

MOSAICO

D2.3: Benchmarking framework for AI agents for SE

September, 2026

DOCUMENT STATUS: SUBMITTED (Pending review and approval from the European Commission)

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.

D2.3 Benchmarking framework for AI agents for SE

Work package	WP2 Repository of AI agents for SE
Task	T2.3 Benchmarking framework for AI agents for SE
Task contributors	UDA, LIST
Due date	30/09/2026
Submission date	30/09/2026
Type of deliverable	Report
Dissemination Level	Public
Deliverable lead	UDA
Version	1.0
Authors	UDA, LIST
Reviewers	UY, IMT, COLLINS
Keywords	Benchmarking, AI Agents, KPI, Quality Model, Observability, Reproducibility

Document Revision History

Version	Date	Description of change	List of contributor(s)
v0.1.0	31/07/2026	First Draft	J. Di Rocco, D. Di Ruscio, U. Zeshan
v0.5.0	08/09/2026	Version for internal review	J. Di Rocco, D. Di Ruscio, U. Zeshan
v1.0.0	30/09/2026	Final version	J. Di Rocco, D. Di Ruscio, U. Zeshan

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This deliverable contains original unpublished work except where clearly indicated otherwise.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

Evaluating AI agents for software engineering requires evidence that is comparable across heterogeneous architectures, reproducible across runs, and traceable from raw execution events to quality indicators. In deliverable D2.1 a taxonomy of agents and benchmark-relevant entities was established, and deliverable D2.2 defined the quality model and the declarative KPI specification layer. This deliverable, provides the operational benchmarking framework that connects those foundations into an executable, evidence-driven process.

The report presents a structured study of current benchmarking practice for agentic software engineering, derives explicit framework requirements, and defines a methodology for benchmark specification, execution, telemetry retention, and KPI computation. The resulting MOSAICO framework is designed for extensibility along three independent axes: metric providers, formula language integration, and agent architecture adapters.

Table of Contents

1 Introduction **9**
1.1 Structure of the document 10

2 Benchmarking AI agents for SE **11**
2.1 Benchmark families 25
2.2 Software engineering task coverage 25
2.3 Agent architecture patterns 26
2.4 What existing benchmarks evaluate and record 27
2.5 Recurring assets in benchmarking infrastructures 29
2.5.1 Main benchmark components 30
2.5.2 Gaps and resulting requirements 31

3 The MOSAICO benchmarking framework **34**
3.1 Design rationale 34
3.2 Architecture overview 35
3.2.1 The run boundary 37
3.2.2 Orchestration Component 38
3.2.3 Service component 39
3.2.4 DSL component and integration 42
3.2.5 Persistence component 43
3.2.6 Interface component 45
3.3 The benchmarking flow 48
3.4 An explanatory use of the MOSAICO benchmarking framework 52
3.4.1 Benchmark specification 52
3.4.2 Quality definition 53

4 Conclusion and future work **56**

List of Figures

1 Main components of benchmarking frameworks for AI agents in software engineering. 11
2 Selection of the primary studies. Of the 46 records returned by the search, four could not be obtained in full text and twenty did not satisfy the eligibility criteria, leaving the 22 primary studies listed in Table 3. 13
3 Architecture of the MOSAICO benchmarking framework. Solid arrows indicate call dependencies between components; dashed arrows indicate integrations with systems outside the framework. Evaluated agents are not hosted by the framework: they report the evidence of a run across the run boundary, and the framework retrieves it from the observability platform. The dashed edge on which that evidence travels is the run boundary of Section 3.2.1. 36
4 The `KPIFormulaParser` interface and its implementation. 44
5 Excerpt of the data model of the benchmarking framework. 45
6 The run lifecycle on the management surface, with the request body and the acknowledgement returned by `POST /api/benchmark-runs`. 46
7 The indicator group, with `POST /api/performance-kpis/validate` expanded: an expression is submitted as text and answered with a verdict and the metric identifiers it resolved against. 47
8 Agent retrieval on the management surface, with the semantic entry point `POST /api/agents/search` expanded beside its four exact-match counterparts. 48
9 MCP tools exposed on the agent-facing surface of the MOSAICO repository, with the parameters of `SearchByTextTool`. 49
10 MCP resources exposed on the same surface, showing the benchmark collection returned as a JSON document. 50
11 MOSAICO benchmark flow from run admission to trace collection, per-trace metric computation, indicator evaluation, persistence, and completion or failure. . . . 51

List of Tables

1 Benchmarks employed by the corpus and their benchmark-level characteristics. Counts indicate the number of primary studies using each benchmark or benchmark family. Because a study may use more than one benchmark, the counts are non-exclusive. 16
2 Software-engineering task categories represented in the review corpus. Counts indicate the number of primary studies addressing each category; because a study may address more than one task category, the counts are non-exclusive. . 18
3 Primary studies included in the structured review and their study-specific evaluation attributes. Benchmark and task identifiers refer to Tables 1 and 2; a study may fall under more than one of each. 19
4 Evidence produced and systematically retained by the 22 reviewed studies. Counts report the number of studies providing explicit, core protocol support (score of 2) for each capability indicator. 28
5 Infrastructure tools examined and selection decisions, based on public documentation available in August 2026. Retained tools are used in the capability comparison; eligible alternatives were not added when their infrastructure class was already represented. 30
6 Coverage of requirements R1–R5 across the examined infrastructure tools, assessed from public documentation in August 2026. 32
7 Derivation of the requirements. R1–R5 are traced back to the study finding that motivates each; R6 and R7 arise from the MOSAICO setting. All are traced forward to the section that addresses them. 35
8 Built-in metric providers 41

Abbreviations

API	Application Programming Interface
DSL	Domain-Specific Language
KPI	Key Performance Indicator
LLM	Large Language Model
MCP	Model Context Protocol
SE	Software Engineering

1 Introduction

The MOSAICO repository is designed to enable software engineers to discover, compare, and select suitable AI agents for concrete software engineering tasks. Effective selection presupposes the availability of comparison evidence that is directly commensurable across candidate agents. Within Work Package 2 (WP2), Deliverable D2.1 (Task 2.1) established the taxonomy used to classify and describe AI agents, while Deliverable D2.2 (Task 2.2) defined the quality model and the domain-specific specification language through which their quality attributes and Key Performance Indicators (KPIs) are declared. Building directly upon these foundations, this deliverable provides the operational benchmarking infrastructure required to evaluate heterogeneous agents under a common protocol. The framework orchestrates benchmark runs, retrieves execution evidence produced by externally executed agents, computes standardised metrics, evaluates declarative KPIs, and persists the resulting evidence and quality history.

Generating reliable evaluation evidence is substantially more complex for AI agents than for conventional software components. An agent is not a static artefact whose behaviour is governed solely by deterministic source code. Instead, the system under measurement encompasses the underlying large language model, system prompts, tool interfaces, orchestration logic, execution environment, token budget, and retry policies; a modification to any single variable can fundamentally alter the observed results [1, 2]. Multi-agent architectures further compound these evaluation challenges. Responsibilities are distributed across specialized roles that exchange intermediate artefacts, review peer outputs, and coordinate via hierarchical, conversational, or workflow-driven topologies [3–6]. Consequently, two distinct multi-agent configurations may reach the exact same final outcome while exhibiting vast differences in operational latency, resource consumption, run-to-run stability, and diagnostic transparency.

Although the evaluation ecosystem has responded rapidly to agentic software engineering with executable benchmarks, addressing domain-specific tasks such as issue resolution [7], multimodal visual software design [8], performance optimization [9], research repository deployment [10], collaborative synchronization [11], vulnerability remediation [12], and dependency migration [13], these efforts remain fragmented. Each benchmark enforces its own isolated task schema, correctness oracle, execution environment, and reporting conventions. This creates a comparability problem rather than a scarcity problem: identical terms such as *resolution rate* or *pass rate* frequently denote conflicting units of analysis and correctness conditions across studies. Furthermore, critical operational variables—including role definitions, model settings, tool execution traces, and retry policies—are rarely captured within standard benchmark schemas.

To rigorously characterize this fragmentation and derive concrete requirements for our infrastructure, Section 2 reports a structured literature review of 22 recent benchmark studies. The review reveals that existing evaluations predominantly focus on final-artefact correctness while providing limited support for retaining the execution evidence needed to analyse efficiency, reproducibility, failures, and collaboration. Agent selection based solely on task completion would therefore omit important information about efficiency, stability, and execution behaviour. Evaluating AI agents must therefore address the full system that produces an artefact rather than evaluating the artefact in isolation. Outcome correctness must be analysed in conjunction with execution time, resource consumption, run-to-run stability, tool usage patterns, and multi-agent interaction behaviour.

Within the broader WP2 roadmap, Task 2.1 (D2.1) establishes how AI agents are classified and indexed within the MOSAICO repository. Task 2.2 (D2.2) provides the quality model and the domain-specific language (DSL) required to formalize metrics and KPIs. Task 2.3 (D2.3, this report) delivers the benchmarking infrastructure that organizes evaluations under controlled

benchmark specifications, collects runtime telemetry, and computes the DSL-defined indicators over externally produced execution evidence. Finally, Task 2.4 (D2.4) consumes these computed KPI profiles to power the semantic search, trade-off analysis, and agent selection services of the repository.

A central requirement of this deliverable is its direct integration with the Task 2.2 KPI specification language. Because quality characteristics cannot be directly measured as single raw values, the Task 2.2 DSL allows benchmark authors to declare structured indicators computed over low-level execution metrics [14]. The language, implemented in Xtext, provides authoring-time syntactic validation and supports a broad set of operations beyond simple weighted sums, including higher-order collection operations (`filter`, `map`, `reduce`), explicit threshold gates, and conditional disjunctions for qualitative evaluations. This deliverable operationalizes that language through a dedicated connector component. By enforcing a clear separation between quality model definitions (D2.2), indicator specifications (authored in the DSL), and runtime telemetry collection (D2.3), the MOSAICO benchmarking infrastructure is designed to support the introduction of new KPIs or modified evaluation rules without requiring modifications or redeployments of the core benchmarking engine.

1.1 Structure of the document

The remainder of this deliverable is organized as follows. Section 2 presents the structured literature review of 22 agentic benchmarks and formalizes the derived system requirements. Section 3 details the architecture and operational mechanisms of the MOSAICO benchmarking framework, including its integration with the Task 2.2 KPI specification language connector. Finally, Section 4 concludes the report and outlines the roadmap toward Task 2.4.

2 Benchmarking AI agents for SE

Benchmarking provides a structured way to evaluate and compare AI agents using defined software engineering tasks, execution environments, artefacts, and evaluation measures. A benchmark therefore specifies what is evaluated, under which conditions it is executed, what evidence is collected, and how the resulting behaviour is measured. Figure 1 summarises these main components and provides the basis for examining how existing benchmarks support the evaluation of AI agents for software engineering.

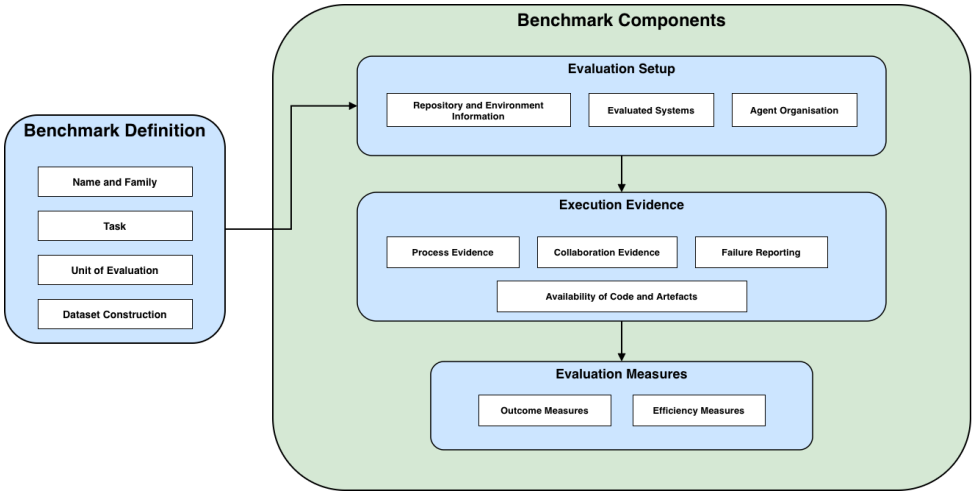


Figure 1: Main components of benchmarking frameworks for AI agents in software engineering.

The *Benchmark Definition* establishes the conceptual scope of an evaluation. It encompasses metadata identifying the benchmark name and family, the specific software engineering task, the targeted unit of evaluation, and the dataset construction methodologies. This declaration directly configures the operational *Benchmark Components*, which structure execution across three sequential phases:

- *Evaluation Setup:* Defines the operational environment prior to execution, detailing repository and execution environment configurations, specifications of the evaluated systems, and the underlying agent organisation topology.
- *Execution Evidence:* Captures runtime telemetry and artefacts during evaluation, collecting process evidence, multi-agent collaboration interactions, structured failure reporting, and the overall availability of code and execution artefacts.
- *Evaluation Measures:* Aggregates execution evidence into quantitative quality assessments, categorizing outcomes into outcome measures (evaluating functional correctness and task success) and efficiency measures (assessing resource usage and latency).

By decomposing the evaluation process into these components, benchmarks can provide a comprehensive and systematic approach to assessing the performance of AI agents in software engineering tasks.

As an example, consider a benchmark for code generation. The task is to generate code based on a given natural language description. The input artefacts include the description and any relevant context, while the evaluated agent is characterised by its model, prompts, and available tools. The execution environment is a development environment with a code editor

and a testing framework. The produced artefact is the generated code, and the evaluation measures include code quality, functionality, and adherence to best practices.

To make an overview of the current state of benchmarking LLM-based software engineering agents, a structured review of the literature has been performed. The review identifies which tasks, agent organisations, and evaluation dimensions are supported in existing benchmarks, and which are not. The findings inform the design of the MOSAICO benchmarking framework, which must accommodate the represented capabilities and fill the identified gaps.

```
TITLE-ABS-KEY(
  ("agentic AI" OR "LLM agent*" OR "AI coding agent*" OR
   "coding agent*" OR "software engineering agent*" OR
   "autonomous software engineer*") AND
  (benchmark* OR "benchmark dataset*" OR
   "evaluation benchmark*" OR "evaluation framework*" OR
   leaderboard) AND
  ("software engineering" OR "software development" OR
   "software repository" OR "repository-level" OR codebase))
```

Listing 1: Search query used for the structured review.

The review has been performed by retrieving and coding primary studies retrieved from Scopus by means of the query shown in Listing 1 that was executed in June 2026 to search for publications from 2022 through 2026, and by applying the following inclusion/exclusion criteria:

- *Inclusion criteria*: Studies were included when they proposed, instantiated, or empirically used a benchmark, dataset, leaderboard, or evaluation framework for LLM-based software engineering agents; evaluated an environment-interacting software engineering task;
- *Exclusion criteria*: Studies were excluded when they addressed a domain other than software engineering, presented only a conceptual discussion without an empirical benchmark or evaluation protocol, evaluated general-purpose agents without a software engineering task, or could not be obtained in full text.

The subsequent coding was performed by two independent reviewers, with disagreements resolved by discussion. The review was conducted according to established software engineering review guidance [15] and the reporting principles of PRISMA [16].

The selection process is summarised in Figure 2. The search returned 46 publications. Four could not be obtained in full text and were removed before detailed assessment; the remaining 42 studies were screened on title, abstract, keywords, and full text. Twenty did not satisfy the eligibility criteria, leaving 22 primary studies.

For each study we extracted the following benchmark attributes:

- *name and family*: the benchmark name and the family it belongs to, if any;
- *the task*: the task(s) the benchmark poses to the agent;
- *the unit of evaluation*: the level at which the benchmark evaluates the agent, e.g., function, file, repository;
- *the dataset construction*: the way the benchmark dataset was constructed, e.g., curated, synthetic, or derived from real-world repositories;
- *repository and environment information*: the information about the repository and environment used in the benchmark, e.g., programming language, framework, operating system;

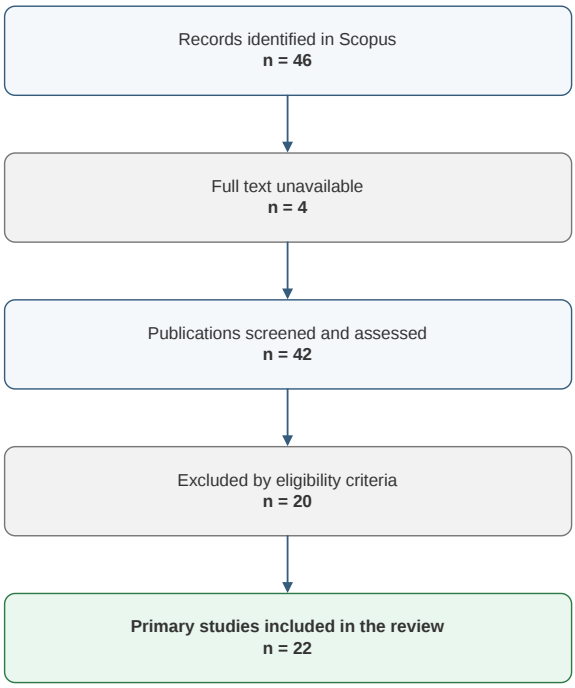


Figure 2: Selection of the primary studies. Of the 46 records returned by the search, four could not be obtained in full text and twenty did not satisfy the eligibility criteria, leaving the 22 primary studies listed in Table 3.

- *the evaluated systems*: the systems evaluated by the benchmark, e.g., LLM-based agents, traditional tools, or human developers;
- *the agent organisation*: the organisation of the evaluated agents, e.g., single-agent, multi-agent, or hybrid;
- *outcome measures*: the metrics used to evaluate the performance of the agents, e.g., accuracy, precision, recall, F1-score;
- *efficiency measures*: the metrics used to evaluate the efficiency of the agents, e.g., run-time, memory usage, energy consumption;
- *process evidence*: the evidence of the process by which the agents were evaluated, e.g., logs, traces, or other observational data;
- *collaboration evidence*: the evidence of how the agents collaborated with each other, e.g., communication logs, shared resources, or joint decision-making processes;
- *failure reporting*: the evidence of how failures were reported and handled by the agents, e.g., error messages, recovery actions, or post-mortem analyses;
- *availability of code and artefacts*: the availability of the code and other artefacts related to the benchmark, e.g., source code, documentation, or datasets.

