive expertise (i.e., 10+ years of experience or/and *Technical Leads*), as well as beginning developers (i.e., 1 to 3 years of experience or/and *Junior Developers*), also provided their input.

The survey was conducted as follows. Each developer was asked to rank all the categories mentioned in the previous section (i.e., *Package management support*, *Possibility of integration with external services*, *Adherence to modern code standards*, *Documentation coverage and quality*, *State of maintenance*, *Automated test support*, *Automated deployment support*, and *Monitoring tools support*) based on their importance when selecting open-source frameworks for product development. The results are summarised in Figure 5, which depicts how often each category was placed in the top three selections. It should be pointed out that the obtained results cannot be seen as a "ground truth" due to the limited number of participants; however, due to the diversity of participants and their strong industrial experience, they can serve as supporting evidence for the findings discussed in this contribution.

It can be observed that the developers indicated that *State of maintenance*, *Package management support*, and *Documentation coverage and quality* are the most critical aspects behind the selection of open-source frameworks. Specifically, (1) the *state of maintenance* was placed in the top three categories by approximately 86% of developers, (2) 56% identified the importance of *package management support*, while (3) 33% included *documentation coverage and quality*. Additionally, *automated test support* and *automated deployment support* were often ranked as the second and third most important categories. These results are not surprising, as all the indicated categories encapsulate methods and tools that assist in software development while ensuring a sufficient level of robustness.

On the other hand, the least important categories were *adherence to modern code standards* (selected in the top three by around 13% of developers) and *monitoring tools support* (selected by none of the developers). Regarding *adherence to modern code standards*, this can be explained by the fact that developers using a given framework are not directly responsible for its maintenance. Therefore, as long as the functionalities work as intended,

in less critical cases, the quality of the code itself is not of the highest priority. Similarly, the configuration of monitoring tools is done/modified so rarely that its support does not have to be particularly advanced.

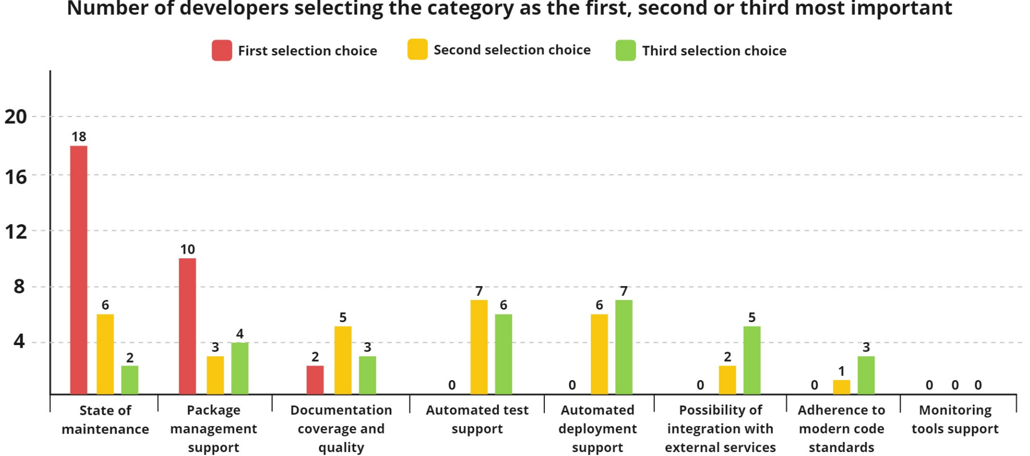


Figure 5. Bar chart summarising how often each category was placed in the top 3 selections.

The results of the pertinent analysis were collectively considered to construct a coverage-importance matrix. Specifically, (1) the conclusions drawn from the evaluation of the state of selected agent platforms (described in Section 5), as well as (2) patterns recognized within existing agent platforms surveys [40,43], were used to determine the coverage for each individual category. The category was classified as *none* when there were almost no platforms that satisfied or had features that would give considerable potential to satisfy a given category. Next, *low* coverage was assigned to the categories supported by a modest number of agent platforms (e.g., for *adherence to modern code standards*, one can mention SPADE), but have plenty of platforms that do not address them at all. On the other hand, the *medium* coverage was assigned for the categories that are at least partially addressed by the majority of agent platforms, but still not fully covered by most. The categories would have been of *high* coverage if almost all agent platforms had offered features that fully addressed them.

