DOI: 10.15514/ISPRAS-2026-38(4)-32

Development of a CASE Platform with Generative Artificial Intelligence and an MCP Agent Layer

Bogdan Polygalov, ORCID: 0009-0003-1256-3003 <bogdan.polygalov@gmail.com>
Viacheslav Lanin, ORCID: 0000-0002-0650-2314 <vlanin@hse.ru>

IT in Business Department, HSE University,
38, Studencheskaya st., Perm, 614070, Russia.

Abstract. This paper presents the architecture, implementation and preliminary service-level evaluation of a prototype CASE platform that uses generative artificial intelligence to support requirements structuring, code generation, test-artifact generation and deployment checks. The platform combines an adapter-based LLM integration layer, versioned artifact storage, explicit human approval checkpoints and a bidirectional Model Context Protocol (MCP) layer for tool-oriented integration. The work is positioned against recent research on LLM-based software engineering, test generation, software agents and tool-augmented language models. The evaluation is intentionally limited and does not claim productivity gains or industrial readiness. It uses five web-application scenarios executed in deterministic mode, six success criteria, backend regression tests, a workflow-property comparison with two baselines and an additional exploratory check with a real commercial LLM provider on two project prompts. The results provide preliminary evidence that the prototype can execute a traceable requirements-to-artifacts workflow, generate structurally valid deployment bundles and preserve run metadata. The evaluation does not measure semantic correctness of generated business logic and should be extended with repeated real-provider experiments, larger benchmarks and controlled user studies.

Keywords: case system; generative artificial intelligence; software engineering; LLM; requirements engineering; code generation; test generation; mcp; software architecture; human-in-the-loop; evaluation.

For citation: Polygalov B.A., Lanin V.V. Development of a CASE Platform with Generative Artificial Intelligence and an MCP Agent Layer. Trudy ISP RAN/Proc. ISP RAS, 2026, vol. 38, issue 4, part 2, pp. 279-292. DOI: 10.15514/ISPRAS-2026-38(4)-32

Разработка CASE-платформы с генеративным искусственным интеллектом и агентным слоем модельного контекста

Б.А. Полыгалов, ORCID: 0009-0003-1256-3003 <bogdan.polygalov@gmail.com>
В. Ланин, ORCID: 0000-0002-0650-2314 <vlanin@hse.ru>

Кафедра информационных технологий в бизнесе, НИУ ВШЭ,
Россия, 614070, Пермь, ул. Студенческая, 38.

Аннотация. В статье представлена архитектура, реализация и предварительная сервисная оценка прототипа CASE-платформы, использующей генеративный искусственный интеллект для структурирования требований, генерации кода, формирования тестовых артефактов и проверки развёртывания. Платформа объединяет адаптерный слой интеграции с большой языковой моделью, версионное хранилище артефактов, явные контрольные точки подтверждения пользователем и двунаправленный слой модельного контекста (MCP) для инструментальной интеграции. Оценка ограничена воспроизводимым детерминированным-режимом, регрессионными тестами серверной части, сравнением свойств рабочего процесса с двумя базовыми режимами и дополнительной разведочной проверкой с реальным коммерческим поставщиком языковой модели на двух проектных запросах. Результаты дают предварительные свидетельства архитектурной реализуемости трассируемого процесса от требований к артефактам, но не доказывают промышленную готовность решения или семантическую корректность всей сгенерированной бизнес-логики.

Ключевые слова: CASE-система; генеративный искусственный интеллект; программная инженерия; большие языковые модели; анализ требований; генерация кода; генерация тестов; протокол модельного контекста; контроль со стороны человека; оценка процесса.

Для цитирования: Полыгалов Б.А., Ланин В.В. Разработка CASE-платформы с генеративным искусственным интеллектом и агентным слоем модельного контекста. Труды ИСП РАН, 2026, том 38, вып. 4, часть 2, стр. 279-292 (на английском языке). DOI: 10.15514/ISPRAS-2026-38(4)-32

1. Introduction

Computer-Aided Software Engineering (CASE) technologies were introduced to reduce the effort required for modeling, documentation, code generation and maintenance of software systems. Classical CASE and model-driven engineering approaches provide structured artifacts and traceability, but they usually require formal models, specialized notations and substantial adoption effort [1-2]. At the same time, large language models (LLMs) have become capable of assisting with code generation, requirements analysis, testing and program repair, as summarized in recent [3-5].

The problem addressed in this paper is not how to call an LLM once, but how to organize an engineering workflow in which requirements, prompts, generated files, tests, deployment artifacts and logs remain traceable. Standalone chat prompting is flexible, but it provides weak reproducibility and weak artifact management. Classical template-based generation is reproducible, but it is less adaptable to informal natural-language requirements. The motivation of the work is therefore to explore a controlled AI-assisted CASE workflow that combines requirements structuring, constrained generation, versioned artifacts, user approvals and deployment checks.