The 22 primary studies (S1–S22) constitute the unit of literature retrieval, but the analytical focus rests on the individual benchmarks (B1–B19) and software engineering tasks (K1–K17) they employ. Because a single publication frequently spans multiple benchmarks and task domains, two derived inventories structure the corpus:

- *Benchmark Inventory (Table 1):* Consolidates variants sharing identical protocols and task schemas into benchmark families (e.g., grouping SWE-bench alongside its Verified, Lite, and Multimodal variants), while retaining distinct variants where studies specifically relied upon them.
- *Task Inventory (Table 2):* Categorizes tasks based on operational benchmarking requirements. Tasks requiring distinct evaluation oracles (e.g., repository deployment vs. dependency migration) are separated, whereas functionally equivalent activities (e.g., bug fixing and patch generation) are unified into single task categories.

As detailed in Table 1, the benchmark inventory reveals a strong concentration around repository-level issue resolution. The SWE-Bench family (B1) dominates the literature, utilized across 12 primary studies. Additionally, 10 studies employ study-specific custom repository tasks (B2). Beyond standard issue resolution, the corpus spans specialized domain benchmarks, including security vulnerability repair (B5, CVE-Bench), test generation (B6, TDD-Bench-Verified), cross-file code completion (B13, CrossCodeEval), enterprise workflows (B15, EnterpriseBench), and tool creation (B16, TM-BENCH).

The task inventory in Table 2 mirrors this distribution across 17 distinct operational categories. GitHub issue resolution, bug fixing, and patch generation (K1) represent the primary focus, addressed by 9 studies. Feature implementation (K16) and repository setup, installation, and deployment (K2) are each addressed by 3 studies. The remaining categories capture specialized software engineering workflows, ranging from collaborative recovery from out-of-sync repository states (K5) and vulnerability repair (K3) to automated debugging (K9), dependency migration (K12), and agent training scale evaluations (K14).

Table 3 details the 22 primary studies (S1–S22) included in the review alongside their specific evaluation attributes, mapping each study to its corresponding benchmark identifiers (Table 1) and software engineering task categories (Table 2). A detailed examination of Table 3 reveals several structural characteristics across the evaluated systems, environments, measurement strategies, and retained evidence as discussed below.

Evaluated Systems and Agent Architectures: Evaluated baselines range from individual foundation models (e.g., Claude 3.5 Sonnet, GPT-4o, DeepSeek-R1, Llama variants) to complex agentic frameworks such as OpenHands, SWE-Agent, Agentless, AutoCodeRover, and Aider. While several studies incorporate sophisticated search or reasoning workflows, such as MCTS in LingmaAgent (S12), static code-graph analysis in CODEXGRAPH (S13), or formal verification with Z3 in PatchPilot (S20), the vast majority of studies evaluate monolithic or single-agent configurations. Explicit multi-agent roles and collaborative organisational topologies remain rare, appearing prominently only in studies specifically designed for collaborative recovery, such as SyncBench (S7), or human-in-the-loop workflows (S8).

Execution Environments and Containerization: Containerized execution environments predominate across the corpus. standardised Docker containers are utilized extensively (e.g., S1, S2, S3, S4, S6, S7, S10, S16, S17) to isolate repository environments, run build pipelines (e.g., .NET in S6, Maven in S18), and execute test suites safely. Sandbox environments with explicit tool access controls are also leveraged for enterprise workflow tasks (S14).

Outcome and Efficiency Measurement: Outcome evaluation relies almost universally on execution-based correctness metrics, such as task resolution rates, pass@k, unit test pass rates, and repair success. Custom task-specific indicators are frequently introduced to capture domain nuances, such as test coverage metrics in Otter (S4), patch complexity in SWE-Sharp-Bench (S6), PR rejection odds in wild PR analyses (S9), and path-feasibility validation in RepoAudit

(S21). Conversely, efficiency accounting remains inconsistent: while token usage, monetary cost, and execution runtime are tracked in dedicated efficiency studies (e.g., S4, S11, S15, S18, S20, S21), explicit resource or interaction budgets are rarely enforced as core benchmark constraints.

Process, Collaboration, and artefact Retention: While open data and code availability are widely supported across the corpus, systemic retention of detailed execution evidence remains a critical gap. Most studies provide only partial run artefacts or intermediate logs. Detailed tool-action and repository-interaction traces are captured in only a fraction of frameworks (e.g., S3, S7, S10, S11, S14, S17, S21), and complete execution trace persistence is limited to a few specific studies (e.g., S4, S9, S17). Similarly, explicit reporting of retries, failure modes, and inter-agent communication events is largely absent outside of specialized failure-analysis studies (S9) or collaborative agent evaluations (S7).

By capturing these study-specific evaluation attributes, Table 3 provides the empirical mapping necessary to assess current state-of-the-art capability coverage and highlights the infrastructure gaps addressed by the proposed benchmarking framework.



Table 1: Benchmarks employed by the corpus and their benchmark-level characteristics. Counts indicate the number of primary studies using each benchmark or benchmark family. Because a study may use more than one benchmark, the counts are non-exclusive.

ID	Benchmark / family	Benchmark purpose / task	Evaluation unit	Dataset origin / construction	# studies
B1	SWE-Bench family (including SWE-Bench, SWE-Bench Verified, SWE-Bench Lite, and SWE-Bench Multimodal)	Repository issue resolution, bug fixing, patch generation, and multimodal issue resolution	Issue / repository	Real GitHub issues linked to historical repository states and executable tests; all SWE-Bench variants are consolidated within this benchmark family	12
B2	GitHub repository-based custom tasks	Study-specific repository-level software-engineering tasks	Repository / issue / pull request, depending on the study	Study-specific collections constructed from real GitHub repositories, issues, pull requests, or project histories	10
B3	GAIA	Generalist problem solving and human-agent support tasks	Task	Curated general-assistant tasks used alongside software-engineering-oriented evaluations	2
B4	CSR-Bench	Deployment of computer-science research repositories	Repository	One hundred computer-science research repositories curated for setup, download, training, inference, and evaluation	1
B5	CVE-Bench	Repair of real-world software vulnerabilities	CVE / repository	A collection of 509 real-world CVEs drawn from 120 repositories and paired with repair and validation environments	1
B6	TDD-Bench-Verified	Issue-driven test generation for validating software patches	Issue	Human-validated issue tasks derived from SWE-Bench Verified and equipped for generated-test evaluation	1
B7	SWE-MERA	Dynamic evaluation of agents on software issue resolution	GitHub issue	Dynamically collected GitHub issue tasks with task-level and regression-test validation	1
B8	SWE-Sharp-Bench	C# software issue resolution and feature implementation	Issue / repository	Reproducible C# repository issue tasks with a .NET build-and-test pipeline	1
B9	SyncBench	Recovery from out-of-sync states in collaborative software engineering	Collaborative task / repository	Repository tasks constructed using simulated collaborator changes that place the evaluated agent out of sync	1
B10	AIDev-POP	Analysis of agent-generated pull requests and repository contributions	Pull request / repository	A collection of 12,433 real agent-generated pull requests obtained from 1,495 GitHub repositories	1
B11	The Agent Company	Generalist completion of realistic, multi-step computer-use tasks	Task	Curated workplace-style scenarios requiring interaction with multiple tools and applications	1
B12	BigCodeBench-R / LiveCodeBench-R	Automated debugging and program repair	Program-repair task	Repair variants derived from BigCodeBench and LiveCodeBench programming tasks	1
B13	CrossCodeEval	Cross-file, repository-level code completion	Code-completion task	A public benchmark constructed from real repositories to require cross-file context	1
B14	EvoCodeBench	Code completion in evolving repositories	Code-completion task	A public benchmark constructed from repository evolution and temporally changing code contexts	1
B15	EnterpriseBench	Enterprise workflow automation involving software and organisational systems	Workflow task	Curated enterprise workflows executed in a sandbox containing software-development and business-system interfaces	1
B16	TM-BENCH	Creation and validation of tools for agents	Tool-creation task	Fifteen curated tool-creation tasks with task-specific execution and unit-test validation	1
B17	Synthetic Java dependency-upgrade benchmark	Dependency migration and software maintenance	Repository	Three controlled Java repository tasks requiring dependency upgrades and associated migration changes	1

continued on next page

Table 1 continued from previous page

ID	Benchmark / family	Benchmark purpose / task	Evaluation unit	Dataset origin / construction	# studies
B18	Real-world code-auditing repositories	Repository-level code auditing and bug discovery	Repository	Fifteen benchmark repositories and nine additional real-world projects used for code-auditing evaluation	1
B19	SWE training instances / generated tests	Training and inference scaling for software-engineering agents	Training / evaluation instance	A collection of 38,000 training instances and 4,630 generated test cases, with evaluation conducted on SWE-Bench Verified	1

Table 2: Software-engineering task categories represented in the review corpus. Counts indicate the number of primary studies addressing each category; because a study may address more than one task category, the counts are non-exclusive.

ID	Task category	# studies
K1	GitHub issue resolution, bug fixing, and patch generation	9
K2	Repository setup, installation, and deployment	3
K3	Vulnerability repair and security-bug fixing	1
K4	Issue-driven test generation and patch validation	1
K5	Out-of-sync recovery and collaborative software engineering	1
K6	Human-agent developer workflow and interaction	1
K7	Pull-request generation and repository contribution	1
K8	Multimodal and generalist software-agent tasks	1
K9	Automated debugging and interactive program repair	1
K10	Enterprise workflow automation using software-engineering tools	1
K11	Repository-level tool creation	1
K12	Dependency migration and software maintenance	1
K13	Repository-level code auditing and bug discovery	1
K14	Software-engineering agent training and inference scaling	1
K15	Repository-level code completion using cross-file context	1
K16	Feature implementation	3
K17	Data analysis and non-software-engineering support tasks	1

Table 3: Primary studies included in the structured review and their study-specific evaluation attributes. Benchmark and task identifiers refer to Tables 1 and 2; a study may fall under more than one of each.

ID	Study	Year	Benchmark IDs	Task IDs	Unit / Dataset	Evaluated Systems / Environment / Agent Organisation	Outcome / Efficiency Measures	Process / Collaboration / Failure Evidence & Artefacts
S1	Louis Milliken, Sungmin Kang, and Shin Yoo, *Beyond pip Install: Evaluating LLM Agents for the Automated Installation of Python Projects*, Proceedings of the 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), DOI: 10.1109/SANER64311.2025.00009 [17]	2025	B2	K2	Repository-level; 40 Python repositories	0 replicable baselines; evaluates INSTALLAMATIC only; API-based GPT-4o-mini agent; Dockerfiles executed inside VM; Python repositories only; organisation: no explicit multi-agent roles	Installation success rate; documentation recall; visibility; informativity; repair-step success	open code/data; partial run artefacts; partial failure/retry/intermediate evidence
S2	Yijia Xiao, Runhui Wang, Luyang Kong, Davor Golac, and Wei Wang, *CSR-Bench: Benchmarking LLM Agents in Deployment of Computer Science Research Repositories*, Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies, DOI: 10.18653/v1/2025.naacl-long.633 [10]	2025	B2, B4	K2	Repository-level; 100 CS research repositories	11 model baselines: Claude, GPT-4, Llama, Mistral variants; API/service-based LLMs; standardized Docker environment; issue retrieval; Perplexity web search; organisation: no explicit multi-agent roles	Completion rate; setup, download, training, inference, evaluation success rates	open code/data; partial run artefacts; partial failure/retry/intermediate evidence
S3	Peiran Wang, Xiaogeng Liu, and Chaowei Xiao, *CVE-Bench: Benchmarking LLM-based Software Engineering Agent's Ability to Repair Real-World CVE Vulnerabilities*, Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies, DOI: 10.18653/v1/2025.naacl-long.212 [18]	2025	B2, B5	K3	CVE/repository-level; 509 CVEs; 120 repositories	1 replicable baseline: SWE-Agent; 3 LLM backends tested; interactive Docker repair environment; Pylint, Bandit, Mypy, Prospector; organisation: no explicit multi-agent roles	Repair rate; unit-test pass rate; failure reasons; repair CDF; tool-use impact	open code/data; partial run artefacts; partial tool traces; partial repository traces; partial failure/retry/intermediate evidence

continued on next page

Table 3 continued from previous page

ID	Study	Year	Benchmark IDs	Task IDs	Unit / Dataset	Evaluated Systems / Environment / Agent Organisation	Outcome / Efficiency Measures	Process / Collaboration / Failure Evidence & Artefacts
S4	Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel, *Otter: Generating Tests from Issues to Validate SWE Patches*, Proceedings of the 42nd International Conference on Machine Learning (ICML), PMLR 267, URL: `https://proceedings.mlr.press/v267/ahmed25b.html` [19]	2025	B1, B6	K4	Issue-level; SWE-Bench Verified-derived issue tasks	4 replicable baselines: Zero-shot, SWE-Agent, SWE-Agent+, LIBRO; also AutoCodeRover and AEGIS-related approaches; local execution; Docker; Flake8; GPT-4o and Mistral-Large; repository checkout and coverage instrumentation; organisation: no explicit multi-agent roles	Fail-to-pass rate; tddScore; test coverage; pass@5; precision; recall; cost per issue; custom: tddScore	open code/data; run artefacts retained; partial repository traces; partial failure/retry/intermediate evidence
S5	Pavel Adamenko, Mikhail Ivanov, Aidar Valeev, Rodion Levichev, Pavel Zadorozhny, Ivan Lopatin, Dmitrii Babaev, Alena Fenogenova, and Valentin Malykh, *SWE-MERA: A Dynamic Benchmark for Agentically Evaluating Large Language Models on Software Engineering Tasks*, Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, DOI: 10.18653/v1/2025.emnlp-demos.30 [20]	2025	B1, B2, B7	K1	GitHub issue-level; dynamic GitHub issue tasks	10 Aider-based model baselines: DeepSeek, Devstral, Qwen, Llama and Codestral families; Docker; HuggingFace; Aider; GitHub; leaderboard; organisation: no explicit multi-agent roles	pass@1; pass@6; localization; patch generation; regression tests; custom: task correctness, test correctness, test completeness, complexity	open code/data; partial run artefacts; partial failure/retry/intermediate evidence
S6	Sanket Mhatre, Yasharth Bajpai, Sumit Gulwani, Emerson Murphy-Hill, and Gustavo Soares, *SWE-Sharp-Bench: A Reproducible Benchmark for C# Software Engineering Tasks*, 2025 IEEE/ACM International Conference on AI-Powered Software (AIware), DOI: 10.1109/AIware69974.2025.00039 [21]	2025	B1, B2, B8	K1, K16	Issue-level; C# issue tasks	2 agent baselines: SWE-Agent and OpenHands; evaluated across 9 LLMs; Docker; GitHub Actions; .NET build pipeline; HuggingFace; organisation: no explicit multi-agent roles	Resolution Rate; Patch Complexity (#files/#hunks/#lines)	open code/data; partial run artefacts; partial failure/retry/intermediate evidence
S7	Xuehang Guo, Xingyao Wang, Yangyi Chen, Sha Li, Chi Han, Manling Li, and Heng Ji, *SyncMind: Measuring Agent Out-of-Sync Recovery in Collaborative Software Engineering*, Proceedings of the 42nd International Conference on Machine Learning (ICML), PMLR 267, URL: `https://proceedings.mlr.press/v267/guo25l.html` [11]	2025	B2, B9	K5	Collaborative task/repository-level; SyncBench collaborative tasks	7 model baselines: Claude, GPT-4o, DeepSeek and Llama families; OpenHands environment; Docker; GitHub repositories; simulated collaborators; organisation: explicit multi-agent roles	Success Rate; Localization Accuracy; Conditional Success Rate; Assistance Seeking Rate; Recovery Efficiency; custom: SR, LA, CSR, ASR, Recovery Efficiency	open code/data; partial run artefacts; partial tool traces; repository traces; failures/retries/intermediate artefacts; communication events; partial handoff/role-contribution evidence

continued on next page

Table 3 continued from previous page

ID	Study	Year	Benchmark IDs	Task IDs	Unit / Dataset	Evaluated Systems / Environment / Agent Organisation	Outcome / Efficiency Measures	Process / Collaboration / Failure Evidence & Artefacts
S8	Valerie Chen, Ameet Talwalkar, Robert Brennan, and Graham Neubig, *Code with Me or for Me? How Increasing AI Automation Transforms Developer Workflows*, Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems, DOI: 10.1145/3772318.3790850 [22]	2026	B1, B3	K6, K16, K17	Participant/task-level; 20 participants; mixed SE and GAIA-derived tasks	2 coding-assistant baselines: GitHub Copilot vs OpenHands; IDE; VSCode; terminal; browser; Jupyter; GitHub; organisation: partial/experiment-specific multi-agent roles	Task correctness; time; cognitive load; satisfaction; flow; understanding	partial code/data availability; partial run artefacts; partial tool traces; partial repository traces; partial failure/retry/intermediate evidence; communication events; partial handoff/role-contribution evidence
S9	Mahd Hindi, Yasir Mahmood, Linda Mohammed, Salah Bouktif, and Mohammed Mediani, *Coding Agents in the Wild: Failure Modes and Rejection Patterns of AI-Generated Pull Requests*, IEEE Access, vol. 14, DOI: 10.1109/ACCESS.2026.3696573 [23]	2026	B2, B10	K7, K16	Pull-request level; 12,433 agent-generated PRs; 1,495 repositories	5 coding-agent baselines: OpenAI Codex, GitHub Copilot, Claude Code, Cursor, Devin; real GitHub PR workflow; CI; code review; governance/policy gates; organisation: no explicit multi-agent roles	PR rejection rate; acceptance rate; rejection odds; temporal trend; failure modes; reviewer comments; silent rejection; custom: failure-mode taxonomy and silent rejection	open code/data; run artefacts retained; repository traces; failures/retries/intermediate artefacts; partial communication evidence
S10	Aditya Bharat Soni, Boxuan Li, Xingyao Wang, Valerie Chen, and Graham Neubig, *Coding Agents with Multimodal Browsing are Generalist Problem Solvers*, Findings of the Association for Computational Linguistics: EACL 2026, DOI: 10.18653/v1/2026.findings-eacl.318 [24]	2026	B1, B3, B11	K8	Benchmark-task level; multiple benchmark suites	8 baselines: SWE-Agent, Agentless, OpenHands, BrowserGym, Browser-use, OpenDeepResearch, OWL, Magentic-One; Bash; Python; browser; search APIs; multimodal files; Docker/OpenHands runtime; organisation: no explicit multi-agent roles	Resolve Rate; Success Rate; Full/Partial Completion; Tool Usage; Cost	open code/data; partial run artefacts; partial tool traces; partial repository traces; partial failure/retry/intermediate evidence
S11	Yunkun Wang, Yue Zhang, Guochang Li, Chen Zhi, Binhua Li, Fei Huang, Yongbin Li, and Shuiguang Deng, *InspectCoder: Dynamic Analysis-Driven Self Repair through Interactive LLM-Debugger Collaboration*, Proceedings of the ACM on Programming Languages, vol. 10, OOPSLA1, DOI: 10.1145/3798238 [25]	2026	B12	K9	Program-repair task level; repair variants of BigCodeBench and LiveCodeBench	6+ baselines: Simple-Debugging, Rubber-Ducking, Trace-Reasoning, AutoSD, LDB variants, SWE-Agent; InspectWare; PDB debugger; runtime inspection; breakpoints; runtime modification; organisation: no explicit multi-agent roles	Resolve Rate; Pass Rate; Fixes/Hour; Fixes/Dollar; Debug Time; Cost	open code/data; partial run artefacts; partial tool traces; partial failure/retry/intermediate evidence