Additionally, the developers' responses were used to specify the importance. In particular, (1) the categories placed in the top three selections by developers more than 10 times were considered of *high importance*, (2) those selected more than 5 times were of *medium importance*, and (3) the remaining categories were of *low importance*. The resulting matrix is presented in Figure 6.

The fields indicated in red mark the technological areas of AOSE where the most attention is needed. As can be observed, this group currently encompasses almost all categories. The most severe issues are in the domains of *automated deployment support* and *automated test support*, which have the highest importance and an absolute lack of coverage, as it has been pointed out in Section 5. Significant work is also required in *package management support* and *state of maintenance*. Here, note that while discussing the technical state of agent platforms, it was noted that many of them are still not deployed in central, open-access repositories. Furthermore, platforms from the representative group tend to use outdated code practices, making them less robust. Finally, framework's ease-of-use and its reliability are among the most essential features that developers seek, thereby these two issues should also receive considerable attention.

Despite the lack of coverage in contemporary AOSE tools, it seems that the *possibility of integration with external services* can be addressed later, as this category was assigned slightly lower importance. The last category in this group is *monitoring tools support*. Despite its low interest among developers, since it has no existing coverage in AOSE frameworks, it should be at least minimally considered.

The next "yellow group" indicates categories that should be considered but are not of the highest priority. It includes (1) *documentation coverage and quality*, which has been partially addressed by existing AOSE frameworks, and (2) *adherence to modern code standards*, which is also slightly covered and of lower importance.

Interestingly, no categories were identified in the "green group", which covers domains of sufficient technological maturity. Consequently, this indicates that work should be done in almost all the aforementioned fields for AOSE tools to meet the standards of industrial-grade systems.

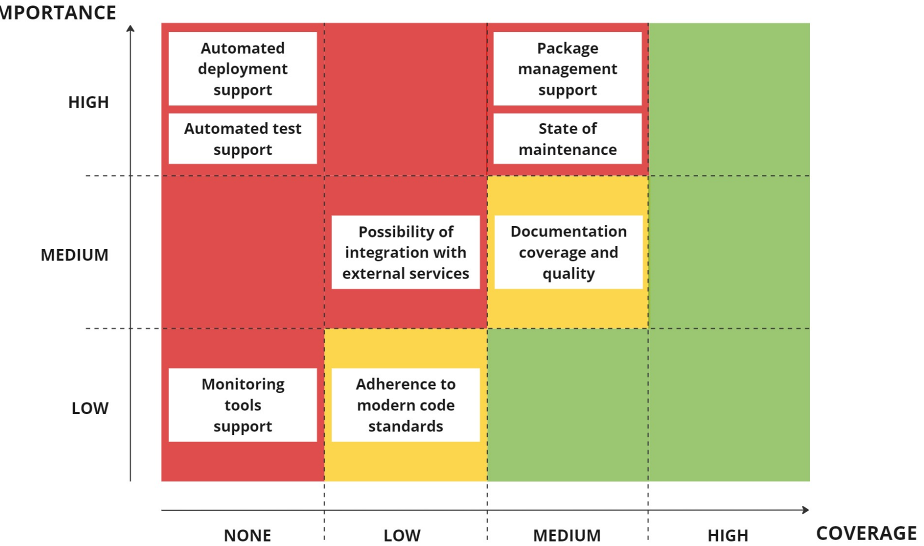


Figure 6. Coverage importance matrix of software development categories.

