UDC 004.8
DOI: 10.15587/2706-5448.2026.360928

Maksym Moskalenko

DEVELOPMENT AND VERIFICATION OF AN ORCHESTRATION ARCHITECTURE OF AI AGENTS FOR AUTOMATED API TESTING WITH A UNIFIED REPRESENTATION OF REQUIREMENTS

The object of this research is the process of automated testing of application programming interfaces (API) in systems developed following Agile and DevOps approaches, where requirements are heterogeneous, frequently changing, and represented in various formats: from formal OpenAPI specifications to textual documentation (Confluence). The problem addressed is the absence of an architectural mechanism for systematic integration of heterogeneous requirements sources and for decoupling requirements interpretation from test generation. Specification-oriented approaches fail to incorporate business rules from textual sources, covering only 70–80% of specification content. LLM-based approaches are unstable: repeated runs with identical prompts yield test sets differing by 20–40%, with coverage standard deviation reaching ±12%.

AI-driven orchestration architecture is proposed, comprising a coordination layer O, requirements-processing agents A, and a protocol-independent unified requirements representation R. The test generation process is formalized as $T = G(O(A(S)))$, where S denotes the set of requirements sources and G the deterministic test generation algorithm. The key property of the architecture is isolation of the stochastic LLM component at the agent level, guaranteeing reproducibility of the test set T for any fixed R.

Verification was conducted through a comparative experiment on a REST API service with 5 endpoints and 12 business rules. API coverage: 88–92% vs. 72–78% (specification-based) and 55–82% (LLM-based). Standard deviation: ±2% vs. ±3% and ±12%. Reproducibility: 0.97 vs. 0.95 and 0.62. Maintenance: 15–20% modified tests vs. 60–70% and 40–55%.

The proposed architecture targets software development teams practising API-First Development and Documentation-Driven Development. Results are applicable to Agile/DevOps environments with frequently changing requirements.

Keywords: *API, testing, orchestration, API requirements, LLM, DevOps, CI/CD, architecture, automation, determinism.*

Received: 06.03.2026
Received in revised form: 01.05.2026
Accepted: 12.05.2026
Published: 29.05.2026

How to cite
Moskalenko, M. (2026). Development and verification of an orchestration architecture of AI agents for automated API testing with a unified representation of requirements. Technology Audit and Production Reserves, 3 (2 (89)), 31–40. https://doi.org/10.15587/2706-5448.2026.360928

1. Introduction

Modern software systems are increasingly being developed using Agile and DevOps approaches, which involve short development cycles, continuous integration (CI), and continuous improvement of functionality. Software engineering exploratory methods [1] and code review practices [2] confirm the increasing complexity of quality assurance in such environments. This approach allows to reduce the time to market and respond quickly to frequent changes. At the same time, this approach significantly complicates software quality assurance, because traditional manual testing does not scale with the pace of system development, and test suites need to be constantly updated with each release. Given the complexity of such development approaches, automated testing has become an integral part of modern software engineering [3, 4]. Among the different levels of testing, API (Application Programming Interface) testing occupies a special place. Unlike user interface tests, API tests are more stable, less sensitive to changes in the presentation layer, and integrate well into CI/CD pipelines (Continuous Integration/Continuous Delivery) [5]. In modern thin-client architectures, where the main business logic is implemented on the server side, the API acts as the main contract between system components [6] – this turns API testing into a key mechanism for checking functional correctness. The theoretical basis for this work is formed by two general sources: the work [3] with the systematization of coverage criteria and test design techniques, and the ISTQB syllabus [7], the terminology of which is used in the formalization of the generation algorithm. The architectural principles of REST (Representational State Transfer) are described in [5], and in [6] it is shown that in microservice systems it is the API that becomes the main contract between components. The general state of research in the field of AI (artificial intelligence) applications in testing is analyzed in [4], and the issue of maturity of testing processes is analyzed in [8]. A detailed analysis of each direction is given below.

Specification-driven testing is the most common approach to API test automation [3, 9]. Test scenarios are generated directly from formal interface descriptions – OpenAPI or Swagger – by analyzing endpoints, request and response schemes, parameter constraints and expected status codes. The advantages of this approach are determinism and suitability for regression testing. Recent publications [3, 9]

ISSN-L 2664-9969; E-ISSN 2706-5448

demonstrate that specification-driven approaches provide coverage of 70–80% of the requirements present in the specification. For example, the RESTTESTGEN tool [10] confirms the effectiveness of black-box approaches based on OpenAPI for testing nominal and error scenarios, but remains limited by the content of the formal specification. If the OpenAPI specification defines the age parameter as an integer without additional restrictions, and the text documentation contains the rule "age >18 years", the specification generator will not generate a negative scenario for "age <18 years". In addition, supporting the new API description format requires significant rework of the generator [3, 11]. Model-Based Testing (MBT) involves building an abstract model of system behavior from which test scenarios are automatically generated [11, 12]. In the context of API testing, such models are represented in the form of state transition graphs or decision tables. Formally, the MBT process is described as a mapping

$$M: Sys \rightarrow Tests, \quad (1)$$

where *Sys* – a formal model of the system (state graph, decision table, etc.), and *Tests* – a generated set of test cases.

In [11, 12] it is shown that MBT provides a systematic exploration of the state space and can theoretically achieve full coverage for systems with a finite number of states. However, practical applicability is limited by the complexity of maintaining models. In an Agile environment, where requirements change every week, adding a new system state requires manual updating of the entire model, which is controversial in CI/CD environments. Rule-based approaches partially automate this process, but remain tied to specific templates [11]. At the same time, approaches based on random testing [13, 14], although not requiring a model, have limited theoretical efficiency compared to deterministic techniques.

According to some publications [15], in 67% of specification-based approaches, manual model updates take from 4 to 12 hours per sprint – that is, from 10 to 30% of the QA (Quality Assurance) team's time is spent not on testing, but on supporting the test infrastructure. It is this practical indicator that serves as a quantitative justification for the need for automatic updating of the requirements representation.

ML (Machine Learning) methods are widely used to optimize test processes: test prioritization, defect prediction, and anomaly detection in API responses [4, 9]. The general model of the ML approach is described as a function $f: D\_train \rightarrow Predictor$, where *D_train* is historical test execution data and defects, and *Predictor* is a trained classifier that prioritizes tests or predicts the probability of a defect.

The practical application of ML in testing demonstrates an increase in the efficiency of defect detection, which is confirmed both by comparative studies of REST API tools [9] and by generalizing reviews of the use of AI (Artificial Intelligence) in testing. In particular, clustering methods allow to automatically detect anomalous patterns of API responses that indicate regression defects [4]. However, ML approaches require a significant amount of high-quality training data and often require complete retraining when changing the system architecture. It is significant that ML copes well with the optimization of existing tests, but cannot create them from scratch. This fundamentally distinguishes ML from the problem solved in this article. The emergence of LLM (Large Language Model) has opened up a qualitatively different level of possibilities for generating tests directly from unstructured documentation [16]. Unlike previous approaches, LLMs are able to semantically analyze text descriptions and generate test cases without formal specifications.

