

MOSAICO

D2.2: Key Performance Indicators of AI agents for SE

March, 2026

Funded by the European Union

Funded by the European Union under the Grant Agreement No 101189664. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency (HADEA). Neither the European Union nor the granting authority can be held responsible for them.

D2.2 Key Performance Indicators of AI agents for SE

Work package	WP2 Repository of AI agents for SE
Task	T2.2 Definition of Key Performance Indicators of AI agents for SE
Task contributors	LIST, UDA
Due date	31/03/2026
Submission date	31/03/2026
Type of deliverable	Demonstrator
Dissemination Level	Public
Deliverable lead	LIST
Version	1.0
Authors	LIST, UDA
Reviewers	Intra, UY, IMT
Keywords	KPI, AI Agents, Quality Model, Benchmarking

Document Revision History

Version	Date	Description of change	List of contributor(s)
v0.1.0	12/03/2026	First Draft	Gwendal JOUNEAUX, Umar Zeshan, Juri Di Rocco
v0.5.0	23/03/2026	Version for internal review	G. JOUNEAUX, U. Zeshan, J. Di Rocco, D. Di Ruscio
v1.0.0	31/03/2026	Final version	G. JOUNEAUX, U. Zeshan, J. Di Rocco, D. Di Ruscio

Partners

























Funding



Funded by
the European Union

Call	Digital and emerging technologies for competitiveness and fit for the Green Deal (HORIZON-CL4-2024-DIGITAL-EMERGING-01)
Topic	HORIZON-CL4-2024-DIGITAL-EMERGING-01-22: Fundamentals of Software Engineering
Type of action	Research and Innovation Action (RIA)

Disclaimer

Funded by the European Union under the Grant Agreement No 101189664. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency (HADEA). Neither the European Union nor the granting authority can be held responsible for them.

Copyright Notice

© MOSAICO Consortium, 2026

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both. Reproduction is authorised provided the source is acknowledged.



Executive Summary

This document presents the methodology and supporting tools for the quality assessment of MOSAICO AI agents. Several quality models have been proposed in software engineering. Deciding whether an AI agent meets the required requirements for adoption in terms of quality and maturity is not a straightforward process. To deal with such problems, MOSAICO is developing an extensible platform to benchmark AI agents in a systematic and automated manner. The analysis and comparison of AI agents requires the definition of a number of Key Performance Indicators (KPIs) and related attributes that have to be calculated and evaluated in order to take the final decision about which of them has to be adopted. In other words, the adoption of a quality model is mandatory. A quality model is typically hierarchically organized since they decompose quality into a hierarchy of criteria and attributes, leading to specific metrics at their lowest level.

In this document, we present initial components of the MOSAICO benchmarking framework (to be fully presented in D2.3) for specifying KPIs of AI agents. After an overview of already developed quality models in the software engineering community, a domain-specific language is introduced with the aim of enabling the specification of KPIs that will streamline the process of discovering, assessing, and selecting AI agents. The KPIs will provide valuable insights into the quality and suitability of AI agents, enabling Software Engineers to make informed decisions about which AI agents to adopt for specific SE tasks.

Table of Contents

1 Introduction **9**

2 Background **11**
2.1 AI Agents in Software Engineering 11
2.2 Quality Characteristics of AI Systems 12

3 Quality Model for AI Agents **13**
3.1 Quality Characteristics of AI Agents 13
3.2 Extension of the Software Quality Model 18

4 KPI Specification **20**
4.1 Abstract syntax 20
4.2 Concrete syntax 23
4.3 AI Agent KPIs 25

5 Specification and use of MOSAICO KPIs **31**
5.1 Distribution and Installation 31
5.1.1 Eclipse Plugin 31
5.1.2 Maven Dependency (Standalone Mode) 33
5.2 Authoring KPI Specifications in Eclipse 34
5.3 Programmatic Use from the MOSAICO Evaluation Backend 36
5.4 Integration with the MOSAICO Benchmarking Architecture 36
5.5 Illustrative Usage Scenario 38

6 Conclusion and future work **40**

List of Figures

1 Dimensions of Agent Capability in Software Engineering. 11
2 Evolution of Software Engineering Benchmarks from Static-Task [6, 26, 27] to Context-Aware [28, 29], Repository-Level [30–32], and Agent-Oriented Paradigms [10, 33–36]. 13
3 ISO/IEC 25010 quality model for software systems with the proposed extension for AI software system 18
4 Metamodel of KPIFormulaDSL (Expressions are removed for clarity) 21
5 Eclipse IDE with the DSL tooling (outline on the right, validation in the gutter and underlined, and content assist providing metric name completion) 34
6 Example of linking between a composite metric reference and its definition . . . 35
7 Example of proposed quick fix for a missing letter in a metric's reference 35
8 Three summary agents that should be evaluated 39

List of Tables

1 Operational Dimensions of Quality Assessment in AI-Assisted Software Engineering . . . 14
2 Mapping of non-functional requirements from [48] to quality characteristics. "—" denotes the absence of matching characteristic . . . 15
3 AI software relevant quality characteristics and sub-characteristics . . . 17
4 Definitions of the new and updated quality (sub-)characteristics . . . 19
5 KPI formula DSL: supported operators. S denotes an expression evaluating to a set; S_i an element of set S; P a predicate (potentially parametrized); a, b arbitrary expressions; W a set of weights $W_i \in \mathbb{R}_{>0}$; $\theta \in [0, 1]$ a threshold; $v(a)$ the computed value of a. . . . 22
6 Evaluation Ingredients of the studied approaches . . . 25
7 Metric and KPI values obtained for three agent configurations on a shared evaluation dataset. KPI values are computed from run-level metric aggregates via the DSL formulas defined above. . . . 40

Abbreviations

KPI	Key Performance Indicator
DSL	Domain-Specific Language
NFR	Non-Functional Requirement

1 Introduction

In recent years, large language models (LLMs) and AI agents have become integral components of modern software engineering workflows. From automated code generation and test synthesis to repository-level modification and multi-agent orchestration, AI-based systems are increasingly embedded across the entire software development lifecycle. These systems now participate in requirement analysis, code transformation, debugging, refactoring, documentation generation, and performance optimization. This evolution has extended the paradigm from CodeWare and AIWare to AgentWare, in which communities of interacting agents collaborate, delegate subtasks, and coordinate reasoning processes to solve complex engineering problems [1]. The shift from single-model assistance to orchestrated agent ecosystems introduces greater autonomy, deeper interaction, and greater contextual awareness. While these advances promise substantial productivity gains and accelerated development cycles, they also introduce new challenges for evaluating, comparing, validating, and governing AI agents operating in dynamic software engineering environments.

The increasing autonomy of AI agents raises fundamental questions about reliability, controllability, reproducibility, and accountability. Unlike static code generation tools, agent-based systems engage in multi-step reasoning, interact with external tools, adapt to evolving repository states, and operate across extended contexts. Their behavior is influenced not only by model parameters but also by orchestration logic, memory mechanisms, tool interfaces, and execution environments [2]. As a consequence, quality assessment cannot be reduced to isolated task accuracy. Instead, it must address systemic properties such as regression stability, dependency integrity, scalability under long-context conditions, robustness against variation, and consistency across iterative reasoning cycles. These properties span both traditional software quality dimensions and AI-specific characteristics, creating a need for structured integration between established standards and emerging evaluation practices.

Despite the rapid proliferation of benchmarks and agent-based frameworks, systematic mechanisms for assessing the quality of AI agents in software engineering remain fragmented. Existing benchmark ecosystems vary considerably in task scope, contextual realism, metric semantics, and reproducibility protocols. Some benchmarks focus on narrowly defined functional correctness, while others emphasize repository realism, performance preservation, or interactive reasoning. [3] The absence of harmonized quality criteria complicates empirical comparison across studies and limits the interpretability of performance claims. Furthermore, traditional software quality standards such as ISO/IEC 25010 were not designed to address properties specific to AI agents, including autonomy, iterative reasoning, tool-mediated execution, long-context scalability, sustainability impact, and telemetry-driven monitoring. Consequently, organizations adopting AI-assisted development tools face methodological uncertainty when evaluating and comparing agent-based solutions.

Within this context, the MOSAICO project addresses the need for a structured, interoperable framework that connects conceptual quality modeling to operational benchmarking practices. Building upon the taxonomy defined in Deliverable D2.1, which established a systematic taxonomy characterizing AI agents and underpinning repository metadata, this deliverable takes the next critical step by formalizing a quality model tailored to AI agents and operationalizing it through measurable Key Performance Indicators (KPIs). Rather than proposing isolated evaluation metrics, the objective is to construct a coherent framework that aligns quality characteristics, benchmark dimensions, and executable measurement mechanisms within a unified architecture. Specifically, this deliverable introduces an extended quality model aligned with ISO standards while incorporating AI-agent-specific characteristics such as autonomy, contextual consistency, scalability, security, sustainability, interaction robustness, and reproducibility.

These characteristics are systematically mapped to benchmark-driven evaluation dimensions observed in contemporary software engineering benchmarks, thereby ensuring methodological compatibility with existing empirical ecosystems. In addition, a domain-specific KPI specification language is defined to enable formal and machine-readable representation of evaluation criteria. This specification supports consistency across use cases, reproducibility of experimental setups, and interoperability between heterogeneous benchmarking scenarios.

The KPI model is embedded within the MOSAICO benchmarking architecture (which is the main subject of Deliverable D2.3), which integrates telemetry collection, controlled execution environments, and reproducible evaluation pipelines. This architecture enables performance-aware comparison of AI agents, traceable computation of metrics, and systematic monitoring of quality properties across diverse industrial contexts. By linking quality characteristics, benchmark taxonomy, formal KPI definitions, and repository-level execution infrastructure, the framework establishes a methodological foundation for rigorous evaluation and governance of AI agents in software engineering. Through this integration of conceptual modeling and executable benchmarking infrastructure, MOSAICO will provide a structured basis for comparability, transparency, and informed decision-making in AI-assisted software engineering environments.

It is important to note that this deliverable serves as the methodological and conceptual foundation for the subsequent evaluation activities within the MOSAICO project. In particular, this document presents the foundational tools, benchmarking paradigms (from Static-Task to Agent-Oriented), and existing quality models (e.g., ISO/IEC 25010 extensions) currently available in the state-of-the-art. It acts as a preparatory study, mapping the landscape of what can be measured and how LLM-assisted engineering affects software quality, sustainability, and maintainability. While D2.2 provides the "toolkit" and the theoretical boundaries, D2.3 will perform the actual selection and specialization of Key Performance Indicators (KPIs). These KPIs will be strictly tailored to the specific requirements and industrial constraints provided by the use case partners. D2.3 will transition from the general state-of-the-art discussed here to a concrete implementation and measurement phase, ensuring that the selected metrics directly address the business and technical goals of the MOSAICO use cases.

This deliverable provides the following contributions:

- An extended quality model for AI agents in software engineering, aligned with ISO/IEC standards and enriched with agent-specific characteristics.
- A structured analysis of benchmark-driven evaluation dimensions across contemporary software engineering benchmarks.
- A formal Key Performance Indicator specification language supporting domain-specific quality assessment.
- An initial set of components that will underpin the benchmarking framework that will be presented in Deliverable D2.3 and that will operationalize quality assessment through telemetry integration, metric computation, and repository-level evaluation workflows.

The remainder of this document is organized as follows: Section 2 provides the background on AI agents in software engineering, discusses quality characteristics of AI systems, and analyzes benchmark-driven evaluation practices. Section 3 introduces the extended quality model for AI agents and presents its alignment with ISO standards. Section 4 defines the Key Performance Indicator specification language and formalizes its syntax and semantics. Section 5 presents an explanatory specification and the use of MOSAICO KPIs. Section 6 concludes the deliverable and outlines the next steps in the MOSAICO roadmap.



2 Background

The rapid advancement of large language models has significantly reshaped research directions in software engineering, expanding the scope of automated systems beyond isolated code generation toward more interactive and workflow-oriented paradigms. In particular, the emergence of AI agents capable of planning, reasoning, and tool invocation has introduced new possibilities for addressing complex software engineering tasks that require iterative decision-making and environment interaction. This shift has not only influenced system design but has also necessitated rethinking how such systems are evaluated in realistic development contexts. As software engineering automation becomes increasingly agent-centric, the design of appropriate benchmarking methodologies has become equally critical. The following subsections first discuss the evolution and characteristics of AI agents in software engineering, and then the corresponding development of software engineering benchmarks to support their systematic evaluation.

2.1 AI Agents in Software Engineering

The increasing capabilities of large language models have shifted software engineering research from isolated code generation toward agent-based systems that operate through iterative interactions with software artifacts and execution environments. Unlike single-step model invocation, AI agents integrate language models with planning mechanisms, memory components, and tool interfaces, enabling multi-step reasoning, adaptive behavior, and environment-aware decision-making.

Early applications focused on task-specific automation such as code generation, translation, and defect detection [4–6]. However, these systems operated under a limited contextual scope. Agent architectures address this limitation by interleaving reasoning and action execution [7], incorporating reflective feedback mechanisms [8], and integrating external development tools [9]. Repository-level systems such as SWE-agent [10] and OpenHands [11] further extend this paradigm to realistic development workflows.

Figure 1 summarizes the primary capability dimensions of AI agents in software engineering, highlighting multi-step reasoning, tool interaction, and repository-level context awareness.

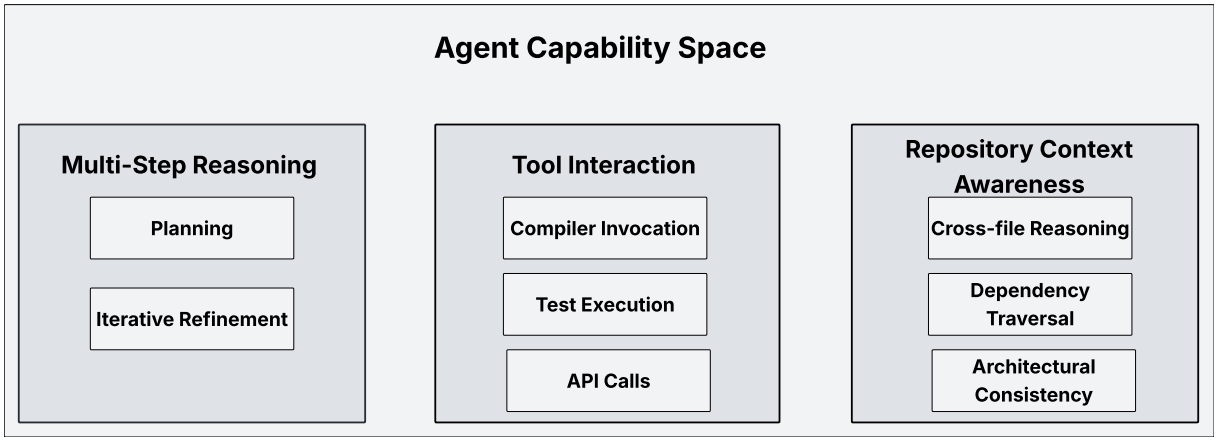


Figure 1: Dimensions of Agent Capability in Software Engineering.