7. Concluding Remarks

From a conceptual perspective, agent-based programming is well-aligned with the requirements of modern distributed systems. However, due to the complexity of agent-related paradigms, developers without extensive understanding of the agent domain may struggle to use them successfully without proper tools. Therefore, it is important to make the advanced mechanisms offered by agent platforms available through modern, well-documented APIs, rendering them more accessible to today's software architects and developers. As emphasised throughout this contribution, for the AOSE tools to achieve industrial-grade acceptance, they must first meet certain technological standards. This work has identified and highlighted the most critical future development directions that need to be inherently addressed. It has become evident that the standards generally adopted within AOSE have fallen years behind those considered common in modern industrial software development. The technological maturity of existing frameworks requires considerable advancements. Therefore, this gap can be seen as one of the most significant limitations hindering the application of MAS-based methods in the development of commercial systems, despite their clear conceptual and architectural advantages and their suitability for many application domains.

Moreover, it can be argued that this area appears to have been neglected by the agent-oriented scientific community. Current studies have been primarily concerned with the application of MAS, whereas the platforms for agent-based development have mostly addressed the specific needs of the research community. However, it should be noted that, as with other fields in computer science, rapid technological development is often impossible without the proper resources associated with industry engagement. A clear example of this is found in some of the discussed agent platforms (e.g., JADE), which undergo irregular updates. Furthermore, as pointed out throughout this work, industry perceives the practical development of systems differently from scientists. Thus, it can be

concluded that advancing the agent system domain requires collaborative efforts between the scientific and industrial communities.

Establishing this collaboration will remain impossible without initiating significant technological progress in AOSE. A possible method of achieving this could be to become affiliated with well-known open-source software foundations. For instance, Java-based agent platforms (e.g., JADE) would benefit from standardising their code in accordance with Apache Software Foundation (ASF) regulations, incorporating them into their projects. It would enable users to more actively participate in further project development and would compel the framework's creators to maintain high-quality standards. Through such initiatives, AOSE may reach a larger group of software engineers, contributing to its popularisation.

These considerations motivate further open questions. Suppose that one would pursue the development of an agent platform that complies with modern coding standards and SE requirements. What would be the most effective pathway to undertake? Would it be feasible to put efforts into restructuring one of the existing agent platforms, making it compliant with common SE standards (e.g., ASF standards or documentation standards)? Alternatively, someone could leverage the experience gained from the analysis of existing platforms' designs to create a new solution from scratch, preemptively bearing strict technical requirements in mind and utilising modern open-source tools and libraries (e.g., Apache Kafka and Apache Pekko) to deliver a truly scalable, resilient, and distributed architecture. Both approaches have their advantages and disadvantages. However, addressing these questions is out of the scope of this contribution. Nevertheless, anyone interested in pursuing either of the two pathways is invited to contact us.

Author Contributions: Conceptualisation, Z.W.; investigation, Z.W., writing—original draft preparation, Z.W.; writing—discussion, review and editing, W.P., M.G., M.P., A.F., G.C., and C.B.; supervision, M.G. and M.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in the study are included in the article further inquiries can be directed to the corresponding author/s.

Conflicts of Interest: The 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.

Abbreviations

The following abbreviations are used in this manuscript:

AC	Active Component
ACL	Agent Communication Language
AMS	Agent Management System
AOSE	Agent-Oriented Software Engineering
BDI	Belief Desire Intention
CEC	Cloud-Edge-Continuum
CI/CD	Continuous Integration and Continuous Delivery
DAI	Distributed Artificial Intelligence
DDS	Design Document Specification
DF	Directory Facilitator
FIPA	Foundation for Intelligent Physical Agent
GDPR	General Data Protection Regulation
GUI	Graphical User Interface
IDE	Integrated Development Environment
IPMS	Inter Platform Mobility Service
JCC	Jadex Control Center

MAS	Multi-Agent System
OOP	Object Oriented Programming
QA	Quality Assurance
QoS	Quality of Service
SCA	Service Component Architecture
SDLC	Software-Development Life Cycle
SE	Software Engineering
SLA	Service Level Agreement
TRL	Technology Readiness Level
TDD	Test-Driven Development
UML	Unified Modelling Language
WSIG	Web Service Integration Gateway
XMPP	Extensible Messaging and Presence Protocol
XP	Extreme Programming