continued on next page

Table 3 continued from previous page

ID	Study	Year	Benchmark IDs	Task IDs	Unit / Dataset	Evaluated Systems / Environment / Agent Organisation	Outcome / Efficiency Measures	Process / Collaboration / Failure Evidence & Artefacts
S12	Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li, *Alibaba LingmaAgent: Improving Automated Issue Resolution via Comprehensive Repository Exploration*, Companion Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, DOI: 10.1145/3696630.3728549 [26]	2025	B1, B2	K1	Issue-level; SWE-Bench Lite + in-house issues	Compared with SWE-Agent, AutoCodeRover, GPT-4, Claude-3 Opus and SWE-Llama; repository knowledge graph; MCTS; AST/Jedi parsing; search APIs; IDE integration; organisation: no explicit multi-agent roles	Resolved Rate; Apply Rate; Fault Localization; Pass@3; Cost	partial code/data availability; partial run artefacts
S13	Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou, *CODEXGRAPH: Bridging Large Language Models and Code Repositories via Code Graph Databases*, Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies, DOI: 10.18653/v1/2025.naacl-long.7 [27]	2025	B1, B13, B14	K1, K15	Benchmark-task level; three public benchmarks	3 main baselines: NO-RAG, BM25, AutoCodeRover; GPT-4o, DeepSeek-Coder-V2 and Qwen2-72B; Neo4j; Cypher; static analysis; graph queries; repository indexing; organisation: no explicit multi-agent roles	CrossCodeEval: EM, ES, ID-EM, ID-F1; SWE-Bench: Pass@1; EvoCodeBench: Pass@1 and Recall@1	partial code/data availability; partial run artefacts
S14	Harsh Vishwakarma, Ankush Agarwal, Ojas Patil, Chaitanya Devaguptapu, and Mahesh Chandran, *Can LLMs Help You at Work? A Sandbox for Evaluating LLM Agents in Enterprise Environments*, Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (EMNLP), DOI: 10.18653/v1/2025.emnlp-main.466 [28]	2025	B15	K10	Enterprise workflow-task level; enterprise workflow tasks	5 LLM baselines: GPT-4o, Claude-3.5 Sonnet, o1-mini, Llama-3.1-8B, Llama-3.3-70B; No Planning, CoT, ReAct and Gold Planning; enterprise sandbox; CRUD APIs; GitHub; HR; CRM; IT tickets; chats; emails; access control; organisation: no explicit multi-agent roles	Task Completion; F1; Human Eval; Prometheus Score; Execution Success; custom: access-control compliance, tool selection, decomposition, factual coverage, execution success	open code/data; partial run artefacts; tool traces; partial repository traces; partial failure/retry/intermediate evidence
S15	Vaibhav Aggarwal, Ojasv Kamal, Abhinav Japesh, Zhijing Jin, and Bernhard Schölkopf, *DARS: Dynamic Action Re-Sampling to Enhance Coding Agent Performance by Adaptive Tree Traversal*, Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics, DOI: 10.18653/v1/2025.acl-long.973 [29]	2025	B1	K1	Issue-level; SWE-Bench Lite	6+ baselines: SWE-Agent, OpenHands, Agentless, MASAI, SWE-Search, Large Language Monkeys; GPT-4o, Claude-3.5 and Gemini models; RepoGraph; Aider; IPython; server execution; undo edit; sandbox; organisation: no explicit multi-agent roles	Pass@1; Pass@k; Resolve Rate; Coverage; Precision; Cost; Attempts; Localization Accuracy; custom: bug fixing score, regression risk score, coverage, precision, reviewer-based patch ranking	open code/data; partial run artefacts; partial failure/retry/intermediate evidence

continued on next page

Table 3 continued from previous page

ID	Study	Year	Benchmark IDs	Task IDs	Unit / Dataset	Evaluated Systems / Environment / Agent Organisation	Outcome / Efficiency Measures	Process / Collaboration / Failure Evidence & Artefacts
S16	Karina Zainullina, Alexander Golubev, Maria Trofimova, Sergei Polezhaev, Ibragim Badertdinov, Daria Litvintseva, Simon Karasik, Filipp Fisin, Sergei Skvortsov, Maksim Nekrashevich, Anton Shevtsov, and Boris Yangel, *Guided Search Strategies in Non-Serializable Environments with Applications to Software Engineering Agents*, Proceedings of the 42nd International Conference on Machine Learning (ICML), PMLR 267, URL: `https://proceedings.mlr.press/v267/zainullina25a.html` [30]	2025	B1	K1	Issue-level; SWE-Bench Verified + Verified-50	4 main baselines: Qwen policy, GPT-4o policy, SWE-Agent and SWE-Gym; compared with LingmaAgent and SWE-Fixer; Docker; SWE-Agent; critic models; 1-step lookahead; trajectory selection; organisation: no explicit multi-agent roles	Success Rate; Pass@N; Pass@5; Pass@15; Average Success Rate; SEM; custom: critic-based action-value and trajectory scoring	open code/data; partial run artefacts; partial failure/retry/intermediate evidence
S17	Georg Wölflein, Dyke Ferber, Daniel Truhn, Ognjen Arandjelovic, and Jakob Nikolas Kather, *LLM Agents Making Agent Tools*, Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics, DOI: 10.18653/v1/2025.acl-long.1266 [31]	2025	B16	K2, K11	Tool-creation task level; 15 benchmark tasks	Main baseline: OpenHands; Bash; file I/O; browsing; Google Drive; Docker; checkpointing; dependency installation; GitHub repositories; organisation: no explicit multi-agent roles	Tool Accuracy; Correct Invocations; Unit Test Pass Rate; Installation Success; Cost; Actions; Tokens; Self-Correction Iterations; custom: task-specific unit tests, invocation correctness, tool correctness, file/output/execution validation	open code/data; run artefacts retained; tool traces; partial repository traces; failures/retries/intermediate artefacts
S18	Vali Tawosi, Salwa Alamir, Xiaomo Liu, and Manuela Veloso, *LLM Agents for Automated Dependency Upgrades*, 2025 IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), DOI: 10.1109/ASEW67777.2025.00016 [32]	2025	B17	K12	Repository-level; 3 Java dependency-upgrade repositories	OpenHands + Claude 3.7 Sonnet baseline; LADU compared against OpenHands; Meta-RAG; Maven; Pom/YAML parsing; compilation; unit tests; migration guides; organisation: no explicit multi-agent roles	Precision; Files Modified; Common Files; Common Lines; Runtime; Steps; Tokens; Cost	partial code/data availability; partial run artefacts; partial tool traces; partial repository traces; partial failure/retry/intermediate evidence

continued on next page

Table 3 continued from previous page

ID	Study	Year	Benchmark IDs	Task IDs	Unit / Dataset	Evaluated Systems / Environment / Agent Organisation	Outcome / Efficiency Measures	Process / Collaboration / Failure Evidence & Artefacts
S19	Atefeh Sohrabizadeh, Jialin Song, Mingjie Liu, Rajarshi Roy, Chankyu Lee, Jonathan Raiman, and Bryan Catanzaro, *NEMOTRON-CORTEXA: Enhancing LLM Agents for Software Engineering Tasks via Improved Localization and Solution Diversity*, Proceedings of the 42nd International Conference on Machine Learning (ICML), PMLR 267, URL: `https://proceedings.mlr.press/v267/sohrabizadeh25a.html` [33]	2025	B1	K1	Issue-level; SWE-Bench Lite and Verified	Agentless plus comparisons with OpenHands, SWE-Agent, AutoCodeRover and Moatless; NV-EmbedCode; AST graph; LSP; dependency analysis; regression/reproduction tests; organisation: no explicit multi-agent roles	Resolution Rate; Recall@k; Precision; Recall; Pass@1; Pass@9; Cost; custom: localization metrics, patch diversity, test filtering, cost-aware evaluation	open code/data; partial run artefacts; partial failure/retry/intermediate evidence
S20	Hongwei Li, Yuheng Tang, Shiqi Wang, and Wenbo Guo, *PatchPilot: A Cost-Efficient Software Engineering Agent with Early Attempts on Formal Verification*, Proceedings of the 42nd International Conference on Machine Learning (ICML), PMLR 267, URL: `https://proceedings.mlr.press/v267/li25cf.html` [34]	2025	B1	K1	Issue-level; SWE-Bench Lite and Verified	Compared against Agentless, OpenHands, AutoCodeRover, Globant and CodeStory; own rule-based workflow; PoC execution; coverage analysis; function/class search; AST; Flake8; CrossHair; Z3; test execution; organisation: no explicit multi-agent roles	Resolved Rate; Average Cost; Stability; Functionality-Test Pass; PoC-Test Pass; Formal Verification Success; custom: ranking metrics combining PoC tests, functionality tests, stability, refinement and formal verification	open code/data; partial run artefacts; partial failure/retry/intermediate evidence
S21	Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang, *RepoAudit: An Autonomous LLM-Agent for Repository-Level Code Auditing*, Proceedings of the 42nd International Conference on Machine Learning (ICML), PMLR 267, URL: `https://proceedings.mlr.press/v267/guo25n.html` [35]	2025	B2, B18	K13	Repository-level; 15 benchmark repositories + 9 additional projects	Compared with CoT prompting, LLMDFA, Meta INFER and Amazon CodeGuru; Claude 3.5/3.7, DeepSeek R1 and OpenAI o3-mini; Tree-sitter; call graphs; data-flow analysis; control-flow validation; agent memory; organisation: no explicit multi-agent roles	TP; FP; Precision; Recall; Bugs Found; Tokens; Cost; Time; custom: path-feasibility, data-flow validation, TP/FP analysis, bug-confirmation/fix tracking	open code/data; partial run artefacts; partial tool traces; partial repository traces; failures/retries/intermediate artefacts
S22	Haoran Wang, Zhenyu Hou, Yao Wei, Jie Tang, and Yuxiao Dong, *SWE-Dev: Building Software Engineering Agents with Training and Inference Scaling*, Findings of the Association for Computational Linguistics: ACL 2025, DOI: 10.18653/v1/2025.findings-acl.193 [36]	2025	B1, B2, B19	K1, K14	Training/evaluation instance level; 38K training instances; 4,630 generated test cases; SWE-Bench Verified evaluation	Compared with SWE-Gym, SWE-SynInfer, SWE-Fixer, GPT-4o, OpenHands, Agentless and AutoCodeRover; OpenHands; test generation; Gherkin scenarios; trajectory sampling; long context; organisation: no explicit multi-agent roles	Resolve Rate; Unit Test Pass; Training Scaling; Inference Scaling; custom: scaling curves, trajectory quality filtering, synthetic test validation	open code/data; partial run artefacts; partial failure/retry/intermediate evidence

2.1 Benchmark families

The SWE-bench family (B1) represents the dominant repository-level ecosystem across the reviewed literature, employed by 12 of the 22 primary studies. It established the standard evaluation protocol wherein an agent receives a repository issue (e.g., a GitHub issue report) and must generate a code patch that passes executable test suites against a historical repository state [7]. Additionally, 10 studies constructed custom repository collections (B2) to evaluate specialized tasks unavailable in standard suites (S1, S18, S21) or to mitigate temporal data contamination (S5) [20].

Beyond generic issue resolution, specialized benchmarks introduce task-specific execution procedures and evaluation criteria. For instance, in SWE-bench (B1), an issue instance such as `sympy__sympy-20590` presents an unexpected `__dict__` attribute on `Symbol` objects. The evaluated agent receives the issue statement alongside the repository context and produces a candidate patch:

```
{
  "instance_id": "sympy__sympy-20590",
  "model_name_or_path": "gpt-4",
  "model_patch":
    "diff --git a/sympy/core/sympify.py b/sympy/core/sympify.py\n
     ...
     -          ValueError)):\n
     +          ValueError, AttributeError)):\n
                return a"
}
```

The test harness applies the candidate patch and executes verification test suites to confirm resolution:

```
python -m swebench.harness.run_evaluation \
  --predictions_path gold \
  --max_workers 1 \
  --instance_ids sympy sympy-20590 \
  --run_id validate-gold
```

While SWE-bench validates functional correctness via patch application and unit test execution, other benchmark families employ distinct operational protocols. For example, deployment benchmarks (B4, CSR-Bench) evaluate environment configuration success, vulnerability repair benchmarks (B5, CVE-Bench) validate security regressions, and tool-creation benchmarks (B16, TM-BENCH) evaluate dynamic API invocation accuracy. Across all families, the benchmark links problem inputs, environment states, agent actions, and evaluation oracles into a unified execution instance.

2.2 Software engineering task coverage

GitHub issue resolution, bug fixing, and patch generation (K1) constitute the primary focus of the literature, represented across nine studies. Repository setup and deployment (K2) and feature implementation (K16) each appear in three studies. Conversely, broader software engineering activities, including vulnerability repair (K3), issue-driven test generation (K4), collaborative out-of-sync recovery (K5), pull-request reviews (K7), dependency migration (K12), and repository auditing (K13), are addressed only by isolated, task-specific benchmarks.

This distribution highlights a significant concentration in current research: while repository-level evaluation represents a major advancement over isolated code completion, the literature

remains heavily focused on maintenance and localized repair. Early-stage activities (e.g., requirements engineering, architectural design) and long-term organisational tasks (e.g., release management, continuous coordination) remain largely unaddressed. To support comprehensive software engineering evaluation, a benchmarking framework must avoid hardcoding issue-resolution assumptions and instead supply a task-agnostic specification model capable of accommodating heterogeneous task schemas.

Because different software engineering tasks rely on fundamentally different correctness oracles, ranging from unit test execution in bug fixing to static policy checking in requirements review, a general framework cannot enforce a uniform validation function across task families. Instead, the framework standardizes operational metadata that is strictly comparable across evaluations, including task identity, agent configuration, execution environment, resource budgets, trace evidence, and metric definitions, while preserving the task-specific correctness criteria defined by individual benchmark specs.

2.3 Agent architecture patterns

The evaluated systems differ substantially not only in underlying foundation models but also in internal structural organisation. Analyzing these organisational variations reveals key evidence requirements for benchmark design. The literature exhibits five recurring architectural patterns:

1. Interactive tool-using agents. Systems such as SWE-Agent [37] and OpenHands [38] alternate reasoning with shell commands, code editor actions, and browser interactions against live environments. Observable evidence consists of sequential tool calls and environment responses.

2. Workflow-based pipelines. Systems such as Agentless [39] structure fault localization, patch generation, and validation as deterministic, multi-stage pipelines. Observable evidence consists of structured stage outputs and phase transitions rather than free-form tool loops.

3. Retrieval-centered systems. Frameworks such as AutoCodeRover [40] prioritize repository search, AST parsing, and context retrieval prior to modification. Observable evidence emphasizes search queries, context graphs, and localization accuracy.

4. Planning and reflection systems. Implementations based on ReAct and Reflexion [1, 2] explicitly separate planning, execution, and self-correction. Observable evidence spans reasoning traces, intermediate critique states, and plan revisions.

5. Multi-agent topologies. Systems such as AutoGen [3], MetaGPT [4], and ChatDev [5] distribute tasks across specialized agent roles communicating via messages or shared artefacts. Observable evidence includes role assignments, message topologies, and task handoffs.

These architectural variations demonstrate that candidate systems organize execution and expose internal states in fundamentally distinct ways. Consequently, a benchmarking framework cannot mandate a uniform internal interface or proprietary communication protocol without excluding major agent topologies. Instead, the framework must establish a stable external observability boundary, normalizing heterogeneous execution traces, event logs, tool calls, and produced artefacts into a unified data model (Section 3).

2.4 What existing benchmarks evaluate and record

In addition to task and benchmark classifications, the review assessed what each benchmark systematically records once an evaluation run completes. A benchmark may pose a realistic software engineering task yet fail to retain traces of how a result was produced. However, it is the retained trace that determines whether an evaluation can be retrospectively diagnosed, audited, or compared across system iterations.

To support systematic synthesis across the corpus, the extracted evidence and task inventories are mapped to six analytical capability dimensions: *Task and Scope* (informed by benchmark definitions, task categories K1–K17, evaluation units, and dataset construction methods); *Execution and Reproducibility* (derived from repository details, environment configurations, evaluated system topologies, and artefact availability); *Outcome and Efficiency* (mapped directly from outcome metrics and resource efficiency measures); *Process-level Evaluation* (built from captured process evidence and structured failure reporting); *Collaboration Evaluation* (informed by agent organisation structures and multi-agent interaction evidence); and *Extensibility* (evaluated through documented mechanisms for introducing new tasks, dataset instances, target systems, or evaluation metrics). This mapping translates heterogeneous descriptive data across individual literature sources into a standardised, multi-dimensional assessment matrix. By aggregating study-specific attributes into these six capability dimensions, the framework provides the structured analytical foundation required to evaluate state-of-the-art benchmark coverage and systematically identify research gaps as discussed in Section 2.5.

To evaluate these capabilities, the review examined 17 specific indicators grouped across the capability dimensions. Each study-indicator pair was evaluated against a three-point ordinal scale. A score of 0 (*Absent*) indicates that the capability is completely absent or unmentioned in the benchmark protocol. A score of 1 (*Partial/Ad hoc*) indicates that support is partial, optional, or restricted to a specific ad hoc experiment rather than integrated into the reusable benchmark infrastructure; for instance, a study receives a score of 1 for *Runtime measured* if execution latency is manually timed and reported in a paper's result table for a single experiment, but the underlying benchmark framework provides no automated timer or logging mechanism to record execution durations across general runs. Finally, a score of 2 (*Explicit Core Support*) indicates that the capability is systematically provided as an automated, native feature of the core benchmark evaluation protocol.