The paper does not claim that the prototype is superior to commercial assistants or that it improves developer productivity. Instead, it evaluates whether a small prototype can implement a reproducible, traceable and structurally validated workflow from requirements to deployment artifacts. The contributions are:

- a CASE-oriented workflow that connects requirements structuring, code generation, test-artifact generation and deployment checks through versioned artifacts;
- an adapter-based LLM integration layer that decouples platform services from concrete provider APIs;
- a bidirectional MCP layer that allows the platform to expose CASE operations as tools and to consume external tools, resources and prompts;

- a reproducible deterministic demo mode and backend regression tests for service-level validation;
- a preliminary evaluation with explicit research questions, success criteria, deterministic scenarios, regression tests, baseline workflow comparison and an exploratory real-provider check.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 formulates the problem and success criteria. Section 4 presents the architecture. Section 5 describes the implementation. Section 6 reports the preliminary evaluation. Section 7 discusses threats to validity. Section 8 concludes the paper.

2. Related Work

2.1 Classical CASE and model-driven engineering

The CASE vision is closely related to model-driven engineering (MDE), where models and design artifacts guide implementation and maintenance. Brambilla, Cabot and Wimmer describe model-driven software engineering as a practice in which models are not only documentation, but also inputs to transformations, code generation and analysis [1]. da Silva surveys model-driven engineering and emphasizes that the benefits of MDE depend on the quality of modeling languages, transformations and tool support [2]. These works support the artifact-centric part of our platform: requirements, structured representations, diagrams and generated bundles are stored as explicit artifacts rather than transient chat messages. However, classical CASE/MDE approaches usually assume more formalized input than many early-stage projects can provide.

2.2 LLMs for software engineering and code generation

Recent surveys report that LLMs are being applied across requirements engineering, code generation, repair, testing, documentation and maintenance [3-5]. Empirical and usability studies also show that AI-assisted coding tools can be useful but require careful evaluation. Vaithilingam et al. studied the gap between developer expectations and experience when using LLM-powered code generation tools [6]. Pearce et al. demonstrated that code generated by Copilot-like systems can contain security weaknesses, which supports the need for validation and human review [7]. Large-scale code generation systems such as AlphaCode [8], Codex [9], Synchromesh [10], CodeGen [11] and InCoder [12] show the progress of LLM-based programming, but they primarily evaluate model capability rather than a CASE-style artifact lifecycle.

The implication for this work is that LLMs should not be treated as a replacement for engineering process control. Our platform uses LLMs inside a constrained workflow: a confirmed requirements structure is converted into a prompt; the model is asked to return a file map; the platform validates required files and records run metadata. This differs from pure code-completion settings, where the output is often evaluated only locally by the developer.

2.3 LLMs for requirements engineering and testing

Requirements engineering is a natural application area for LLMs because requirements are often written in informal language. Recent reviews on LLMs for requirements engineering indicate potential in elicitation, classification, ambiguity detection and transformation of natural-language requirements, but also report limitations related to hallucinations, missing domain context and evaluation datasets [13, 14]. Our platform therefore introduces a mandatory intermediate artifact: the requirements structure is stored as both JSON and Markdown and must be confirmed by the user before code generation.

Testing is another area where LLMs are increasingly studied. Siddiq et al. evaluate LLMs for JUnit test generation and report that generated tests can suffer from correctness and quality problems [15]. Schäfer et al. study automated unit-test generation using LLMs and show that generated tests can

improve coverage, but require execution feedback and repair [16]. Traditional tools such as EvoSuite and Pynguin remain important baselines for automatic test generation [17-18]. These results motivate the test-artifact layer of our prototype. The platform does not claim to generate semantically complete tests; it generates a test bundle, traceability matrix and smoke checks that make validation more explicit.

2.4 Agentic software engineering systems

Agentic software engineering systems attempt to move beyond single-turn code completion. MetaGPT organizes multi-agent collaboration around standardized operating procedures [19]. ChatDev explores chat-based collaboration between role-specialized agents [20]. AutoDev investigates autonomous execution of software engineering tasks with guarded access to files, commands and development environments [21]. SWE-agent shows how agent-computer interfaces can improve an agent's ability to inspect files, edit repositories and run tests [22]. These systems are important because they demonstrate longer workflows, tool use and feedback loops. Their limitation for the present problem is that they usually focus on agent performance on coding or issue-resolution tasks, while our work focuses on CASE-style artifact management, requirements confirmation and reproducible service-level execution.

2.5 Benchmarks, self-improvement and tool use

Evaluation of LLM-based software engineering has also become more rigorous. SWE-bench introduced real GitHub issue resolution as an executable benchmark [23]. ReAct [24], Reflexion [25] and Self-Refine [26] show that reasoning, acting and iterative feedback can improve LLM workflows. Tool-oriented research such as Toolformer [27], Gorilla [28], API-Bank [29] and ToolLLM [30] demonstrates that LLMs can be connected to external APIs and tools, but also that tool selection, argument formation and error handling are difficult. MCP standardizes part of this integration problem by exposing tools, resources and prompts through a common protocol [31].