References

1. General Data Protection Regulation. Available online: https://gdpr-info.eu/ (accessed on 15 April 2024).
2. Lann, G.L. Distributed Systems—Towards a Formal Approach. In Proceedings of the IFIP Congress, Toronto, ON, Canada, 8–12 August 1977.
3. Gruver, W.A. Distributed Intelligence Systems: A New Paradigm for Systems Integration. In Proceedings of the 2007 IEEE International Conference on Information Reuse and Integration, Las Vegas, NV, USA, 13–15 August 2007; pp. nil14–nil15. [CrossRef]
4. Davis, R. *Report on the Workshop on Distributed AI*; Technical Report; MIT Artificial Intelligence Laboratory: Cambridge, MA, USA, 1980.
5. Wooldridge, M.; Jennings, N.R. Intelligent agents: Theory and practice. *Knowl. Eng. Rev.* **1995**, *10*, 115–152. [CrossRef]
6. Goonatilleke, S.; Hettige, B. Past, Present and Future Trends in Multi-Agent System Technology. *J. Eur. Des Syst. Autom.* **2022**, *55*, 723–739. [CrossRef]
7. AgentLink Phase I. 1998. Available online: https://cordis.europa.eu/project/id/27225 (accessed on 12 April 2024).
8. AgentLink phase II. 2000. Available online: https://cordis.europa.eu/project/id/IST-1999-29003 (accessed on 15 April 2024).
9. AgentLink Phase III. 2004. Available online: https://cordis.europa.eu/project/id/002006 (accessed on 15 April 2024).
10. Luck, M.; McBurney, P.; Shehory, O.; Willmott, S. *Agent Technology, Computing as Interaction: A Roadmap for Agent Based Computing*; University of Southampton on Behalf of AgentLink III; University of Southampton Department of Electronics & Computer Science: Southampton, UK, 2005.
11. Berdonosov, V.; Zhivotova, A.; Sycheva, T. TRIZ Evolution of the Object-Oriented Programming Languages. *Procedia Eng.* **2015**, *131*, 333–342. [CrossRef]
12. Osterweil, L.J.; Ghezzi, C.; Kramer, J.; Wolf, A.L. Determining the Impact of Software Engineering Research on Practice. *Computer* **2008**, *41*, 39–49. [CrossRef]
13. Belecheanu, R.A.; Munroe, S.; Luck, M.; Payne, T.; Miller, T.; McBurney, P.; Pěchouček, M. Commercial applications of agents: Lessons, experiences and challenges. In Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), Hakodate, Japan, 8–12 May 2006; pp. 1549–1555. [CrossRef]
14. Belecheanu, R.; Munroe, S.; Luck, M.; Delaney, T.; Fletcher, M. Intelligent Human Variability in Computer Generated Forces Agent Oriented Software—A Case Study. 2005. Available online: https://grasia.fdi.ucm.es/aose08/specialtracks/assets/aos2.pdf (accessed on 15 April 2024).
15. JACK Agent Platform. Available online: https://aosgrp.com.au/jack/ (accessed on 15 April 2024).
16. Belecheanu, R.; Luck, M.; Darley, V. Agent-Based Factory Modelling—Eurobios and SCA Packaging a Case Study. 2005. Available online: https://grasia.fdi.ucm.es/aose08/specialtracks/assets/eurobios2.pdf (accessed on 15 April 2024).
17. Skobelev, P.; Glaschenko, A.; Grachev, I.; Inozemtsev, S. MAGENTA technology case studies of magenta i-scheduler for road transportation. In Proceedings of the International Conference on Autonomous Agents, Honolulu, HI, USA, 14–18 May 2007; p. 273. [CrossRef]
18. LS/ATN—Living Systems Adaptive Transportation Networks, Whitestein Technologies. Available online: https://www.whitestein.com/sites/default/files/WhitesteinTechnologies_LS-ATN_ProductBrochure.pdf (accessed on 15 April 2024).
19. Munroe, S.; Belecheanu, R.; Luck, M.; Bryan, W. Agent-Based Simulation for Logistics and Plant Management—Nutech Solutions and Air Liquide America A Case Study. 2005. Available online: https://grasia.fdi.ucm.es/aose08/specialtracks/assets/nutech1.pdf (accessed on 15 April 2024).
20. Aart, C.v.; Marcke, K.V.; Pels, R.; Smulders, J. International insurance traffic with software agents. In Proceedings of the ECAI'02—15th European Conference on Artificial Intelligence, Lyon, France, 21–26 July 2002; IOS Press: Geneva, Switzerland, 2002; pp. 623–627.
21. Actoron HomePage. Available online: https://www.actoron.com/#/home (accessed on 15 April 2024).