The scoring process adhered to four strict guidelines: scores evaluate capabilities natively supplied by the benchmark framework rather than independent actions taken by evaluated agents; every score is tied to explicit text or repository evidence; ambiguous cases receive the lower score; and every score of 1 records the specific gaps preventing core protocol integration. Table 4 reports the count of studies achieving a score of 2 for each indicator. Reporting score 2 isolates infrastructure-level capabilities that are systematically supported out-of-the-box, excluding ad hoc manual measurements reported only within individual study experiments.

A detailed breakdown across the six capability dimensions highlights significant variations in how existing benchmarks record execution telemetry, account for resource efficiency, and support architectural extensibility.

Execution and Reproducibility. This dimension evaluates environment stability and configuration persistence across five indicators. Sixteen of the 22 studies explicitly provide an *Executable environment*, enabling tasks to be run dynamically rather than statically evaluated. Similarly, 16 studies specify an *Isolated, reproducibly specified environment*, typically leveraging Docker containers to prevent host contamination (e.g., S1, S2, S3, S4, S6, S7, S10, S16, S17). The SWE-bench family (B1) serves as a representative example, running evaluations

Table 4: Evidence produced and systematically retained by the 22 reviewed studies. Counts report the number of studies providing explicit, core protocol support (score of 2) for each capability indicator.

Dimension	Indicator	Explicit Support (Score 2)
Execution and reproducibility	Executable environment	16/22
	Isolated, reproducibly specified environment	16/22
	Agent configuration recorded	0/22
	Open data and code	18/22
	Run artefacts retained	3/22
Outcome and efficiency	Execution-based correctness	21/22
	Runtime measured	3/22
	Token or monetary cost measured	3/22
	Resource or interaction budget defined	2/22
Process-level evaluation	Tool-action traces	2/22
	Repository-interaction traces	2/22
	Retries, failures, intermediate artefacts	4/22
Collaboration evaluation	Explicit multi-agent roles	1/22
	Communication events	2/22
	Handoffs or role-associated contributions	0/22
Extensibility	Extensible instances or tasks	3/22
	Extensible metrics or KPIs	0/22

against fixed historical repository states with automated test execution [7]. Public accessibility is well supported, with 18 studies supplying *Open data and code*. Conversely, systemic retention of execution state remains critically low: only three studies explicitly enforce *Run artefacts retained* to persist complete logs, intermediate patches, and outputs after execution (S4, S9, S17), and zero studies systematically record *Agent configuration recorded* (capturing exact model parameters, system prompts, tool schemas, and operational budgets) as part of the reusable benchmark evidence.

Outcome and Efficiency. Performance verification and resource accounting are evaluated across four indicators. *Execution-based correctness* represents the most widely supported capability across the corpus, provided natively by 21 of the 22 studies through automated test suites, build pipelines, or dynamic execution scripts. In contrast, efficiency tracking is rarely integrated into reusable benchmark protocols. *Runtime measured* is supported as a core benchmark capability in only three studies (S8, S11, S18), *Token or monetary cost measured* is natively tracked in three studies (S4, S10, S11), and a *Resource or interaction budget defined*

(enforcing hard timeouts or maximum tool limits) is established as a benchmark constraint in only two studies (S14, S15).

Process-Level Evaluation. Three indicators assess visibility into intermediate agent reasoning and tool usage. Capturing process evidence is essential for root-cause failure analysis, yet benchmark-level support remains sparse. Only two studies systematically persist *Tool-action traces* (S3, S17), and only two capture *Repository-interaction traces* logging file edits, directory navigation, and git operations (S3, S9). Furthermore, only four studies explicitly retain *Retries, failures, intermediate artefacts* (S1, S2, S7, S9), allowing non-deterministic retries, syntax errors, or partial repair attempts to be analysed retrospectively.

Collaboration Evaluation. Multi-agent dynamics are evaluated across three indicators, representing the sparsest dimension in the literature. *Explicit multi-agent roles* are supported as a core benchmark capability in only one study (S7), *Communication events* capturing inter-agent message passing are retained in two studies (S7, S8), and zero studies explicitly record *Handoffs or role-associated contributions* to isolate individual agent contributions within a collaborative task. SyncBench (B9, S7) provides a notable exception where collaborative recovery and agent communication form part of the evaluation protocol [11], but such capabilities remain isolated across the broader corpus.

Extensibility. Infrastructure adaptability is evaluated through two indicators. *Extensible instances or tasks* are supported by three studies (S5, S17, S22), primarily via documented dataset construction pipelines or synthetic task generators. Conversely, *Extensible metrics or KPIs* are completely unsupported (0/22): evaluation metrics remain hardcoded inside benchmark runner scripts, requiring manual codebase modifications to evaluate new indicators or rescore existing runs.

The pronounced imbalance between outcome measurement and process retention severely limits the long-term utility of benchmark results. First, it obscures failure diagnosis: when an execution is recorded solely as a binary pass/fail result, environment timeouts, invalid patches, missing tool dependencies, and genuine logic errors become indistinguishable. Second, it prevents evidence reuse: without persisted execution traces and artefacts, scoring a benchmark against a newly defined metric requires repeating agent executions at full computational cost. Third, it hides total resource consumption. Evaluating agentic systems requires extending Green AI principles [41, 42] from isolated LLM inference calls to entire multi-step workflows, reporting task success directly alongside total latency, token consumption, and financial cost.

2.5 Recurring assets in benchmarking infrastructures

While the preceding review analysed primary literature studies proposing task-specific benchmarks, this section examines reusable evaluation and observability infrastructure tools. Because infrastructure components (e.g., telemetry collectors, observability backends, evaluation harnesses) are typically distributed as software packages rather than formal academic studies, they lie outside the systematic literature retrieval protocol.

To complement the literature review, an infrastructure analysis was conducted over candidate tools using four purposive selection criteria. First, candidates had to supply reusable evaluation, telemetry, or observability infrastructure rather than a single task-specific benchmark dataset. Second, tools were required to possess comprehensive, publicly available documentation. Third, the retained set had to span all three core infrastructure classes identified in our taxonomy: telemetry conventions, observability backends, and evaluation harnesses. Fourth, additional candidates from an already represented class were retained only when exposing distinct architectural design points or serving as direct components of the MOSAICO platform.



Table 5: Infrastructure tools examined and selection decisions, based on public documentation available in August 2026. Retained tools are used in the capability comparison; eligible alternatives were not added when their infrastructure class was already represented.

Tool	Class	Decision	Reason
OpenTelemetry GenAI	Telemetry convention	Retained	Provides a public convention for describing generative-AI and agent telemetry.
Langfuse	Observability backend	Retained	Publicly documented observability infrastructure and representative open-source backend.
LangSmith	Observability backend	Retained	Publicly documented agent and LLM tracing and evaluation backend.
HELM	Evaluation harness	Retained	General evaluation infrastructure with extensible scenarios and metrics.
Inspect	Evaluation harness	Retained	Extensible evaluation framework supporting tasks, solvers, and scoring.
Arize Phoenix	Observability backend	Eligible	Not retained because the same infrastructure class is already represented by the selected observability backends.
AgentOps	Observability backend	Eligible	Not retained because adding another representative of the same infrastructure class would not broaden the comparison.
Weights & Biases Weave	Observability backend	Eligible	Not retained because its observability and evaluation functionality overlaps with an infrastructure class already represented.

This selection process yielded five representative infrastructure tools across the three classes: OpenTelemetry GenAI (telemetry convention), Langfuse and LangSmith (observability backends), and HELM and Inspect (evaluation harnesses). Table 5 details both retained and non-retained candidate tools alongside selection rationales based on public documentation available in August 2026. Non-retained eligible tools (e.g., Arize Phoenix, AgentOps, Weights & Biases Weave) provide valuable observability capabilities, but were omitted because their underlying design points overlap with the selected backends.

2.5.1 Main benchmark components

The tools are evaluated in the structured order of Table 5, grouped by their corresponding infrastructure classes:

Telemetry Conventions. Telemetry conventions specify how execution traces should be structured and named without dictating execution runtime or scoring logic. The OpenTelemetry Semantic Conventions for Generative AI serve as the reference standard, defining span hierarchies, parent-child relationships across agent and tool invocations, and standardised attribute names for token usage and model parameters. For instance, an agent execution is represented

as a root span containing nested child spans for LLM completions and tool calls, attaching attributes such as `gen_ai.request.model` directly to individual spans. Although client-level model spans have stabilized, agent-level telemetry conventions remain under active, experimental development.

Observability Backends. Observability backends capture, persist, and index execution trace evidence, enabling post hoc queryability and manual or automated annotation. Langfuse represents an open-source backend on which the MOSAICO framework directly integrates (Section 3), while LangSmith provides a commercial alternative. These platforms persist trace hierarchies, model parameters, and token counts per run, allowing evaluation scores (e.g., LLM-as-a-judge outputs or human annotations) to be attached directly to stored traces after execution. For instance, a Langfuse trace records full input/output structures alongside nested tool executions and token costs, rendering the complete run execution tree inspectable rather than returning only final task outputs.

Evaluation Harnesses. Evaluation harnesses supply execution management, task specifications, and scoring pipelines. HELM provides general evaluation machinery separating benchmark scenarios from evaluation metrics; however, its primary evaluation unit targets an isolated model evaluated against prompt tasks rather than multi-step agentic systems interacting with complex environments. Conversely, Inspect separates tasks into explicit *tasks*, *solvers* (agents), and *scorers* (oracles). This separation allows new scoring functions to be executed over existing outputs without redefining task configurations, illustrating the separation between execution and evaluation.

2.5.2 Gaps and resulting requirements

Although each infrastructure class addresses specific evaluation concerns, no individual class supplies a complete benchmarking environment, nor do they compose seamlessly into an integrated pipeline. Three primary infrastructure gaps emerge:

Partial benchmark specifications. Observability backends model individual execution *runs* rather than declarative *benchmarks*. Crucial evaluation parameters, including dataset references, agent configurations, environmental dependencies, execution budgets, and indicator formulas, exist only as arbitrary attributes attached to traces after execution, preventing pre-execution validation or reusable benchmark declaration. Conversely, evaluation harnesses supply structured benchmark objects, but typically optimize for single-model evaluations rather than multi-agent, tool-using workflows.

Absence of standardised indicator catalogs. Observability backends provide technical interfaces to attach scores to traces, but deliberately omit standardised metric catalogs, domain-specific indicators, or specification languages for declaring composite metrics. Consequently, adopting teams repeatedly implement ad hoc scoring scripts, preventing cross-team result comparability even when sharing an observability backend.

SDK-dependent collaboration evidence. Multi-agent telemetry (e.g., roles, inter-agent messages, handoffs) is captured in observability backends only when explicit proprietary SDKs are embedded within the evaluated agents. Because a benchmarking framework cannot assume that competing agent architectures share an internal framework or SDK, collaboration evidence remains fragmented. While OpenTelemetry aims to standardize agent-level telemetry, its current conventions remain experimental and insufficient to serve as a benchmark-level contract.

These identified gaps establish five core operational requirements (R1–R5) that the MOSAICO benchmarking framework must satisfy:



R1. Common specification. Tasks, datasets, agent configurations, environments, metrics, artefacts, and publication metadata must be represented explicitly and uniformly, so that what is evaluated is declared independently of execution logic.

R2. Reproducible execution. Repository versions, dependencies, model parameters, prompts, tool definitions, timeouts, and execution budgets must be bound to each run, ensuring results are traceable to exact execution conditions.

R3. Modular evaluation. Metric computation must be decoupled from execution, allowing task-specific correctness oracles and cross-cutting indicators to coexist, while enabling retained trace evidence to be rescored without re-executing agents.

R4. Collaboration awareness. Roles, messages, handoffs, tool actions, failures, and intermediate artefacts must be observable and persisted whenever exposed by evaluated systems.

R5. Extensibility. New tasks, agent topologies, metrics, telemetry backends, and execution environments must be integrable without modifying core framework protocols.

In addition to R1–R5, two further requirements stem directly from the MOSAICO agent repository architecture (Deliverable D2.1) and quality model (Deliverable D2.2). In MOSAICO, benchmarking evidence supports semantic search, discovery, and automated agent selection. This requires persisting evaluation results alongside traceable execution provenance and exposing them to external consensus agents via structured interfaces:

R6. Declarative indicator specification. Key performance indicators must be declared in a dedicated specification language anchored to the MOSAICO quality model rather than hardcoded in the evaluation engine, ensuring quality models evolve independently of underlying infrastructure.

R7. Repository integration. Benchmarking results must be persisted, traceable, and queryable by the MOSAICO repository, ensuring measured quality directly informs semantic search and automated agent selection mechanisms.

Table 6: Coverage of requirements R1–R5 across the examined infrastructure tools, assessed from public documentation in August 2026.

Requirement	**OTel GenAI** *convention*	**Langfuse** *backend*	**LangSmith** *backend*	**HELM** *harness*	**Inspect** *harness*
R1 Common specification	none	partial	partial	partial	partial
R2 Reproducible execution	partial	partial	partial	full	full
R3 Modular evaluation	none	full	full	full	full
R4 Collaboration awareness	partial	partial	partial	none	partial
R5 Extensibility	full	full	partial	full	full

full: supported as a core native capability. *partial*: supported partially or dependent on agent instrumentation. *none*: outside the defined scope of the tool.

Table 6 evaluates the extent to which the examined infrastructure tools satisfy requirements R1–R5 based on public documentation as of August 2026 by supporting the planned capabilities of the MOSAICO benchmarking framework.

The review and infrastructure comparison provide a structured basis for deriving the framework requirements, but they do not constitute an empirical validation of the MOSAICO implementation. The literature review is based on a Scopus search covering publications from 2022 to 2026 and includes 22 eligible primary studies. The infrastructure comparison is purposive and based on publicly available documentation. Consequently, the findings may not cover all relevant benchmarks or infrastructure tools, and the capability scores should be interpreted as a structured assessment rather than as an exhaustive or independently validated ranking. Empirical evaluation of the framework across industrial and experimental environments remains beyond the scope of this deliverable and will be conducted during the deployment and support of the industrial partner use cases.

3 The MOSAICO benchmarking framework

This section describes the MOSAICO benchmarking framework,[1] which realises the requirements of Section 2.5.2. In particular, Section 3.1 states the design principles and maps each requirement onto the mechanism that answers it. Section 3.2 depicts the architecture of the framework. Section 3.3 presents the MOSAICO benchmark flow: how runtime components collaborate, how evidence moves through the lifecycle, and how results are retained for repository consumption.

3.1 Design rationale

Four principles govern the design of the MOSAICO benchmarking framework, directly addressing the infrastructural gaps identified in Section 2.5. To ground these requirements operationally, the first three principles are contextualized using industrial use cases from Deliverable D5.1, while the fourth dictates the internal framework architecture.

A declared object, not an executable script (R1). The evaluation target must exist independently of its execution procedure. It must function as an artefact capable of being statically validated prior to runtime, systematically compared across studies, and deterministically re-instantiated. The common practice of encoding benchmarks as execution scripts conflates specification with runtime behaviour, rendering independent inspection impossible. As demonstrated in Section 2.4, this coupling prevents the durable retention and reuse of agent configurations by omitting a dedicated declarative layer. This separation is operationally critical, for instance, when an agent has to assist engineers in proper requirements decomposition and management (Deliverable D5.1, Section 5).

Evidence and its interpretation are separate concerns (R3, R6). Execution telemetry and metric interpretation are strictly decoupled. Decoupling them enables retrospective rescoring of retained evidence and allows indicators to be declared using the quality model DSL (Deliverable D2.2) rather than hard-coded into the evaluation engine. As identified in Section 2.5, current observability backends typically support trace scoring but lack a standardised measurement catalog. By separating measurement from interpretation, heterogeneous constraints can be evaluated as distinct indicators over a single dataset, eliminating redundant evaluation runs.

A stable evidence boundary instead of a mandated agent interface (R4, R5). The framework does not require evaluated agents to implement a common internal architecture or expose the same agent-level API. Such a requirement would artificially constrain heterogeneous systems and would make the benchmark less extensible. Instead, MOSAICO defines a stable evidence boundary: evaluated systems remain outside the framework, but they must expose the execution evidence needed to interpret a run, to the extent that such evidence is applicable to their architecture. This evidence may include roles, messages, handoffs, tool actions, generated artefacts, and trace metadata. The boundary also protects evaluation integrity: adapters and instrumentation may report what happened during execution, but they may not alter task definitions, reference datasets, correctness oracles, metric specifications, or validation rules.

Reuse existing infrastructure, build only what is missing (R2, R7). Existing telemetry conventions, observability backends, and evaluation harnesses already supply foundational components of a benchmarking framework, while the MOSAICO repository manages agent discovery and evidence consumption. Rather than duplicating these systems, the framework

[1] https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/mosaico-repository

Table 7: Derivation of the requirements. R1–R5 are traced back to the study finding that motivates each; R6 and R7 arise from the MOSAICO setting. All are traced forward to the section that addresses them.

	Requirement	Motivating finding	Addressed in
R1	Common specification	High task concentration and absence of a unified correctness oracle across task domains.	Sect. 3.2
R2	Reproducible execution	Systematic omission of stored agent configurations, operational budgets, and run artefacts.	Sect. 3.2.1, 3.2.2
R3	Modular evaluation	Evaluation formulas are hard-coded into runners, preventing retrospective rescoring of traces.	Sect. 3.2.2, 3.2.4
R4	Collaboration awareness	Heterogeneous evidence exposure across topologies; inter-agent roles and handoffs are rarely retained.	Sect. 3.2.1, 3.2.3
R5	Extensibility	Lack of shared internal agent APIs and rigid evaluation harnesses hindering custom metrics.	Sect. 3.2.3, 3.2.4
Arising from the MOSAICO setting rather than from the study (Sect. 2.5.2)			
R6	Declarative indicator specification	The quality model of D2.2 must be operationalised without encoding indicators in the evaluation engine	Sect. 3.2.4
R7	Repository integration	Agent selection in the MOSAICO repository requires measured quality to be persisted, traceable and queryable	Sect. 3.2.5, 3.2.6

delegates core responsibilities: execution evidence is retained directly within the observability backend, and evaluation outcomes are persisted as `KPIHistory` entries inside the repository (Section 3.2.5). This historical accumulation preserves the temporal evolution of an agent's measured quality rather than maintaining an isolated framework-specific log. The framework itself is therefore a layer that orchestrates the evaluation process, computes metrics, resolves indicators, and persists results, while relying on existing infrastructure for telemetry retention and agent discovery.

Table 7 traces each requirement to the finding that motivates it and to the section that addresses it.