2.6 Gap addressed by this work

The reviewed literature leaves three relevant gaps. First, many LLM-based tools are evaluated as assistants or agents, while less attention is paid to CASE-style management of generated artifacts, runs and approvals. Second, requirements engineering, code generation, testing and deployment are often studied separately; the prototype integrates them into one traceable workflow. Third, tool-oriented integration is usually discussed at the agent level, whereas the present platform combines provider adapters and bidirectional MCP with a CASE workflow. The contribution is therefore architectural and empirical at a preliminary service level, not a claim of scientific novelty in individual mechanisms such as adapter patterns or human-in-the-loop approval.

References to non-peer-reviewed primary artifacts are retained only where they are necessary to identify a technical report, protocol specification or prototype artifact, for example Codex, MCP and the public repository. The related-work argument itself relies primarily on peer-reviewed surveys, empirical studies, conference papers and journal articles.

3. Problem Statement and Success Criteria

The platform is intended to support an early-stage software development scenario. A user creates a project, provides a requirements description, launches requirements structuring, reviews and confirms the structure, generates application files, generates test artifacts and performs a deployment check. The evaluation asks whether this workflow can be executed in a reproducible and traceable way.

The following research questions were used.

- RQ1. Can the prototype complete the end-to-end service-level workflow for several small web-application scenarios in deterministic mode?

- RQ2. Does the platform enforce structural constraints on generated artifacts, including a file-map format, required Docker Compose bundle and user-facing frontend?
- RQ3. Does the workflow preserve traceability through run metadata, artifact versions and human approval checkpoints?

The success criteria were defined before running the evaluation.

- SC1 – workflow completion: the platform completes requirements upload, structuring, confirmation, code generation, test-artifact generation and deployment check.
- SC2 – constrained generation format: generated code is represented as a file map and includes docker-compose.generated.yml and a user-facing frontend.
- SC3 – validation and repair: invalid or incomplete file maps are rejected, repaired or replaced by deterministic fallback artifacts according to the selected mode.
- SC4 – traceability: each run stores provider, model, task type, prompt snapshot, input payload, output payload, status and artifact links.
- SC5 – human approval: requirements confirmation, code generation, test generation, patch application and real deployment require explicit approval.
- SC6 – reproducibility: the source code and deterministic mode allow independent repetition of the service-level evaluation.

4. System Design

4.1 Overall architecture

The implemented MVP is a modular web platform rather than a set of fully independent microservices. This choice reduces prototype complexity while preserving clear boundaries between services. The backend contains services for projects, requirements, generation, testing, deployment, artifact storage, LLM provider integration, MCP integration and agent execution. The frontend exposes a project workspace where the user can configure providers, upload requirements, review structures, start generation and inspect artifacts. The overall architecture of the prototype is shown in Fig. 1.

The main pipeline is: raw requirements; structured requirements; confirmed structure; generated code; generated test artifacts; deployment check; post-generation refinement. Each transition creates or updates persistent metadata. This design supports traceability and allows the user to inspect intermediate artifacts rather than accepting an opaque model response.

The project data model is centered around the entities Project, RequirementDocument, RequirementStructure, GenerationRun, GeneratedArtifact, TestRun, DeploymentRun, ProviderConfig and MCP-related entities such as MCPServerConnection, MCPTool, MCPResource, MCPPrompt and MCPInvocationLog. This structure makes it possible to store not only the current project state, but also the history of operations: which requirements document was uploaded, which structured version was confirmed, which generation run produced a given set of artifacts, and which testing or deployment-related files were created. Fig. 2 presents a simplified class diagram of the prototype, focusing on the main domain entities and service-level relationships.

4.2 Requirements structuring

The requirements module supports direct upload of Markdown/text requirements and generation of a draft requirements document from a short description. The structuring step converts requirements

into a JSON/Markdown artifact containing functional requirements, non-functional requirements, domain entities, user stories, acceptance criteria, constraints, UI screens, backend modules, test scenarios, risks and gaps. This intermediate artifact is central to the platform: code and test generation are not started from raw text alone, but from the latest confirmed structure.

4.3 Generation service and validation logic

The generation service builds prompts from the confirmed requirements structure and the project description. The code-generation prompt instructs the selected LLM to return only JSON with a files map. The platform then extracts the map, checks the required files, determines referenced Docker build files, normalizes conflicting preview ports and verifies that the bundle contains a user-facing frontend. If the response is malformed, the service attempts to repair it by requesting a strict JSON transformation. In non-strict mode, missing mandatory artifacts can be filled with deterministic templates. This validation does not prove semantic correctness, but it prevents a common failure mode where an LLM returns explanations, partial snippets or an API-only bundle instead of a deployable web application.

