https://doi.org/10.31891/2219-9365-2026-85-13
UDC 004.4, 004.8

BOIKO Viacheslav
Khmelnitskyi National University
https://orcid.org/0009-0004-9443-0286
e-mail: bviacheslav.12@gmail.com
MARTYNIUK Valerii
Khmelnitskyi National University
https://orcid.org/0000-0001-5758-4244
e-mail: mac30973097@gmail.com

METHOD OF AUTOMATED INTEGRATION TEST GENERATION AND CONTINUOUS EXECUTION IN LARGE-SCALE CODE REPOSITORIES

Maintaining high-quality tests in large-scale code repositories remains a critical challenge due to frequent code changes and the high cost of manual test creation and maintenance. This paper proposes a method of automated test generation and continuous execution integrated directly into a .NET Azure DevOps pipeline. The method leverages large language models to automatically generate and update integration tests in response to every code commit. Once generated, the tests are compiled and executed within the same pipeline, while a self-healing mechanism attempts to automatically recover failing AI-generated tests. If recovery fails, the pipeline halts to prevent defective deployments. The approach ensures that the test suite evolves continuously alongside the codebase, enabling real-time validation of new functionality. Experimental evaluation on a large-scale project demonstrated improved test coverage, reduced manual testing effort, and enhanced defect detection during continuous integration. The research highlights the synergy between AI-driven code analysis and continuous testing, showing how automated test generation can strengthen DevOps practices. Future work will focus on fine-tuning language models for domain-specific test generation and improving self-healing accuracy to further reduce developer intervention.

Keywords: automated test generation, AI-driven testing, integration tests, software quality assurance, object-oriented programming, Azure DevOps, large-scale repositories.

БОЙКО В’ячеслав, МАРТИНЮК Валерій
Хмельницький національний університет

МЕТОД АВТОМАТИЗОВАНОГО ГЕНЕРУВАННЯ ІНТЕГРАЦІЙНИХ ТЕСТІВ ТА ЇХ НЕПЕРЕРВНОГО ВИКОНАННЯ У ВЕЛИКОМАСШТАБНИХ РЕПОЗИТОРІЯХ КОДУ

Підтримання високої якості тестування у великомасштабних репозиторіях коду залишається складним завданням через часті зміни програмного забезпечення та високу вартість мануального тестування. У статті запропоновано метод автоматизованого генерування тестів та їх безперервного виконання, інтегрований безпосередньо в конвеєр збірки Azure DevOps для .NET-проєктів. Метод використовує великі мовні моделі для автоматичного створення та оновлення інтеграційних тестів у відповідь на кожен коміт коду. Згенеровані тести компілюються та виконуються в тому ж конвеєрі, а механізм самовідновлення намагається автоматично виправити збої у тестах, згенерованих ШІ. Якщо відновлення неможливе, конвеєр припиняє виконання, запобігаючи некоректним розгортанням. Запропонований підхід забезпечує еволюцію тестового набору синхронно з розвитком коду, дозволяючи здійснювати перевірку нової функціональності в реальному часі. Експериментальна оцінка на великому проєкті продемонструвала підвищення покриття тестами, зменшення обсягу ручної роботи та покращення виявлення дефектів у процесі безперервної інтеграції. Дослідження підкреслює синергію між ШІ-керованим аналізом коду та неперервним тестуванням, демонструючи, як автоматизоване генерування тестів може зміцнити практики DevOps. Подальші дослідження зосереджуватимуться на точному налаштуванні мовних моделей для доменно-специфічного тестування та вдосконаленні точності самовідновлення з метою мінімізації втручання розробників..

Ключові слова: автоматизоване генерування тестів; тестування на основі штучного інтелекту; інтеграційні тести; забезпечення якості програмного забезпечення; об’єктно-орієнтоване програмування; Azure DevOps; великомасштабні репозиторії.

Стаття надійшла до редакції / Received 04.12.2025
Прийнята до друку / Accepted 02.02.2026
Опубліковано / Published 05.03.2026

ANALYSIS OF THE LATEST RESEARCH

Recent advancements in AI-driven software testing suggest that intelligent tools can automatically create detailed test cases, adapt to code changes, and even identify risky code areas – enabling continuous testing with self-healing capabilities [1].