Study [16] using the ChatUniTest framework based on GPT-4 demonstrates that LLMs are able to generate functionally correct tests for 65–75% of the described scenarios. And some recent studies [17] show that with structured prompt engineering, the quality of generated tests approaches that of manually written ones. However, when reruns with identical contexts, test sets differ by 20–40% due to the probabilistic nature of generation. The standard deviation of coverage reaches ±12%, which is critical for CI/CD environments where reproducibility is a mandatory requirement.

LLMs also lack a mechanism for formal consistency between different sources of requirements, for example, if a business rule is described in Confluence, and a parameter constraint is in OpenAPI, the model may take into account only part of the information or generate conflicting tests.

Of particular note is the class of approaches based on RAG (Retrieval-Augmented Generation), which gained significant popularity in 2023–2025. In the RAG architecture, LLM is supplemented by an external knowledge base, where relevant document fragments are extracted by semantic similarity and passed into the model context along with the query. RAG significantly improves the quality of LLM generation compared to the conventional context approach, since relevant fragments increase the accuracy and reduce the "hallucinations" of models. However, RAG retains a fundamental limitation of LLM approaches, test generation remains a probabilistic function of LLM. If repeated calls are made with an identical context, the test sets differ due to the stochastic nature of autoregressive generation. Even with parameters that regulate the operation (*temperature* = 0), different versions of models or changes in the prompt give different results.

The fundamental architectural difference of this work from RAG is as follows. In RAG, extracted document fragments are transferred directly to the context of the LLM, which generates tests, then the LLM performs two functions at the same time, it interprets requirements and generates tests, which makes determinism impossible.

To summarize, RAG approaches do not solve the problem of determinism of test generation, since they retain the direct dependence of tests on the probabilistic result of the LLM. In addition, RAG does not have a mechanism for formal reconciliation of contradictions between different sources of requirements – this gap is considered in the following sections.

All this allows to argue that it is advisable to conduct research devoted to an architectural mechanism that would systematically integrate heterogeneous sources of requirements and separate their interpretation from deterministic test generation.

The object of research is the process of automated API testing in software systems developed using the Agile and DevOps approaches.

The subject of research is architectural methods and algorithms for integrating heterogeneous sources of requirements for deterministic generation of test scripts in CI/CD environments.

The aim of research is to develop architecture of AI-driven orchestration of automated API testing, which provides deterministic integration of heterogeneous sources of requirements and reproducible generation of test scripts in CI/CD environments.

To achieve the aim, the following objectives are set:

1. To develop a formal model of AI-driven orchestration architecture and verify whether it increases API coverage.
2. To develop and implement a deterministic algorithm for generating test scripts based on a unified representation using test design techniques.
3. To conduct a comparative experiment to quantitatively test the research hypotheses regarding coverage, stability, and maintenance effort.

2. Materials and Methods

2.1. Research questions and research methods

The following research questions are tested within the framework of research:

– *RQ*1 – does a unified representation of requirements increase the completeness of API coverage compared to approaches that use a single source;

– *RQ*2 – does an intermediate representation reduce the variability of test generation results compared to LLM approaches;
– *RQ*3 – does the proposed approach reduce the complexity of test support during the evolution of requirements.

Four scientific methods were used to solve the tasks set. Each of them is applied at a specific stage of the research and has its own justification.

Analysis and comparison were used at the literature review stage. Existing approaches to automated API testing are quite diverse: specification, model, ML, LLM, RAG. Without their systematization, it is difficult to show where exactly the scientific gap is located and why it requires an architectural, rather than an algorithmic, solution.

Formalization and mathematical modeling were needed when describing the architecture. The proposed composition $T = G(O(A(S)))$ and the proof of Property 1 require a strictly formal notation. A verbal description does not allow to verify that the same input always generates the same output. Other methods of proving determinism (empirical, statistical) are not suitable for this task, because they show only a high probability of coincidence, but not its guarantee.

System design was applied in the development of four-level architecture. Decomposition into agents, an orchestration layer, a representation R and an algorithm G is a design solution, and it requires the definition of interfaces between levels, responsibilities of each component and rules for their interaction. Without this, it is impossible to ensure the extensibility of the system to new types of sources of requirements.

The experimental method was chosen to test hypotheses. The metrics $C(T)$, σ, *R_rep* and *M_maint* contain a stochastic component – the behavior of the LLM agent during repeated launches. It is impossible to predict σ or *R_rep* analytically, so 20 independent runs and statistical processing of the results are required.

2.2. Tested system and experiment setup

To verify the proposed approach, a public REST API service was used that implements CRUD (Create, Read, Update, Delete) over domain-level entities (Table 1). The service was developed in Python/FastAPI and deployed in a Docker container. The choice of the system is due to its typicality for modern backend architectures.

Table 1

API endpoints used in the experiment

Method	Endpoint	Description	Parameters
GET	/api/resources	List of objects	page, limit, filter
GET	/api/resources/{id}	Object by ID	id (UUID)
POST	/api/resources	Create object	name, value, type
PUT	/api/resources/{id}	Update object	id, name, value
DELETE	/api/resources/{id}	Delete object	id (UUID)

Hardware and software environment of the experiment. Experimental runs were performed on a MacBook Air laptop with an Apple M3 processor (8-core CPU (Central Processing Unit), 10-core GPU (Graphics Processing Unit), 16-core Neural Engine) and 16 GB of RAM. Operating system – macOS Tahoe 26. Software: Python 3.11, FastAPI 0.104, Docker 24, pytest 7.4, OpenAI Python SDK (Software Development Kit) 1.3, openapi-spec-validator 0.7, pydantic 2.5, scikit-learn 1.3 (for calculating TF-IDF and cosine similarity). External services: OpenAI API (model gpt-4-0125-preview, parameters: *temperature* = 0.1, *seed* = 42, *max_tokens* = 2048).

This experiment uses a subset of two requirements sources

$$S = \{S\_spec, S\_doc\}, \quad (2)$$

where *S_spec* – the OpenAPI 3.0 specification (5 endpoints, 18 parameters, 24 response schemes); *S_doc* – the text documentation (12 business rules, including value constraints, validation rules, and inter-parameter dependencies). Three business rules are present only in the text documentation and are absent from the specification – this provides differentiation of approaches. In the current research, work with two main requirements sources was verified; the issue tracker analysis agent (*S_issues*) is part of the proposed architecture and will be verified in further researches on larger industrial systems.

Three approaches were compared. Specification: $T = G\_spec(S\_spec)$ – test scripts are generated exclusively from OpenAPI. LLM approach: $T = G\_LLM(S\_doc)$ – tests are generated by LLM (GPT-4) from the text context *S_doc* with an identical prompt in each of the 20 runs. Proposed: $T = G(O(A(S)))$ – tests are generated from the unified representation $R = O(A(S))$.