4.4 LLM provider abstraction

The LLM provider layer uses the adapter pattern. A base adapter defines operations for text generation, structured generation, code generation, test generation, model listing and health checks. Concrete adapters can be implemented for different providers, while platform services depend only on the common interface. This design reduces coupling and supports provider replacement, fallback policies and deterministic execution without external API keys.

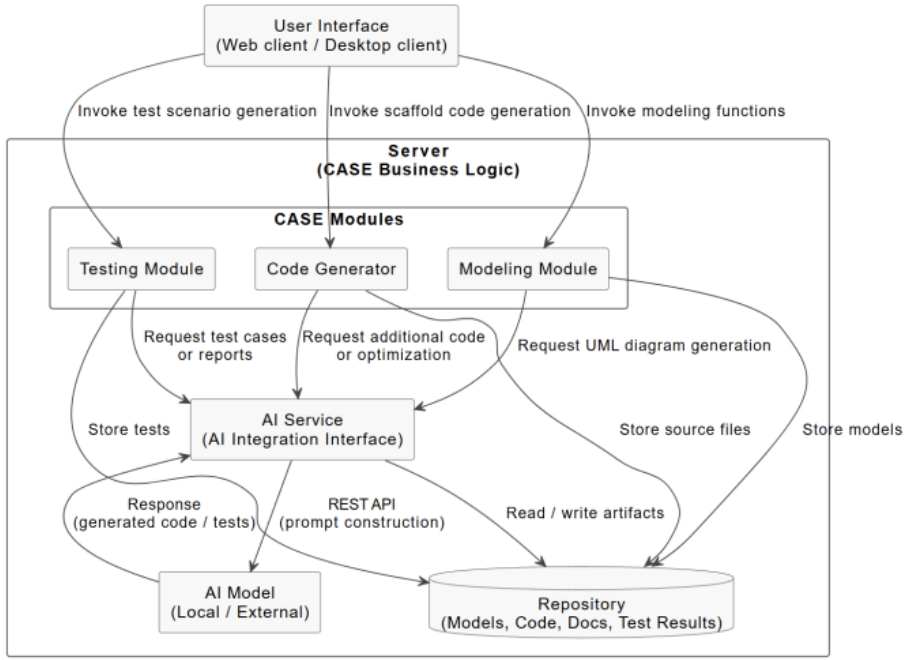

Fig. 1. Architecture of the CASE platform prototype.

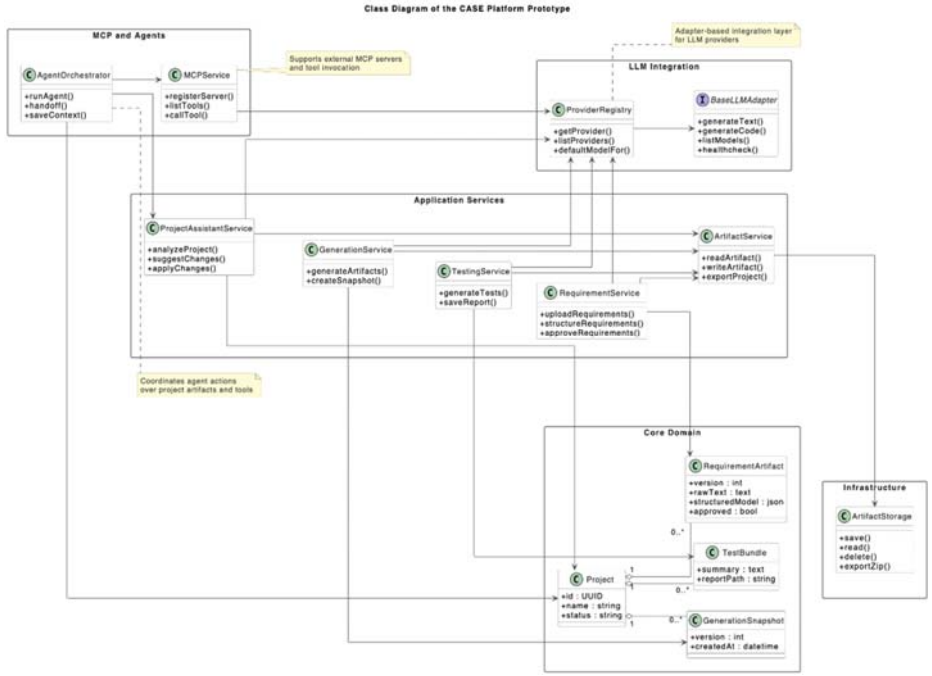

Fig. 2. Simplified class diagram of the CASE platform prototype.

4.5 MCP layer and human control

The MCP layer is bidirectional. As an MCP server, the platform can expose CASE operations such as project creation, requirements upload, structuring, code generation and artifact listing as tools. As an MCP client, it can register external MCP servers and synchronize tools, resources and prompts. The current implementation should be interpreted as an integration layer for future agentic scenarios, not as a fully evaluated multi-agent system.