2.2 Quality Characteristics of AI Systems

The integration of AI-based components into software-intensive systems fundamentally challenges established notions of software quality. Unlike deterministic systems whose behavior is fully specified at design time, AI systems exhibit probabilistic outputs, sensitivity to training data distributions, and performance variability under contextual shifts [12]. These characteristics complicate classical verification and validation processes and necessitate the extension of established quality frameworks to account for adaptive and learning-driven behavior.

Several systematic investigations have analyzed quality models tailored to AI-intensive systems [13, 14]. Some models are constructed independently of existing standards. In contrast, others explicitly align their attributes with ISO IEC 25010 [15–20]. The ISO-aligned approach preserves compatibility with functional suitability, reliability, usability, performance efficiency, maintainability, and security while introducing refinements for AI-specific characteristics. However, empirical analyses indicate that these mappings frequently provide only partial coverage of emerging concerns. Sustainability is explicitly addressed in only a subset of ISO-aligned extensions [15, 20], despite growing awareness of the environmental cost associated with large-scale model training and inference [21]. Similarly, explainability, accountability, and transparency are treated inconsistently across models [17, 19], even though they are central to responsible AI development and governance discussions [22]. These gaps suggest that existing quality models require further refinement to capture the systemic properties of AI-enabled systems.

Beyond ISO-based extensions, research on non-functional requirements for AI systems further expands the quality landscape. Requirements engineering studies emphasize interpretability, fairness, adaptability, safety, and traceability as critical system properties [23]. These properties extend traditional software quality considerations into socio-technical domains, where trust, governance, and accountability play a central role. In addition, technical debt in machine learning systems introduces maintainability risks that are not fully addressed in conventional quality models [24]. Data dependencies, configuration drift, and pipeline complexity can degrade system quality over time, reinforcing the need to incorporate evolvability and lifecycle awareness into quality assessment.

Reproducibility also requires explicit consideration in AI contexts. Variability introduced by stochastic training procedures and dynamic execution environments challenges assumptions of repeatable behavior. Initiatives such as MLPerf demonstrate how standardized execution environments and controlled measurement protocols can strengthen reproducibility and comparability in AI evaluation [25]. These principles are directly relevant to AI-assisted software engineering, where benchmark reproducibility influences the credibility of empirical results. In practice, quality characteristics of AI systems are operationalized through benchmarking. Structured benchmarking frameworks provide datasets, evaluation pipelines, and measurable criteria that instantiate abstract quality dimensions.

Figure 2 illustrates the progressive evolution of software engineering benchmarks, tracing a shift from isolated evaluations to holistic, system-wide assessments. The trajectory begins with Static-Task paradigms—exemplified by datasets like HumanEval and MBPP—which focus on discrete code generation units. This evolved into Context-Aware and Repository-Level frameworks (e.g., RepoBench and SWE-bench), which require models to navigate cross-file dependencies and larger codebases. The culmination of this trend is the Agent-Oriented paradigm, which features benchmarks such as SWE-agent and DevBench. In this final stage, the evaluation focuses on the autonomous capabilities of AI agents to perform end-to-end tasks, such as bug fixing and repository maintenance, in complex and dynamic environments. This evolution highlights a growing necessity for benchmarks to mirror the intricate, socio-technical realities of

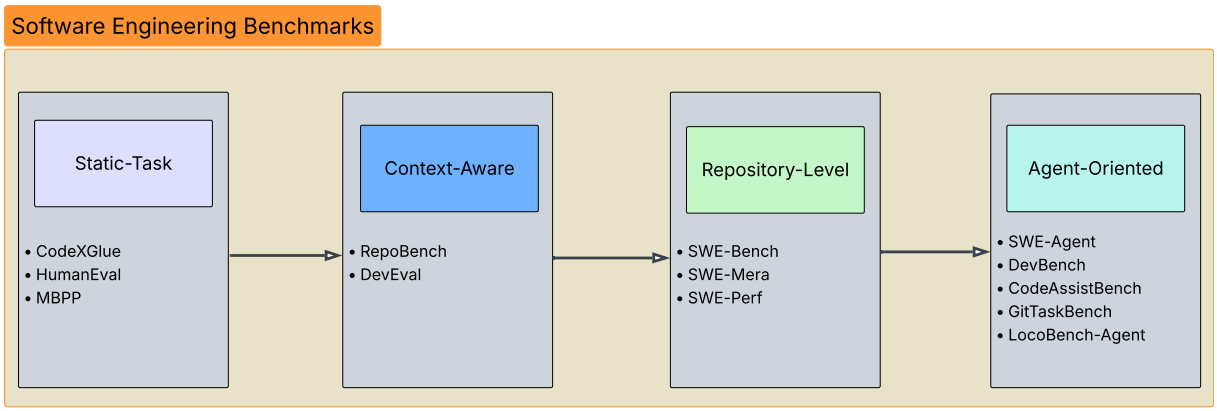


Figure 2: Evolution of Software Engineering Benchmarks from Static-Task [6, 26, 27] to Context-Aware [28, 29], Repository-Level [30–32], and Agent-Oriented Paradigms [10, 33–36].

modern software development.

Table 1 summarizes the primary operational dimensions observed across contemporary software engineering benchmarks and maps them to representative benchmark families. Software engineering benchmarks span diverse task categories and contextual scopes. Early executable validation datasets such as [37, 38, 45] grounded evaluation in regression test outcomes and patch correctness. Code generation benchmarks such as [4, 5] introduced probabilistic correctness metrics, while CodeXGLUE [6] diversified evaluation tasks across translation, defect detection, clone detection, and summarization. Repository-level benchmarks introduce environmental realism by requiring modifications within complete software projects. SWE-bench [30] captures dependency management and regression stability within realistic development contexts, and its extensions [31, 32] incorporate dynamic adaptability and performance preservation. Interactive benchmarks such as [33, 34, 40] emphasize multi-stage reasoning and workflow coherence. Long-context benchmarks [36, 39] address scalability and architectural consistency, while security-focused datasets [42, 46] extend evaluation toward vulnerability resilience.

The diversity of benchmark families illustrates that quality characteristics of AI systems are instantiated through varied evaluation structures. Differences in task abstraction, contextual realism, metric semantics, and execution constraints directly shape the interpretation of empirical results in AI-assisted software engineering.

3 Quality Model for AI Agents

In the current landscape of AI software quality, there is no consensus on a general quality model for AI agents. The goal of this section is to discuss relevant quality characteristics and propose the related quality model.

3.1 Quality Characteristics of AI Agents

As a starting point for our selection of quality characteristics, we looked at the ISO 25059 Quality model for AI systems, an extension of the ISO 25010 Quality model for software products, which includes specific characteristics to encompass additional aspects of these systems.

Table 1: Operational Dimensions of Quality Assessment in AI-Assisted Software Engineering

Quality Assessment Dimension	**Evaluation Operationalization in Benchmarks**	**Representative Benchmark Families**
Functional Correctness	Validation through executable test suites, unit tests, or reference comparisons	HumanEval [26]; MBPP [27]; Defects4J [37]; SWE-bench [30]
Regression Stability	Preservation of previously passing behaviors after modification	SWE-bench [30]; Defects4J [37]; BugsJS [38]
Contextual Coherence	Cross-file consistency, dependency integrity, architectural alignment	RepoBench [28]; LoCoBench [39]; CoreCodeBench [40]
Performance Efficiency	Runtime improvement and resource reduction under correctness constraints	SWE-Perf [32]; MLPerf-style evaluations [25]
Robustness Under Variation	Stability across task configurations, input perturbations, or repeated executions	DevBench [33]; LoCoBench [39]; AgentBench [41]
Security and Vulnerability Resistance	Detection or avoidance of exploitable weaknesses in generated code	Devign [42]; SecurityEval [43]
Scalability and Long-Context Reasoning	Performance retention under large repository contexts or architectural complexity	LoCoBench [39]; LoCoBench-Agent [36]
Reproducibility and Experimental Control	Deterministic execution pipelines and standardized reporting protocols	ModelXGlue [44]; MLPerf [25]; SWE-bench [30]

In addition, we looked at which quality characteristics were included in existing quality models [14]. We counted six main characteristics that made consensus: **Correctness** indicates that the AI software produces the expected result, encompassing accuracy, precision, and recall of the model (maps to Functional correctness in ISO 25010). **Model Relevance** represents the adequacy of the model architecture to the data and problem to solve. **Efficiency** describes the time behavior of the AI software, such as Time-To-First-Token [47] or end-to-end response time [47]. **Robustness** denotes the resilience to perturbations and is often associated with

security concerns. **Fairness** identifies the different types of bias (e.g., racism, sexism) of the AI software. Finally, **Interpretability** characterizes the ability to understand how features impact the output of the system.

Furthermore, another recurrent aspect that we noted is the critical need to understand the decision process of AI components. This is why properties like transparency, explainability, or accountability have been introduced in some quality models [15, 17, 18, 20] to complement the interpretability aspect. In our model, we regroup the transparency, explainability, and interpretability aspects as sub-characteristics of the **Understandability** quality characteristic.

In addition to existing quality models, we also looked at studies regarding non-functional requirements of AI systems [16, 23, 48–51]. Among them, Habibullah *et al.* [48] surveyed practitioners from both industry and academia to derive a list of 35 important non-functional requirements for machine learning systems. As this list includes the requirements elicited in other studies, we used it to map each requirement to one or more of the quality characteristics from ISO 25059 or the ones we extracted from existing quality models. This mapping is represented in Table 2.

Table 2: Mapping of non-functional requirements from [48] to quality characteristics. "—" denotes the absence of matching characteristic

Requirement name	Quality (sub-)characteristic
Accuracy	Functional Correctness
Adaptability	Functional Adaptability
Bias	Fairness
Completeness	Functional Completeness
Complexity	Functional Appropriateness
Consistency	—
Correctness	Functional Correctness
Domain Adaptation	Reusability
Efficiency	Performance Efficiency
Ethics	—
Explainability	Explainability
Fairness	Fairness
Fault Tolerance	Fault Tolerance
Flexibility	Functional Adaptation
Integrity	Integrity
Interpretability	Interpretability
Interoperability	Interoperability
Justifiability	—
Maintainability	Maintainability
Performance	Performance Efficiency

Requirement name	Quality (sub-)characteristic
Portability	Adaptability
Privacy	Confidentiality + Integrity
Reliability	Reliability
Repeatability	—
Retrainability	Modifiability
Reproducibility	—
Reusability	Reusability
Safety	Safety
Scalability	Scalability
Security	Security
Testability	Testability
Transparency	Transparency
Traceability	Accountability
Trust	Security
Usability	Interaction Capabilities

Based on this mapping, we observe a gap with what we already extracted from the literature on the following topics: justifiability, ethics, consistency, repeatability, and reproducibility

The justifiability requirement differs from our other understanding sub-characteristics, as not only a way of understanding the output or the way it was produced, but rather of understanding why this output should be considered right or reasonable. This aspect is especially relevant in contexts such as LLM-as-a-Judge [52] where explaining the thought process is the main goal.

The ethics requirement is concerned with ensuring moral behaviors. This differs from bias and fairness aspects that are concerned about ensuring equal treatment independently of some features (*e.g.,* age, race, religion). Bias and fairness issues can lead to ethical issues (*e.g.,* AI in judicial systems), but ethical issues can arise without bias (*e.g.,* AI inciting violence).

Finally, consistency, repeatability, and reproducibility revolve around the ability of the system to produce a similar output in a similar context (with varying degrees of similarity for each term). In Habibullah *et al.* [48], the definition for consistency does not specify whether the data used as input differ or not, only that the method and system are the same. For this reason, we consider consistency as an overlapping term that denote both repeatability and reproducibility. To model the general idea behind these terms, we define three quality characteristics: *Repeatability*, *Reproducibility*, and *Replicability*. The following definitions for these terms are inspired by the terminology adopted by ACM for its artifact evaluation process.[1]

Repeatability denotes the ability to obtain the same or similar results with the same input and using the same model.

Reproducibility denotes the ability to obtain the same or similar results with similar but different inputs and using the same model.

[1] https://www.acm.org/publications/policies/artifact-review-and-badging-current

Replicability denotes the ability to obtain the same or similar results with similar but different inputs and using a similar but different model.

For our quality model, we extend this set with additional characteristics that should also be accounted for. Table 3 summarizes all the relevant quality characteristics and their sub-characteristics. This includes new dimensions we consider important, such as Sustainability.

Table 3: AI software relevant quality characteristics and sub-characteristics

Correctness	Interoperability
– *Accuracy*	Autonomy
– *Precision*	Understandability
– *Recall*	– *Interpretability*
Model relevance	– *Transparency*
Robustness	– *Explainability*
– *Reliability*	– *Justifiability*
– *Security*	Output properties
Efficiency	– *Creativity*
– *Time-To-First-Token*	– *Diversity*
– *E2E response time*	– *Conciseness*
– *Training* time	– *Repeatability*
Fairness	– *Replicability*
Ethics	– *Reproducibility*
Sustainability	

Indeed, the rising interest in sustainable AI, also known as Green AI [53], shows that sustainability is also one of the main extra-functional properties of AI software to be addressed. There is no yet an existing standard for consistently collecting sustainable metrics [54]. Thus, we based our quality characteristics and metrics on the Sustainability Model Card [55], adding a **Sustainability** characteristic in the quality model addressing the training impact, inference impact, and mitigation for these impacts tackled by the model or platform provider.