2.3. Evaluation metrics

Four metrics are defined to answer *RQ*1–*RQ*3. The first metric, $C(T)$, is the API coverage (%), which is the ratio of the number of functional requirements covered by at least one scenario to the total number of requirements. Requirements are defined as the intersection of the contents of *S_spec* and *S_doc* (42 unique functional requirements). σ – the standard deviation of $C(T)$ and is calculated over 20 independent runs; it characterizes the stability of the approach. *R_rep* – the reproducibility, which is the proportion of test scenarios that are reproduced in all 20 runs (from 0 to 1). *M_maint* – the maintenance effort (%): the proportion of tests that required manual modification after three control changes to the requirements, such as:

a) adding a new parameter constraint;
b) changing a business rule;
c) adding a new endpoint.

3. Results and Discussion

3.1. Formal model of AI-driven orchestration architecture

A systematic analysis of five approaches shows that each of them solves only part of the problem of automated API testing. Specification-oriented approaches provide determinism, but are limited by the content of the formal specification. Model-oriented approaches are systematic, but not suitable for frequent changes. ML methods optimize existing tests, but do not generate new ones. LLMs are flexible, but unstable. The results of the comparisons of the approaches are presented in Table 2.

The identified scientific gap is the lack of a formalized orchestration architecture that would systematically integrate heterogeneous sources of requirements, formalize extracted fragments into a protocol-independent representation, separate the interpretation of requirements from test generation, and guarantee reproducibility in CI/CD. A comparative analysis of the approaches is given in Table 2.

The formal statement can be presented in the following way. Let the set of API requirements sources

$$S = \{S_1, S_2, ..., S_n\}, \quad (3)$$

where each S_i can be an OpenAPI specification, a page in Confluence, a ticket in Jira, or a user story. It is necessary to construct a set of test scenarios

$$T = \{t_1, t_2, ..., t_k\}. \quad (4)$$

The simplest approach is a direct mapping Φ: $S \to T$, when the test generator works directly with the source sources. In practice, this doesn't work well: the sources have different formats, some of the business rules are implicit in the text, and the generator either ignores them or interprets them differently each time it is run.

ISSN-L 2664-9969; E-ISSN 2706-5448

Table 2

Comparative analysis of existing approaches to automated API testing

Approach	Requirements Source	Source Integration	Generation stability	Support Intensity	CI/CD suitability
Specification	OpenAPI	No	High (±3%)	High (60–70%)	High
Model-based	Formal model	No	High	Very High	Low
ML optimization	Historical data	No	Medium	Medium	Medium
LLM approach	Text documentation	Partial	Low (±12%)	Medium (40–55%)	Low
RAG approach	Multiple sources	Partial	Low (±10–15%)	Medium (35–50%)	Low

To solve this problem, instead of a direct mapping, the process is broken down into three sequential steps

$$T = G(O(A(S))), \quad (5)$$

where A – agents, each of which processes its own source of requirements; O – an orchestration layer that collects the results of the agents together, eliminating duplication and contradictions; $R = O(A(S))$ – an intermediate unified representation of the requirements, which no longer depends on the format of the input sources; G – a deterministic algorithm that generates tests exclusively from R.

The meaning of such decomposition is in the separation of responsibility. All the "fuzziness" associated with the interpretation of the text and the work of the LLM is concentrated at the level of agents A. Starting from the level R, the process becomes fully deterministic: the same R always generates the same set of tests T.

Stabilization property (Property 1). If G is deterministic with respect to R, then for any two sets of sources S and S' containing semantically equivalent information, the result of test generation will be the same.

Proof. Let S and S' describe the same API, but in different formats – for example, S contains the OpenAPI specification, and S' – the same information in the form of text documentation. Since agents A and orchestration layer O normalize both sets to a single structure, there is

$$O(A(S)) = O(A(S')) = R. \quad (6)$$

And since G – a deterministic mapping $R \to T$, let's get $T = G(R)$ in both cases. In practice, this means that when changing the documentation format (for example, transferring requirements from Wiki to Confluence), the set of generated tests does not change – provided that the content of the requirements remains the same.

The proposed architecture consists of four functional levels, each of which corresponds to a separate function in the test generation process. The general scheme of the architecture is shown in Fig. 1.

Level 1 – Requirements processing agents. Each agent specializes in processing a specific type of requirements and is an independent component of the system. For example, the specification analysis agent processes OpenAPI/Swagger data, extracts endpoints, HTTP (HyperText Transfer Protocol) methods, parameters, data types, status codes. The documentation analysis agent applies NLP (Natural Language Processing) methods to detect business rules and constraints in text sources. The application of NLP techniques to improve API testing is investigated in some works [18] and confirms the promising nature of this approach. This agent is implemented using the GPT-4 API with a structured JSON (JavaScript Object Notation) prompt (*temperature* = 0.1, *seed* = 42 for reproducibility) that returns an array of {entity, constraint_type, value, source_sentence} objects. The following issue tracker analysis agent extracts edge-cases and non-functional constraints from Jira or GitHub Issues. This modularity of agents ensures extensibility, so that adding a new type of requirement source only requires implementing a new agent without changing the rest of the architecture.