Human-in-the-loop control is implemented as explicit approval checkpoints. The platform does not treat an LLM as an autonomous software engineer. The user confirms structured requirements, approves generation and decides whether to apply assistant-generated patches or start real deployment. This follows the empirical evidence that LLM outputs require verification, especially for security, correctness and requirements conformance [6-7, 15-16].

5. Prototype Implementation

The backend is implemented in Python 3.12 using FastAPI for HTTP routes, SQLAlchemy for database access and PostgreSQL for persistent metadata. Redis and Celery are included for asynchronous and background operations. Generated artifacts are stored in a local project storage hierarchy and registered in the database. The frontend is implemented with React and TypeScript. Docker Compose is used for reproducible local deployment of the platform and generated applications.

The main backend services are RequirementService, GenerationService, TestService, DeploymentService, ArtifactService and the provider registry. RequirementService creates

versioned raw and structured requirements. GenerationService creates code and diagram artifacts, validates file maps and records GenerationRun metadata. TestService creates a test bundle with a test plan, traceability matrix, backend smoke tests, frontend smoke tests and JUnit-style report. DeploymentService checks generated Docker Compose bundles through dry-run configuration validation and can perform real local deployment when Docker access is available.

At the implementation level, ProviderRegistry performs centralized provider selection, while BaseLLMAdapter and HTTPBasedLLMAdapter define the provider-independent contract and shared HTTP-oriented behavior for concrete integrations. GenerationRun stores provider, model, task type, prompt snapshot, input and output payloads, status, token information and correlation identifier. ArtifactStorage saves versioned files in functional directories, and MCPInvocationLog records calls to external MCP tools and resources. ProjectAssistantService uses the stored project state and deployment logs to support post-generation explanations and patch proposals, but patch application remains controlled by explicit user approval.

The implementation includes a deterministic FakeLLMAdapter. It is not intended to simulate real LLM quality. Its purpose is reproducibility: regression tests and the service-level evaluation can be repeated without API keys, network variability or changing commercial model versions. Real providers can be connected through adapters; an additional exploratory check with a real commercial LLM provider is reported separately as a qualitative sanity check rather than as a full model benchmark.

6. Preliminary Evaluation

6.1 Evaluation design

The evaluation was designed to provide preliminary empirical evidence for the proposed architecture. It is preliminary and service-level: it measures workflow completion, artifact structure, metadata preservation and deployment-check behavior in deterministic mode, and it additionally reports a qualitative real-provider sanity check. It does not measure developer productivity, full semantic correctness of generated business logic, usability in a real team or comparative performance of commercial LLMs.

Five scenarios were used: task management, pizza ordering, library catalog, simple CRM and event registration. Each scenario contained a short natural-language requirements description. For each scenario, the same sequence was executed: create project; upload or draft requirements; structure requirements; confirm structure; generate code; generate test artifacts; run deployment check; inspect runs and artifacts.

The service-level procedure was executed at the repository state cited in [32]. For each scenario, a clean project was created and the deterministic FakeLLMAdapter was selected explicitly. The raw evidence recorded by the platform consisted of GenerationRun, TestRun and DeploymentRun records, prompt and payload snapshots, and generated artifact directories for the corresponding project. The regression-test evidence was the backend pytest service-suite result reported in Section 6.3.

6.2 Metrics and results

The collected metrics were workflow completion, presence of `'docker-compose.generated.yml'`, presence of a user-facing frontend, test-bundle generation, artifact count, run metadata presence and deployment-check status. Table 1 reports service-level results. Timings are intentionally not interpreted as real LLM performance because deterministic generation removes external API latency.

Table 1. Preliminary service-level evaluation results.

Scenario	Workflow	Compose file	Frontend	Test bundle	Artifacts
Task management	completed	yes	yes	generated	27
Pizza ordering	completed	yes	yes	generated	27
Library catalog	completed	yes	yes	generated	27
Simple CRM	completed	yes	yes	generated	27
Event registration	completed	yes	yes	generated	27

The evaluation satisfied SC1 for all five scenarios: the service-level workflow completed in deterministic mode. SC2 was satisfied because each generated bundle contained `'docker-compose.generated.yml'` and a frontend entry. SC4 was satisfied because GenerationRun, TestRun and DeploymentRun records were produced and linked to artifacts. SC5 was satisfied by explicit approval flags before requirements confirmation, code generation, test generation and deployment. SC6 was satisfied by the public repository [32] and deterministic provider. SC3 was partially evaluated by regression tests for malformed and incomplete generation outputs.

6.3 Regression tests and validation checks

The backend regression suite contained seven service tests and passed 7/7 tests. The covered cases included requirements drafting and structuring, code generation, diagram generation, provider registry behavior, test generation, artifact versioning and deployment-check paths. One negative test verifies that a generated Docker bundle containing only an API service is rejected because it lacks a user-facing frontend. Another test verifies that saved text artifacts create a new version rather than overwriting the previous artifact. These tests support RQ2 and RQ3, but they remain structural tests rather than semantic business-logic tests.