3.2 Architecture overview

The MOSAICO benchmarking framework is designed to be modular and extensible, with a clear separation of concerns. It consists of five main components (see Figure 3): interface, orchestration, service, KPI DSL, and persistence. Each component is responsible for a specific aspect of the benchmarking process as described below.

The *orchestration component* manages the execution and tracks the state of each evaluation run using the orchestrator pattern [43]. In contrast, the underlying services remain stateless, processing tasks on demand without storing run information. Centralizing state management satisfies requirement R2: if an evaluation fails mid-execution, its current progress is preserved in a single inspectable record rather than fragmented across multiple services. Additionally, a dedicated scheduler component monitors stored execution schedules and automatically triggers evaluations when due.

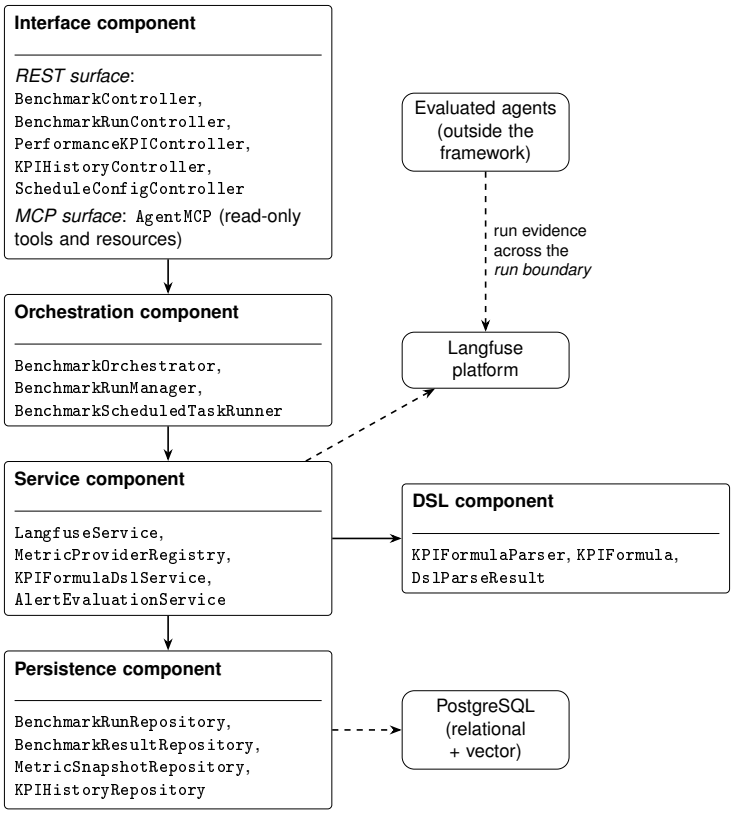


Figure 3: Architecture of the MOSAICO benchmarking framework. Solid arrows indicate call dependencies between components; dashed arrows indicate integrations with systems outside the framework. Evaluated agents are not hosted by the framework: they report the evidence of a run across the run boundary, and the framework retrieves it from the observability platform. The dashed edge on which that evidence travels is the run boundary of Section 3.2.1.

The *service component* holds four domain capabilities. Telemetry retrieval obtains what a run produced from the observability backend and normalises it into the internal evidence model. Metric computation maintains the catalogue of measurable quantities and dispatches over it. Indicator resolution turns a stored specification into an evaluable form by delegating to the formula language. Alert evaluation compares computed indicator values against the bounds configured for a benchmark and dispatches notifications on the configured channels.

The *DSL component* is the integration point between measured evidence and declared indicators. It keeps the indicator language outside the evaluation engine: metrics are produced as named values, while formulae are parsed, validated and compiled behind a dedicated parser interface. As a result, benchmark designers can revise how measurements are combined without changing the orchestration logic, and the framework can reject an indicator that names unavailable metrics before a run is executed.

The *persistence component* retains benchmarks, runs, per-trace evidence and indicator history. It also maintains the vector index on which discovery depends, which is built over the textual representation of each agent and enriched with skill and KPI metadata derived from persisted evaluations. Its structure, and the extent of the traceability it supports, are described in Section 3.2.5.

The *interface component* is the entry point at which incoming requests are validated and authenticated before any downstream component sees them. It carries two surfaces as detailed in Section 3.2.6: a management surface over which benchmarks, indicators and runs

are administered, and a read-only surface through which agentic consumers reach measured quality.

Although not a concrete software component, the *run boundary* represents a key conceptual interface of the framework architecture. As depicted by the dashed arrow labeled "run evidence across the run boundary" in Figure 3, it denotes the decoupling point between external agents and the evaluation infrastructure. The framework never invokes or executes an agent directly; instead, it passively ingests execution traces and evidence deposited by evaluated agents into the Langfuse platform.

The delegation patterns across components isolate operational responsibilities, ensuring that changes within a specific domain do not impact the broader framework. As illustrated in Figure 3, the `BenchmarkScheduledTaskRunner` within the *Orchestration component* executes no evaluations directly; it reads stored recurrences, determines when a run is due, and delegates execution to the `BenchmarkOrchestrator`. Consequently, automated and manual execution paths remain identical, differing only in recorded provenance.

The *BenchmarkOrchestrator* manages run lifecycle without performing metric computations directly. Instead, it delegates telemetry retrieval from the *Langfuse platform* to the *LangfuseService* inside the *Service component*, resolves providers via the `MetricProviderRegistry`, and delegates formula parsing to the `KPIFormulaDslService`. In turn, the `KPIFormulaDslService` encapsulates the *DSL component* by invoking the `KPIFormulaParser`. This design isolates the core orchestration logic from changes to metric provider implementations or expression language syntaxes.

Similarly, the `LangfuseService` serves as the sole interface to the external *Langfuse platform*, decoupling the remaining system components from the underlying observability infrastructure. Data persistence is managed centrally: the `BenchmarkOrchestrator` updates execution state via the *Persistence component* (`BenchmarkRunRepository`, `BenchmarkResultRepository`) backed by *PostgreSQL*. The *Interface component* subsequently reads this persisted state across both its REST and MCP surfaces without interacting directly with the orchestrator, enforcing a strict separation between evaluation execution and result reporting.

The remainder of this section follows the structure of Figure 3: the run boundary is detailed first to define how execution evidence enters the framework, followed by dedicated subsections for each component and its integration responsibilities.

3.2.1 The run boundary

The point at which execution evidence becomes available lies outside the five core framework components and is designated as the *run boundary*: the interface through which an evaluated agent reports execution outputs. For agents built on the MOSAICO base agent library[2], this boundary is defined by an Agent-to-Agent (A2A) extension carrying three optional metadata fields:

```
MosaicoObservabilityMetadata = TypedDict(
    "MosaicoObservabilityMetadata", {
        "mosaico-root-task-id":    str,    # run trace
        "mosaico-super-task-id":   str,    # caller span
        "mosaico-root-task-name":  str,    # trace name
    }, total=False)
```

An agent receiving this metadata creates its own span under the designated trace and parent span, allowing multi-agent operations on a single task to be recorded as a unified trace

[2] https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/mosaico-base-agents

tree. Operationally, collaboration evidence is reconstructed directly from the trace topology: a *handoff* corresponds to a span opened beneath another agent's span, while a *role* is defined by the name an agent assigns to its own span.

Execution evidence is persisted in the Langfuse platform, which the base library targets by default. During execution, the `execute` method resolves the incoming metadata, opens the corresponding Langfuse span, and invokes `execute_agent`. Because tracing logic is inherited rather than manually implemented, agents built on the base library record execution traces automatically. This abstraction holds regardless of the underlying agent topology, whether implementing a bare executor, direct model completions, a LangGraph ReAct loop, or BDI performatives. Consequently, MOSAICO agents are benchmarkable out of the box by having the framework read directly from the shared Langfuse instance, satisfying requirement R5.

Because metadata fields are optional and parsing non-blocking, tracing failures degrade gracefully without causing execution errors. However, a single-span trace can indicate either an agent that performed no delegation or one that delegated without propagating trace metadata. Indicators dependent on roles or handoffs are therefore evaluated conditionally: if the required instrumentation is absent, the indicator is omitted rather than evaluated as zero.

While adapters and custom instrumentation may expose additional telemetry, they are strictly prohibited from modifying the task instructions, reference datasets, correctness oracles, metric definitions, or validation rules specified by the benchmark. Agents developed outside the MOSAICO ecosystem require targeted integration confined entirely to their respective adapters or internal instrumentation.

3.2.2 Orchestration Component

The orchestration component serves as the control plane for benchmark execution and is implemented by three core elements:

- `BenchmarkRunManager`: Manages the persisted `BenchmarkRun` entity, serving as the single authority for progress counters and lifecycle state transitions.
- `BenchmarkOrchestrator`: Coordinates the execution workflow by initiating runs, resolving target agents and benchmarks, retrieving trace evidence via `LangfuseService`, invoking metric providers through `MetricProviderRegistry`, resolving indicator formulas using `KPIFormulaDslService`, persisting per-trace results and KPI history, and evaluating alerts before closing the run.
- `BenchmarkScheduledTaskRunner`: Monitors recurrence configurations, identifies due tasks, instantiates them as standard runs, and delegates execution to the orchestrator.

The execution state model is explicitly defined to support full operational traceability (R2). A `BenchmarkRun` transitions through a deterministic state lifecycle: created as `PENDING`; moved to `RUNNING` by `BenchmarkRunManager.startRun`; marked as `COMPLETED` by `completeRun` upon storing computed metrics and trace counts; assigned to `FAILED` via `failRun` if errors occur; or set to `CANCELLED` via `cancelRun` upon operator intervention. Execution state is persisted directly within the run record alongside the benchmark ID, agent ID, trigger type, triggering identity, optional schedule ID, Langfuse run name, timestamps, retry counts, and progress counters. All execution triggers converge on a unified asynchronous pipeline. Calling `BenchmarkRunManager.createRun` persists a pending run with its associated provenance metadata and returns a unique run identifier. The caller then invokes the method `BenchmarkOrchestrator.executeBenchmarkRunAsync`, allowing HTTP clients, background schedulers, or event handlers to proceed without blocking on evidence retrieval or metric computation.



On-demand execution. Clients initiate runs via a `POST /api/benchmark-runs` request specifying the benchmark, agent, and optional trace identifiers:

```
POST /api/benchmark-runs
{
    "benchmarkId"     : "<benchmarkId>",
    "agentId"         : "<agentId>",
    "triggeredBy"     : "api-demo",
    "langfuseRunName" : "experiment_00 - 2026-01-22T10:44:46Z"
}
```

The controller validates required fields, initializes a run with a `MANUAL` trigger type, initiates asynchronous processing, and immediately returns a `202 Accepted` response. Clients track progress by querying the run identifier or can abort evaluations by calling the cancellation endpoint, which updates the state to `CANCELLED`.

Scheduled execution. Automated recurrences are configured using a `ScheduleConfig`, which binds a benchmark–agent pair to a cron expression and time zone while maintaining operational status (e.g., active state, execution timestamps, failure thresholds, and success/failure counters):

```
POST /api/schedules
{
    "name"                   : "daily-quality-check",
    "benchmarkId"            : "<benchmarkId>",
    "agentId"                : "<agentId>",
    "cronExpression"         : "0 0 6 * * ?",
    "timezone"               : "UTC",
    "maxConsecutiveFailures" : 3,
    "autoDisableOnFailure"   : true
}
```

A background worker periodically queries for due schedules, creates a run with a `SCHEDULED` trigger type (attributing origin to `scheduler`), calculates the subsequent execution time, and dispatches processing to the orchestrator. Isolating failure counts and auto-disable policies within the schedule model protects system health without corrupting individual run histories.

Event-driven execution. The trigger model also supports `EVENT` and `WEBHOOK` types. These enable automated benchmarking in response to repository updates, dataset modifications, new trace availability, or external system webhooks.

By persisting trigger types alongside originating identities, the framework attributes computed quality indicators directly to the operational context and business rationale behind each evaluation run, satisfying requirement R2.

3.2.3 Service component

The service component groups domain services invoked by the orchestration component during an evaluation run. Each service encapsulates a specific technical responsibility behind a narrow interface, enabling the orchestrator to sequence pipeline execution without embedding backend communications, metric logic, DSL parsing, or alerting rules. The component comprises four primary domain capabilities:

- `LangfuseService`: Serves as the gateway to the observability backend. It instantiates a Langfuse client using agent-specific credentials, falling back to framework-level configurations when necessary. For each run, `fetchTracesFromRun` joins the selected dataset run, dataset items (carrying inputs and expected outputs), complete execution traces, and

pre-existing backend scores. It normalizes this evidence into a `TraceData` object passed directly to metric providers, isolating backend pagination and trace extraction logic from the orchestrator and metric implementations.

- `MetricProviderRegistry`: Acts as the dispatch authority for framework-computed metrics. It indexes Spring-managed provider beans by metric key and exposes them to the orchestrator for scoring.

```
public MetricProviderRegistry(List<MetricProvider<?>> providers) {
    for (var p : providers) {
        byKey.put(p.key(), p);
    }
}
```

- `KPIFormulaDslService`: Operates as a facade between raw metric scores and declared indicators. It parses stored `KPISpecification` records into evaluable `KPIFormula` objects, validates formula syntax prior to execution, and verifies expressions against runtime metric keys. This isolates the grammar and parsing logic from the rest of the framework (Section 3.2.4).

- `AlertEvaluationService`: Converts computed KPI history into operational notifications. Following run persistence, it evaluates active alert rules against threshold conditions for the benchmark and dispatches alerts accordingly.

Metric computation relies on a standardised provider interface encapsulated within the registry:

```
public interface MetricProvider<K extends MetricKey> {
    Metric compute(Agent agent, String referenceText,
                   String generatedText, TraceData trace);
    Class<K> key();
}
```

Providers operate in strict isolation, accepting the evaluated agent, expected/actual outputs, and trace telemetry to compute a single metric value. The registry auto-discovers provider implementations at startup, allowing new metrics to be added without modifying existing pipeline code, satisfying requirements R3 and R5. This design establishes two key architectural properties:

- *Decoupled indicator integration:* New metrics become immediately referenceable within DSL expressions, as their exported keys serve as recognized formula identifiers.

- *Uniform provider execution:* The orchestrator executes domain-specific metrics, LLM-based judges, and deterministic calculations (e.g., token usage) through a single unified interface.

Furthermore, `TraceData` includes pre-computed scores from the observability platform (such as human annotations or agent-level instrumentation), allowing providers to incorporate external feedback without dedicated integrations.

3.2.3.1 Catalogue of metrics

Metric key	Provider	Operational definition	Range
`ROUGE`	`Rouge MetricProvider`	Computes a ROUGE-L-style F-measure over the longest common subsequence between tokenised reference and generated outputs.	$[0, 1]$ score
`BLEU`	`Blue MetricProvider`	Computes the standard BLEU score between tokenised reference and generated outputs.	$[0, 1]$ score
`ACCURACY`	`Accuracy MetricProvider`	Computes token-set overlap as the size of the intersection between reference and generated token sets divided by their union.	$[0, 1]$ score
`PRECISION`	`Precision MetricProvider`	Computes the share of unique generated tokens that also occur in the reference output.	$[0, 1]$ score
`RECALL`	`Recall MetricProvider`	Computes the share of unique reference tokens that are recovered in the generated output.	$[0, 1]$ score
`F1_SCORE`	`F1Score MetricProvider`	Computes the harmonic mean of the token-set precision and recall defined above.	$[0, 1]$ score

Table 8: Built-in metric providers currently exposed by the MOSAICO benchmarking framework.

While the extension mechanism defines how new metrics are registered and executed, the actual evaluative capability of the framework depends on the concrete metrics provided. Decoupling the extensible architecture from the specific catalogue contents ensures that measurement mechanisms remain independent of the metrics currently implemented.

The metric catalogue acts as the operational bridge between raw framework measurements and the overarching quality model. Because quality indicators rely directly on these underlying computations, each metric in the catalogue maps to a specific quality characteristic defined within the target quality framework.

The current implementation exposes the built-in metric keys listed in Table 8. They are provider-backed metrics: each key corresponds to a `MetricProvider` implementation discovered by the `MetricProviderRegistry`, computed for each trace for which both the expected and generated outputs are available, and then persisted as a `MetricSnapshot`. At this stage the built-in catalogue is therefore output-oriented. All providers receive the trace context through the common interface described above, but the implemented metrics primarily compare the reference output with the generated output.

The catalogue should therefore be interpreted as a set of low-level measurements rather than as a fixed set of high-level quality indicators. Indicator meaning is assigned by the KPI specification layer: the same metric can contribute to different quality characteristics depending on the formula and benchmark context declared through the quality model of Deliverable D2.2. Conversely, the current names `ACCURACY`, `PRECISION`, `RECALL`, and `F1_SCORE` denote lexical token-overlap measures in the framework implementation; they must not be read as task-independent classification metrics unless the benchmark specification defines the reference and generated outputs accordingly.

In addition to provider-computed values, the framework can consume metric-like scores already attached to execution traces in the observability backend. These externally produced values, such as `ROUGE1_F`, `ROUGEL_F`, `COSINE_PRED_GOLD`, `COSINE_PRED_SOURCE`, and `LEN_RATIO`,



are registered as formula keys when required by a benchmark and normalised by the trace metrics aggregation layer. They extend the metric vocabulary available to KPI formulae, but they are not built-in providers unless a corresponding `MetricProvider` is supplied.

3.2.3.2 Extending the catalogue of metrics

Metric providers receive four parameters: the evaluated agent, the reference output, the generated output, and the execution trace (`TraceData`). While text-comparison metrics rely primarily on expected and actual outputs, behavioural metrics evaluate the execution trace itself. The implementation below illustrates how to introduce a trace-based metric that counts total tool invocations:

```
public class ToolCallCountMetric implements MetricKey { }

@Service
public class ToolCallCountMetricProvider
        implements MetricProvider<ToolCallCountMetric> {

    @Override
    public Metric compute(Agent agent, String referenceText,
                          String generatedText, TraceData trace) {
        long calls = trace.trace.getObservations().stream()
                .filter(this::isToolInvocation)
                .count();

        Metric metric = new Metric();
        metric.setName("Tool calls");
        metric.setFloatValue((float) calls);
        metric.setUnit("count");
        return metric;
    }

    @Override
    public Class<ToolCallCountMetric> key() {
        return ToolCallCountMetric.class;
    }
}
```

Adding a metric consists of adding a key class and a provider that declares it: no existing class is edited and no list is extended, which is what makes the catalogue open to extension and closed to modification. The new identifier becomes usable in indicator expressions once it is published to the language through the interface of the next section. Nothing in the mechanism distinguishes a measure of an agent's behaviour from a measure of its output, which is what allows future catalogue entries to combine outcome, trace, cost, collaboration, or judge-based evidence under the same execution model.