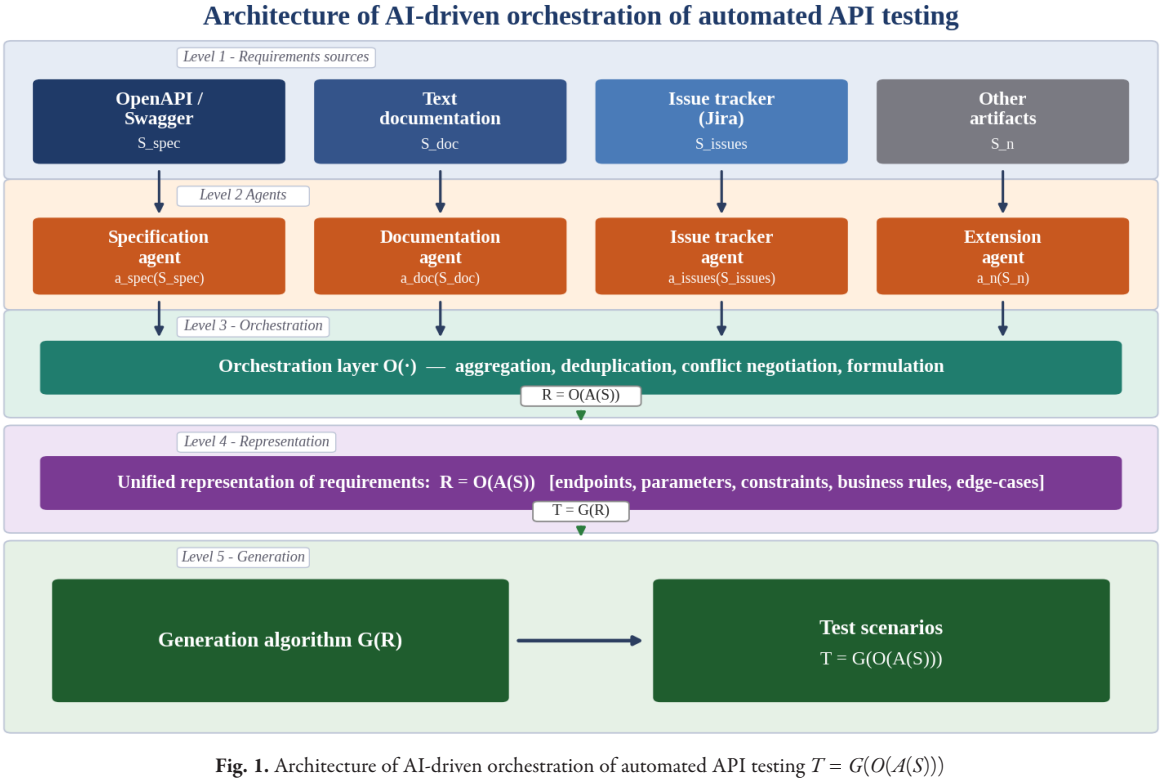

Fig. 1. Architecture of AI-driven orchestration of automated API testing $T = G(O(A(S)))$

Layer 2 – Orchestration layer. Coordinates the work of agents and performs aggregation of requirement fragments. Eliminates duplication of the same endpoint from different sources. Reconciles conflicts between sources using a "strict constraint takes precedence" strategy. Forms a single data structure.

Running agents in parallel reduces the total execution time from 90 to about 47 seconds, but makes it more difficult to handle conflicts, for example, agents can simultaneously return conflicting constraints for the same parameter. In the implemented prototype, this problem is solved by priorities in the orchestration layer, where the specification agent always finalizes its output before the merge begins, which provides a stable basis for agreement.

Level 3 – Unified requirements representation. This is a structured protocol-independent representation containing a list of endpoints; query parameters (type, mandatory, constraints); expected responses for each scenario; business rules from text sources; edge-cases from issue trackers. Formally, $R = O(A(S))$.

Level 4 – Test generation algorithm. It applies classical test design techniques (equivalent decomposition, boundary value analysis, and negative scenario generation) to the structured requirements representation received from the orchestration layer.

3.2. Deterministic algorithm for generating test scenarios based on a unified representation of requirements

The representation R is essentially a structured JSON document that contains everything that agents have extracted from different sources. Both the completeness of test coverage and the possibility of deterministic generation depend on its organization. In the current implementation, R consists of five components.

The first component is a list of endpoints

$$E = \{e_1, e_2, ..., e_n\}. \quad (7)$$

In this statement, each endpoint is described by a triple (method, path, description) and for each endpoint, input parameters and expected responses are specified. The second component is the parameter description, which is denoted by P. It will record the type of values (string, integer, boolean), the mandatory nature of values, the format (email, date-time, regex), and numeric or string constraints (min/max, minLength/maxLength). The third component includes RS response schemes: what structure and what fields the server returns for each status code. The fourth includes business rules

$$BR = \{br_1, br_2, ..., br_k\}. \quad (8)$$

The choice of such a format instead of, for example, OpenAPI is due to the fact that OpenAPI is focused on the description of the interface and does not provide for business rules of the type "the client's age must be over 18, if the type of operation is a loan". Because OpenAPI is focused on the description of the interface, and it does not provide for business rules of the type "the client's age must be over 18, if the type of operation is a loan". In R, such rules are a full-fledged component. Unlike unstructured text, which is usually fed into the LLM context, R has a fixed schema and is therefore subject to deterministic processing by the G algorithm.

It is also worth noting four properties that were built into the R structure, namely protocol independence (the same format for REST, GraphQL (Graph Query Language), gRPC (gRPC Remote Procedure Calls)), extensibility (a new type of requirement is simply a new component, without changing existing ones), determinism (the same set of requirements always gives the same R), and traceability (each element of R stores a link to the source from which it was extracted).

The full cycle from input sources to finished tests is described step by step below.

At the input of the system – $S = \{S\_spec, S\_doc, S\_issues\}$ – specification, text documentation, and data from the issue tracker. The output is a set of test scenarios $T = \{t_1, t_2, ..., t_k\}$.

First, the sources are processed, all sources are processed in parallel. Each agent a_i receives its source S_i, then analyzes it (syntactically in the case of OpenAPI, semantically in the case of text documentation). After processing, a set of structured fragments of requirements $F_i = a_i(S_i)$ is formed. Parallel launch allows to speed up the process, and in this case the specification agent works in fractions of a second. At the processing stage, the documentation agent waits for a response from GPT-4 for up to 40 seconds, and there is no point in executing them sequentially.

In the second step, the orchestration layer O enters. It collects the results of all agents $F = \cup F_i$ and performs two operations. The first is deduplication (if two fragments describe the same thing – Jaccard similarity over 0.8 – one remains) and performs conflict reconciliation (if the specification says "age ≥ 0", and the documentation says "age > 18", the stricter constraint wins). The result will be a formed representation of the requirements $R = O(F)$.

Step 3. Formation of test conditions. At this step, test conditions are formed: for each parameter $p \in R$, the class of equivalent values $EC(p)$, the boundary values

$$BV(p) = \{\min - 1, \min, \min + 1, \max - 1, \max, \max + 1\}, \quad (9)$$

and the unacceptable values $IV(p)$ [3, 17] are determined.

Step 4. Generation of test scenarios. For each test condition TC, one or more scenarios are formed

$$T = G(R) = \cup\{t_i \mid t_i \leftarrow TC\}. \quad (10)$$

Step 5. Preparation for execution. The scenarios are serialized into the format of HTTP requests or automated tests for pytest/JUnit. An artifact for the CI/CD pipeline is generated.

A prototype of the proposed architecture was implemented as part of the experiment. The specification analysis agent is implemented as a Python module using the openapi-spec-validator library for schema validation, plus pydantic for structure parsing. For each endpoint, the agent generates a structured description with method, path, required and optional parameters, data types and constraints, response schemes for all status codes. The text documentation analysis agent is implemented using the GPT-4 API (gpt-4-0125-preview) via a structured prompt with forced JSON output (response_format: {type: "json_object"}). In the prompt, the model receives the role of an "API business rule extractor" and a rigidly specified response format. To help the model better understand what is expected of it, two examples are also added to the prompt: a fragment of the documentation text and the corresponding JSON format. At the output, for each found rule, the agent returns an object with the following fields: entity, endpoint, constraint_type (one of value_range, required_if, forbidden_if, format), value, confidence, and source_sentence. The last field – source_sentence – stores the original sentence from the documentation, which allows to check where this rule came from if necessary. The confidence field plays a filtering role: the orchestration layer rejects everything below 0.7 – such rules are considered unreliable. To reduce the variance between runs, temperature is set to 0.1 (not 0, because some API versions behave unpredictably when zero), seed is fixed at 42, max_tokens is limited to 2048. The orchestration layer is written as a separate Python service. Inside, it does three things. First, the aggregation module collects the results of all agents, then groups them by pair (entity, endpoint) – that is, all fragments related to one endpoint fall into one group. Next, the deduplication module checks whether there are any repetitions among the collected fragments. The check is two-stage: first, a fast filter by Jaccard-similarity of tokens with a threshold of 0.8 filters out obviously repeating fragments, after which TF-IDF (Term Frequency – Inverse

ISSN-L 2664-9969; E-ISSN 2706-5448

Document Frequency) cosine similarity is calculated for the pairs that passed the filter. If it exceeds 0.85, the fragments are considered duplicates, one of them is removed. The two-stage method was chosen to reduce computational costs – Jaccard works on raw tokens and is cheap, while TF-IDF requires the construction of a vector representation, therefore it is applied only to candidates that passed the first filter. Finally, the conflict reconciliation module handles cases where different sources say different things about one parameter. The rule is simple: the stricter restriction wins. For example, the specification allows age ≥ 0, and the documentation says "client must be of legal age" – *R* will have the restriction age > 18.