We also add the **Autonomy** characteristic. In accordance with current trends in AI research, Agentic AI systems and AI agents are becoming increasingly common. This new paradigm suggests that these agents have the capacity for independent thought and can refine their answers autonomously. However, this refinement may sometimes require additional information from the user, leading to varying levels of autonomy based on how often this occurs.

Finally, the goal of the majority of the AI software is to provide an output to the user in the form of generated text, code, or an image. The quality of these outputs can change significantly while remaining correct with respect to the provided prompt. To tackle these concerns, we propose to add a new **Output properties** characteristic to our quality model. This dimension encompasses extra-functional properties of the generated output, such as the creativity of the model (in the sense of Boden [56]), the diversity of possible outputs, the conciseness of the answer, and the consistency aspects discussed earlier.

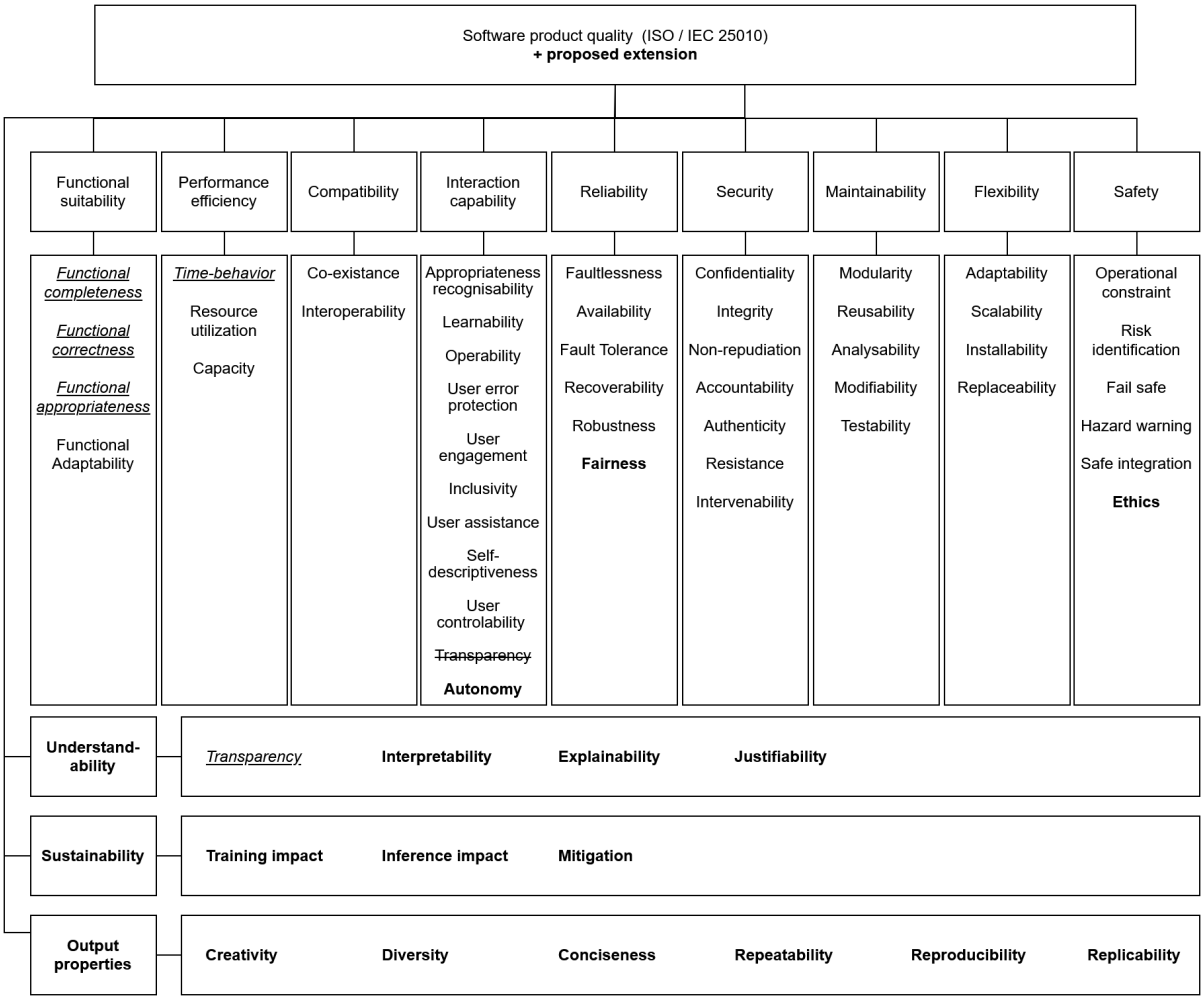


Figure 3: ISO/IEC 25010 quality model for software systems with the proposed extension for AI software system

3.2 Extension of the Software Quality Model

As AI software is still software, we decided to formalize our quality model as an extension of the ISO/IEC 25010 standard for software product quality, with its extension for AI systems defined by the ISO/IEC 25059 standard. Figure 3 shows the existing standard and our proposed extension, denoted by bold text when something was added, by italicized and underlined text when something was modified, and by strikethrough text when moved/removed. The existing model is structured into nine sections, each composed of multiple quality characteristics.

In addition to the existing sections, we added three new sections addressing understandability, the sustainability aspect, and output properties of the AI software, regrouping the related characteristics previously detailed. The remaining characteristics were either added to an appropriate existing section or extended an existing characteristic's definition to encompass the additional concern. The **Autonomy** characteristic, representing the level of User/Agent interaction required, was included in the *Interaction capability* section. **Fairness** represents the *Reliability* of the agent concerning biases. **Ethics** represents the ability of the agent to ensure moral *Safety*.

Extensions of sub-characteristics include *Functional completeness*, *Functional correctness*,

and *Functional appropriateness*, broadening the scope to completeness of the model capabilities, accuracy of the result, and model relevance, respectively. *Time-behavior* is extended to encompass the model training time and Time-To-First-Token, emerging from the iterative nature of agents. Finally, *Transparency* was moved from *Interaction* capability to *Understanding*, which regroups all the sub-characteristics revolving around understanding the behavior of AI models. Table 4 gives the definitions for the new and updated quality (sub-)characteristics.

Table 4: Definitions of the new and updated quality (sub-)characteristics

Characteristic	Definition
Functional completeness	Degree to which training and evaluation data reflect real-world scenarios, and the set of functions covers all the specified tasks and intended users' objectives.
Functional correctness	Degree to which a product or system provides results achieving a level of accuracy exceeding specified requirements when used by intended users.
Functional appropriateness	Degree to which the training process and architecture of machine learning models are relevant for the task, and their functions facilitate the accomplishment of specified tasks and objectives.
Time-behavior	Degree to which the (partial-)response time, training time, and throughput rates of a product or system, when performing its functions, meet requirements.
Autonomy	Degree to which an AI agent can perform a set of actions without requiring human intervention.
Fairness	Degree to which an AI system can operate in an unbiased manner.
Ethics	Degree to which an AI system can ensure moral behaviors.
Understandability	This characteristic represents the degree to which an AI component behavior can be understood intuitively, in terms of features impact, its decision process, and the adequacy of this decision process.
Transparency	Degree to which adequate information about the AI system is communicated to users to understand its behavior intuitively.
Interpretability	Degree to which the impact of input features on the output can be understood.
Explainability	Degree to which the internal mechanics of an AI system can be explained in human terms.
Justifiability	Degree to which an AI system can demonstrate the output to be right or reasonable.
Sustainability	This characteristic represents the degree to which an AI system is sustainable in terms of its training impact, inference impact and actions to mitigate its impacts.
Training Impact	Degree to which the impact of the training process of an AI system is sustainable.
Inference Impact	Degree to which the impact of the inference process of an AI system is sustainable
Mitigation	Degree to which mitigation actions are put in place to reduce training and inference impacts.
Output Properties	This characteristic represents the degree to which the output of an AI system can meet different non-functional requirements.

Characteristic	Definition
Creativity	Degree to which an AI system can produce new, surprising, and valuable output.
Diversity	Degree to which an AI system can create a range of widely different valid result.
Conciseness	Degree to which an AI system can reduce its output size to the minimum required.
Repeatability	Degree to which an AI system with a fixed model can repeatedly produce the same or a similar output for a given input.
Reproducibility	Degree to which an AI system with a fixed model can repeatedly produce the same or a similar output for similar but different inputs.
Replicability	Degree to which an AI system can repeatedly produce the same or a similar output for similar but different inputs with a similar model.

4 KPI Specification

In this section, we introduce a domain-specific language (DSL) defined by analyzing the quality models summarized in the previous section and of the proposed extension, to support the formal specification of key performance indicators (KPIs) for AI agents. A DSL is a specialized language designed to model and address problems within a specific domain. It is typically composed of two principal components: abstract syntax and concrete syntax.

The abstract syntax defines the structural elements of the language and specifies how its primitives can be composed, independently of any particular representation. In contrast, the concrete syntax specifies one or more notational forms of the language, encompassing textual and/or graphical representations of its primitives.

The DSL proposed in this document enables the specification of both quantitative and qualitative KPIs by requiring, for each indicator, an explicit declaration of the computation formula and the corresponding value type. The following sections present a detailed description of the DSL's abstract syntax and concrete syntax, then present the list of KPIs we extracted from the literature and their formalization using our DSL. The complete implementation of the DSL and the associated KPIs definition file are available on GitLab: `https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/kpi-formula-dsl`

4.1 Abstract syntax

Figure 4 illustrates the abstract syntax of the proposed DSL as a metamodel that organizes the language constructs, their attributes, and the relationships among them. This metamodel is composed of two main parts: one to describe a quality model and the other to describe KPIs and their formulas.

The abstractions provided to model the quality model include the `QualityModel`, `QualityCharacteristic`, `QualityAttribute`, and `Metric` concepts. These concepts are used to model the quality model presented in section 3 and the metrics associated with the quality attributes. In the metamodel, the `Metric` concept refers to all types of metrics. However, metrics defined as part of the quality model represent metrics evaluated by the MOSAICO benchmarking platform (in opposition to composite metrics detailed later). In addition to the corresponding quality attribute, these metrics are also linked to a description of their provider.

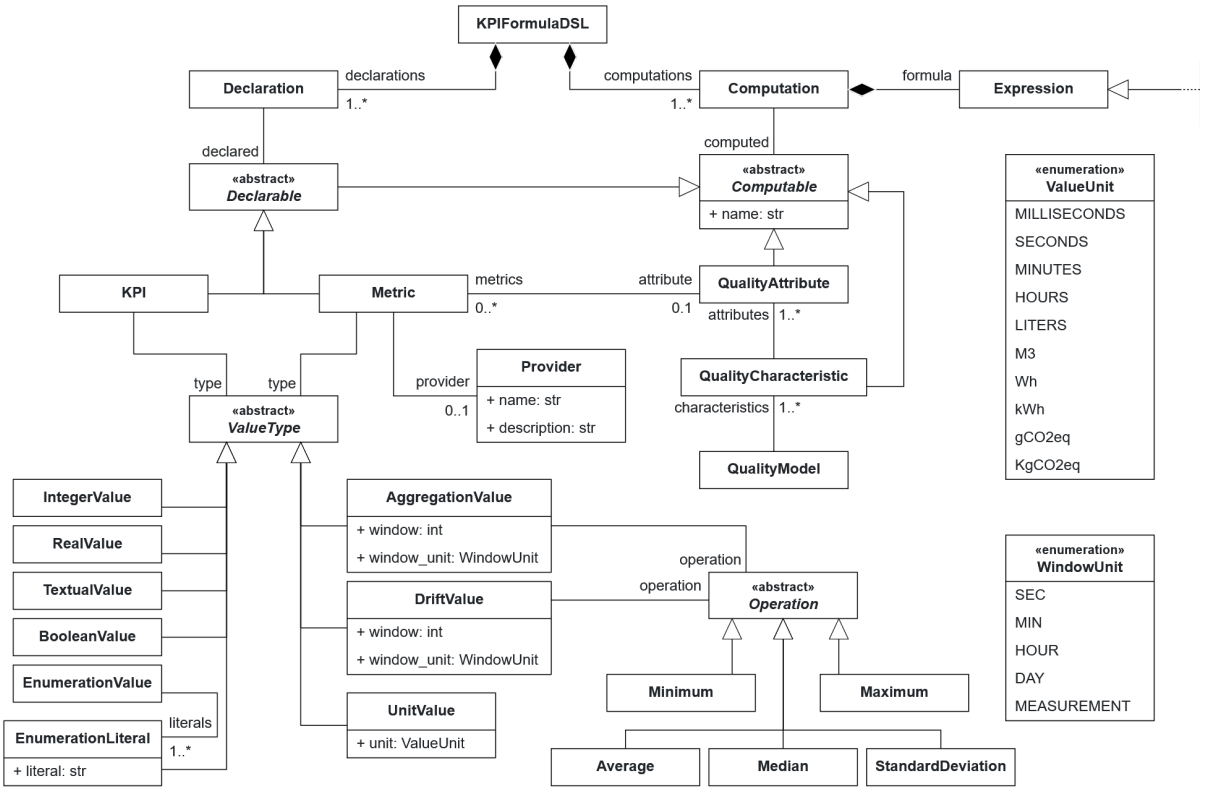


Figure 4: Metamodel of KPIFormulaDSL (Expressions are removed for clarity)

To model the KPIs and their formula, we defined the `KPIFormulaDSL` concept that represents the complete model for KPI specification and comprises two main components: a set of declarations and a set of computations.

A `Declaration` corresponds to the definition of either a KPI or a composite metric. Each declaration is characterized by a unique name and an associated `ValueType`. The `ValueType` concept specifies the data type to which a KPI or metric evaluates. The DSL supports both qualitative and quantitative data types. Qualitative types include textual, boolean, and enumeration values. Quantitative types include integers, real numbers, values associated with measurement units (e.g., liters, watt-hours, seconds), and more complex types such as `AggregationValue` and `DriftValue`.

An `AggregationValue` represents the aggregation of multiple measurements into a single value. It is defined by (i) an aggregation function (e.g., average, maximum) and (ii) a measurement window specifying the scope of data considered (e.g., the last 10 seconds or the last 20 observations). In contrast, a `DriftValue` captures the temporal evolution of a metric, enabling the assessment of changes in quality characteristics over time. This is particularly relevant for AI agents that adapt dynamically to evolving contexts. A `DriftValue` is also parameterized by an aggregation function and a measurement window. However, unlike `AggregationValue`, it computes the difference between two comparable aggregation results: the value obtained over the specified window and the value computed over the immediately preceding, non-overlapping window. For example, given a 10-second window and an average aggregation function, the drift corresponds to the average value between t_{-10} and t_0 minus the average between t_{-20} and t_{-10}.