6.4 Baseline comparison

A full quantitative comparison with commercial tools would require a controlled user study and fixed model versions; it was not performed. However, to make the evaluation less purely descriptive, we compared the workflow properties of the proposed platform with two baseline interaction modes. Baseline 1 is standalone chat-based prompting, where a user asks an LLM to generate a project and manually copies files. Baseline 2 is classical template-only generation, where files are produced from deterministic templates without LLM adaptation to informal requirements. Table 2 summarizes the comparison.

Table 2. Workflow-property comparison with baseline interaction modes.

Property	Standalone chat prompting	Classical template-only generation	Proposed CASE platform
Requirements confirmation	manual and external	usually fixed by template input	explicit platform checkpoint
Structured file-map generation	not enforced	predefined	enforced JSON files map
Artifact versioning	manual	possible but tool-specific	built into artifact storage
Run metadata	absent or external	limited	provider, model, prompt, status and payloads stored

Reproducible deterministic mode	no	yes	yes, via FakeLLMAdapter
Generated test bundle	ad hoc	template-defined	generated and stored with traceability matrix
Deployment-check artifact	manual	template-defined	dry-run/compose check recorded
Human approval checkpoints	implicit	tool-specific	explicit before critical transitions

The comparison does not demonstrate superiority in productivity or quality. It shows that the proposed prototype combines adaptability of LLM-based prompting with reproducibility and traceability properties usually associated with CASE or template-based tools. This directly addresses RQ3 and supports the architectural claim of a traceable deterministic workflow.

6.5 Exploratory real-provider check

In addition to the deterministic service-level evaluation, an exploratory manual check was performed with a real commercial LLM provider. Two additional project prompts were submitted through the platform: an e-commerce website for ordering goods and a browser-based drawing application with basic graphic-editor capabilities. For both prompts, the platform completed the same controlled workflow: requirements were structured, the intermediate structure was reviewed, code generation was launched through the provider adapter, generated files were stored as versioned artifacts, and deployment/test artifacts were registered.

Across these two runs, no file-map format violations or missing mandatory deployment artifacts were observed. Manual inspection showed that the generated screen and component structure corresponded to the uploaded requirements at the structural level: the ordering scenario produced user-facing e-commerce pages and order-related artifacts, while the drawing scenario produced a frontend-oriented application structure with drawing/editor functionality. This check is treated as a qualitative sanity check only. It is not included in Table 1 because it was not repeated enough times to estimate variance and it does not replace semantic assessment, security review or a controlled comparison of LLM providers.

6.6 Reproducibility package

The source code is publicly available [32]. The evaluated artifact version was identified through the public repository commit dd15d1594301980d4c56f004672a2d30db264df9, retrieved on 7 June 2026. The repository contains the backend, frontend, Docker Compose configuration, deterministic fake provider, tests and documentation. The project targets Python 3.12. The repository README specifies quick start with `` `cp .env.example .env` `` and `` `docker compose up --build` ``; local development uses `` `python3 -m pip install -e ".[dev]"` ``, frontend `` `npm install` `` and `` `docker compose -f docker-compose.dev.yml up --build` ``.

The evaluation can be reproduced by running the backend regression tests with pytest and by executing the same five service-level scenarios using the deterministic provider. The deterministic part of the evaluation can be reproduced without external API keys; the exploratory real-provider check requires access to the corresponding provider and may vary across model versions.

6.7 Interpretation

The results provide preliminary evidence of architectural viability. They show that the prototype can complete the intended deterministic workflow, create structurally valid deployment bundles, preserve run metadata and support deterministic reproduction. The additional real-provider check indicates basic compatibility with a real commercial LLM provider on two manually submitted project prompts. These results do not prove industrial readiness, full semantic correctness, security

of generated applications or productivity gains. Such claims require systematic real-provider experiments, larger datasets, human evaluators and comparison against established tools.

7. Discussion and Threats to Validity

7.1 Internal validity

The main internal validity threat is deterministic generation. It makes the evaluation reproducible but hides variability of real LLM outputs. The paper mitigates this threat by avoiding claims about model quality and by limiting conclusions to workflow and structural validation.

7.2 External validity

The scenarios are small web applications. The results cannot be generalized to enterprise systems, mobile applications, embedded software or safety-critical domains. The platform also generates relatively simple deployment bundles. Further work should evaluate larger projects and richer requirements sets.

7.3 Construct validity

The metrics measure workflow completion, structural validity and traceability. They do not measure semantic correctness of generated business logic. A generated application may contain all required files and pass health checks while still failing to implement the expected domain behavior. This is why the platform includes human approval checkpoints and why future work should include expert assessment and executable acceptance tests derived from requirements.

7.4 Reliability

The deterministic provider improves reliability of the service-level experiment. The exploratory real-provider check demonstrates that the workflow can also be executed with a real commercial provider, but it was not repeated enough times to estimate variance. Systematic experiments with real providers require fixed model versions, temperatures, prompts, seeds where available, and repeated runs to characterize stability.