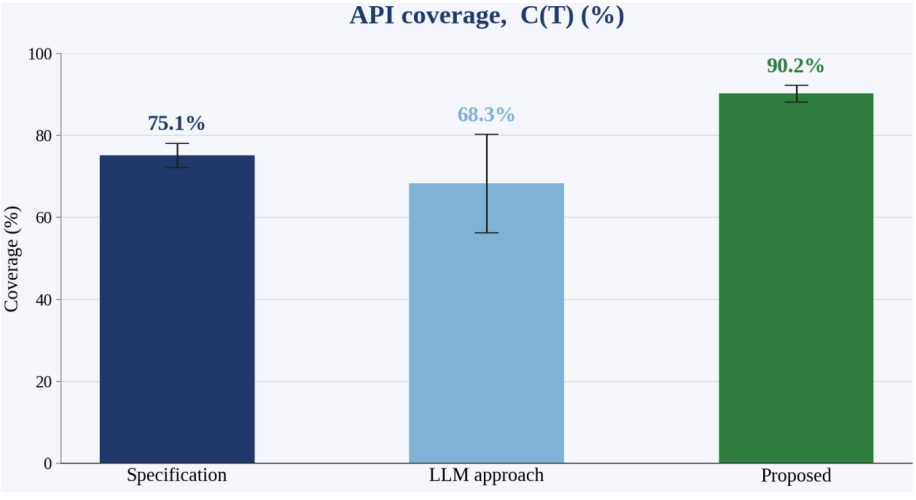

Fig. 2. Comparative analysis of experimental study results (n = 20 runs): API coverage, $C(T)$ (%)

The *G* algorithm is a regular Python script without any LLM calls. It runs through R, builds equivalence classes, boundary values, and invalid combinations for each parameter, and writes the result as pytest files with parameterized tests. The entire cycle $T = G(O(A(S)))$ for the experimental system (5 endpoints, 18 parameters) is executed in 47–63 seconds. Of these, about 40 seconds are spent waiting for a response from GPT-4 in the documentation agent. The rest – specification parsing, orchestration, and actual test generation – takes less than 10 seconds.

3.3. Comparative experiment: verification of research hypotheses

3.3.1. API coverage by test scripts (*RQ*1)

The results of the assessment of API coverage by test scripts for 20 independent runs for each approach are given in Table 3.

Table 3

API coverage by test scripts (n = 20 runs)

Approach	Minimum $C(T)$, %	Maximum $C(T)$, %	Average $C(T)$, %	σ, ±%
Specification	72	78	75.1	3
LLM approach (GPT-4)	55	82	68.3	12
Proposed	88	92	90.2	2

The specification-oriented approach provides a stable but limited coverage of 72–78%. The limitation is due to the fact that 3 out of 12 business rules are present exclusively in the text documentation and are not reflected in the OpenAPI specification. The LLM approach shows significant variability: 55–82%. Under favorable conditions (quality context), the coverage can exceed the specification approach, but the instability makes it practically unsuitable for CI/CD. The proposed approach provides 88–92% due to the integration of both sources of requirements into a single R representation.

The difference between the proposed approach and the specification approach (88–92% vs. 72–78%) is explained by the inclusion of business rules from the text documentation via the *S_doc* processing agent. The coverage does not reach 100% due to implicit limitations in the documentation that were not detected by the NLP agent in the current implementation.

*RQ*1 – an increase of 13–17 percentage points of coverage relative to the specification approach and up to 37 pp. relative to the LLM approach proves: the integration of heterogeneous sources through *R* fundamentally expands the area of test generation. The result is shown in Fig. 2.

The 15 percentage point increase in coverage compared to the specification approach is explained by a specific reason: three out of twelve business rules were contained exclusively in the text documentation *S_doc* and were not reflected in the OpenAPI specification at all. It is their extraction by the NLP agent and inclusion in the *R* representation that forms the difference between 75.1% and 90.2%. This is not a random result – it confirms the well-known practical problem that formal specifications in real projects do not cover all aspects of business logic [6]. Tools such as RESTler [19] and EvoMaster [20] demonstrate good results within the specification, but do not have a mechanism for working with non-formalized sources. Zhang & Arcuri's review [15] directly points to this as a systemic gap in most REST API testing approaches – and it is this gap that the proposed architecture closes through a unified R representation.

3.3.2. Stability and reproducibility of test generation (*RQ*2)

The results of the stability and reproducibility assessment of test generation are given in Table 4.

Table 4

Stability and reproducibility of test generation (n = 20 runs)

Approach	σ, ±%	*R_rep*	Minimum number of tests	Maximum number of tests
Specification	±3	0.95	47	52
LLM approach (GPT-4)	±12	0.62	31	58
Proposed	±2	0.97	64	69

The LLM approach is characterized by significant instability: σ = ±12%, reproducibility – only 0.62. This means that with each run approximately 38% of the tests are new. For a CI/CD pipeline, where a predictable result is expected, this is not suitable. The specification approach, on the contrary, is stable (σ = ±3%, *R_rep* = 0.95), but it generates fewer tests – because it works with only one source.

The proposed approach showed σ = ±2% and *R_rep* = 0.97. It is important to explain where these ±2% came from if the algorithm *G* is fully deterministic. After running the tests, the logs of all 20 runs were analyzed, from which it follows that the variability concerns three specific business rules, which are formulated ambiguously in the documentation – with conditional constructs like "may be required if...". The documentation agent sometimes treats them differently at different runs, and this leaks into *R*. Algorithm *G* is useless here because it works the same way every time R is the same. So, the way to σ = 0% is

to standardize prompts for such ambiguous constructs, which is a very specific engineering task.

RQ2 – σ reduction by a factor of 6 (from ±12% to ±2%) and *R_rep* increase from 0.62 to 0.97 (Fig. 3).

The obtained result is explained not by "better settings", but by the architectural solution. When tests are generated directly from LLM, the whole process is stochastic: the model chooses a different generation path each time, even at *temperature* = 0. In the proposed approach, LLM is involved only in agents – it extracts business rules and writes them in a structured format. Then the orchestrator forms *R*, and the algorithm *G* generates tests from *R* without any reference to the model. That is, *R* works as a "wall" between the stochastic part (agents) and the deterministic part (test generation) – this is the essence of Property 1 in practice.