3.2.4 DSL component and integration

Requirement R6 mandates that quality indicators be declared in a dedicated specification language anchored to the underlying quality model rather than hardcoded within the evaluation engine. Within the framework, a KPI specification is treated as an immutable string. It is authored independently, stored as written, and evaluated without the engine holding explicit internal models of the grammar or operators. Instead, formula parsing and validation are completely delegated to implementations of the `KPIFormulaParser` interface.

The KPIFormulaParser interface defines the contract governing expression parsing, validation, and metric key registration:

```
public interface KPIFormulaParser {
    DslParseResult parse(String expression);
    DslParseResult validate(String expression);
    Set<String> getKnownMetricKeys();
    void registerMetricKeys(Set<String> keys);
}
```

These four methods encapsulate the framework's entire interaction with the specification language. The parse method transforms a stored formula string into an evaluable KPIFormula instance. The validate method checks syntactic and semantic correctness at benchmark declaration time, returning structured diagnostics (including error locations and severity levels) before any evaluation runs are initiated. The registerMetricKeys and getKnownMetricKeys methods maintain a shared vocabulary, allowing the framework to inform the parser of available metric providers so that formulas referencing unsupported metrics are rejected during declaration.

KPI expressions explicitly declare input metrics before combining them. The listing below illustrates a representative DSL specification that declares two input metrics and computes their arithmetic mean:

```
1 ROUGE of type real
2 BLEU  of type real
3 KPI quality of type real
4 quality = AVERAGE([ROUGE, BLEU])
```

Expression processing occurs across two distinct execution phases:

- **Declaration Phase:** The expression is validated by the parser. The engine verifies syntax correctness and ensures that all referenced metric keys (ROUGE, BLEU) match registered providers. If an expression references an unmapped metric identifier, declaration fails immediately with precise diagnostic locations.
- **Evaluation Phase:** The framework retrieves trace evidence, executes the corresponding metric providers, passes the resulting metric vector to the instantiated KPIFormula object, and records the computed scalar result.

The primary implementation bound to this interface is XtextKPIFormulaParser, which wraps the parser generated from the KPI Formula DSL defined in Deliverable D2.2. The framework supplies the set of active provider keys, while the Xtext parser manages grammar parsing, identifier resolution, and AST construction.

Figure 4 illustrates the component architecture. The concrete XtextKPIFormulaParser encapsulates the translation logic between the external DSL and the internal KPIFormula interface, insulating the core evaluation engine from changes to the grammar or parsing infrastructure.

3.2.5 Persistence component

Figure 5 illustrates an excerpt of the relational data model, which organizes framework domain entities into three primary logical groups: benchmark configurations, execution run records, and KPI historical tracking.

A Benchmark represents a reusable evaluation configuration. It specifies the dataset reference (datasetRef), the target task definition (taskDef), and associated indicator definitions.

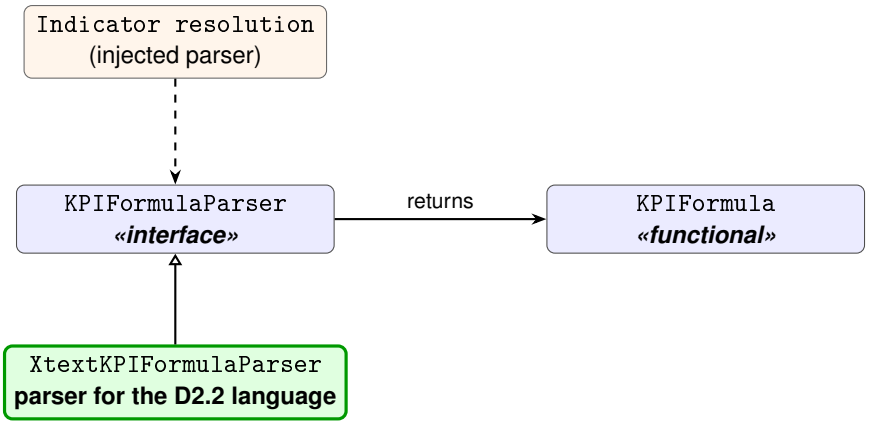


Figure 4: The `KPIFormulaParser` interface and its implementation.

Decoupling configuration from execution enables identical benchmarks to be repeatedly evaluated across multiple agent versions or temporal runs.

A `PerformanceKPI` binds individual indicator definitions to a parent `Benchmark`. It holds human-readable metadata (`description`) and embeds a `KPISpecification`. `KPISpecification` is an embedded value object storing the exact, immutable formula definition. It retains the raw authoring text (`dslText`), formula type (`formulaType`), and language version (`dslVersion`), alongside a transient parsed object (`formula`). Preserving the raw specification alongside the compiled representation guarantees long-term auditability even as the DSL grammar evolves.

A `BenchmarkRun` records a discrete execution of a benchmark against an evaluated agent. It maintains execution lifecycle state (`status`) across `PENDING`, `RUNNING`, `COMPLETED`, `FAILED`, or `CANCELLED` transitions, alongside execution provenance (`triggeredBy`).

`BenchmarkResult` captures evaluation outputs for an individual dataset item, linking the associated trace identifier (`traceId`) to its computed indicator map (`kpiValues`). Inspecting item-level results allows aggregate evaluation anomalies to be isolated to specific dataset instances.

`MetricSnapshot` stores fine-grained, item-level metric measurements (`metricKey`, `value`, `timestamp`). Retaining observations at this level satisfies requirement R3, enabling historical traces to be re-evaluated under newly declared indicator formulas without re-executing target agents.

`KPIHistory`: tracks aggregated indicator scores across evaluation runs (`value`, `baseline`, `delta`, `status`). Appending historical entries rather than overwriting existing values records performance trajectories over time, helping distinguish performance shifts caused by prompt updates, model iterations, or updated datasets.

Auditability across evaluation runs is supported by a dual-store strategy (R7). Detailed execution traces, span trees, and raw tool interactions are archived within the Langfuse observability platform. Concurrently, structural execution metadata, spanning `BenchmarkRun`, `BenchmarkResult`, `MetricSnapshot`, and `KPIHistory` records, is maintained within a relational PostgreSQL instance[3] via Spring Data JPA[4]. The external trace identifier (`traceId`) serves as the join key between the relational metadata and the observability backend, establishing an end-to-end audit chain from high-level KPI trends down to raw interaction logs.

In addition to relational domain entities, the PostgreSQL database leverages the `pgvector` extension[5] managed through Spring AI vector stores. Co-locating relational records and vector

[3] `https://www.postgresql.org/`
[4] `https://spring.io/projects/spring-data-jpa`
[5] `https://github.com/pgvector/pgvector`

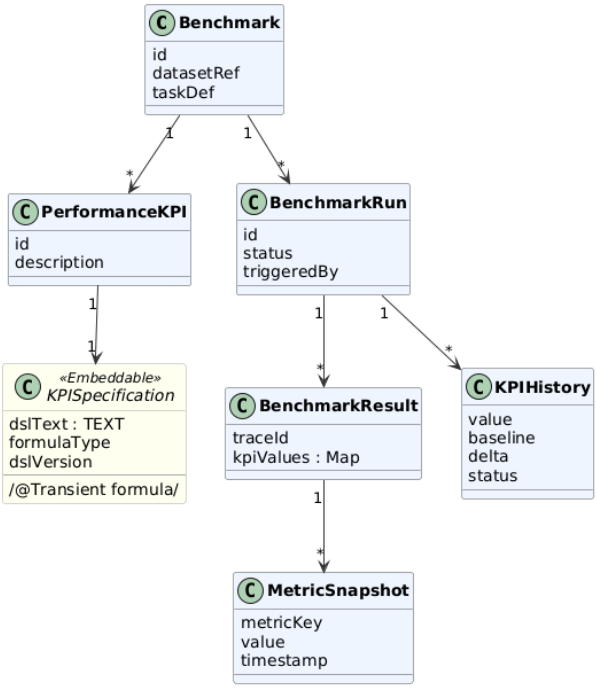


Figure 5: Excerpt of the data model of the benchmarking framework.

embeddings within a single database instance eliminates cross-system synchronization overhead, ensuring that vector-based agent discovery remains consistent with recorded evaluation evidence.

Persisted evaluation evidence also contributes to agent indexing. When an agent is indexed, `VectorSearchService` builds its embedded representation from declared metadata, skills, tools, benchmark KPI specifications, and the distinct KPI names already measured for that agent. The numerical values remain in `BenchmarkResult` and `KPIHistory`; they are exposed through MCP tools for historical retrieval and filtering against KPI thresholds. Semantic MCP search can therefore match agents by skill and evaluation vocabulary, while KPI-based selection remains grounded in persisted records rather than in the embedding itself.

3.2.6 Interface component

Satisfying requirement R7 requires programmatic accessibility to stored benchmarking evidence, moving agent selection beyond static filtering by declared capabilities toward ranking by demonstrated performance (Section 2.5.2). To support distinct access patterns, the framework separates administration from consumption into two programmatic surfaces: a management surface for human operators and CI pipelines, and a lightweight, read-only surface for automated agents.

3.2.6.1 REST API

The management interface exposes benchmarks, agents, indicator specifications, runs, schedules, and alert configurations as HTTP resources. Beyond baseline CRUD operations, three core architectural mechanisms support system requirements:

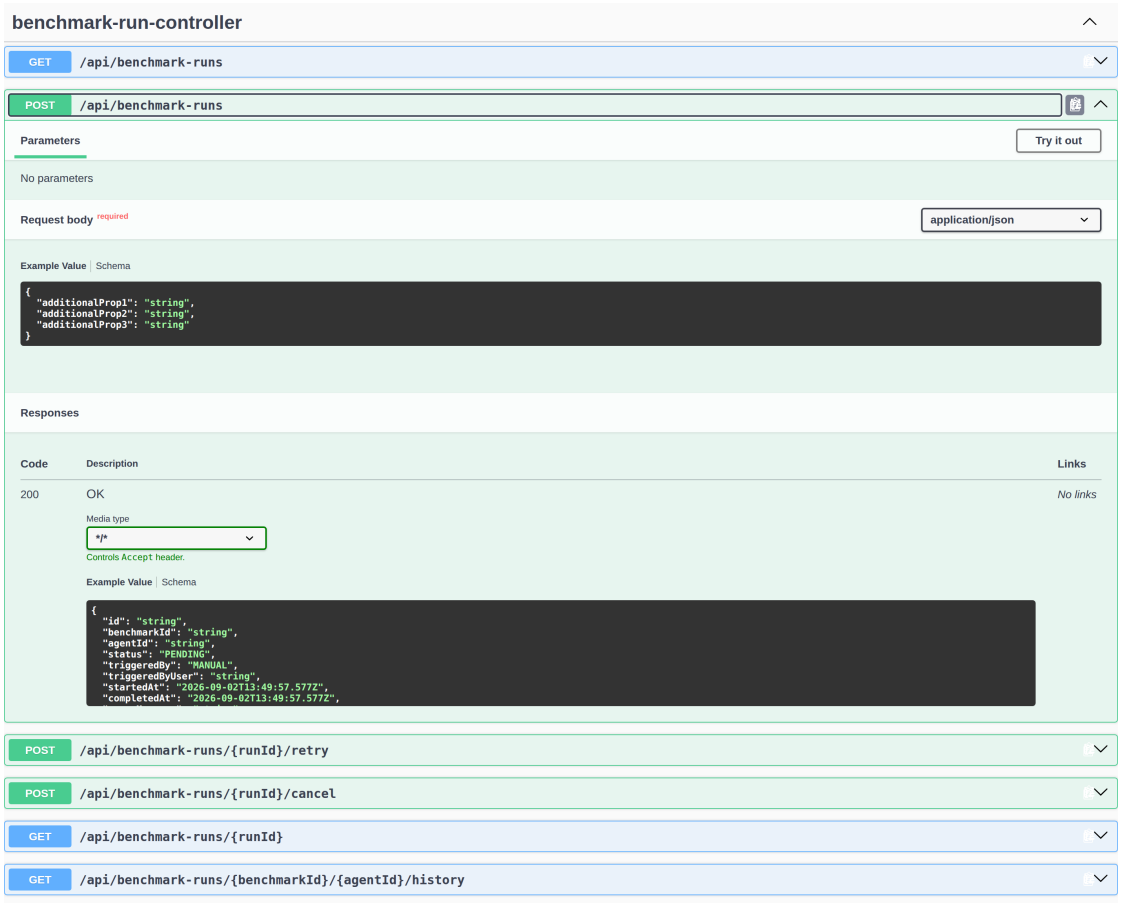


Figure 6: The run lifecycle on the management surface, with the request body and the acknowledgement returned by `POST /api/benchmark-runs`.

- *Asynchronous execution handling:* Immediate asynchronous run acknowledgement returns a unique run identifier prior to execution. Callers monitor progress by polling this identifier, guaranteeing trace-to-run linkage (R2) while keeping partially completed executions inspectable. Figure 6 shows the six operations through which a run is governed.
- *Pre-execution formula validation:* Formula validation (see Figure 7) parses and verifies indicator specifications against registered namespaces without triggering evaluations, enabling static correctness checking prior to runtime (Section 3.1).
- *Time-series KPI retrieval:* Historical series querying exposes continuous evaluation records rather than isolated execution snapshots, enabling longitudinal quality tracking across model and prompt iterations. Figure 8 shows the agent group, whose search operations are the management-side counterpart of the available MCP tools.

3.2.6.2 MCP Resources and tools

To support automated selection by consensus agents, evaluation evidence is exposed natively via MCP, which is the surface that organizes access into two distinct abstractions:

- *Tools:* Executable query interfaces enabling skill-based filtering, natural-language semantic search, historical evaluation retrieval, and filtering against KPI thresholds.

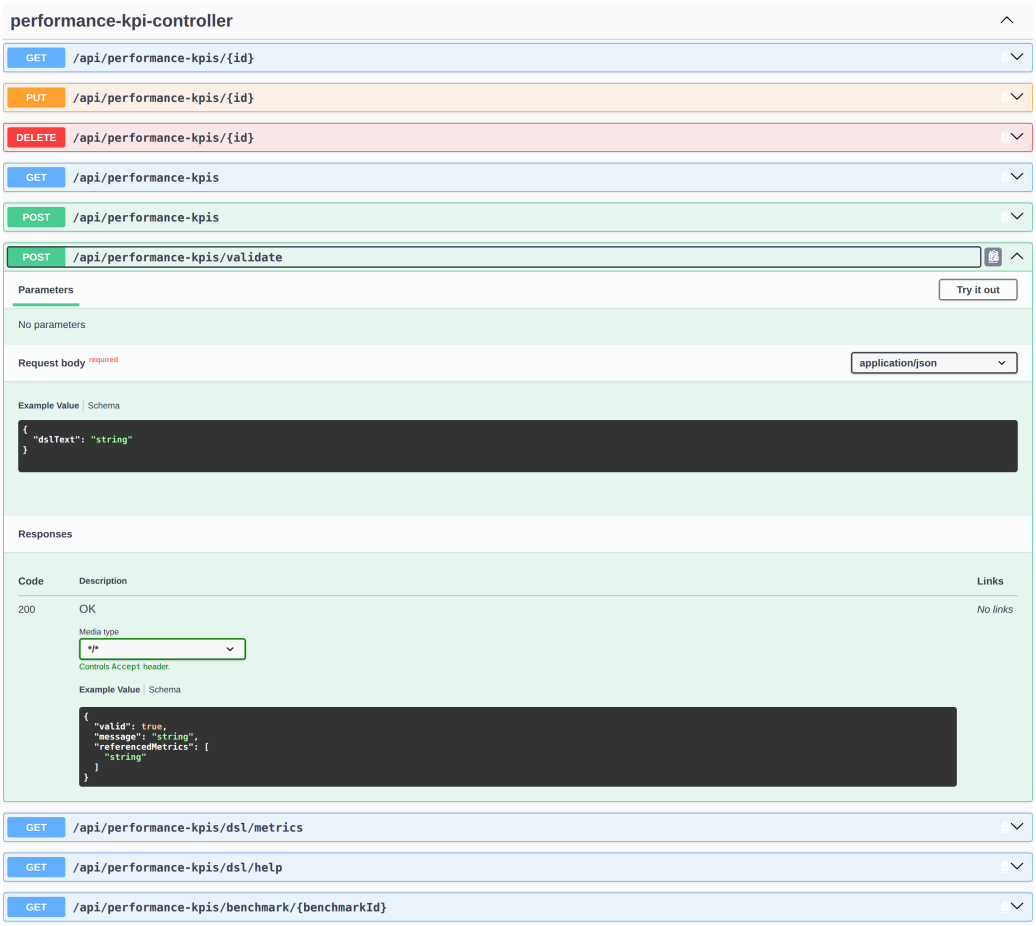


Figure 7: The indicator group, with `POST /api/performance-kpis/validate` expanded: an expression is submitted as text and answered with a verdict and the metric identifiers it resolved against.

- *Resources:* Read-only documents providing static definitions of agents, skills, and benchmark configurations.

Figure 9 shows the search-oriented part of the tool surface provided by the repository server. The `SearchByTextTool` serves as the primary semantic entry point, accepting natural-language queries optionally bounded by required skill names and supported model architectures. `SearchBySkillTool` and `BenchmarksBySkillTool` provide exact-match discovery, while `AgentBenchmarkResultsTool` and `AgentLastBenchmarkRunTool` retrieve historical performance measurements. A separate query over KPI thresholds returns agents whose persisted KPI values satisfy the configured threshold constraints for a requested skill. Detailed parameter schemas embedded within each tool specification enable consuming agents to dynamically discover and invoke these endpoints without requiring custom integrations. KPI thresholds operationalize requirement R7 by returning candidate agents that satisfy configured numerical constraints for a given skill. Retrieval combines exact skill matching, semantic vector search over enriched agent descriptions (Section 3.2), and deterministic checks against persisted KPI values.