8. Conclusion

The paper presented a prototype CASE platform with generative artificial intelligence and an MCP agent layer. The platform is evaluated through an expanded related-work analysis, explicit success criteria, a preliminary service-level evaluation, a baseline workflow comparison, reproducibility information and an exploratory real-provider check. The contribution should be interpreted as an architectural and engineering prototype supported by preliminary evidence, not as proof of productivity gains or superiority over existing tools.

The preliminary evaluation showed that the deterministic prototype completed five requirements-to-code-to-tests scenarios, generated structurally valid Docker Compose bundles with frontend entries, produced test artifacts and preserved run metadata. An additional real-provider check on an e-commerce website for ordering goods and a browser-based drawing application with graphic-editor capabilities produced structurally valid artifacts consistent with the uploaded requirements. Future work should include repeated runs with real LLM providers, semantic assessment of generated applications, security review, controlled user studies and stronger comparison with AI-assisted development tools and agentic software engineering systems. A separate engineering direction is the transition from adapter-extensible integration to a schema-driven plugin architecture for LLM providers, including locally deployed models through OpenAI-compatible endpoints, native REST APIs, local SDKs and MCP wrappers.

References

[1]. Brambilla M., Cabot J., Wimmer M. Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering. Morgan & Claypool, 2012, 182 p. DOI: 10.2200/S00441ED1V01Y201208SWE001.

[2]. da Silva A.R. Model-driven engineering: A survey supported by a unified conceptual model. Computer Languages, Systems & Structures, 2015, vol. 43, pp. 139-155. DOI: 10.1016/j.cl.2015.06.001.

[3]. Fan A., Gokkaya B., Harman M., Lyubarskiy M., Sengupta S., Yoo S., Zhang J.M. Large Language Models for Software Engineering: Survey and Open Problems. In Proc. of the IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE), 2023, pp. 31-53. DOI: 10.1109/ICSE-FoSE59343.2023.00008.

[4]. Hou X., Zhao Y., Liu Y., Yang Z., Wang K., Li L., Luo X., Lo D., Grundy J., Wang H. Large Language Models for Software Engineering: A Systematic Literature Review. ACM Transactions on Software Engineering and Methodology, 2024, vol. 33, no. 8, article 220. DOI: 10.1145/3695988.

[5]. Zhang Q., Fang C., Xie Y., Zhang Y., Yang Y., Sun W., Yu S., Chen Z. A Survey on Large Language Models for Software Engineering. Science China Information Sciences, 2026, vol. 69, article 141102. DOI: 10.1007/s11432-025-4670-0.

[6]. Vaithilingam P., Zhang T., Glassman E.L. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In Proc. of the CHI Conference on Human Factors in Computing Systems, 2022, article 332, pp. 1-18. DOI: 10.1145/3491101.3519665.

[7]. Pearce H., Ahmad B., Tan B., Dolan-Gavitt B., Karri R. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In Proc. of the IEEE Symposium on Security and Privacy, 2022, pp. 754-768. DOI: 10.1109/SP46214.2022.9833571.

[8]. Li Y., Choi D., Chung J., Kushman N., Schrittwieser J., Leblond R., Eccles T., et al. Competition-level code generation with AlphaCode. Science, 2022, vol. 378, no. 6624, pp. 1092-1097. doi: 10.1126/science.abq1158.

[9]. Chen M., Tworek J., Jun H., Yuan Q., Pinto H.P.O., Kaplan J., Edwards H., et al. Evaluating Large Language Models Trained on Code. Preprint, 2021. DOI: 10.48550/arXiv.2107.03374.

[10]. Poesia G., Polozov O., Le V., Tiwari A., Soares G., Meek C., Gulwani S. Synchromesh: Reliable code generation from pre-trained language models. In Proc. of the International Conference on Learning Representations, 2022.

[11]. Nijkamp E., Pang B., Hayashi H., Tu L., Wang H., Zhou Y., Savarese S., Xiong C. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In Proc. of the International Conference on Learning Representations, 2023.

[12]. Fried D., Aghajanyan A., Lin J., Wang S., Wallace E., Shi F., Zhong R., et al. InCoder: A Generative Model for Code Infilling and Synthesis. In Proc. of the International Conference on Learning Representations, 2023.

[13]. Arora C., Sabetzadeh M., Briand L.C., Zimmer F. Machine learning and natural language processing for requirements engineering: A systematic mapping study. ACM Computing Surveys, 2022, vol. 54, no. 3, article 55. DOI: 10.1145/3444689.

[14]. Zadenoori M.A., Dabrowski J., Alhoshan W., Zhao L., Ferrari A. Large Language Models for Requirements Engineering: A Systematic Literature Review. Preprint, 2025. DOI: 10.48550/arXiv.2509.11446.