22. Braubach, L.; Pokahr, A.; Lamersdorf, W. Jadex: A BDI-Agent System Combining Middleware and Reasoning. In *Software Agent-Based Applications, Platforms and Development Kits*; Unland, R., Calisti, M., Klusch, M., Eds.; Springer: Basel, Switzerland, 2005; pp. 143–168.
23. Simudyne Platform. Available online: https://simudyne.com/ (accessed on 15 April 2024).
24. Insights: How Barclays Is Predicting the Future. Available online: https://home.barclays/news/2019/7/insights--how-barclays-is-predicting-the-future-/ (accessed on 15 April 2024).
25. AnyLogic Platform. Available online: https://www.anylogic.com/ (accessed on 15 April 2024).
26. Abuhaimed, S.; Sen, S. Team Performance and User Satisfaction in Mixed Human-Agent Teams. In Proceedings of the AAMAS '24—23rd International Conference on Autonomous Agents and Multiagent Systems, Auckland, New Zealand, 6–10 May 2024; International Foundation for Autonomous Agents and Multiagent Systems: Richland, SC, USA, 2024; pp. 4–12.
27. Aguilera, A.; Montes, N.; Curto, G.; Sierra, C.; Osman, N. Can Poverty Be Reduced by Acting on Discrimination? An Agent-based Model for Policy Making. In Proceedings of the AAMAS '24—23rd International Conference on Autonomous Agents and Multiagent Systems, Auckland, New Zealand, 6–10 May 2024; International Foundation for Autonomous Agents and Multiagent Systems: Richland, SC, USA, 2024; pp. 22–30.
28. Nguyen, J.; Powers, S.T.; Urquhart, N.; Farrenkopf, T.; Guckert, M. An overview of agent-based traffic simulators. *Transp. Res. Interdiscip. Perspect.* **2021**, *12*, 100486. [CrossRef]
29. Bilò, V.; Flammini, M.; Monaco, G.; Moscardelli, L.; Vinci, C. On Green Sustainability of Resource Selection Games with Equitable Cost-Sharing. In Proceedings of the AAMAS '24—23rd International Conference on Autonomous Agents and Multiagent Systems, Auckland, New Zealand, 6–10 May 2024; International Foundation for Autonomous Agents and Multiagent Systems: Richland, SC, USA, 2024; pp. 207–215.
30. Champagne Gareau, J.; Lavoie, M.A.; Gosset, G.; Beaudry, E. Cooperative Electric Vehicles Planning. In Proceedings of the AAMAS '24—23rd International Conference on Autonomous Agents and Multiagent Systems, Auckland, New Zealand, 6–10 May 2024; International Foundation for Autonomous Agents and Multiagent Systems: Richland, SC, USA, 2024; pp. 290–298.
31. AAMAS Conference HomePage. Available online: https://www.ifaamas.org/index.html (accessed on 23 September 2024).
32. EMAS Conference HomePage. Available online: https://emas.in.tu-clausthal.de/2024/ (accessed on 23 September 2024).
33. *AAMAS '24—Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems, Auckland, New Zealand, 6–10 May 2024*; International Foundation for Autonomous Agents and Multiagent Systems: Richland, SC, USA, 2024.