Figure 10 depicts the resource structure, which exposes three primary collection URIs (`agents`, `skills`, `benchmarks`) along with parametrized resource templates for identifier-based access. Served as structured JSON documents, these endpoints allow agents to retrieve static

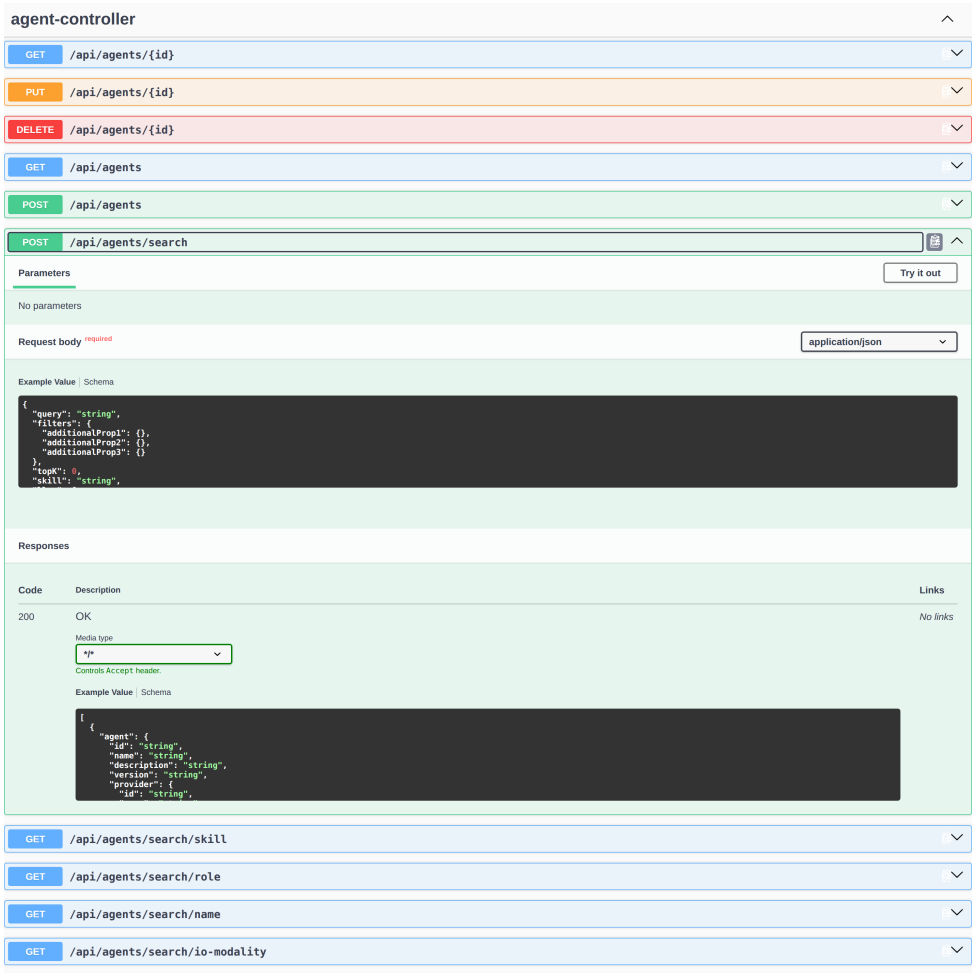


Figure 8: Agent retrieval on the management surface, with the semantic entry point `POST /api/agents/search` expanded beside its four exact-match counterparts.

metadata, such as benchmark dataset references, evaluated agent roles, objectives, and declared attributes, alongside empirical execution evidence through a single unified surface.

To protect system resources and maintain evaluation integrity, the MCP surface operates under strict read-only constraints. Benchmark execution, schedule management, and administrative configuration remain exclusively restricted to authenticated callers on the REST management surface.

3.3 The benchmarking flow

The dynamic behaviour of the benchmarking engine is governed by an asynchronous execution flow that processes evaluation requests, collects execution telemetry, computes low-level metrics, and resolves high-level indicators. Figure 11 depicts the sequence of interactions across internal framework services and external storage systems. An evaluation lifecycle progresses through five consecutive phases sequenced by the `BenchmarkOrchestrator`:

1. **Admission:** A client issues an evaluation request specifying a benchmark and target agent identifier to the `RESTController`. The `BenchmarkRunManager` initializes a new `BenchmarkRun` record in the PostgreSQL database with a `PENDING` status and returns a

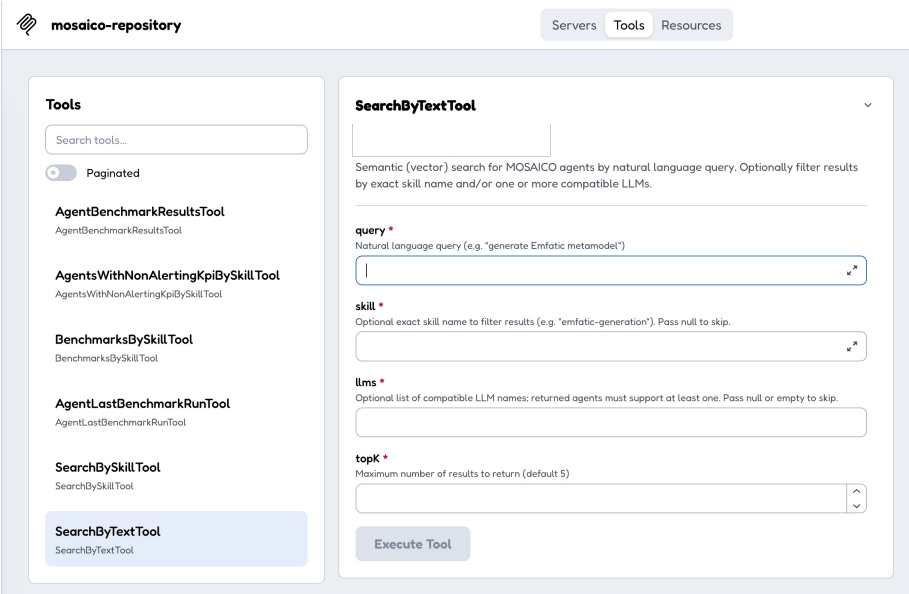


Figure 9: MCP tools exposed on the agent-facing surface of the MOSAICO repository, with the parameters of `SearchByTextTool`.

unique `runId`. The controller immediately responds with `202 Accepted`, allowing clients to poll execution status asynchronously while guaranteeing that run artefacts are anchored to a valid identifier before processing begins.

2. **Validation:** Upon initiating execution via `executeBenchmarkRun`, the `BenchmarkRunManager` updates the run status to `RUNNING` and loads the referenced `Benchmark` and `Agent` specifications. If either entity fails to resolve, the run transitions directly to `FAILED` status and aborts, preventing unnecessary telemetry processing costs.

3. **Trace Collection:** The `LangfuseService` queries the external Langfuse API for dataset items and execution traces associated with the target run. Retained traces, containing input prompts, generated outputs, expected references, and tool invocation metadata, are fetched, normalized into the internal framework trace model, and returned to the orchestrator. Where several agents contributed to one task, what is retrieved is a tree rather than a flat list, since the spans were linked at the run boundary; the nesting is what any collaboration metric is computed over.

4. **Per-Trace Metric Computation:** The orchestrator queries the `MetricProviderRegistry` for all registered measurement providers. For each normalized trace, every provider computes its assigned metric value using the target agent, reference output, generated output, and trace context. Each calculated metric is immediately written to PostgreSQL as an immutable `MetricSnapshot`, preserving partial evaluation evidence even in the event of subsequent process interruptions.

5. **Aggregation and KPI Evaluation:** Once per-trace metrics are stored, the orchestrator aggregates snapshot values into a unified metric vector per key using an arithmetic mean. For each `PerformanceKPI` defined by the benchmark, the `KPIFormulaDslService` parses the declarative expression string into an executable lambda instance. The expression evaluates over the metric vector, and the resulting KPI values are persisted into `KPIHistory`.

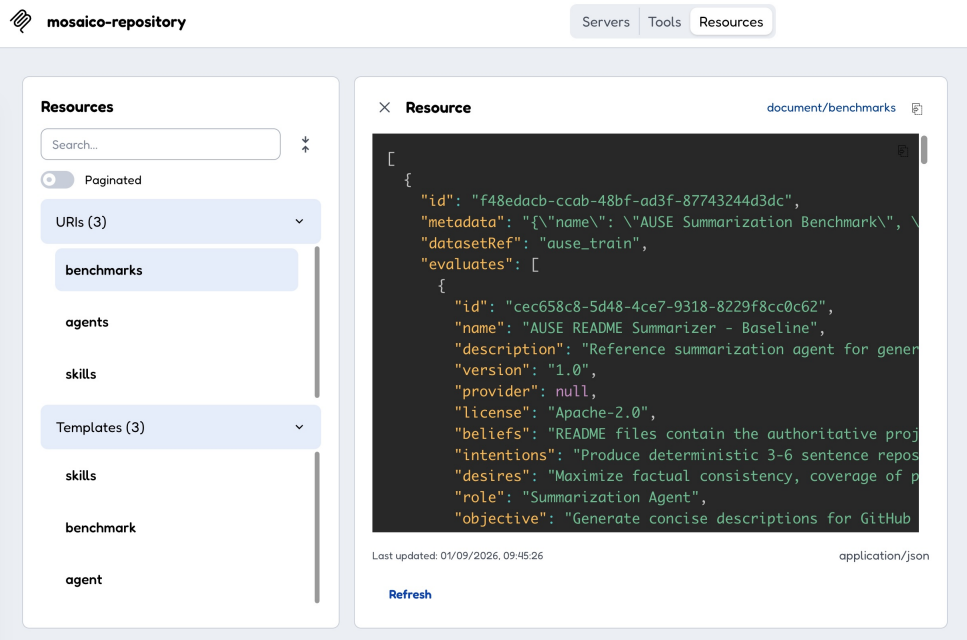


Figure 10: MCP resources exposed on the same surface, showing the benchmark collection returned as a JSON document.

Upon successful completion of all evaluation loops, the orchestrator invokes `completeRun`, recording execution counters and marking the run status as `COMPLETED`. Any unhandled exception during any phase triggers an explicit fail-safe transition, updating the run status to `FAILED` alongside an error message.

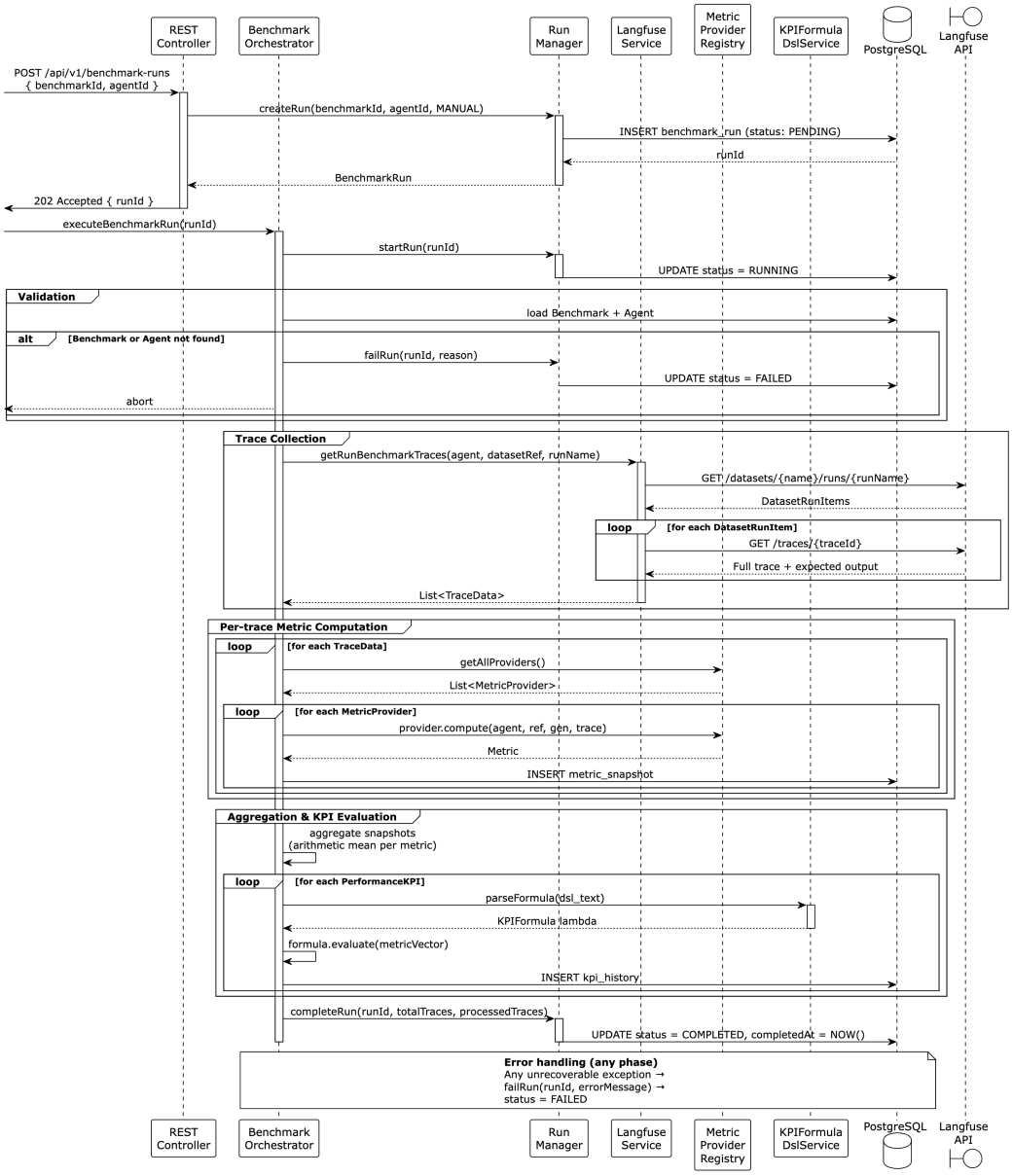


Figure 11: MOSAICO benchmark flow from run admission to trace collection, per-trace metric computation, indicator evaluation, persistence, and completion or failure.

3.4 An explanatory use of the MOSAICO benchmarking framework

This section provides a conceptual example of how the MOSAICO benchmarking framework can be configured for assessing AI agents that identify and correct issues in software requirements. The example illustrates the specification of a benchmark, the declaration of KPI indicators, and the implementation of a domain-specific metric provider. It is intended to demonstrate the framework's configuration and extension mechanisms; it does not report an empirical evaluation of the framework or of the evaluated agents.

3.4.1 Benchmark specification

A benchmark is declared through the management surface (Section 3.2.6.1) as a set of records: the skills it assesses, the benchmark, and one or more KPI indicators. These records instantiate the specification model of Section 3.2.4 and are persisted as the entities of Section 3.2.5. Listing 2 gives them for this scenario. The listing provides an abstraction of the sequence of data required by the system to fully specify a benchmark. It captures the main records involved in the benchmark definition, including the benchmark itself, its indicators, and the runs associated with it, while preserving the field names used by the implementation. A complete step-by-step description of how these elements are instantiated programmatically is provided in the benchmark system guide.[6]

```
{
  "metadata": "{\"name\": \"Requirement review\", \"version\": \"1.0\",
               \"task\": \"requirements-review\"}",
  "datasetRef": "requirement_review",
  "taskDef": "Given a set of requirements and the writing rules
              in force, report every requirement that violates a rule, one
              row per finding, as (requirement id, rule id, issue)",
  "features": "unambiguity,verifiability,completeness,traceability",
  "protocolVersion": "1.0",
  "assess": [
    { "id": "skill-requirements-review" },
    { "id": "skill-requirements-decomposition" }
  ],
}

// one further record per indicator, bound to the benchmark by identifier
{
  "benchmarkId": "<the benchmark declared above>",
  "description": "Review effectiveness",
  "dslText":     "<the specification of Listing 3>"
}

// one run per agent, naming the recorded execution whose traces are read
{
  "benchmarkId":     "<the benchmark declared above>",
  "agentId":         "<the agent under evaluation>",
  "triggeredBy":     "wp5-demo",
  "langfuseRunName": "<the execution recorded in the observability backend>"
}
```

Listing 2: Abstraction of the data sequence required to fully specify a benchmark in the system.

Two fields carry the task-specific content. `taskDef` states the task and the required output format. The exchange of Deliverable D5.1, Section 5.3.2.4, already prescribes that format. Adherence to it is the condition under which an answer is comparable to a reference. `datasetRef`

[6] https://gitlab.eclipse.org/eclipse-research-labs/mosaico-project/mosaico-repository/-/blob/master/docs/DEMO_INTERACTIVE_BENCHMARK_DSL_EN.md?ref_type=heads.

names the dataset whose items supply input and expected output: one item per requirement examined in a review round, the expected output being the issues that requirement carries and the rules they violate. `assess` references skill entities, which agents also reference to declare their capabilities. The link is required by the agent-facing surface (Section 3.2.6.2), where KPI thresholds select agents by skill and persisted indicator state (R7). The two skills declared here, reviewing statements against a rule set and decomposing a parent requirement into children, are exercised by different turns of the exchange and may be satisfied to different degrees by one agent.

Indicators are not fields of the benchmark. Each is a separate record bound to it by identifier and carrying its specification as text in `dslText`, which is the form the indicator language reads and the form that is retained (Section 3.2.5). Three consequences follow. An indicator can be added, reweighted or withdrawn without modifying the benchmark. A specification can be validated before storage, the response naming the metric keys the expression refers to, so that a formula is checked against the catalogue at authoring time rather than at run time (Section 3.2.4). And retaining the authored text, not only its evaluable form, keeps a recorded value readable back to the expression that produced it. A run is requested per agent. The request names the benchmark, the agent, and, in `langfuseRunName`, the execution whose traces are to be read. The framework does not invoke the agent (Section 3.2.1), so the evidence must already be recorded under that name; a name matching no recorded execution is the condition under which a run does not progress. The call returns a run identifier immediately and evaluation proceeds asynchronously through the states of Section 3.2.5, reporting on completion the number of traces processed and of measurements computed. Runs may equally be started on a schedule or on a repository event (Section 3.2.2), which is how the indicator series that `Recall_Drift` reads is produced. Execution leaves the benchmark record unchanged, which is the condition under which several agents, or one agent at successive times, are compared against the same declaration.