[15]. Siddiq M.L., Santos J.C.S., Tanvir R.H., Ulfat N., Al Rifat F., Lopes V.C. Using Large Language Models to Generate JUnit Tests: An Empirical Study. In Proc. of the International Conference on Evaluation and Assessment in Software Engineering (EASE), 2024, pp. 313-322. DOI: 10.1145/3661167.3661216.

[16]. Schäfer M., Nadi S., Eghbali A., Tip F. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. IEEE Transactions on Software Engineering, 2024, vol. 50, no. 1, pp. 85-105. DOI: 10.1109/TSE.2023.3334955.

[17]. Fraser G., Arcuri A. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In Proc. of the ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering, 2011, pp. 416-419. DOI: 10.1145/2025113.2025179.

[18]. Lukasczyk S., Kroiss F., Fraser G. Automated Unit Test Generation for Python. In Proc. of the International Symposium on Search Based Software Engineering, 2020, pp. 9-24. DOI: 10.1007/978-3-030-59762-7_2.

[19]. Hong S., Zhuge M., Chen J., Zheng X., Cheng Y., Zhang C., Wang J., et al. MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework. In Proc. of the International Conference on Learning Representations, 2024.

[20]. Qian C., Liu W., Liu H., Chen N., Dang Y., Li J., Yang C., Chen W., Su Y., Cong X., Xu J., Li D., Liu Z., Sun M. ChatDev: Communicative Agents for Software Development. In Proc. of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL), 2024, pp. 15174-15186. DOI: 10.18653/v1/2024.acl-long.810.
[21]. Tufano M., Agarwal A., Jang J., Moghaddam R.Z., Sundaresan N. AutoDev: Automated AI-Driven Development. Preprint, 2024. DOI: 10.48550/arXiv.2403.08299.
[22]. Yang J., Jimenez C.E., Wettig A., Lieret K., Yao S., Narasimhan K., Press O. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In Advances in Neural Information Processing Systems, 2024.
[23]. Jimenez C.E., Yang J., Wettig A., Yao S., Pei K., Press O., Narasimhan K. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In Proc. of the International Conference on Learning Representations, 2024.
[24]. Yao S., Zhao J., Yu D., Du N., Shafran I., Narasimhan K., Cao Y. ReAct: Synergizing Reasoning and Acting in Language Models. In Proc. of the International Conference on Learning Representations, 2023.
[25]. Shinn N., Cassano F., Gopinath A., Narasimhan K., Yao S. Reflexion: Language Agents with Verbal Reinforcement Learning. In Advances in Neural Information Processing Systems, 2023, vol. 36.
[26]. Madaan A., Tandon N., Gupta P., Hallinan S., Gao L., Wiegreffe S., Alon U., et al. Self-Refine: Iterative Refinement with Self-Feedback. In Advances in Neural Information Processing Systems, 2023.
[27]. Schick T., Dwivedi-Yu J., Dessi R., Raileanu R., Lomeli M., Hambro E., Zettlemoyer L., Cancedda N., Scialom T. Toolformer: Language Models Can Teach Themselves to Use Tools. In Advances in Neural Information Processing Systems, 2023.
[28]. Patil S.G., Zhang T., Wang X., Gonzalez J.E. Gorilla: Large Language Model Connected with Massive APIs. In Advances in Neural Information Processing Systems, 2024, vol. 37.
[29]. Li M., Zhao Y., Yu B., Song F., Li H., Yu H., Li Z., Huang F., Li Y. API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs. In Proc. of the Conference on Empirical Methods in Natural Language Processing, 2023, pp. 3102-3116. DOI: 10.18653/v1/2023.emnlp-main.187.
[30]. Qin Y., Liang S., Ye Y., Zhu K., Yan L., Lu Y., Lin Y., et al. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. In Proc. of the International Conference on Learning Representations, 2024.
[31]. Anthropic. Model Context Protocol Specification. 2024-2026. Available at: https://modelcontextprotocol.io/, accessed 07.06.2026.
[32]. Polygalov B. HSE-CASE-AI: Prototype CASE Platform with Generative AI. GitHub repository, commit dd15d1594301980d4c56f004672a2d30db264df9, 2026. Available at: https://github.com/miamib34ch/HSE-CASE-AI, accessed 07.06.2026.

Information about authors / Информация об авторах

Богдан Александрович ПОЛЫГАЛОВ – магистрант НИУ ВШЭ, кафедра информационных технологий в бизнесе. Область научных интересов: программная инженерия, CASE-системы, генеративный искусственный интеллект, инструменты на основе LLM и архитектура программных систем.

Bogdan Alexandrovich POLYGALOV – master's student at HSE University, IT in Business Department. Research interests: software engineering, CASE systems, generative artificial intelligence, LLM-based tools and software architecture.

Вячеслав Владимирович ЛАНИН – НИУ ВШЭ, кафедра информационных технологий в бизнесе. Область научных интересов: программная инженерия, архитектура информационных систем и бизнес-аналитика.

Viacheslav Vladimirovich LANIN – HSE University, IT in Business Department. Research interests: software engineering, information systems architecture and business analytics.