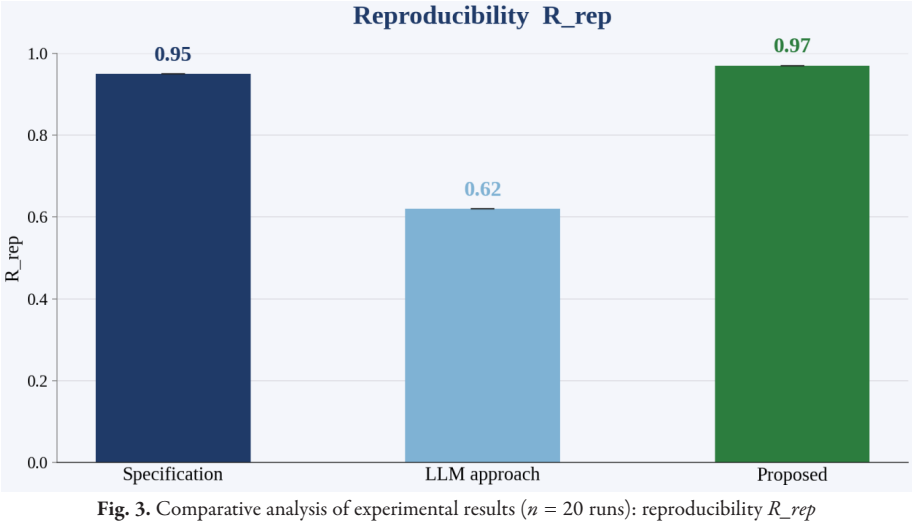

Fig. 3. Comparative analysis of experimental results (n = 20 runs): reproducibility *R_rep*

It should be noted that the limitation: *R_rep* = 0.97 means that 97% of tests syntactically coincide between runs. The experiment did not test whether different tests do not test the same thing – that is, they are not semantic duplicates. This is a separate issue that requires a semantic equivalence metric, and it is planned to develop it in further research. As for the practical side, "flaky" tests that run only once are a known problem in microservice environments [11]. Reducing σ by a factor of six means that a significant part of such instability disappears automatically, without manual intervention by the team.

3.3.3. Test maintenance effort when changing requirements (*RQ3*)

To assess the maintenance effort, three control changes to the requirements were introduced into the system:

1) adding a new parameter constraint value: min = 0;

2) changing the business rule for the POST operation;

3) adding a new endpoint PATCH/api/resources/{id}. For each approach, the proportion of test scripts that required manual modification was calculated (Table 5).

In the specification approach, a change in requirements directly affects the specification structure and requires extensive modification of tests. In the proposed approach, the change in requirements is reflected in a unified representation of *R* requirements, after which the test scripts are generated again automatically. Manual modification (17.7% on average) applies only to tests with tight dependencies on specific implementation details not covered by *R*.

RQ3 – the reduction in the proportion of tests requiring manual updates from 65.7% to 17.7% – by a factor of 3.7 – reflects the practical value of decoupling between requirements sources and test scripts through *R*. A comparative analysis is presented in Fig. 4.

The value of 15–20% of tests that required manual updates after three control changes to requirements is not only a better number compared to 60–70% and 40–55%. There is a specific mechanism behind this indicator, for example, in traditional approaches, a test is rigidly tied to its source and any change in the specification cascades out of date the tests. In the proposed approach, the only place of change is the *R* representation, after which the full set of tests is automatically regenerated. Manual intervention remains only where the tests had tight dependencies on implementation details that go beyond the scope of *R*.

The above results were obtained under the following assumptions: *S_spec* and *S_doc* are filled in by the team conscientiously – contradictions between them occur, but they are unintentional. It is possible to rely on GPT-4 with *temperature* = 0.1 and *seed* = 42 as a source with acceptable spread – not deterministic, but ±2% variability for our task is tolerable. The Jaccard 0.8 and TF-IDF 0.85 thresholds were empirically selected for this specific documentation corpus. Perhaps on another project they will probably need to be recalibrated.

Table 5

Labor intensity of test support when requirements change

Approach	Change 1: new parameter, %	Change 2: new rule, %	Change 3: new endpoint, %	Average, %
Specification	62	70	65	65.7
LLM approach	45	55	40	46.7
Proposed	18	20	15	17.7

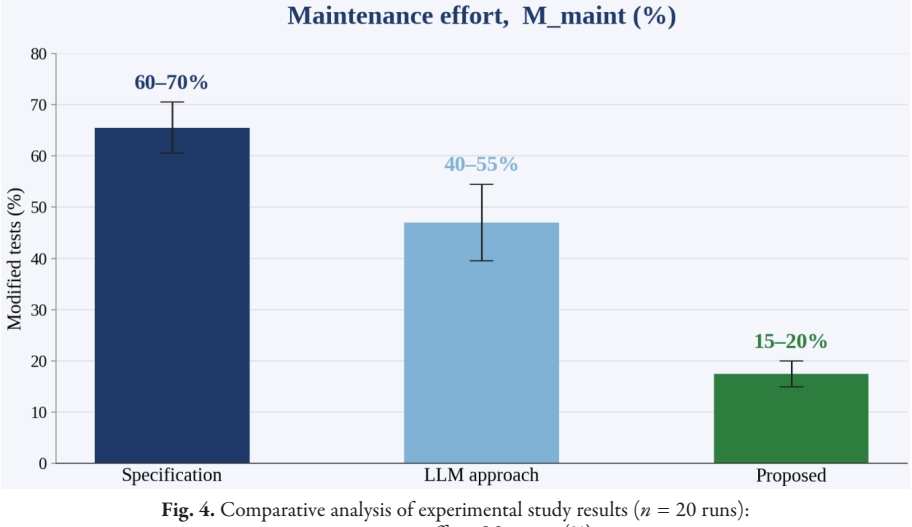

Fig. 4. Comparative analysis of experimental study results (n = 20 runs): maintenance effort, *M_maint* (%)

ISSN-L 2664-9969; E-ISSN 2706-5448

A separate issue is the style of the documentation itself. The research used texts where business rules are formulated in the form of "if-then", which is typical for technical requirements. If the team maintains the documentation descriptively, without explicit conditional constructs, the NLP agent will produce a worse result – this is not a defect of the architecture, but due to the limits of its applicability. Similarly, the whole idea makes sense only on condition that the documentation is actually updated together with the code – in teams where it lags behind the implementation for years, generating tests from it makes no sense at all.