This promises more efficient regression testing, broader test coverage, and faster release cycles. In particular, large language models (LLMs) like OpenAI’s GPT have emerged as powerful engines for generating and improving test code. Researchers have rapidly embraced LLMs for automated test generation, seeing remarkable results but also encountering new challenges. A 2025 systematic review analyzed 105 papers on LLM-based testing and found

significant interest in using LLMs to generate tests and even oracles, while highlighting unresolved issues (e.g. inconsistent outputs, integration complexities) and the need for better prompt strategies and model tuning [2]. Likewise, Celik and Mahmoud report that LLMs have improved metrics such as test coverage, correctness, and developer usability, yet their performance can be inconsistent, with issues like compilation errors and hallucinated assertions still common [3]. These surveys underscore that although LLM-generated tests can boost automation, careful engineering is required to ensure reliability and maintainability.

Concurrently, empirical studies have demonstrated the potential of LLM-based test generation on real codebases. Schäfer et al. introduced TestPilot, a JavaScript unit test generator using GPT-3.5, which achieved a median 70.2% statement coverage – significantly higher than a state-of-the-art search-based technique (51.3% coverage) on the same projects [4]. This shows that even without special fine-tuning, LLMs can produce thorough tests that cover most program logic. In parallel, Yaraghi et al. developed a system called TaRGET that fine-tunes an LLM for automated test repair. TaRGET successfully fixed 66% of broken test cases (exact-match) in a dataset of over 45k failing tests [5]. Such results illustrate two important points of recent research: using AI to generate new tests from scratch and to repair/adapt existing tests when code evolves. Both aim to reduce the manual effort of maintaining test suites in large projects.

Industry practitioners have also begun leveraging AI for testing at scale. A notable example is Meta's TestGen-LLM, an internal tool that integrates LLMs into the test creation workflow for mobile code. Deployed in "test-a-thon" events at Facebook and Instagram, TestGen-LLM automatically extended existing test suites with new cases covering previously untested behaviors. In trials on Instagram's code, 75% of AI-generated test cases built and ran successfully, over 57% passed reliably, and 25% of the new tests increased coverage of the code base [6]. Impressively, when engineers reviewed these additions, 73% of the AI-recommended tests were accepted and merged into production, indicating that LLM-generated tests can meet real-world quality standards in a large-scale repository. This is one of the first reports of industrial-scale deployment of LLM-generated code in testing, and it demonstrates tangible benefits like higher coverage and faster test creation.

Beyond unit tests, researchers are tackling integration and system-level testing with AI. Yachamaneni explores AI-driven test automation for enterprise integration scenarios, where systems of systems must be validated. Key emerging techniques include autonomous testing agents, AI-powered test orchestration, generative AI for test case creation, predictive analytics for detecting likely failure points, and cognitive/self-healing automation [7]. These innovations aim to produce more intelligent and adaptable integration tests that can dynamically respond to changing requirements and system configurations. For example, self-healing test frameworks use AI to automatically adjust tests when an interface or workflow changes (e.g., finding a moved UI element or updating an API call), thereby reducing flaky test failures and maintenance overhead. Such capabilities are especially valuable in large-scale integration testing, where traditional scripted tests often break as systems evolve. The literature indicates that by leveraging AI's ability to analyze runtime behavior and past failures, next-generation test suites can "eliminate repetitive maintenance tasks and proactively identify issues before they hit production" [7]. This aligns with Wavhal's observation that AI-driven testing can continuously monitor and adapt tests, leading to more resilient and scalable quality assurance processes [1].

It is worth mentioning that current AI test generation methods are not without limitations. Andrzejewski et al. found that while LLMs can readily generate unit tests for simple, well-structured code, their effectiveness drops as code complexity increases [8]. In their experiments, GPT-based models often produced tests that were syntactically correct but lacked the semantic depth or diversity of human-written tests (e.g., similar assertions repeated, or missing edge-case scenarios). They also reported that tailoring prompts and model choices is critical: different LLMs varied in coverage achieved, and none consistently outperformed human tests on more complex functionalities. These findings target the need for prompt engineering and possibly fine-tuning to guide LLMs in generating high-quality tests. Another common challenge is test correctness. LLMs may introduce incorrect assumptions or expected values. Research emphasizes adding verification steps or filters. For instance, Meta's approach incorporated a filtration layer to discard any AI-generated tests that did not demonstrably improve coverage or that might cause regressions.