The `Computation` concept constitutes the core mechanism of the DSL, as it associates a computable element with the formula that determines its value. The DSL is considered as

`Computable`: (1) any quality characteristic or attribute, enabling the calculation of its overall metric value, (2) composite metrics defined within the model, and (3) declared KPIs. Each computable element is defined through an `Expression`, which may consist of nested sub-expressions. The expression language supports arithmetic, comparison, and boolean operators, as well as literals corresponding to the supported data types and references to other computable elements.

Furthermore, we provide a set of core operators, summarized in Table 5. Among these, we find generic set operations such as *Cardinality*, *Filter*, *Map*, and *Reduce* operators. These generic operators are completed by specific set operators commonly used as composition patterns, such as: composite quality scores (via *Weighted sum*, *Sum* or *Average*), and conservative quality assessors (via *Minimum* or *Maximum*). These operators allow modeling KPIs composing the values of metrics reporting multiple results. An example of such metrics is the test results, which return a value for each test executed, indicating whether the test passed or not. In addition to set operations, the core operators also include a generic *Conditional disjunction* allowing, among other things, the definition of qualitative KPIs and a *Threshold* operator to define pass/fail quality gates. The combination of these expressions allows the creation of arbitrarily complex composite indicators.

Table 5: KPI formula DSL: supported operators. S denotes an expression evaluating to a set; S_i an element of set S; P a predicate (potentially parametrized); a, b arbitrary expressions; W a set of weights $W_i \in \mathbb{R}_{>0}$; $\theta \in [0, 1]$ a threshold; $v(a)$ the computed value of a.

Operator	Syntax	Semantics
Cardinality	$\lvert S \rvert$	$\lvert v(S) \rvert$
Filter	`filter` $P(S_i)$ `for` S_i `in` S	$\{S_i \in v(S) \mid P(S_i)\}$
Map	`map` $f(S_i)$ `for` S_i `in` S	$\{f(S_i) \mid S_i \in v(S)\}$
Reduce	`reduce` $f(S_i, accumulator)$ `starting` a `for` S_i `in` S	$u_{\lvert v(S) \rvert + 1}\, with\, u_0 = a\, and\, u_{i+1} = f(S_i,\, u_i)$
Average	`AVERAGE(`S`)`	$\frac{1}{n} \sum_{i=1}^{n} v(S_i)$
Weighted sum	`WEIGHTED_SUM(`S`,` W`)`	$\sum_{i=1}^{n} v(W_i) \cdot v(S_i)$
Sum	`SUM(`S`)`	$\sum_{i=1}^{n} v(S_i)$
Minimum	`MIN(`S`)`	$\min_i v(S_i)$
Maximum	`MAX(`S`)`	$\max_i v(S_i)$
Conditional Disjunction	`{` P `-> a;` `otherwise -> b` `}`	$\begin{cases} v(a) & \text{if } P \\ v(b) & \text{otherwise} \end{cases}$
Threshold gate	`THRESHOLD(`a`,` θ`)`	$\mathbf{1}[v(a) \geq \theta]$

4.2 Concrete syntax

Listing 1 presents an excerpt of the concrete syntax of the proposed DSL through a grammar representing the rules describing the syntactical structures composing the notation. This grammar is defined using the Xtext grammar notation. In this grammar, we named the grammar rules based on the meta-class represented by this rule (the *AggregationValue* rule, when parsed, will produce an `AggregationValue` instance).

At the beginning, we find the entry point of the notation, the *KPIFormulaDSL* rule. This rule is what we call a non-terminal rule, as it calls other rules as part of its definition. In this case, to successfully parse a *KPIFormulaDSL* we need to first parse a sequence of *Declaration* and then a sequence of *Computation*.

Parsing a *Computation* is simple, first we parse a reference to a `Computable` model element, then the equal sign, and finally the expression. In Xtext grammars, references to model elements are denoted using square brackets enclosing the type of the element (here *[Computable]*). Under the hood, this will try to parse an identifier and find a model element of the specified type that has a name attribute equal to the parsed identifier.

A *Declaration* is defined as parsing a single *Declarable*, which can either be a *KPI* or a *Metric* (composite in this case). Both of those rules are defined similarly, first parsing an identifier as the name of the declarable, then the "of type" keyword, followed by a *ValueType*. KPIs are identified from composite metrics by prepending the "KPI" keyword to the declaration.

Parsing a *ValueType* is defined as parsing one of its subtypes of value. For each subtype, we define a rule that parses the concrete type. In most cases, parsing a concrete type is simply matching the name of the type or unit in the case of *UnitValue*. However, *AggregationValue* and *DriftValue* require more information regarding the aggregation function to use and the window definition. For aggregation functions, the *Operation* rule makes a disjunction similar to *ValueType* and parses the concrete operation name. For windows, we first parse the size of the window representer by an integer value, and then its unit based on an enumeration of the possible units.

With this grammar, we can generate a parser that will recognize the specified notation and instantiate the corresponding model. Then, this model can be traversed to compute the described formulas and return KPI values to the MOSAICO benchmark platform.



```
KPIFormulaDSL returns KPIFormula:
    (declarations+=Declaration | computations+=Computation)*
;

Declaration returns Declaration: declared=Declarable;
Computation returns Computation: computed=[Computable] "=" formula=Expression
    ;

Declarable returns Declarable: KPI | Metric;
Metric returns Metric: name=ID "of" "type" type=ValueType;
KPI returns KPI: "KPI" name=ID "of" "type" type=ValueType;

ValueType returns ValueType:
  IntegerValue | RealValue | TextualValue | BooleanValue |
  EnumerationValue | AggregationValue | DriftValue | UnitValue
;
IntegerValue :      {IntegerValue} "int";
RealValue:          {RealValue} "real";
TextualValue:       {TextualValue} "text";
BooleanValue:       {BooleanValue} "bool";
EnumerationValue:   {EnumerationValue} "{" literals+=EnumerationLiteral+ "}";
EnumerationLiteral: {EnumerationLiteral} name=ID;
UnitValue:          {UnitValue} unit=ValueUnit;

AggregationValue returns AggregationValue:
  "aggregate"  operation=Operation "for" window=INT windowUnit=WindowUnit;
DriftValue returns DriftValue:
  "drift" "of" operation=Operation "for" window=INT windowUnit=WindowUnit;

Operation returns Operation: Minimum | Maximum | Average | Median |
    StandardDeviation;
Minimum: {Minimum} "Minimum";
Maximum: {Maximum} "Maximum";
Average: {Average} "Average";
Median:  {Median}  "Median";
StandardDeviation: {StandardDeviation} "Standard" "deviation";

enum WINDOW_UNIT:
    SEC="seconds"             |
    MIN="minutes"             |
    HOUR="hours"              |
    DAY="days"                |
    MEASUREMENT="measurements"
;

enum VALUE_UNIT:
    MILLI="milliseconds" |
    SEC="seconds"        |
    MIN="minutes"        |
    HOUR="hours"         |
    LITERS="liters"      |
    M3="cubic meter"     |
    Wh="Wh"              |
    kWh="kWh"            |
    gCO2eq="gCO2eq"      |
    kgCO2eq="kgCO2eq"
;
```

Listing 1: Excerpt of the DSL Xtext grammar

4.3 AI Agent KPIs

In this section, we detail the KPIs for AI agents extracted from the literature and existing AI agent benchmarks. For each of the six approaches studied, we detail the KPIs proposed, regrouping for each KPI its name, formula, semantics, and known baseline or threshold. This information is summarized in Table 6. Then, for each KPI detailed in these approaches, we translate the provided formula into our KPI Formula DSL, yielding a set of formalized KPIs that will be usable in the future MOSAICO platform. The complete formalization is presented in Listing 2.

Table 6: Evaluation Ingredients of the studied approaches

Metric Name	Mathematical Formula	Semantics & Description	Baselines & Thresholds
SWE-Agent Evaluation Ingredients [10]			
% Resolved (Pass@1)	$\frac{\text{Resolved Instances}}{\text{Total Instances}}$	The proportion of task instances for which all unit tests pass successfully after the agent's patch is applied.	12.47% on the full SWE-bench. 18.00% on Lite.
Pass@k	$\max(v(\text{Run}_1), \ldots, v(\text{Run}_k))$	Calculates the best resolution rate across k independent runs to evaluate variance.	For $k = 6$, the baseline pass rate on SWE-bench Lite is 32.67%.
Average Cost	$\frac{1}{N}\sum_{i=1}^{N} \text{Cost}_i$	The average API inference cost incurred across all *successfully resolved* instances.	Median cost of successful instances: $1.21. Max limit: $4
File Localization F1	$2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$	Evaluates whether the agent edited/removed the correct files compared to the human-written "gold patch".	Baseline F1 Score for GPT-4 Turbo: 59.05%.
Trajectory Length	$\sum$ Interaction Steps	The total number of interaction turns an agent takes to finish a task episode.	Median steps for successful instances: 12 turns.
Patch Size	$\sum$ Lines Added + $\sum$ Lines Removed	Tracks the size of the generated edits compared to human-written gold patches.	Median lines added for resolved instances: 3.0.
LoCoBench-Agent Evaluation Ingredients [36]			
Execution Success Rate (ESR)	$\frac{\lvert\text{unique_tools_used}\rvert}{\lvert\text{total_tools_available}\rvert} \times \frac{\lvert\text{successful_tool_calls}\rvert}{\lvert\text{Total Tool Calls}\rvert}$	Rewards agents that successfully explore diverse tools rather than repeatedly failing.	-
Multi-Session Memory (MMR)	$\alpha \cdot \text{ref_consistency} + \beta \cdot \text{topic_coherence}$	Evaluates context retention and factual consistency across conversation turns.	$\alpha = 0.6$ $\beta = 0.4$

Metric Name	Mathematical Formula	Semantics & Description	Baselines & Thresholds
Cross-File Consistency (CFC)	$\frac{1}{\lvert F_m \rvert} \cdot \sum (w_1 \cdot \text{naming}(f) + w_2 \cdot \text{imports}(f) + w_3 \cdot \text{style}(f))$	Measures cohesion across modified files (F_m) regarding naming, imports, and style.	$w_1 = 0.4$ $w_2 = 0.4$ $w_3 = 0.2$
Dependency Traversal (DT)	$\frac{\lvert \text{resolved_imports} \rvert}{\lvert \text{total_imports} \rvert} \times \frac{\lvert \text{valid_references} \rvert}{\lvert \text{total_references} \rvert}$	Assesses the accuracy of import resolutions and cross-file references.	-
Solution Usability (SU)	$\gamma \cdot \text{readability} + \delta \cdot \text{maintainability} + \epsilon \cdot \text{documentation}$	Evaluates the maintainability and readability of the generated code.	$\gamma = 0.4$ $\delta = 0.4$ $\epsilon = 0.2$
Runtime Efficiency (RE)	$1 - \left(\sum \frac{\text{complexity_penalty}(f)}{\lvert F_m \rvert} \right)$	Analyzes time complexity by identifying nested loops or inefficient algorithms.	-
Memory Efficiency (ME)	$1 - \left(\sum \frac{\text{memory_penalty}(f)}{\lvert F_m \rvert} \right)$	Evaluates space complexity (e.g., detecting large memory allocations).	-
Information Coverage (IC)	$\frac{\lvert F_r \cap \text{relevant_files} \rvert}{\lvert F_m \rvert + 1}$	Measures thorough codebase exploration before modification.	-
Long-Range Dependency (LRDR)	$\frac{\lvert \{ f \in F_r \text{ before modification} \} \rvert}{\lvert F_m \rvert}$ $with\ f \in F_m$	Evaluates proper "read-before-write" validation patterns across the codebase.	-
LCBA-Comp. Score	$\frac{\text{ESR} + \text{MMR} + \text{CFC} + \text{DT} + \text{SU}}{5}$	Aggregate arithmetic mean of the 5 comprehension metrics.	Specialist Threshold: ≥ 0.74
LCBA-Eff. Score	$\frac{\text{RE} + \text{ME} + \text{IC} + \text{LRDR}}{4}$	Aggregate arithmetic mean of the 4 efficiency metrics.*	Specialist Threshold: ≥ 0.63
DevBench Evaluation Ingredients [33]			
Pass@Example Usage (Env. Setup)	$\frac{\text{Successful Executions}}{\text{Total Example Usages}}$	Evaluates if the generated dependency file successfully sets up an environment where the repo's example code runs.	GPT-4-Turbo: 41.7%. GPT-3.5: 33.3%.
Pass@Accept. Test (Implementation)	$\sum \left(\text{Pass Rate}_i \cdot \frac{\text{Lines}_i}{\text{Total Lines}} \right)$	Evaluates if the generated codebase passes the predefined acceptance tests, weighted by repo difficulty (lines of code).	GPT-4-Turbo: 7.1%. DeepSeek-33B: 4.4%.
Pass@Unit Test (Implementation)	$\sum \left(\text{Pass Rate}_i \cdot \frac{\text{Lines}_i}{\text{Total Lines}} \right)$	Evaluates if the generated codebase passes the predefined unit tests, weighted by repo difficulty.	GPT-4-Turbo: 8.0%. DeepSeek-33B: 5.5%.