3.4.2 Quality definition

Listing 3 formalizes the evaluation scheme for assessing candidate requirement-review agents by enforcing a strict acceptance gate on detection recall, assigning intermediate weight to rule attribution, and reporting precision separately as an explicit review burden.

```
// the quantities the domain providers expose, declared before use
TruePositives         of type real
FalseNegatives        of type real
FalsePositives        of type real
RulePrecision         of type real
ReqRuleRecall         of type real
RulesSolved           of type real
RulesNewlyViolated    of type real
RulesViolated         of type real
ReviewedRequirements  of type real

KPI Detection_Recall       of type real
KPI Detection_Precision    of type real
KPI Attribution_F1         of type real
KPI Review_Effectiveness   of type real
KPI Repair_Yield           of type real
KPI Review_Burden          of type real
KPI Safety_Gate            of type real
KPI Recall_Drift           of type drift of Average for 5 measurements

Detection_Recall      = TruePositives / (TruePositives + FalseNegatives)
Detection_Precision   = TruePositives / (TruePositives + FalsePositives)
Attribution_F1        = 2 * ((RulePrecision * ReqRuleRecall) /
                             (RulePrecision + ReqRuleRecall))
```

```
25
26 // Recall dominates: a missed defect propagates into code and test
27 // procedures, whereas a spurious finding costs reviewer time only.
28 Review_Effectiveness = WEIGHTED_SUM([Detection_Recall, Attribution_F1,
29                                     Detection_Precision],
30                                    [0.50, 0.30, 0.20])
31
32 // A rewriting that resolves one rule by breaking another is not a repair.
33 Repair_Yield          = (RulesSolved - RulesNewlyViolated) / RulesViolated
34 Review_Burden         = FalsePositives / ReviewedRequirements
35
36 Safety_Gate           = THRESHOLD(Detection_Recall, 0.95)
37 Recall_Drift          = Detection_Recall
```

Listing 3: Indicators for the requirements-review scenario, in the KPI Formula DSL.

Listing 3 therefore places the acceptance gate on recall, admits attribution into the composite at an intermediate weight, and keeps precision in the composite while also exposing it separately as a review burden, that is the number of spurious findings a reviewer must process per document, which is the form in which imprecision is actually felt.

This scenario requires a domain-specific provider that compares the expert-reviewed issue table with the issue table produced by the agent. The implementation in Listing 4 follows the extension mechanism of Section 3.2.3.1: the provider parses both artefacts as sets of `(requirementId, ruleId)` pairs and reports the share of expected rule violations recovered by the agent. The resulting provider key is published to the indicator layer as `ReqRuleRecall` and used directly by Listing 3.

```java
public class ReqRuleRecallMetric implements MetricKey { }

@Service
public class ReqRuleRecallMetricProvider
        implements MetricProvider<ReqRuleRecallMetric> {

    @Override
    public Metric compute(Agent agent, String referenceText,
                          String generatedText, TraceData trace) {
        Set<String> expected = parseFindingPairs(referenceText);
        Set<String> reported = parseFindingPairs(generatedText);

        Set<String> recovered = new HashSet<>(expected);
        recovered.retainAll(reported);

        float value = expected.isEmpty()
                ? 1.0f
                : (float) recovered.size() / expected.size();

        Metric metric = new Metric();
        metric.setName("ReqRuleRecall");
        metric.setFloatValue(value);
        metric.setUnit("score");
        return metric;
    }

    private Set<String> parseFindingPairs(String text) {
        if (text == null || text.isBlank()) {
            return Set.of();
        }
        Set<String> pairs = new HashSet<>();
        for (String line : text.split("\\R")) {
            String cleaned = line.trim();
            if (cleaned.isBlank() || cleaned.startsWith("#")) {
```

```
                continue;
            }
            String[] cells = cleaned.split("[,;]");
            if (cells.length < 2) {
                continue;
            }
            String requirementId = cells[0].trim().toLowerCase();
            String ruleId = cells[1].trim().toLowerCase();
            if (!requirementId.isBlank() && !ruleId.isBlank()) {
                pairs.add(requirementId + "::" + ruleId);
            }
        }
        return pairs;
    }

    @Override
    public Class<ReqRuleRecallMetric> key() {
        return ReqRuleRecallMetric.class;
    }
}
```

Listing 4: Self-contained provider for requirement rule recall.

4 Conclusion and future work

This deliverable completes the operational step of WP2 by connecting the classification layer of D2.1 and the quality/KPI layer of D2.2 to an executable benchmarking infrastructure. Through a structured literature review, an analysis of existing infrastructure tools, and an implementation-oriented architectural design, the document defines how heterogeneous AI agents for software engineering can be evaluated under a common process while preserving reproducibility, extensibility, and traceability.

The primary outcome of this work is a reusable benchmarking infrastructure that satisfies three key operational properties: (i) tasks, configurations, and execution environments are explicitly and uniformly declared; (ii) execution evidence is retained independently of indicator formulas; and (iii) metric and architectural evolution is managed via stable extension boundaries. Together, these properties establish the necessary architectural foundation for generating comparable empirical evidence across heterogeneous agent systems, directly supporting repository-driven discovery and automated selection decisions, including the following operational properties:

- *Explicit specification decoupling:* Evaluation tasks, agent configurations, dataset references, and execution environments are declared uniformly and independently of execution logic.
- *Decoupled evidence retention:* Execution traces are retained independently of indicator formulas, enabling retrospective rescoring as quality models evolve without requiring agent re-execution.
- *Stable extension boundaries:* The framework accommodates evolving custom metric providers, and external observability platforms across stable interfaces.

Future work will proceed along two complementary directions. Within WP2, Task 2.4 will continue refining and extending the repository infrastructure to onboard new metric providers as requested by industrial partners during ongoing use-case implementations. Concurrently, close integration with WP5 will advance the full empirical deployment and evaluation of the industrial demonstrators, validating the framework across real-world software engineering environments.

References

[1] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao, "ReAct: Synergizing Reasoning and Acting in Language Models", *in*: *International Conference on Learning Representations*, 2023, URL: https://openreview.net/forum?id=WE_vluYUL-X. [Cited on pages 9, 26]

[2] 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 pages 9, 26]

[3] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al., "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation", *in*: *arXiv preprint arXiv:2308.08155* (2023). [Cited on pages 9, 26]

[4] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber, "MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework", *in*: *International Conference on Learning Representations*, 2024. [Cited on pages 9, 26]

[5] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun, "ChatDev: Communicative Agents for Software Development", *in*: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024, pp. 15174–15186, DOI: 10.18653/v1/2024.acl-long.810. [Cited on pages 9, 26]

[6] Junda He, Christoph Treude, and David Lo, "LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision and the Road Ahead", *in*: *ACM Transactions on Software Engineering and Methodology* (2025), DOI: 10.1145/3712003. [Cited on page 9]

[7] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan, "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?", *in*: *International Conference on Learning Representations*, 2024, URL: https://www.swebench.com/. [Cited on pages 9, 25, 28]

[8] John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press, "SWE-bench Multimodal: Do AI Systems Generalize to Visual Software Domains?", *in*: *The Thirteenth International Conference on Learning Representations*, 2025, URL: https://openreview.net/forum?id=riTiq3i21b. [Cited on page 9]

[9] 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*: *arXiv preprint arXiv:2507.12415* (2025). [Cited on page 9]

[10] Yijia Xiao, Runhui Wang, Luyang Kong, Davor Golac, and Wei Wang, "CSR-Bench: Benchmarking LLM Agents in Deployment of Computer Science Research Repositories", *in*: *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, ed. by Luis Chiruzzo, Alan Ritter, and Lu Wang, Albuquerque, New Mexico: Association for Computational Linguistics, Apr. 2025, pp. 12705–12723, ISBN: 979-8-89176-189-6, DOI: 10.18653/v1/2025.naacl-long.633, URL: https://aclanthology.org/2025.naacl-long.633/. [Cited on pages 9, 19]

[11] Xuehang Guo, Xingyao Wang, Yangyi Chen, Sha Li, Chi Han, Manling Li, and Heng Ji, "SyncMind: Measuring Agent Out-of-Sync Recovery in Collaborative Software Engineering", *in*: *Proceedings of the 42nd International Conference on Machine Learning*, vol. 267, Proceedings of Machine Learning Research, PMLR, 2025, pp. 20993–21066, URL: https://proceedings.mlr.press/v267/guo25l.html. [Cited on pages 9, 20, 29]

[12] Yuxuan Zhu, Antony Kellermann, Dylan Bowman, Philip Li, Akul Gupta, Adarsh Danda, Richard Fang, Conner Jensen, Eric Ihli, Jason Benn, Jet Geronimo, Avi Dhir, Sudhit Rao, Kaicheng Yu, Twm Stone, and Daniel Kang, "CVE-Bench: a benchmark for AI agents' ability to exploit real-world web application vulnerabilities", *in*: *Proceedings of the 42nd International Conference on Machine Learning*, ICML'25, Vancouver, Canada: JMLR.org, 2025. [Cited on page 9]

[13] Ryo Fujii, Makoto Morishita, Kazuki Yano, and Jun Suzuki, "TimeMachine-bench: A Benchmark for Evaluating Model Capabilities in Repository-Level Migration Tasks", *in*: *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics*, 2026, pp. 8233–8264, DOI: 10.18653/v1/2026.eacl-long.385. [Cited on page 9]

[14] Sergio Morales, Robert Clarisó, and Jordi Cabot, "A DSL for testing LLMs for fairness and bias", *in*: *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, 2024, pp. 203–213. [Cited on page 10]

[15] Barbara Kitchenham and Stuart Charters, *Guidelines for Performing Systematic Literature Reviews in Software Engineering*, tech. rep. EBSE-2007-01, Keele University and Durham University, 2007. [Cited on page 12]

[16] Matthew J. Page, Joanne E. McKenzie, Patrick M. Bossuyt, Isabelle Boutron, Tammy C. Hoffmann, Cynthia D. Mulrow, Larissa Shamseer, Jennifer M. Tetzlaff, Elie A. Akl, Sue E. Brennan, et al., "The PRISMA 2020 Statement: An Updated Guideline for Reporting Systematic Reviews", *in*: *BMJ* 372 (2021), n71, DOI: 10.1136/bmj.n71. [Cited on page 12]

[17] Louis Milliken, Sungmin Kang, and Shin Yoo, "Beyond pip Install: Evaluating LLM Agents for the Automated Installation of Python Projects", *in*: *Proceedings of the 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2025, pp. 1–11, DOI: 10.1109/SANER64311.2025.00009. [Cited on page 19]

[18] Peiran Wang, Xiaogeng Liu, and Chaowei Xiao, "CVE-Bench: Benchmarking LLM-based Software Engineering Agent's Ability to Repair Real-World CVE Vulnerabilities", *in*: *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, Association for Computational Linguistics, 2025, pp. 4207–4224, DOI: 10.18653/v1/2025.naacl-long.212, URL: https://aclanthology.org/2025.naacl-long.212/. [Cited on page 19]

[19] Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel, "Otter: Generating Tests from Issues to Validate SWE Patches", *in*: *Proceedings of the 42nd International Conference on Machine Learning*, vol. 267, Proceedings of Machine Learning Research, PMLR, 2025, pp. 752–771, URL: https://proceedings.mlr.press/v267/ahmed25b.html. [Cited on page 20]

[20] Pavel Adamenko, Mikhail Ivanov, Aidar Valeev, Rodion Levichev, Pavel Zadorozhny, Ivan Lopatin, Dmitrii Babaev, Alena Fenogenova, and Valentin Malykh, "SWE-MERA: A Dynamic Benchmark for Agentically 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 pages 20, 25]

[21] Sanket Mhatre, Yasharth Bajpai, Sumit Gulwani, Emerson Murphy-Hill, and Gustavo Soares, "SWE-Sharp-Bench: A Reproducible Benchmark for C# Software Engineering Tasks", *in*: *2025 2nd IEEE/ACM International Conference on AI-powered Software (AIware)*, IEEE, 2025, pp. 277–280. [Cited on page 20]

[22] Valerie Chen, Ameet Talwalkar, Robert Brennan, and Graham Neubig, "Code with Me or for Me? How Increasing AI Automation Transforms Developer Workflows", *in*: *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*, ACM, 2026, pp. 1–19, DOI: 10.1145/3772318.3790850. [Cited on page 21]

[23] Mahd Hindi, Yasir Mahmood, Linda Mohammed, Salah Bouktif, and Mohammed Mediani, "Coding Agents in the Wild: Failure Modes and Rejection Patterns of AI-Generated Pull Requests", *in*: *IEEE Access* 14 (2026), pp. 83075–83094, DOI: 10.1109/ACCESS.2026.3696573. [Cited on page 21]

[24] Aditya Bharat Soni, Boxuan Li, Xingyao Wang, Valerie Chen, and Graham Neubig, "Coding Agents with Multimodal Browsing are Generalist Problem Solvers", *in*: *Findings of the Association for Computational Linguistics: EACL 2026*, Association for Computational Linguistics, 2026, pp. 6052–6069, DOI: 10.18653/v1/2026.findings-eacl.318, URL: https://aclanthology.org/2026.findings-eacl.318/. [Cited on page 21]

[25] Yunkun Wang, Yue Zhang, Guochang Li, Chen Zhi, Binhua Li, Fei Huang, Yongbin Li, and Shuiguang Deng, "InspectCoder: Dynamic Analysis-Driven Self Repair through Interactive LLM-Debugger Collaboration", *in*: *Proceedings of the ACM on Programming Languages* 10.OOPSLA1 (2026), pp. 1041–1069, DOI: 10.1145/3798238. [Cited on page 21]

[26] Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li, "Alibaba LingmaAgent: Improving Automated Issue Resolution via Comprehensive Repository Exploration", *in*: *Companion Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, ACM, 2025, pp. 238–249, DOI: 10.1145/3696630.3728549. [Cited on page 22]

[27] Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou, "CodexGraph: Bridging Large Language Models and Code Repositories via Code Graph Databases", *in*: *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, Association for Computational Linguistics, 2025, pp. 142–160, DOI: 10.18653/v1/2025.naacl-long.7, URL: https://aclanthology.org/2025.naacl-long.7/. [Cited on page 22]

[28] Harsh Vishwakarma, Ankush Agarwal, Ojas Patil, Chaitanya Devaguptapu, and Mahesh Chandran, "Can LLMs Help You at Work? A Sandbox for Evaluating LLM Agents in Enterprise Environments", *in*: *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, 2025, pp. 9167–9201, DOI: 10.18653/v1/2025.emnlp-main.466, URL: https://aclanthology.org/2025.emnlp-main.466/. [Cited on page 22]

[29] Vaibhav Aggarwal, Ojasv Kamal, Abhinav Japesh, Zhijing Jin, and Bernhard Schölkopf, "DARS: Dynamic Action Re-Sampling to Enhance Coding Agent Performance by Adaptive Tree Traversal", *in*: *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Association for Computational Linguistics, 2025, pp. 19808–19855, DOI: 10.18653/v1/2025.acl-long.973, URL: https://aclanthology.org/2025.acl-long.973/. [Cited on page 22]

[30] Karina Zainullina, Alexander Golubev, Maria Trofimova, Sergei Polezhaev, Ibragim Badertdinov, Daria Litvintseva, Simon Karasik, Filipp Fisin, Sergei Skvortsov, Maksim Nekrashevich, Anton Shevtsov, and Boris Yangel, "Guided Search Strategies in Non-Serializable Environments with Applications to Software Engineering Agents", *in*: *Proceedings of the 42nd International Conference on Machine Learning*, vol. 267, Proceedings of Machine Learning Research, PMLR, 2025, pp. 73957–73973, URL: https://proceedings.mlr.press/v267/zainullina25a.html. [Cited on page 23]

[31] Georg Wölflein, Dyke Ferber, Daniel Truhn, Ognjen Arandjelovic, and Jakob Nikolas Kather, "LLM Agents Making Agent Tools", *in*: *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Association for Computational Linguistics, 2025, pp. 26092–26130, DOI: 10.18653/v1/2025.acl-long.1266, URL: https://aclanthology.org/2025.acl-long.1266/. [Cited on page 23]

[32] Vali Tawosi, Salwa Alamir, Xiaomo Liu, and Manuela Veloso, "Llm agents for automated dependency upgrades", *in*: *2025 40th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, IEEE, 2025, pp. 34–37. [Cited on page 23]

[33] Atefeh Sohrabizadeh, Jialin Song, Mingjie Liu, Rajarshi Roy, Chankyu Lee, Jonathan Raiman, and Bryan Catanzaro, "Nemotron-CORTEXA: Enhancing LLM Agents for Software Engineering Tasks via Improved Localization and Solution Diversity", *in*: *Proceedings of the 42nd International Conference on Machine Learning*, vol. 267, Proceedings of Machine Learning Research, PMLR, 2025, pp. 56085–56100, URL: https://proceedings.mlr.press/v267/sohrabizadeh25a.html. [Cited on page 24]

[34] Hongwei Li, Yuheng Tang, Shiqi Wang, and Wenbo Guo, "PatchPilot: A Cost-Efficient Software Engineering Agent with Early Attempts on Formal Verification", *in*: *Proceedings of the 42nd International Conference on Machine Learning*, vol. 267, Proceedings of Machine Learning Research, PMLR, 2025, pp. 35922–35941, URL: https://proceedings.mlr.press/v267/li25cf.html. [Cited on page 24]

[35] Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang, "RepoAudit: An Autonomous LLM-Agent for Repository-Level Code Auditing", *in*: *Proceedings of the 42nd International Conference on Machine Learning*, vol. 267, Proceedings of Machine Learning Research, PMLR, 2025, pp. 21083–21100, URL: https://proceedings.mlr.press/v267/guo25n.html. [Cited on page 24]

[36] Haoran Wang, Zhenyu Hou, Yao Wei, Jie Tang, and Yuxiao Dong, "SWE-Dev: Building Software Engineering Agents with Training and Inference Scaling", *in*: *Findings of the Association for Computational Linguistics: ACL 2025*, Association for Computational Linguistics, 2025, pp. 3742–3761, DOI: 10.18653/v1/2025.findings-acl.193, URL: https://aclanthology.org/2025.findings-acl.193/. [Cited on page 24]

[37] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press, "SWE-Agent: Agent-Computer Interfaces Enable Automated Software Engineering", *in*: *Advances in Neural Information Processing Systems* 37 (2024), pp. 50528–50652. [Cited on page 26]

[38] 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*: *International Conference on Learning Representations*, vol. 2025, 2025, pp. 65882–65919. [Cited on page 26]



[39] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang, "Demystifying LLM-Based Software Engineering Agents", *in*: *Proc. ACM Softw. Eng.* 2.FSE (June 2025), DOI: 10.1145/3715754, URL: https://doi-org.univaq.idm.oclc.org/10.1145/3715754. [Cited on page 26]

[40] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury, "AutoCodeRover: Autonomous Program Improvement", *in*: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, DOI: 10.1145/3650212.3680384. [Cited on page 26]

[41] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni, "Green AI", *in*: *Communications of the ACM* 63.12 (2020), pp. 54–63, DOI: 10.1145/3381831. [Cited on page 29]

[42] 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, DOI: 10.1002/widm.1507. [Cited on page 29]

[43] Chris Richardson, *Microservices patterns: with examples in Java*, Simon and Schuster, 2018. [Cited on page 35]