As for the limitations of the experiment itself, the main one is scale. Five endpoints and two sources of requirements are a working minimum for testing the hypothesis, but far from industrial. The behavior of the architecture on systems with 50+ endpoints was not tested in this research; the protocol-independence of *R* gives reason to hope that the effect will persist, but until there is an experiment, this is just a hypothesis. Similarly to GraphQL and gRPC, the issue tracker analysis agent *S_issues* is built into the design, but is not involved in the experiment – it requires a real project with an accumulated ticket history, not a synthetic bench. The last limitation is the use of a single LLM model – gpt-4-0125-preview. Other models have different stochastic behavior, and the confidence threshold of 0.7, which it is possible to use to filter out rules, would have to be selected separately for each of them.

The proposed architecture can be integrated into the CI/CD pipeline as a separate pipeline stage between the requirements gathering and test execution stages.

Each commit to the repository will trigger a pipeline that in turn triggers agents that automatically analyze the current requirements sources: *S_spec* is updated from the Git repository, *S_doc* is updated via the Confluence API or a similar interface. The orchestration layer generates the current *R*, and the algorithm *G* generates an updated test suite *T*. The entire cycle is performed within 2–5 minutes for a medium-sized API (5–20 endpoints) and is presented in Fig. 5.

The approach is particularly useful for teams practicing DDD (Documentation-Driven Development) or API-First Development, where documentation and specifications are updated synchronously with code. A typical workflow looks like this. At the sprint planning stage, an analyst updates the text documentation in Confluence by adding a new business rule. A developer updates the OpenAPI specification by adding a new parameter. When the code is uploaded to the repository, the CI/CD pipeline automatically triggers an update of the *R* representation. Unlike the traditional approach, where a tester manually examines changes and updates tests, in the proposed approach, human intervention is required only to verify the correctness of the saved tests.

Organizationally, the proposed architecture is close to the architectures of multi-agent systems (MAS, Multi-Agent System), where several specialized AI components jointly solve the testing problem. Similar solutions are considered in reviews of AI applications in software testing. The key difference is in the interaction model: in classic MAS, agents communicate directly and may have conflicting goals, while in the proposed architecture; agents process their sources of requirements independently, without direct communication with each other. All coordination is performed by the orchestration layer, which makes the system behavior more predictable. The similarity with SOA (Service-Oriented Architecture) and microservices [6] is manifested in the decomposition of the system into independent components with clearly defined interfaces. However, unlike microservices, agents in the proposed architecture are not autonomous services – they are managed by the orchestration layer and do not have their own state between runs, which allows the system to be easily deployed as a stateless component of the CI/CD pipeline.

3.4. Limitations and directions for further research development

The results obtained have applicability limits that are important to consider – both in the practical implementation of the architecture and in its further theoretical development.

Limitations of the methodology. The architecture assumes that the team fills in *S_spec* and *S_doc* conscientiously: discrepancies between them occur, but are unintentional. GPT-4 with *temperature* = 0.1 and *seed* = 42 was used as a source with an acceptable spread – not fully deterministic, but with a variability of ±2%, which is tolerable for the task at hand. The Jaccard 0.8 and TF-IDF 0.85 thresholds were selected empirically for a specific corpus of documentation; on another project, they will most likely have to be recalibrated.

Limitations on the documentation style. The experiment used texts where business rules are formulated in the form of "if-then" – this is how most technical requirements are written. If the team maintains documentation descriptively, without explicit conditional constructs, the NLP agent will produce a worse result – this is not a defect in the architecture, but a consequence of what it knows how to work with. Moreover, the idea itself makes sense only when the documentation is updated together with the code. In teams where it lags behind the implementation for years, generating tests from it makes no sense at all.

Limitations of the experiment. Five endpoints and two sources of requirements are a working minimum for testing a hypothesis, but far from the industrial level. How the architecture will behave on 50+ endpoints is not tested in the current research. The protocol-independence of *R* gives reason to hope that the effect will persist, but until there is an experiment, this is just an assumption. The same applies to GraphQL and gRPC. The issue tracker analysis agent *S_issues* is included in the architecture design, but is not used in the experiment: its verification requires a real project with an accumulated ticket history. And lastly, there is only one LLM model in the experiment, gpt-4-0125-preview. Other models have different stochastic behavior, and the confidence threshold of 0.7, which is used to filter out unreliable extracted rules, would have to be selected separately for each of them.

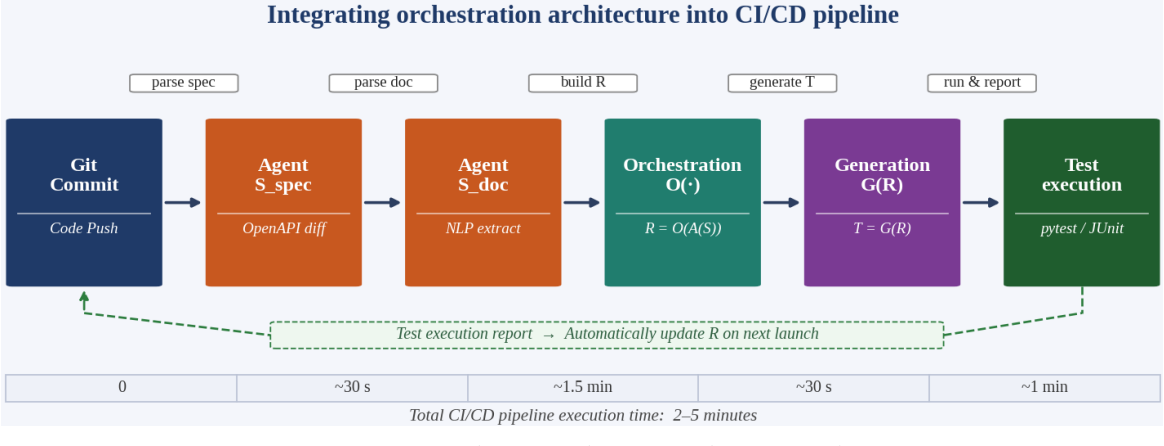

Fig. 5. Integrating orchestration architecture into the CI/CD pipeline

Priority areas for further research are:
– expansion of the architecture to support GraphQL and gRPC;
– standardization of agent prompts to eliminate residual variability of ±2%;
– assessment of scalability on systems with 50 + endpoints;
– development of a semantic reproducibility metric instead of a structural one;
– verification of the issue tracker analysis agent (*S_issues*) on industrial systems.

4. Conclusions

1. A formal model of AI-driven orchestration architecture $T = G(O(A(S)))$ is developed and the stabilization property (Property 1) is proven: a deterministic algorithm G, working with a unified representation of requirements *R*, generates identical tests regardless of the format of the input sources. The reproducibility $R\_rep = 0.97$ confirms this property in practice.

2. The generation algorithm *G* is implemented as Python script. It receives *R* as input and automatically builds equivalence classes $EC(p)$, boundary values $BV(p)$ and unacceptable values $IV(p)$ for each parameter. The result will be pytest files with parameterized tests. In the experiment, the algorithm provided coverage within 88–92%, this is at $\sigma = \pm 2\%$. For comparison, let's take the LLM approach, which gave $\sigma = \pm 12\%$ – that is, our result is six times more stable. This difference is not a coincidence, but a direct consequence of the fact that *G* works with a fixed *R* and does not refer to the language model.