34. EMAS Accepted Papers 2024. Available online: https://emas.in.tu-clausthal.de/2024/accepted/ (accessed on 23 September 2024).
35. Müller, J.; Fischer, K. Application Impact of Multiagent Systems and Technologies: A Survey. In *Agent-Oriented Software Engineering: Reflections on Architectures, Methodologies, Languages, and Frameworks*; Springer: Berlin/Heidelberg, Germany, 2014; pp. 27–53. [CrossRef]
36. Leitão, P.; Karnouskos, S. A Survey on Factors that Impact Industrial Agent Acceptance. In *Industrial Agents Emerging Applications of Software Agents in Industry*; Elsevier Inc.: Amsterdam, The Netherlands, 2015; pp. 421–429. [CrossRef]
37. Gorodetsky, V.; Skobelev, P. System engineering view on multi-agent technology for industrial applications: Barriers and prospects. *Cybern. Phys.* **2020**, *9*, 13–30. [CrossRef]
38. Ndumu, D.T.; Nwana, H.S. Research and development challenges for agent-based systems. *IEE Proc.-Softw.* **1997**, *144*, 2–10. [CrossRef]
39. Chmiel, K.; Tomiak, D.; Gawinecki, M.; Karczmarek, P.; Szymczak, M.; Paprzycki, M. Testing the Efficiency of JADE Agent Platform. In Proceedings of the Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, Cork, Ireland, 7 July 2004; pp. 49–56. [CrossRef]
40. Wrona, Z.; Buchwald, W.; Ganzha, M.; Paprzycki, M.; Leon, F.; Noor, N.; Pal, C.V. Overview of Software Agent Platforms Available in 2023. *Information* **2023**, *14*, 348. [CrossRef]
41. Hołda, P.; Rachwał, K.; Sawicki, J.; Ganzha, M.; Paprzycki, M. Agents Assembly: Domain Specific Language for Agent Simulations. In *Advances in Practical Applications of Agents, Multi-Agent Systems, and Complex Systems Simulation. The PAAMS Collection, Proceedings of the 20th International Conference, PAAMS 2022, L'Aquila, Italy, 13–15 July 2022*; Dignum, F., Mathieu, P., Corchado, J.M., De La Prieta, F., Eds.; Springer: Cham, Switzerland, 2022; pp. 487–492.
42. Ribeiro, B.; Pereira, H.; Gomes, L.; Vale, Z. Python-Based Ecosystem for Agent Communities Simulation. In Proceedings of the 17th International Conference on Soft Computing Models in Industrial and Environmental Applications (SOCO 2022), Salamanca, Spain, 5–7 September 2022; Springer: Cham, Switzerland, 2023; pp. 62–71.
43. Cardoso, R.C.; Ferrando, A. A Review of Agent-Based Programming for Multi-Agent Systems. *Computers* **2021**, *10*, 16. [CrossRef]
44. JADE Platform. Available online: https://jade.tilab.com/ (accessed on 15 April 2024).
45. SPADE Platform. Available online: https://spade-mas.readthedocs.io/en/latest/readme.html (accessed on 15 April 2024).
46. Jadex Platform. Available online: https://www.activecomponents.org/#/project/newsl (accessed on 15 April 2024).
47. JaCaMo Agent Platform. Available online: https://jacamo-lang.github.io/ (accessed on 15 April 2024).
48. SARL Agent Platform. Available online: http://www.sarl.io/ (accessed on 15 April 2024).