Metric Name	Mathematical Formula	Semantics & Description	Baselines & Thresholds
Oracle Test (Testing Tasks)	$\frac{1}{N}\sum_{i=1}^{N}\mathbf{1}[\text{Output}_i = \text{Oracle}_i]$	Evaluates whether the generated tests correctly assess standard implementation code by comparing actual vs. expected output.	GPT-4-Turbo: 29.2% (Acceptance), 36.5% (Unit).
Statement Coverage (Unit Testing)	$\frac{\text{Executed Statements}}{\text{Total Statements}}$	Provides a quantitative measure of test comprehensiveness across the codebase.	GPT-4-Turbo: 33.2% (overall), 66.3% (when executable).
Design Quality (Software Design)	LLM-as-a-Judge pairwise comparison against a baseline model.	Evaluates UML and architecture design for cohesion, decoupling, complexity, and adherence to the PRD.	GPT-4-Turbo win rate vs GPT-3.5: 97.9%.
CodeAssistBench Evaluation Ingredients [34]			
Correctness (Resolution Rate) (%)	$\frac{\text{Correct Instances}}{\text{Total Instances}}$	Evaluates if the agent's final answer satisfies all extracted conditions, is technically correct, and passes environment execution (Docker).	ChatGPT 4.1 Mini: 16.49% (Recent), 29.14% (All-Time).
Partially Correct / Incorrect Rate	$\frac{\text{Partial or Incorrect}}{\text{Total Instances}}$	Tracks the proportion of interactions that fail to fully satisfy the conditions or fail the Docker validation.	ChatGPT 4.1 Mini Incorrect: 53.09% (Recent).
Average CAB Turn (Conversation Length)	$\frac{1}{N}\sum_{i=1}^{N}\text{Turns}_i$	The average number of dialogue turns between the simulated user and the maintainer agent. Correct answers typically resolve in fewer turns.	ChatGPT 4.1 Mini (Correct): 2.94 turns (Recent), 2.35 turns (All).
Judge Alignment (Condition Accuracy)	$\frac{\text{Matching Labels}}{\text{Total Annotations}}$	Measures the reliability and accuracy of the automated LLM judge's extracted satisfaction conditions against human expert baseline.	Overall Accuracy: 86.27%. Inter-annotator Agreement: 82.39%.
GitTaskBench Evaluation Ingredients [35]			
Execution Completion Rate (ECR)	$\frac{\text{Completed Executions}}{\text{Total Tasks}}$	Measures basic operational capability: whether the agent successfully executes the repository and generates outputs in an acceptable format.	Top ECR: 57.41% (OpenHands + Claude 3.7).
Task Pass Rate (TPR)	$\frac{\text{Passed Tasks}}{\text{Total Tasks}}$	Quantifies actual task performance. Outputs must satisfy predefined domain-specific quality standards to pass.	Top TPR: 48.15% (OpenHands) to 62.96% (RepoMaster).

Metric Name	Mathematical Formula	Semantics & Description	Baselines & Thresholds
Alpha Value (α-score)	$\frac{1}{n}\sum_{i=1}^{n}[(T \cdot MV \cdot Q) - C]$	Quantifies the economic benefit/net gain of agent performance compared to human labor costs and API token usage.	Highly variable. High-MV tasks yield up to \$95; low-MV tasks require costs $<$\$1-\$2 to stay positive.
Quality Factor (Q)	Majority Vote $\in \{0, 0.25, 0.5, 0.75, 1\}$	Measures how closely the agent's output approximates human-produced ground truth via expert assessment.	1 = Indistinguishable from human; 0 = Far below human.
Operational Cost (C)	$\sum(\text{Tokens} \cdot \text{Price})$	Total API inference cost incurred during the end-to-end repository setup and execution.	DeepSeek-V3 generally yields the highest cost-to-performance ratio.
SWE-Bench Evaluation Ingredients [30]			
% Resolved (Pass Rate)	$\frac{\text{Resolved Instances}}{\text{Total Instances}}$	An instance is only considered resolved if the patch applies successfully and *all* F2P and P2P tests pass.	Claude 2 (BM25): 1.96%. Claude 2 (Oracle): 4.80%.
% Applied (Patch Rate)	$\frac{\text{Applied Patches}}{\text{Total Instances}}$	Measures the LLM's ability to generate strictly formatted Unix patch files that can be injected into the codebase without syntax errors.	Claude 2 (BM25): 43.07%. GPT-4 (Oracle): 48.65%.
Resolution Categorization	Category $\in$ {Resolved, Breaking, Partial, Regression, No-Op}	Fine-grained categorization: e.g., "Breaking Resolved" means F2P tests passed, but some previously working P2P tests failed.	-
Patch Edit Scope	$\sum \text{Lines}_{\text{add}} + \sum \text{Lines}_{\text{rem}}$	Evaluates the conciseness and scope of the LLM's edit compared to the human "gold" patch.	Claude 2 avg lines: 27.2. Gold avg lines: 61.6.
Cyclomatic Complexity	$E - N + 2P$ (McCabe's formula via Radon package)	Evaluates the logical complexity, efficiency, and readability of the integrated code blocks.	-

```
//>---------------------- SWE-Agent  Evaluation Ingredients ----------------------<//
KPI PassAt1 of type real
KPI PassAtK of type aggregate Maximum for 10 measurements // Here K = 10
KPI AverageCost of type real
KPI FileLocalizationF1 of type real
KPI TrajectoryLength of type int
KPI PatchSize of type int

PassAt1 = ResolvedInstances / TotalInstances
PassAtK = PassAt1
AverageCost = AVERAGE(Costs)
FileLocalizationF1 = 2 * ((Precision*Recall) / (Precision+Recall))
TrajectoryLength = SUM(InteractionSteps)
PatchSize = SUM(LinesAdded) + SUM(LinesRemoved)

//>------------------- LoCoBench-Agent  Evaluation Ingredients -------------------<//

KPI ESR of type real
KPI MMR of type real
KPI CFC of type real
KPI DT of type real
KPI SU of type real

ESR = (unique_tools_used / total_tools_available) *
      (successful_tool_calls / total_tool_calls)
CFC = reduce accumulator + WEIGHTED_SUM([naming, imports, style], [0.4, 0.4, 0.2])
      starting 0.0 for f in FilesModified
MMR = 0.6 * ref_concistency + 0.4 * topic_coherence
DT = (resolved_imports/total_imports) * (valid_references/total_references)
SU = 0.4 * readability + 0.4 * maintainability + 0.2 * documentation

KPI RE of type real
KPI ME of type real
KPI IC of type real
KPI LRDR of type real

RE = 1 - AVERAGE(map complexity_penalty for f in FilesModified)
ME = 1 - AVERAGE(map memory_penalty for f in FilesModified)
IC = | FilesRead and relevant_files | / (| FilesModified | + 1)
LRDR = reduce accumulator + | [f] and ReadBeforeModification |
       starting 0.0 for f in FilesModified
// |[elem] and Set| = 1 if elem in Set, 0 otherwise

KPI LCBA_comp of type real
KPI LCBA_eff of type real

LCBA_comp = ESR + CFC + MMR + DT + SU / 5
LCBA_eff = RE + ME + IC + LRDR / 4

//>----------------------- DevBench Evaluation Ingredients -----------------------<//

KPI PassAtExampleUsage of type real
KPI PassAtAcceptTest of type real
KPI PassAtUnitTest of type real
KPI OracleTest of type real
KPI StatementCoverage of type real

PassAtExampleUsage = | SuccessfulExecution | / TotalExampleUsage
PassAtAcceptTest = WEIGHTED_SUM(AcceptanceLines, AcceptancePassRate)
                   / AcceptanceTotalLines
PassAtUnitTest = WEIGHTED_SUM(UnitLines, UnitPassRate)
                 / UnitTotalLines
OracleTest = AVERAGE(map {
  OracleI == OutputI -> 1;
  otherwise          -> 0
} for OracleI in Oracle, OutputI in  Output)
StatementCoverage = ExecutedStatements / TotalStatements

KPI DesignQuality of type real
DesignQuality = 0 // No formula specified
```

```
//>---------------- CodeAssistBench (CAB)  Evaluation Ingredients ----------------<//

KPI ResolutionRate of type real
KPI IncorrectRate of type real
KPI AverageCABTurn of type real
KPI JudgeAlignment of type real

ResolutionRate = CorrectInstances / TotalInstances
IncorrectRate = PartialOrIncorrect / TotalInstances
AverageCABTurn = AVERAGE(Turns)
JudgeAlignment = | JudgeLabels and HumanLabels | / TotalAnnotations

//>---------------------- GitTaskBench Evaluation Ingredients ----------------------<//

KPI ECR of type real
KPI TPR of type real
KPI QualityFactor of type real
KPI OperationalCost of type real
KPI AlphaScore of type real

// Intermediate Composite Metrics
Vote0 of type int
Vote25 of type int
Vote50 of type int
Vote75 of type int
Vote100 of type int
MaxBucketSize of type int

Vote0   = | filter vote == 0.00 for vote in HumanVotes |
Vote25  = | filter vote == 0.25 for vote in HumanVotes |
Vote50  = | filter vote == 0.50 for vote in HumanVotes |
Vote75  = | filter vote == 0.75 for vote in HumanVotes |
Vote100 = | filter vote == 1.00 for vote in HumanVotes |
MaxBucketSize = MAX([Vote0, Vote25, Vote50, Vote75, Vote100])

// KPI Formulas
ECR = | CompleteExecution | / TotalTasks
TPR = | PassedTasks | / TotalTasks
QualityFactor = {
  Vote0 == MaxBucketSize      -> 0.0  ;
  Vote25 == MaxBucketSize     -> 0.25 ;
  Vote50 == MaxBucketSize     -> 0.50 ;
  Vote75 == MaxBucketSize     -> 0.75 ;
  Vote100 == MaxBucketSize    -> 1.0  ;
  otherwise                   -> 0.50
}
OperationalCost = WEIGHTED_SUM(Tokens, Price)

//>---------------------- SWE-Bench  Evaluation Ingredients ----------------------<//

KPI PassRate of type real
KPI PatchRate of type real
KPI ResolutionCategorization of type {Resolved Breaking Partial Regression NoOp}
KPI PatchEditScope of type int
KPI CyclomaticComplexity of type int

PassRate = ResolvedInstances / TotalInstances
PatchRate = AppliedPatches / TotalInstances
ResolutionCategorization = { // domain specific conditions to be defined
  false -> ResolutionCategorization.Resolved;
  false -> ResolutionCategorization.Breaking;
  false -> ResolutionCategorization.Partial;
  false -> ResolutionCategorization.Regression;
  otherwise -> ResolutionCategorization.NoOp
}
PatchEditScope = SUM(LinesAdded) + SUM(LinesRemoved)
CyclomaticComplexity = E - N + 2 * P
```

Listing 2: Evaluation ingredients of the studied approaches formalized using the KPI Formula DSL

5 Specification and use of MOSAICO KPIs

Section 4 introduced the *KPI Formula DSL* and its formal underpinnings: the metamodel, the Xtext grammar (Listing 1), and the complete set of KPI definitions extracted from six representative agent benchmarks (Listing 2). This section closes the loop between formal specification and operational practice. It describes how the language tooling is distributed and installed, how quality engineers interact with it, both interactively in an Eclipse-based editor and programmatically from within the MOSAICO evaluation backend, and how it integrates into the broader MOSAICO benchmarking architecture to enable end-to-end, reproducible quality assessment of AI agents. The complete implementation of the DSL and the associated KPI definition file are publicly available on the MOSAICO project GitLab repository: `https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/kpi-formula-dsl`.

The remainder of this section is structured as follows: Section 5.1 describes the distribution and installation of the DSL tooling, Section 5.2 illustrates interactive authoring support in Eclipse, Section 5.3 presents its programmatic use within the MOSAICO evaluation backend, and Section 5.4 explains how KPI computation integrates into the overall MOSAICO benchmarking architecture.

5.1 Distribution and Installation

The KPI Formula DSL is distributed in two complementary forms, targeting two distinct usage scenarios: interactive model authoring inside an Eclipse IDE, and programmatic parsing and evaluation inside Java-based backend services.

5.1.1 Eclipse Plugin

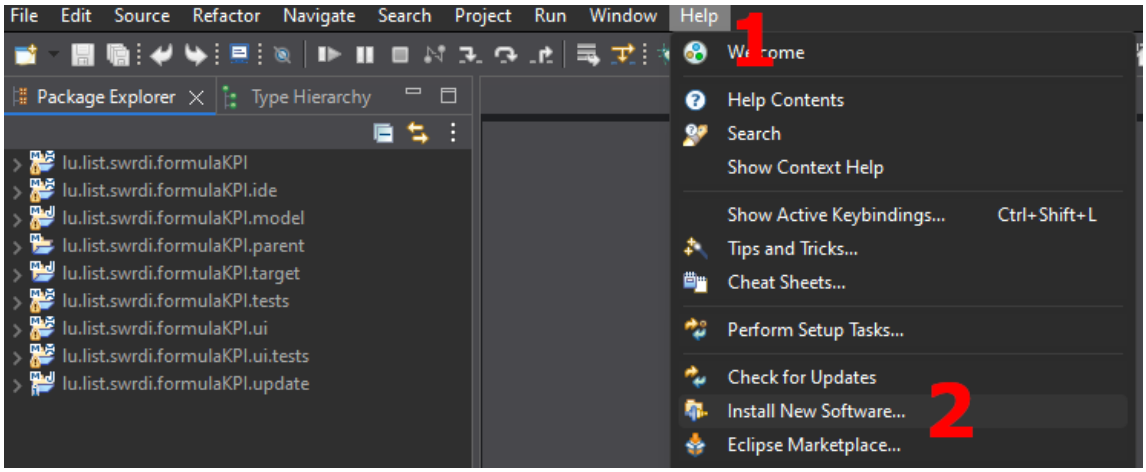
The Eclipse plugin provides a full-featured editor for `.kpi` specification files, including syntax highlighting, background validation, content assist, outline view, and quick fixes. It is packaged as an Eclipse update site and can be installed into any Eclipse IDE 2023-12 or later as follows.

1. Clone the repository and build the update site with Apache Maven (requires Java 21 and Maven 3.9+):

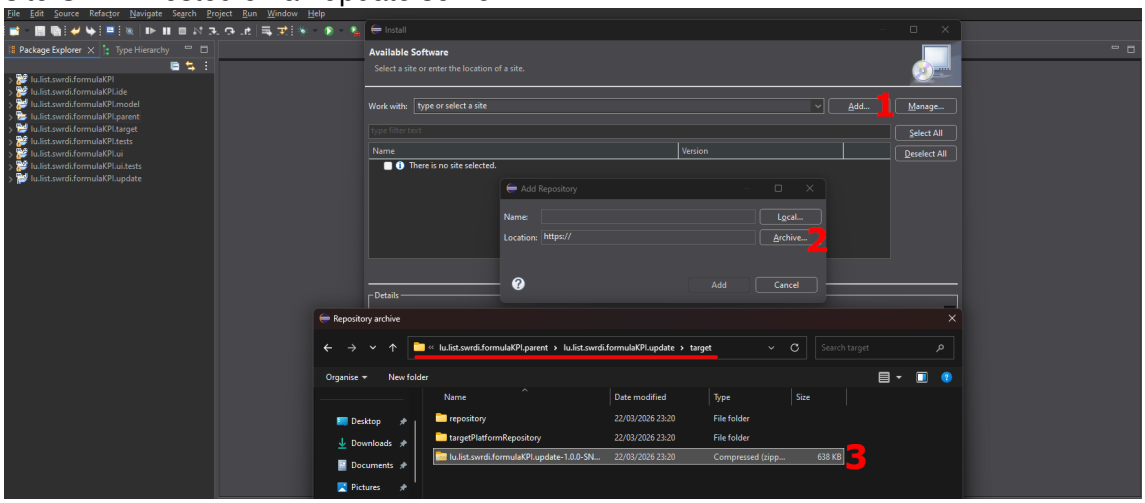
```
git clone https://gitlab.eclipse.org/eclipse-research-labs/
          mosaico-project/kpi-formula-dsl
cd kpi-formula-dsl
mvn clean package
```

   The update site is produced at `lu.list.swrdi.formulaKPI.update/target/repository/`.