In summary, the state-of-the-art suggests that automated test generation and maintenance are becoming feasible even for large codebases, powered by generative AI. We now have evidence of LLMs generating unit tests with substantial coverage [4], updating tests in response to code changes [5], and augmenting enterprise-level test suites with minimal human intervention [6]. The latest research has progressively moved from small-scale demos towards addressing scalability, reliability, and integration with development workflows. However, challenges remain in achieving consistent accuracy, handling complex logic, and integrating these AI tools seamlessly into Continuous Integration pipelines. Indeed, continuous integration and continuous testing (CI/CT) are viewed as essential practices in modern DevOps to maintain software quality at speed, and any AI-based solution must complement these practices [9, 10]. The method proposed in this paper builds upon these insights, aiming to leverage LLM-driven test generation within a CI pipeline to achieve continuous, automated testing for a large-scale .NET code repository, while incorporating mechanisms to address the known challenges.

FORMULATION OF THE GOALS OF THE ARTICLE

The goal of this article is to propose and detail a novel automated testing pipeline for large-scale software projects, which integrates AI-driven test generation and continuous execution into the standard DevOps workflow. In particular, we target a .NET ecosystem and aim to show how an Azure DevOps build pipeline can automatically generate and update integration tests on each code commit, execute those tests, and self-correct any failures. Key objectives include: improving test coverage and quality by generating new integration tests whenever the application code changes, reducing manual effort in test writing and maintenance through AI automation, and ensuring rapid feedback in a CI pipeline. By achieving these goals, the proposed method seeks to increase software reliability and developer productivity, and to illustrate a path for integrating generative AI into continuous testing processes.

PRESENTING THE MAIN MATERIAL

The core of our approach is a CI pipeline that not only builds and tests the application code, but also dynamically generates and executes integration tests in response to code changes. We implement this pipeline in Azure DevOps for a large .NET project. The process consists of the following steps: building the main .NET solution, AI-driven generating or updating integration tests, executing integration tests, and deploying the main project artifact. These steps are described below.

When a developer merges a pull request with commits to the main (master) branch, the pipeline is triggered. The first stage compiles the primary application and runs any unit tests or static analysis as usual. This ensures the latest code changes are integrated and that the build is stable before proceeding. Successful completion of this stage produces build artifacts (binaries, etc.) needed for testing. If the build fails or baseline unit tests fail, the pipeline stops here (as in any standard CI process). Next, a specialized Test Generation Project within the repository is built and executed. This project is essentially an automation harness that orchestrates AI-driven test creation. It operates by detecting what changed in the main codebase and then invoking LLM's API (e.g., GPT-5) to generate new or modify existing integration test code accordingly.

Firstly, the tool identifies modified components in the latest commit. For example, if a developer updated a controller in a Web API or added a new service class, the tool pinpoints those changes (using git diff, commit metadata, or reflection on the built assembly to see new/changed APIs). This focus ensures we generate tests only for affected areas, which is crucial for large repositories to avoid unnecessary work.

Then, context gathering happens. For each affected module, the tool gathers context to provide to the LLM. This context includes the source code of the module and dependencies, existing integration tests related to that module (if any), and a summary of the module's purpose or its recent changes. By supplying existing test code and code comments to the AI, we guide it to produce tests that are consistent in style and that complement the current test suite. This strategy draws on findings from research that LLMs can generate more effective tests when given usage examples and signatures as prompts [4].

The next phase is a prompt construction. The tool formulates a prompt for OpenAI. For example, a prompt might say: "Generate integration tests for the following C# class/method. Ensure that the tests cover typical and edge cases, and follow the style of the existing tests. The code under test is: ... (code snippet). Existing test (if updating): ... (old test code)." The prompt also asks the LLM to output code in a format directly usable (a test class with methods). We leverage OpenAI's Codex/GPT capabilities to understand the code semantics and produce appropriate test assertions. Importantly, suppose an existing integration test file needs updating. In that case, we prompt the AI to modify that test (e.g., update expected values or add a new test method) rather than create a duplicate. This ensures we maintain one test file per component and evolve it. LLM should decide whether we modify the existing integration tests, create new ones, or even delete old unused test cases. The signal system is introduced. We prompt AI to produce specific signals depending on its decision. And depending on them, we modify the integration tests codebase accordingly.