49. Collier, R.W.; Russell, S.; Lillis, D. Reflecting on Agent Programming with AgentSpeak(L). In Proceedings of the PRIMA 2015: Principles and Practice of Multi-Agent Systems, Bertinoro, Italy, 26–30 October 2015; Springer: Cham, Switzerland, 2015; pp. 351–366.
50. ASTRA Agent Platform. Available online: https://guide.astralanguage.com/en/latest/ (accessed on 15 April 2024).
51. Foundation for Intelligent Physical Agents, Agent Communication Language. 1997. Available online: http://www.fipa.org/specs/fipa00018/OC00018.pdf (accessed on 15 April 2024).
52. Rao, A.S.; George, M.P. BDI agents: From theory to practice. In Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95), San Francisco, CA, USA, 12–14 June 1995; pp. 312–319.
53. Omonije, A. Agile Methodology: A Comprehensive Impact on Modern Business Operations. *Int. J. Sci. Res.* **2024**, *13*, 132–138. [CrossRef]
54. Hannay, J.E.; Dybå, T.; Arisholm, E.; Sjøberg, D.I. The effectiveness of pair programming: A meta-analysis. *Inf. Softw. Technol.* **2009**, *51*, 1110–1122. [CrossRef]
55. Article: What Is Pair Programming? 2021. Available online: https://www.codecademy.com/resources/blog/what-is-pair-programming/ (accessed on 15 April 2024).
56. Article: What Is Test-Driven Development? Available online: https://testdriven.io/test-driven-development/ (accessed on 15 April 2024).
57. Azhar, R.; Malik, A.A. An Empirical Study of the Impact of Software Process Patterns on Software Quality and Team Productivity. In Proceedings of the 2023 25th International Multitopic Conference (INMIC), Lahore, Pakistan, 17–18 November 2023; pp. 1–6. [CrossRef]
58. WADE Agent Platform. Available online: https://jade.tilab.com/wadeproject/index.php (accessed on 15 April 2024).
59. Jadescript Agent Platform. Available online: https://github.com/aiagents/jadescript (accessed on 15 April 2024).
60. CellNet Agent Platform. Available online: https://github.com/jcburguillo/CellNet (accessed on 15 April 2024).
61. MASON Agent Platform. Available online: https://github.com/eclab/mason/ (accessed on 15 April 2024).
62. piaf Agent Platform. Available online: https://gitlab.com/ornythorinque/piaf/ (accessed on 15 April 2024).
63. JS-son Agent Platform. Available online: https://github.com/TimKam/JS-son (accessed on 15 April 2024).
64. Martin, R.C. *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed.; Prentice Hall PTR: Hoboken, NJ, USA, 2008.
65. Akka Actor Platform. Available online: https://akka.io/ (accessed on 15 April 2024).
66. Sonar: Cognivite Complexity Metric. Available online: https://www.sonarsource.com/resources/cognitive-complexity/ (accessed on 17 April 2024).
67. Venigalla, A.S.M.; Chimalakonda, S. Understanding Emotions of Developer Community Towards Software Documentation. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS), Madrid, ES, USA, 25–28 May 2021; pp. 87–91. [CrossRef]
68. Cresco Agent Platform. Available online: https://cresco.io/ (accessed on 17 April 2024).
69. CloneMap Agent Platform. Available online: https://fein-aachen.org/en/projects/clonemap/ (accessed on 17 April 2024).
70. OsBrain Agent Platform. Available online: https://osbrain.readthedocs.io/en/stable/basic_patterns.html (accessed on 17 April 2024).
71. Palanca, J.; Terrasa, A.; Julian, V.; Carrascosa, C. SPADE 3: Supporting the New Generation of Multi-Agent Systems. *IEEE Access* **2020**, *8*, 182537–182549. [CrossRef]
72. NetLogo Agent Platform. Available online: https://ccl.northwestern.edu/netlogo/index.shtml (accessed on 17 April 2024).
73. SpaDES Agent Platform. Available online: https://spades.predictiveecology.org/ (accessed on 17 April 2024).
74. JADE Rule Based Extension. Available online: https://github.com/Extended-Green-Cloud/jrba (accessed on 17 April 2024).
75. Gonçalves, E.M.N.; Machado, R.A.; Rodrigues, B.C.; Adamatti, D. CPN4M: Testing Multi-Agent Systems under Organizational Model Moise+ Using Colored Petri Nets. *Appl. Sci.* **2022**, *12*, 5857. [CrossRef]
76. Wrona, Z.; Ganzha, M.; Paprzycki, M.; Krzyżanowski, S.; Bădică, A.; Fidanova, S. Scalability of Extended Green Cloud Simulator. In Proceedings of the 2024 International Conference on INnovations in Intelligent SysTems and Applications (INISTA), Craiova, Romania, 4–6 September 2024; pp. 1–6. [CrossRef]

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.