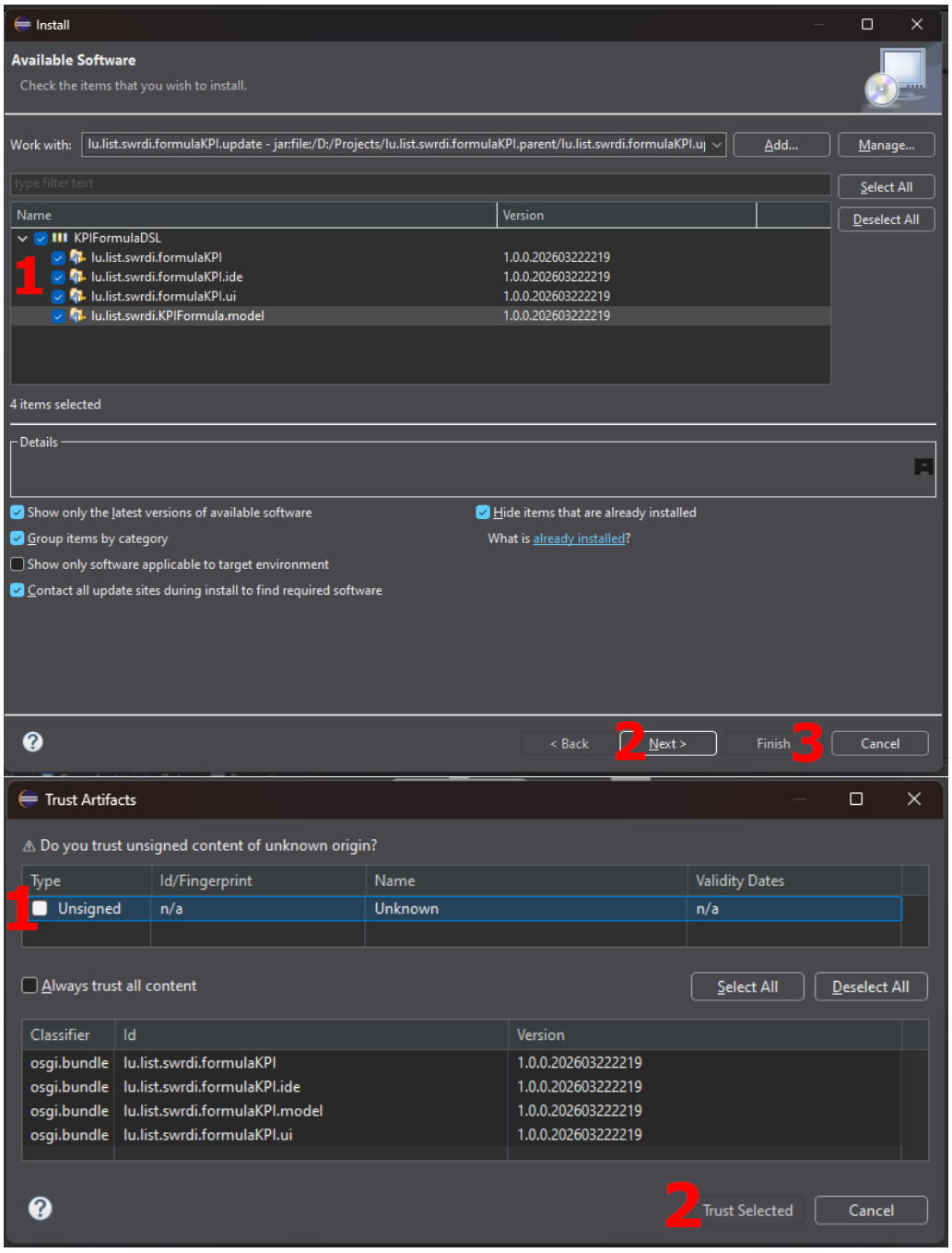
2. In Eclipse, navigate to **Help** → **Install New Software...**



3. Click **Add...** → **Archive...** and select the built update site archive, or enter the remote site URL if hosted on an update server.



4. Select the feature `lu.list.swrdi.formulaKPI.feature`, accept the licence, and complete the wizard.



5. Restart Eclipse when prompted.

After restarting, Eclipse automatically associates the editor with all files bearing the `.kpi` extension. The installation registers the following Eclipse extension points: syntax colouring, content assist, background validation, quick fixes, outline view, and rename refactoring.

5.1.2 Maven Dependency (Standalone Mode)

For integration into Java applications that do not run inside an Eclipse environment, such as the MOSAICO Repository that has a continuous benchmarking pipeline, the core DSL artifacts are consumed as plain Maven dependencies. Two modules are required, as shown in Listing 3.

```xml
<!-- EMF metamodel: provides the AST node types -->
<dependency>
    <groupId>lu.list.swrdi.formulaKPI</groupId>
    <artifactId>lu.list.swrdi.formulaKPI.model</artifactId>
    <version>1.0.0-SNAPSHOT</version>
</dependency>

<!-- Core DSL: parser, evaluator, and quality model -->
<dependency>
    <groupId>lu.list.swrdi.formulaKPI</groupId>
    <artifactId>lu.list.swrdi.formulaKPI</artifactId>
    <version>1.0.0-SNAPSHOT</version>
</dependency>
```

Listing 3: Maven dependencies for programmatic use of the KPI Formula DSL.

The artifact `lu.list.swrdi.formulaKPI.model` provides the EMF-generated Java classes representing the Abstract Syntax Tree (AST) nodes of a parsed KPI specification (`KPIFormula`, `Metric`, `KPI`, all `Expression` subtypes, and so on). The artifact `lu.list.swrdi.formulaKPI` provides the Xtext grammar, the ANTLR-generated parser, the AST interpreter (`KPIFormulaEvaluator`), and the quality model utilities. Both artifacts transitively depend on Xtext 2.41.0, EMF Ecore 2.42.0, and ANTLR 3.2; these are resolved automatically by Maven. No Eclipse runtime is required: the Xtext infrastructure is bootstrapped via a standalone setup class (`KPIFormulaDSLStandaloneSetup`), making the parse-and-evaluate pipeline suitable for any Java 21 environment.

5.2 Authoring KPI Specifications in Eclipse

Once the plugin is installed, a quality engineer creates a new file with the `.kpi` extension in any Eclipse project. The editor activates immediately with full language support.

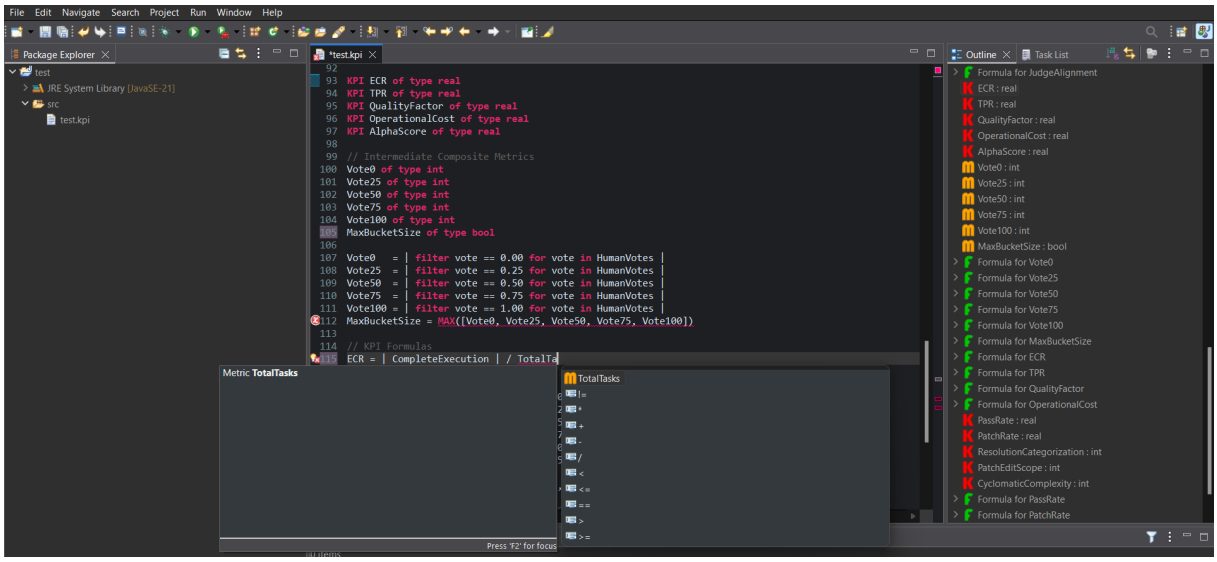


Figure 5: Eclipse IDE with the DSL tooling (outline on the right, validation in the gutter and underlined, and content assist providing metric name completion)

Background validation: On each file modification, the editor runs the Xtext validator in the

background and reports errors directly in the gutter. Typical diagnostics include references to metric names that are neither declared in the file nor pre-registered in the quality model, type mismatches between the declared value type and the inferred type of the formula, and syntactic violations. For instance, referencing a metric that is not in scope produces a structured error message.

Content assist: Pressing `Ctrl+Space` at any point in the specification triggers context-sensitive proposals. The editor suggests metric names pre-registered in the MOSAICO quality model, including all the evaluation ingredients formalized in Listing 2 such as `ResolvedInstances`, `TotalInstances`, `Precision`, `Recall`, or `FilesModified`, as well as KPI names already declared in the current file, type keywords, and operator names. This significantly reduces authoring errors and makes the quality model namespace directly discoverable from the editor.

```
105 MaxBucketSize of type int
106
107 Vote0   = | filter vote == 0.00 for vote in HumanVotes |
108 Vote25  = | filter vote == 0.25 for vote in HumanVotes |
109 Vote50  = | filter vote == 0.50 for vote in HumanVotes |
110 Vote75  = | filter vote == 0.75 for vote in HumanVotes |
111 Vote100 = | filter vote == 1.00 for vote in HumanVotes |
112 MaxBucketSize = MAX([Vote0, Vote25, Vote50, Vote75, Vote100])
113
114 // KPI Formulas
115 ECR = | CompleteExecution | / TotalTasks
116 TPR = | PassedTasks | / TotalTasks
117 QualityFactor = {
118     Vote0 == MaxBucketSize      -> 0.0  ;
119     Vote25 == MaxBucketSize     -> 0.25 ;
```

Figure 6: Example of linking between a composite metric reference and its definition

Outline view and hyperlinking. The outline view (*Window* → *Show View* → *Outline*) displays the list of declarations and computations in the current specification, enabling rapid navigation across larger KPI catalogs. Holding `Ctrl` and clicking on any metric or KPI reference in a formula navigates directly to its declaration — either within the same file or, for pre-registered names, to the corresponding element in the quality model resource.

```
MaxBucketSize = MAX([Vote0, Vote25, Vote50, Vote75, Vote100])

// KPI Formulas
ECR = | CompleteExecution | / TotalTasks
TPR = | PassedTasks | / TotalTasks
QualityFactor = {
    Vote0 == MaxBucktSize       -> 0.0  ;
    Vote25 ==
    Vote50 ==
    Vote75 ==
    Vote100 =
    otherwise
}
Couldn't resolve reference to Computable 'MaxBucktSize'.
1 quick fix available:
Change to 'MaxBucketSize'
Press 'F2' for focus
```

Figure 7: Example of proposed quick fix for a missing letter in a metric's reference

Quick fixes. For a subset of detected errors, the editor offers one-click quick fixes: unknown metric names that closely match a known name trigger a rename suggestion; missing `otherwise` branches in conditional formulas are offered as completions.

5.3 Programmatic Use from the MOSAICO Evaluation Backend

The KPI Formula DSL exposes a clean Java API designed for integration into automated evaluation pipelines. Listing 4 illustrates the complete parse-and-evaluate lifecycle for the SWE-Agent `PassAt1` and `AverageCost` KPIs defined in Listing 2. Line 1 instantiates the parser. The constructor bootstraps the Xtext infrastructure and loads the MOSAICO quality model resource (quality characteristics, sub-attributes, and pre-registered metric names). Lines 2-5 register any runtime metric keys produced by the benchmarking platform that are not in the built-in quality model. Lines 6-13 parse the KPI formula specification (or a fragment thereof). The `DslParseResult` DTO returned at line 6 is immutable. On failure it carries a list of `DslValidationError` objects, each holding `line`, `column`, `message`, and `severity` (`ERROR` / `WARNING` / `INFO`), making errors straightforward to surface in REST responses or audit logs. Lines 14-18 handle parse or validation errors. On success, the `getReferencedMetrics()` (Line 19) method returns the precise set of input names consumed by the formula; this set is used by the MOSAICO orchestration layer to perform targeted metric collection rather than materialising the entire trace. Finally, lines 21-24 evaluate the formula with metric values collected at runtime. The `KPIFormula` functional interface exposes a single method, `double evaluate (Map<?, Double> metricValues)`, accepting either `String` keys (metric names) or type-safe marker classes. The underlying interpreter traverses the AST using Xtend dispatch methods, one per node type, and returns a typed `Value`, either a scalar `DoubleValue`, a `ListValue`, or a `NullValue` for absent inputs. The `NullValue` semantics ensure that a missing metric at evaluation time does not raise an exception, but instead propagates a neutral element through the formula, allowing partial evaluations to complete gracefully.