3. A comparative experiment (20 independent runs) confirmed all three research hypotheses. The proposed approach provides coverage of 88–92%, which is higher than 72–78% of the specification approach and 55–82% of the LLM approach. In terms of reproducibility, the orchestration approach has 0.97 against 0.95 and 0.62. In terms of maintenance complexity, the result is 15–20% of tests that required manual updating after changes in requirements. Compared to the same 60–70% in the specification approach and 40–55% in LLM. The last indicator, the most important from a practical point of view, namely a 3.7-fold decrease in test maintenance. This means that after typical changes in requirements (new parameter, new business rule, new endpoint), the team needs to manually correct only every fifth or sixth test, not two-thirds of the set.

Conflict of interest

The authors declare that they have no conflict of interest in relation to this research, whether financial, personal, authorship or otherwise, that could affect the research and its results presented in this paper.

Financing

The research was performed without financial support.

Data availability

Data will be provided upon reasonable request.

Use of artificial intelligence

AI model and version: GPT-4 API (gpt-4-0125-preview, OpenAI).

Where exactly was AI used: Section 3.2 – as part of the text documentation analysis agent, which is a technical component of the implemented prototype of the proposed architecture.

What exactly was done using AI tools: The GPT-4 API was used to structure business rules from text documentation sources into a unified *R* requirements representation format (an array of JSON objects). Call parameters: *temperature* = 0.1, *seed* = 42, *max_tokens* = 2048.

How the results provided by AI were verified: All 12 business rules extracted by the agent from the text documentation were manually verified by comparing them with the source text. Rules with confidence < 0.7 were rejected by the orchestration layer. The correctness of the final set of rules was confirmed in 20 independent runs of the comparative experiment ($R\_rep = 0.97$).

As a result, AI influenced the conclusions of the research: the AI component is the direct object of the research, not the means of conducting it. The isolation of the stochastic LLM component at the agent level and its separation from the deterministic algorithm *G* is a key architectural contribution of the work, verified quantitatively ($R\_rep = 0.97$, $\sigma = \pm 2\%$).

Authors' contributions

Maksym Moskalenko: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Writing – original draft, Writing – review and editing, Visualization.

References

1. Harman, M., Mansouri, S. A., Zhang, Y. (2012). Search-based software engineering. *ACM Computing Surveys, 45 (1),* 1–61. https://doi.org/10.1145/2379776.2379787
2. McIntosh, S., Kamei, Y., Adams, B., Hassan, A. E. (2015). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering, 21 (5),* 2146–2189. https://doi.org/10.1007/s10664-015-9381-9
3. Ammann, P., Offutt, J. (2016). *Introduction to Software Testing.* Cambridge University Press, 768. Available at: https://www.cambridge.org/highereducation/books/introduction-to-software-testing/95E57CCADEA697EC8594F03729F47311
4. Amalfitano, D., Faralli, S., Hauck, J. C. R., Matalonga, S., Distante, D. (2023). Artificial Intelligence Applied to Software Testing: A Tertiary Study. *ACM Computing Surveys, 56 (3),* 1–38. https://doi.org/10.1145/3616372
5. Richardson, L., Amundsen, M. (2013). *RESTful Web APIs.* O'Reilly Media, 406. Available at: https://www.oreilly.com/library/view/restful-web-apis/9781449359713/
6. Cerny, T., Donahoo, M. J., Trnka, M. (2018). Contextual understanding of microservice architecture. *ACM SIGAPP Applied Computing Review, 17 (4),* 29–45. https://doi.org/10.1145/3183628.3183631
7. *ISTQB Certified Tester Foundation Level Syllabus. Version 4.0* (2023). International Software Testing Qualifications Board (ISTQB). Available at: https://istqb.org/istqb-releases-certified-tester-foundation-level-v4-0-ctfl/
8. Garousi, V., Felderer, M., Hacaloğlu, T. (2017). Software test maturity assessment and test process improvement: A multivocal literature review. *Information and Software Technology, 85,* 16–42. https://doi.org/10.1016/j.infsof.2017.01.001
9. Kim, M., Xin, Q., Sinha, S., Orso, A. (2022). Automated test generation for REST APIs: no time to rest yet. *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis,* 289–301. https://doi.org/10.1145/3533767.3534401
10. Corradini, D., Zampieri, A., Pasqua, M., Viglianisi, E., Dallago, M., Ceccato, M. (2022). Automated black-box testing of nominal and error scenarios in RESTful APIs. *Software Testing, Verification and Reliability, 32 (5).* https://doi.org/10.1002/stvr.1808
11. Utting, M., Pretschner, A., Legeard, B. (2011). A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability, 22 (5),* 297–312. https://doi.org/10.1002/stvr.456
12. Utting, M., Legeard, B. (2010). *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann, 424. Available at: https://www.sciencedirect.com/book/9780123725011/practical-model-based-testing
13. Arcuri, A., Briand, L. (2012). Formal Analysis of the Probability of Interaction Fault Detection Using Random Testing. *IEEE Transactions on Software Engineering, 38 (5),* 1088–1099. https://doi.org/10.1109/tse.2011.85
14. Arcuri, A., Briand, L. (2011). Adaptive random testing. Proceedings of the *2011 International Symposium on Software Testing and Analysis*, 265–275. https://doi.org/10.1145/2001420.2001452
15. Golmohammadi, A., Zhang, M., Arcuri, A. (2023). Testing RESTful APIs: A Survey. *ACM Transactions on Software Engineering and Methodology, 33 (1),* 1–41. https://doi.org/10.1145/3617175

ISSN-L 2664-9969; E-ISSN 2706-5448

16. Chen, Y., Hu, Z., Zhi, C., Han, J., Deng, S., Yin, J. (2024). ChatUniTest: A Framework for LLM-Based Test Generation. *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 572–576. https://doi.org/10.1145/3663529.3663801

17. Tang, Y., Liu, Z., Zhou, Z., Luo, X. (2024). ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation. *IEEE Transactions on Software Engineering, 50 (6),* 1340–1359. https://doi.org/10.1109/tse.2024.3382365

18. Kim, M., Corradini, D., Sinha, S., Orso, A., Pasqua, M., Tzoref-Brill, R., Ceccato, M. (2023). Enhancing REST API Testing with NLP Techniques. *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1232–1243. https://doi.org/10.1145/3597926.3598131

19. Atlidakis, V., Godefroid, P., Polishchuk, M. (2019). RESTler: Stateful REST API Fuzzing. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE),* 748–758. https://doi.org/10.1109/icse.2019.00083

20. Arcuri, A. (2019). RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology, 28 (1),* 1–37. https://doi.org/10.1145/3293455

Maksym Moskalenko, *PhD Student, Department of Information Systems, Educational and Scientific Institute "Ukrainian State University of Chemical Technology", Dnipro, Ukraine, e-mail: mosmax3@gmail.com, ORCID: https://orcid.org/0009-0009-0767-2233*