5.4 Integration with the MOSAICO Benchmarking Architecture

The KPI Formula DSL does not operate in isolation: it is the computation layer of the MOSAICO benchmarking architecture that will be completely defined in deliverable D2.3, which will provide end-to-end infrastructure for the systematic assessment of AI agents registered in the MOSAICO repository. The architecture will operate through the following pipeline:

1. **Orchestration.** The orchestration layer schedules and executes evaluation runs against AI agents registered in the MOSAICO repository. For each agent and benchmark task, it manages execution sandboxes, timeout enforcement, and run reproducibility, ensuring that every evaluation episode is logged with sufficient provenance to support later auditing.

2. **Trace collection and normalisation.** Agent execution traces, comprising tool invocations, code edits, test outcomes, API calls, and token-level cost records, are collected through an external observability layer and normalised into a common trace schema. This decouples the evaluation infrastructure from the internal implementation details of any particular agent framework.

3. **Base metric computation.** A configurable catalogue of base metrics is computed over the normalised traces.

```
KPIFormulaParser parser = new KPIFormulaXtextParser();
parser.registerMetricKeys(
    Set.of("ROUGE", "BLEU", "F1_Score",
           "Precision", "Recall"));
DslParseResult result = parser.parse("""
    KPI Overall_Quality of type real

    Overall_Quality =  WEIGHTED_SUM(
                         [ROUGE, BLEU, Precision, Recall, F1_Score],
                         [0.25,  0.20, 0.15,      0.10,   0.15]
                         )
    """);
if (!result.isSuccess()) {
    result.getErrors().forEach(e ->
        log.error(e.getFormattedMessage()));
    return;
}
Set<String> needed = result.getReferencedMetrics();
KPIFormula formula = result.getFormula();
double Overall_Quality = formula.evaluate(Map.of(
    "ROUGE",      0.384,
    "BLEU",       0.326,
    "F1_Score",   0.698,
    "Precision",  0.712,
    "Recall",     0.685));
```

Listing 4: Parsing and evaluating the `PassAt1` and `AverageCost` KPIs programmatically from the MOSAICO evaluation backend.

4. **KPI computation via the DSL.** Trace base metric values are aggregated into KPI scores by evaluating the formula expressions defined in the KPI Formula DSL. The `KPIFormula-XtextParser` parses the KPI catalog (such as Listing 2) once at startup, producing a set of compiled `KPIFormula` instances. At evaluation time, the orchestration layer calls `formula.evaluate(...)` for each agent run, passing the metric values retrieved from the metric store. The separation between formula *definition* and formula *execution* is a deliberate architectural choice: KPI specifications can be updated, extended, or replaced without modifying the evaluation engine, and the same engine can serve multiple concurrent benchmarking campaigns.

5. **Result persistence and semantic search.** Computed KPI values, together with their provenance metadata, are persisted in a structured results store. This store is further indexed to support semantic search over agent quality profiles, enabling Software Engineers to query the MOSAICO repository not only by agent name or task type, but also by quality characteristics — for example, to retrieve agents with high `PassAt1` and low `AverageCost`, or to compare sustainability profiles across agents belonging to the same benchmark family.

The KPI Formula DSL contributes to this architecture two properties that would be difficult to achieve with ad-hoc scripting: *i) auditability*: every KPI score is traceable to an explicit, human-readable formula stored in a versioned file, and *ii) extensibility*: new KPIs are added by appending declarations and computations to the catalog file, without touching the evaluation engine or the metric store schema. Both properties are essential for a research infrastructure

that must evolve continuously as new benchmarks, agent architectures, and quality dimensions emerge within the MOSAICO project lifetime.

5.5 Illustrative Usage Scenario

To illustrate how the DSL can be used, we describe a comparative evaluation of three GitHub repository description agents whose task is to generate concise natural-language summaries of software projects from their `README` files [57]. The evaluation uses the `ause_train` dataset, which supplies reference descriptions for each repository [57]. The three agents differ in the underlying language model and inference parameters: a *Baseline* agent (llama3.2:3b, temperature 0), a *Creative* variant (llama3.1:8b, temperature 0.8), and a *Deterministic* variant (mistral:7b, temperature 0.2). Figure 8 shows an excerpt of the three agents we involved in this illustrative example.

A single `Benchmark` entity referencing the `ause_train` [58] dataset is created and associated with all three agents. Four `PerformanceKPI` instances are then registered, each backed by a DSL expression persisted in the corresponding `KPISpecification.dsl_text` column:

```
KPI Overall_Quality of type real
KPI Text_Similarity of type real
KPI Balanced_F1     of type real
KPI Quality_Gate    of type int

Overall_Quality =  WEIGHTED_SUM([ROUGE, BLEU, PRECISION, RECALL, F1_SCORE],
                                [0.25 , 0.20, 0.15     , 0.10  , 0.15])
Text_Similarity =  AVERAGE([ROUGE, BLEU])
Balanced_F1 =      WEIGHTED_SUM([PRECISION, RECALL, F1_SCORE],
                                [0.3      , 0.3   , 0.4])
Quality_Gate =     THRESHOLD(F1_SCORE, 0.40)
```

The *Overall Quality* KPI aggregates all six metric families with differential weights, emphasising lexical overlap (ROUGE, BLEU) over classification metrics. *Text Similarity* isolates overlap quality. *Balanced F1* is a recall-precision composite centered on the F_1 harmonic mean. *Quality Gate* is a binary acceptance criterion that returns 1 if and only if the F_1 score clears the minimum threshold.

Three independent evaluation runs are submitted to the MOSAICO evaluation backend, one per agent under assessment, each referencing a shared benchmark identifier and the corresponding execution trace as described in Section 5.4. The orchestrator transitions each run to an active state, retrieves the associated traces from the telemetry layer (i.e., Langfuse[2]), invokes all registered metric providers for each trace, and persists the results as metric snapshots. Once all traces have been processed, metric values are aggregated per agent, and the KPI DSL formulas are compiled and evaluated against the resulting metric vector. Table 7 reports the run-level metric and KPI values obtained for each agent.

As a concrete verification, the *Text Similarity* KPI for the Baseline agent is computed directly from the aggregated metric vector as:

$$\mathrm{KPI}^{\text{base}}_{\text{sim}} = \frac{\overline{\text{ROUGE}} + \overline{\text{BLEU}}}{2} = \frac{0.384 + 0.326}{2} = 0.355 \tag{1}$$

in exact agreement with the value produced by the DSL evaluator. All three agents clear the *Quality Gate* threshold ($\overline{\text{F1}} \geq 0.40$), confirming that the DSL-driven threshold check behaves as specified. The *Deterministic* variant achieves the highest *Overall Quality* score (0.513), while the *Creative* variant underperforms on all composite indicators, suggesting that higher sampling temperature degrades output fidelity on this task.

[2] `https://langfuse.com/`

sum_agent.py > ...

```python
#!/usr/bin/env python3
"""
Baseline: Uses llama3.2:1b - a good balance of speed and quality for summarization tasks
"""
import os
import json
import argparse
from dataclasses import dataclass
from typing import Any, Dict, List

from langfuse import Langfuse

from ollama_agent import AgentConfig, build_agent

# Specific configuration for this variant
VARIANT_NAME = "variant1_llama32_1b_fast"
VARIANT_MODEL = "llama3.2:1b"
VARIANT_OPTIONS = {
    "temperature": 0.3,
    "top_p": 0.9,
    "top_k": 40,
    "num_predict": 256,
}


@dataclass
class RunCfg: ...

def load_config(path: str) -> Dict[str, Any]: ...

def resolve_env(default: str, env_key: str, ignore_env: bool = False) -> str: ...

def make_cfg(j: Dict[str,Any], split: str) -> RunCfg: ...

def main(): ...

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

sum_agent_variant1.py > ...

```python
#!/usr/bin/env python3
"""
Variant 1: Uses llama3.2:1b - a good balance of speed and quality for summarization tasks
"""
import os
import json
import argparse
from dataclasses import dataclass
from typing import Any, Dict, List

from langfuse import Langfuse

from ollama_agent import AgentConfig, build_agent

# Specific configuration for this variant
VARIANT_NAME = "variant1_llama32_1b_fast"
VARIANT_MODEL = "llama3.2:1b"
VARIANT_OPTIONS = {
    "temperature": 0.3,
    "top_p": 0.9,
    "top_k": 40,
    "num_predict": 256,
}


@dataclass
class RunCfg: ...

def load_config(path: str) -> Dict[str, Any]: ...

def resolve_env(default: str, env_key: str, ignore_env: bool = False) -> str: ...

def make_cfg(j: Dict[str,Any], split: str) -> RunCfg: ...

def main(): ...

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

sum_agent_variant2.py > ...

```python
#!/usr/bin/env python3
"""
Variant 2: Uses qwen2.5:0.5b - an ultra-compact and fast model
"""
import os
import json
import argparse
from dataclasses import dataclass
from typing import Any, Dict, List

from langfuse import Langfuse

from ollama_agent import AgentConfig, build_agent

# Specific configuration for this variant
VARIANT_NAME = "variant2_qwen25_05b_compact"
VARIANT_MODEL = "qwen2.5:0.5b"
VARIANT_OPTIONS = {
    "temperature": 0.2,
    "top_p": 0.8,
    "top_k": 20,
    "num_predict": 256,
    "repeat_penalty": 1.1,
}

@dataclass
class RunCfg: ...

def load_config(path: str) -> Dict[str, Any]: ...

def resolve_env(default: str, env_key: str, ignore_env: bool = False) -> str: ...

def make_cfg(j: Dict[str,Any], split: str) -> RunCfg: ...

def main(): ...

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

Figure 8: Three summary agents that should be evaluated

Table 7: Metric and KPI values obtained for three agent configurations on a shared evaluation dataset. KPI values are computed from run-level metric aggregates via the DSL formulas defined above.

Metric / KPI	Baseline	Creative	Deterministic
ROUGE	0.384	0.352	0.410
BLEU	0.326	0.299	0.352
Accuracy	0.651	0.599	0.682
Precision	0.712	0.654	0.746
Recall	0.685	0.612	0.701
F1	0.698	0.633	0.723
Overall Quality	0.482	0.439	0.513
Text Similarity	0.355	0.325	0.381
Balanced F1	0.698	0.633	0.723
Quality Gate	1.0	1.0	1.0

The example highlights three key properties of the approach. First, the DSL formulas are defined once and applied identically across all agent configurations: no agent-specific evaluation logic is required. Second, the full audit trail, i.e., individual metric values recorded for every trace and every provider, remains available for post-hoc inspection independently of the aggregate KPI scores. Third, the *Quality Gate* KPI can be directly integrated into a continuous integration pipeline to enforce minimum acceptance criteria across agent releases, decoupling the pass/fail decision from the specific model or inference parameters employed.

6 Conclusion and future work

This deliverable has presented a comprehensive framework for the systematic assessment of AI agent quality within the MOSAICO project. Building upon the foundations of ISO/IEC 25010 and its AI-oriented extension ISO/IEC 25059, we proposed an extended quality model specifically tailored to the distinctive characteristics of AI agents. By integrating additional quality attributes identified through an extensive literature review and practitioner surveys, the proposed model addresses the dynamic, adaptive, and context-aware nature of AI agents, thereby filling a gap not fully covered by existing standards.

In addition, this report introduced a domain-specific language (DSL) designed to operationalize the proposed quality model. The DSL enables the formal declaration and computation of both quantitative and qualitative KPIs, supporting advanced constructs such as aggregation over temporal windows and drift analysis. These mechanisms allow not only the measurement of static properties but also the monitoring of their evolution over time—an essential requirement for AI agents operating in continuously changing environments.

Finally, this report presents the initial architecture of the MOSAICO benchmarking framework, providing a concrete computational infrastructure to operationalize the KPIs computations defined with the proposed DSL. This architecture ensures the separation between definition and

execution of formulas, as they are authored independently, provides a metric catalogue open to extension and closed to modification, and decouples the benchmarking engine from agent deployment details. Overall, this architecture allows: *(i)* orchestrating the execution of evaluation runs against registered AI agents; *(ii)* collecting and normalizing agent execution traces through an external observability layer; *(iii)* computing a configurable catalogue of base metrics over those traces; *(iv)* aggregating per-trace metric values into KPI scores through a formula expressions defined in a domain-specific language (DSL); and *(v)* Persist the benchmarking results and leverage them to enhance semantic search mechanisms.

Together, they provide a coherent and extensible foundation for defining, computing, and monitoring AI agent quality in a structured and reproducible manner. This contribution strengthens the methodological and technical basis of MOSAICO's approach, ensuring that quality requirements can be systematically translated into measurable indicators and integrated into runtime monitoring. Future work will focus on the full-scale operationalization of the MOSAICO benchmarking framework that is the subject of deliverable D2.3.

References

[1] Ahmed E. Hassan, Dayi Lin, Gopi Krishnan Rajbahadur, Keheliya Gallaba, Filipe Roseiro Cogo, Boyuan Chen, Haoxiang Zhang, Kishanthan Thangarajah, Gustavo Oliva, Jiahuei (Justina) Lin, Wali Mohammad Abdullah, and Zhen Ming (Jack) Jiang, "Rethinking Software Engineering in the Era of Foundation Models: A Curated Catalogue of Challenges in the Development of Trustworthy FMware", *in*: *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, Porto de Galinhas, Brazil: Association for Computing Machinery, 2024, pp. 294–305, ISBN: 9798400706585, DOI: 10.1145/3663529.3663849, URL: https://doi-org.univaq.idm.oclc.org/10.1145/3663529.3663849. [Cited on page 9]

[2] Junda He, Christoph Treude, and David Lo, "LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead", *in*: 34.5 (May 2025), ISSN: 1049-331X, DOI: 10.1145/3712003. [Cited on page 9]

[3] Xing Hu, Feifei Niu, Junkai Chen, Xin Zhou, Junwei Zhang, Junda He, Xin Xia, and David Lo, "Assessing and Advancing Benchmarks for Evaluating Large Language Models in Software Engineering Tasks", *in*: *ACM Trans. Softw. Eng. Methodol.* (Dec. 2026), ISSN: 1049-331X, DOI: 10.1145/3786771, URL: https://doi-org.univaq.idm.oclc.org/10.1145/3786771. [Cited on page 9]

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al., "Evaluating large language models trained on code", *in*: (2021), URL: https://arxiv.org/abs/2107.03374. [Cited on pages 11, 13]

[5] Frank F Xu, Bogdan Vasilescu, and Graham Neubig, "In-ide code generation from natural language: Promise and challenges", *in*: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.2 (2022), pp. 1–47. [Cited on pages 11, 13]

[6] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al., "Codexglue: A machine learning benchmark dataset for code understanding and generation", *in*: (2021), URL: https://arxiv.org/abs/2102.04664. [Cited on pages 11, 13]

[7] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao, "React: Synergizing reasoning and acting in language models", *in*: *The eleventh international conference on learning representations*, 2022. [Cited on page 11]

[8] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao, "Reflexion: Language agents with verbal reinforcement learning", *in*: *Advances in Neural Information Processing Systems* 36 (2023), pp. 8634–8652. [Cited on page 11]

[9] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun, *ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs*, 2023, arXiv: 2307.16789 [cs.AI], URL: https://arxiv.org/abs/2307.16789. [Cited on page 11]

[10] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering", *in*: *Advances in Neural Information Processing Systems* 37 (2024), pp. 50528–50652. [Cited on pages 11, 13, 25]

[11] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al., "Openhands: An open platform for ai software developers as generalist agents", *in*: (2024), URL: https://arxiv.org/abs/2407.16741. [Cited on page 11]

[12] Michael Felderer and Rudolf Ramler, "Quality assurance for AI-based systems: Overview and challenges", *in*: *Software Quality: Future Perspectives on Software Engineering Quality: 13th International Conference, SWQD 2021, Vienna, Austria, January 19–21, 2021, Proceedings 13*, Springer, 2021, pp. 33–42. [Cited on page 12]

[13] Bahar Gezici and Ayça Kolukısa Tarhan, "Systematic literature review on software quality for AI-based software", *in*: *Empirical Software Engineering* 27.3 (2022), p. 66. [Cited on page 12]

[14] Mohamed Abdullahi Ali, Ng Keng Yap, Abdul Azim Abd Ghani, Hazura Zulzalil, Novia Indriaty Admodisastro, and Amin Arab Najafabadi, "A systematic mapping of quality models for AI systems, software and components", *in*: *Applied Sciences* 12.17 (2022), p. 8700. [Cited on pages 12, 14]

[15] Julien Siebert, Lisa Joeckel, Jens Heidrich, Koji Nakamichi, Kyoko Ohashi, Isao Namba, Rieko Yamamoto, and Mikio Aoyama, "Towards guidelines for assessing qualities of machine learning systems", *in*: *Quality of Information and Communications Technology: 13th International Conference, QUATIC 2020, Faro, Portugal, September 9–11, 2020, Proceedings 13*, Springer, 2020, pp. 17–31. [Cited on pages 12, 15]

[16] Koji Nakamichi, Kyoko Ohashi, Isao Namba, Rieko Yamamoto, Mikio Aoyama, Lisa Joeckel, Julien Siebert, and Jens Heidrich, "Requirements-driven method to determine quality characteristics and measurements for machine learning software and its evaluation", *in*: *2020 IEEE 28th international requirements engineering conference (RE)*, IEEE, 2020, pp. 260–270. [Cited on pages 12, 15]

[17] Alexander Poth, Burkhard Meyer, Peter Schlicht, and Andreas Riel, "Quality Assurance for Machine Learning–an approach to function and system safeguarding", *in*: *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, 2020, pp. 22–29. [Cited on pages 12, 15]

[18] Hiroshi Kuwajima and Fuyuki Ishikawa, "Adapting square for quality assessment of artificial intelligence systems", *in*: *2019 IEEE international symposium on software reliability engineering workshops (ISSREW)*, IEEE, 2019, pp. 13–18. [Cited on pages 12, 15]

[19] Hiroshi Kuwajima, Hirotoshi Yasuoka, and Toshihiro Nakae, "Engineering problems in machine learning systems", *in*: *Machine Learning* 109.5 (2020), pp. 1103–1126. [Cited on page 12]

[20] Lena Pons and Ipek Ozkaya, "Priority quality attributes for engineering ai-enabled systems", *in*: (2019), URL: https://arxiv.org/abs/1911.02912. [Cited on pages 12, 15]

[21] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni, "Green ai", *in*: *Communications of the ACM* 63.12 (2020), pp. 54–63. [Cited on page 12]

[22] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al., "On the opportunities and risks of foundation models", *in*: (2021), URL: https://arxiv.org/abs/2108.07258. [Cited on page 12]

[23] Andreas Vogelsang and Markus Borg, "Requirements engineering for machine learning: Perspectives from data scientists", *in*: *2019 IEEE 27th international requirements engineering conference workshops (REW)*, IEEE, 2019, pp. 245–251. [Cited on pages 12, 15]

[24] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison, "Hidden technical debt in machine learning systems", *in*: *Advances in neural information processing systems* 28 (2015). [Cited on page 12]

[25] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou, *MLPerf Inference Benchmark*, 2020, arXiv: 1911.02549 [cs.LG], URL: https://arxiv.org/abs/1911.02549. [Cited on pages 12, 14]

[26] Daniel Li and Lincoln Murr, "HumanEval on Latest GPT Models–2024", *in*: (2024), URL: https://arxiv.org/abs/2402.14852. [Cited on pages 13, 14]

[27] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al., "Program synthesis with large language models", *in*: (2021), URL: https://arxiv.org/abs/2108.07732. [Cited on pages 13, 14]

[28] Tianyang Liu, Canwen Xu, and Julian McAuley, "Repobench: Benchmarking repository-level code auto-completion systems", *in*: (2023), URL: https://arxiv.org/abs/2306.03091. [Cited on pages 13, 14]

[29] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Zhi Jin, Hao Zhu, Huanyu Liu, Kaibo Liu, Lecheng Wang, Zheng Fang, et al., "Deveval: Evaluating code generation in practical software projects", *in*: (2024), URL: https://arxiv.org/abs/2401.06401. [Cited on page 13]

[30] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swebench: Can language models resolve real-world GitHub issues?", *in*: (2023), URL: https://arxiv.org/abs/2310.06770. [Cited on pages 13, 14, 28]

[31] Adamenko Pavel, Ivanov Mikhail, Aidar Valeev, Rodion Levichev, Pavel Zadorozhny, Ivan Lopatin, Dmitrii Babaev, Alena Fenogenova, and Valentin Malykh, "SWE-MERA: A Dynamic Benchmark for Agenticly Evaluating Large Language Models on Software Engineering Tasks", *in*: *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2025, pp. 440–452. [Cited on page 13]

[32] Xinyi He, Qian Liu, Mingzhe Du, Lin Yan, Zhijie Fan, Yiming Huang, Zejian Yuan, and Zejun Ma, "Swe-perf: Can language models optimize code performance on real-world repositories?", *in*: (2025), URL: https://arxiv.org/abs/2507.12415. [Cited on pages 13, 14]

[33] Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, et al., "Devbench: A comprehensive benchmark for software development", *in*: *CoRR* (2024). [Cited on pages 13, 14, 26]

[34] Myeongsoo Kim, Shweta Garg, Baishakhi Ray, Varun Kumar, and Anoop Deoras, "CodeAssistBench (CAB): Dataset & Benchmarking for Multi-turn Chat-Based Code Assistance", *in*: (2025), URL: https://arxiv.org/abs/2507.10646. [Cited on pages 13, 27]

[35] Ziyi Ni, Huacan Wang, Shuo Zhang, Shuo Lu, Ziyang He, Wang You, Zhenheng Tang, Yuntao Du, Bill Sun, Hongzhang Liu, et al., "Gittaskbench: A benchmark for code agents solving real-world tasks through code repository leveraging", *in*: (2025), URL: https://arxiv.org/abs/2508.18993. [Cited on pages 13, 27]

[36] Jielin Qiu, Zuxin Liu, Zhiwei Liu, Rithesh Murthy, Jianguo Zhang, Haolin Chen, Shiyu Wang, Ming Zhu, Liangwei Yang, Juntao Tan, et al., "LoCoBench-Agent: An Interactive Benchmark for LLM Agents in Long-Context Software Engineering", *in*: (2025), URL: https://arxiv.org/abs/2511.13998. [Cited on pages 13, 14, 25]

[37] René Just, Darioush Jalali, and Michael D Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs", *in*: *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440. [Cited on pages 13, 14]

[38] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah, "BUGSJS: a benchmark and taxonomy of JavaScript bugs", *in*: *Software Testing, Verification and Reliability* 31.4 (2021), e1751. [Cited on pages 13, 14]

[39] Jielin Qiu, Zuxin Liu, Zhiwei Liu, Rithesh Murthy, Jianguo Zhang, Haolin Chen, Shiyu Wang, Ming Zhu, Liangwei Yang, Juntao Tan, et al., "Locobench: A benchmark for long-context large language models in complex software engineering", *in*: (2025), URL: https://arxiv.org/abs/2509.09614. [Cited on pages 13, 14]

[40] Lingyue Fu, Hao Guan, Bolun Zhang, Haowei Yuan, Yaoming Zhu, Jun Xu, Zongyu Wang, Lin Qiu, Xunliang Cai, Xuezhi Cao, et al., "CoreCodeBench: A Configurable Multi-Scenario Repository-Level Benchmark", *in*: (2025), URL: https://arxiv.org/abs/2507.05281. [Cited on pages 13, 14]

[41] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al., "Agentbench: Evaluating llms as agents", *in*: (2023), URL: https://arxiv.org/abs/2308.03688. [Cited on page 14]

[42] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks", *in*: *Advances in neural information processing systems* 32 (2019). [Cited on pages 13, 14]

[43] Mohammed Latif Siddiq and Joanna CS Santos, "Securityeval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques", *in*: *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*, 2022, pp. 29–33. [Cited on page 14]

[44] José Antonio Hernández López, Jesús Sánchez Cuadrado, Riccardo Rubei, and Davide Di Ruscio, "ModelXGlue: a benchmarking framework for ML tools in MDE: JAH López et al.", *in*: *Software and Systems Modeling* 24.4 (2025), pp. 1035–1058. [Cited on page 14]

[45] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama, "QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge", *in*: *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, 2017, pp. 55–56. [Cited on page 13]

[46] Jiexin Wang, Xitong Luo, Liuwen Cao, Hongkui He, Hailin Huang, Jiayuan Xie, Adam Jatowt, and Yi Cai, "Is your ai-generated code really safe? evaluating large language models on secure code generation with codeseceval", *in*: (2024), URL: https://arxiv.org/abs/2407.02395. [Cited on page 13]

[47] Kishanthan Thangarajah, Arthur Leung, Boyuan Chen, and Ahmed E. Hassan, *SLA-Awareness for AI-assisted coding*, 2025, arXiv: 2503.19876 [cs.SE], URL: https://arxiv.org/abs/2503.19876. [Cited on page 14]

[48] Khan Mohammad Habibullah, Gregory Gay, and Jennifer Horkoff, "Non-functional requirements for machine learning: Understanding current use and challenges among practitioners", *in*: *Requirements Engineering* 28.2 (2023), pp. 283–316. [Cited on pages 15, 16]

[49] Hans-Martin Heyn, Eric Knauss, Amna Pir Muhammad, Olof Eriksson, Jennifer Linder, Padmini Subbiah, Shameer Kumar Pradhan, and Sagar Tungal, "Requirement engineering challenges for ai-intense systems development", *in*: *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)*, IEEE, 2021, pp. 89–96. [Cited on page 15]

[50] Nagadivya Balasubramaniam, Marjo Kauppinen, Antti Rannisto, Kari Hiekkanen, and Sari Kujala, "Transparency and explainability of AI systems: From ethical guidelines to requirements", *in*: *Information and Software Technology* 159 (2023), p. 107197, ISSN: 0950-5849, DOI: https://doi.org/10.1016/j.infsof.2023.107197, URL: https://www.sciencedirect.com/science/article/pii/S0950584923000514. [Cited on page 15]

[51] Hugo Villamizar, Tatiana Escovedo, and Marcos Kalinowski, "Requirements engineering for machine learning: A systematic mapping study", *in*: *2021 47th Euromicro conference on software engineering and advanced applications (SEAA)*, IEEE, 2021, pp. 29–36. [Cited on page 15]

[52] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al., "Judging llm-as-a-judge with mt-bench and chatbot arena", *in*: *Advances in neural information processing systems* 36 (2023), pp. 46595–46623. [Cited on page 16]

[53] Roberto Verdecchia, June Sallou, and Luís Cruz, "A systematic review of Green AI", *in*: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 13.4 (2023), e1507. [Cited on page 17]

[54] Luís Cruz, João Paulo Fernandes, Maja H. Kirkeby, Silverio Martínez-Fernández, June Sallou, Hina Anwar, Enrique Barba Roque, Justus Bogner, Joel Castaño, Fernando Castor, Aadil Chasmawala, Simão Cunha, Daniel Feitosa, Alexandra González, Andreas Jedlitschka, Patricia Lago, Henry Muccini, Ana Oprescu, Pooja Rani, João Saraiva, Federica Sarro, Raghavendra Selvan, Karthik Vaidhyanathan, Roberto Verdecchia, and Ivan P. Yamshchikov, *Greening AI-enabled Systems with Software Engineering: A Research Agenda for Environmentally Sustainable AI Practices*, 2025, arXiv: 2506.01774 [cs.SE], URL: https://arxiv.org/abs/2506.01774. [Cited on page 17]

[55] Gwendal Jouneaux and Jordi Cabot, *Towards Sustainability Model Cards*, 2025, arXiv: 2507.19559 [cs.CY], URL: https://arxiv.org/abs/2507.19559. [Cited on page 17]

[56] Margaret A Boden, *The creative mind: Myths and mechanisms*, Routledge, 2004. [Cited on page 17]

[57] Duc SH Nguyen, Bach G Truong, Phuong T Nguyen, Juri Di Rocco, and Davide Di Ruscio, "Teamwork makes the dream work: LLMs-Based Agents for GitHub README. MD Summarization", *in*: *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, 2025, pp. 621–625. [Cited on page 38]

[58] Duc SH Nguyen, Minh T Nguyen, Phuong T Nguyen, Juri Di Rocco, and Davide Di Ruscio, "Automated summarization of software documents: an LLM-based multi-agent approach", *in*: *Automated Software Engineering* 33.2 (2026), p. 43. [Cited on page 38]