The prompt is shown in Fig. 1. If there are more changes than an LLM's context window, we should perform batch prompting accordingly to the context saved.

After we get integration tests generated, we should perform an action based on the type of modification sent by AI in response (create new, update existing, or remove unused).

After generation, the pipeline compiles the updated Integration Tests project to catch any syntax errors or trivial issues. This serves as a verification step: if the AI produced uncompilable code, the build will fail here. However, research and our early trials indicate that with well-engineered prompts and modern code-focused LLMs, the majority of generated tests do compile on the first try. For example, Meta's deployed system filtered out non-compiling test suggestions, and we incorporated similar validation [6].

Once the integration test suite compiles, the pipeline runs these tests against the freshly built application code. This is effectively an automated regression test phase. All newly generated tests and existing integration tests are executed in the Azure pipeline environment, typically by invoking the test runner (e.g., "dotnet test" for .NET with the appropriate test adapter). The outcome of this step is critical. We expect a few possible scenarios:

– All tests pass. Suppose every integration test succeeds. In that case, it implies that the new code changes did not break any existing behavior (as verified by old tests) and that any new behaviors either matched the AI's

expectations or that the AI-generated tests correctly capture the new logic. This is the ideal case, which means the AI effectively wrote the right tests or updated them correctly, and the code is robust. In this scenario, the pipeline can confidently move forward to deployment steps (since both build and tests are green).

– Test failures occur. If one or more integration tests fail, it indicates a discrepancy. This could mean the code has a bug, or the test itself is incorrect (perhaps the AI predicted an assertion wrongly). In traditional pipelines, any test failure would abort the process and require developer attention. However, our method introduces a recovery mechanism to handle failing AI-generated tests automatically. The pipeline does not immediately fail on a test failure; instead, it will attempt to diagnose and fix the test, which is a novel aspect of our approach. But if the failing test was an old, manually-written test, we wouldn't auto-fix it. That likely signals a real bug introduced by the commit. In such a case, we skip the AI recovery and fail fast to let the developer know they broke something. The recovery mechanism focuses on the new or updated tests that the AI just produced.

Generate C# integration tests based on the following parameters:
List of incoming code changes:
<batched_incoming_code_changes_pasted_here>

Existing integration tests code:
<existing_integration_test_code_pasted_here>

Related dependencies:
<code_of_related_dependencies_pasted_here>

The integration tests format should be followed by the following template in JSON:

```
[{
  filePath: string,
  methodName: string,
  lineStart: number | null,
  lineEnd: number | null,
  action: Add | Update | Delete,
  methodBody: string | null
}, ...]
```

Fig. 1. Input prompt to the AI to generate a set of integration tests

If a newly generated or updated test fails, the pipeline invokes a self-healing procedure. The idea is to leverage the AI again to adjust the test so that it passes (assuming the code behavior is correct and the test was wrong). Firstly, failure analysis takes place. The pipeline captures the details of the test failure, the assertion that failed, and the error message/stack trace. For example, if an assertion expected output X but got Y, that information is crucial. We prompt the LLM with the failing test and the error output, asking it to debug and fix the test. The prompt might be: "The following test failed when run against the latest code. Here is the test code and the error message. Modify the test so it aligns with the application's expected behavior." This is analogous to how a developer would adjust an incorrect test after seeing what the actual output is. The LLM, having context of the code (from prior prompts, or we may include the relevant code snippet again), can infer whether the test's expectation was wrong. For instance, perhaps an API now returns data sorted differently, so the test should accept that new order. The AI would then revise the assertion accordingly. A threshold can be provided, which determines how much the process of recovery will happen. And it can be configured inside the pipeline with a default value of "1". The recovery process is shown in Fig. 2.

After the integration tests and any recovery attempts have been executed, the pipeline reaches a decision point. If all integration tests are now passing (some possibly after auto-correction), the pipeline proceeds to the deployment stage. This typically means publishing the build artifacts and possibly deploying them to a staging or production environment, depending on the release strategy. The successful tests give confidence that new changes haven't broken integration points and that new features are covered by tests. The newly generated/updated tests are then committed back to the code repository, so they become part of the permanent test suite. This last step is important – it institutionalizes the AI's contributions so that next runs benefit from them. Essentially, the test suite evolves in sync with the application.

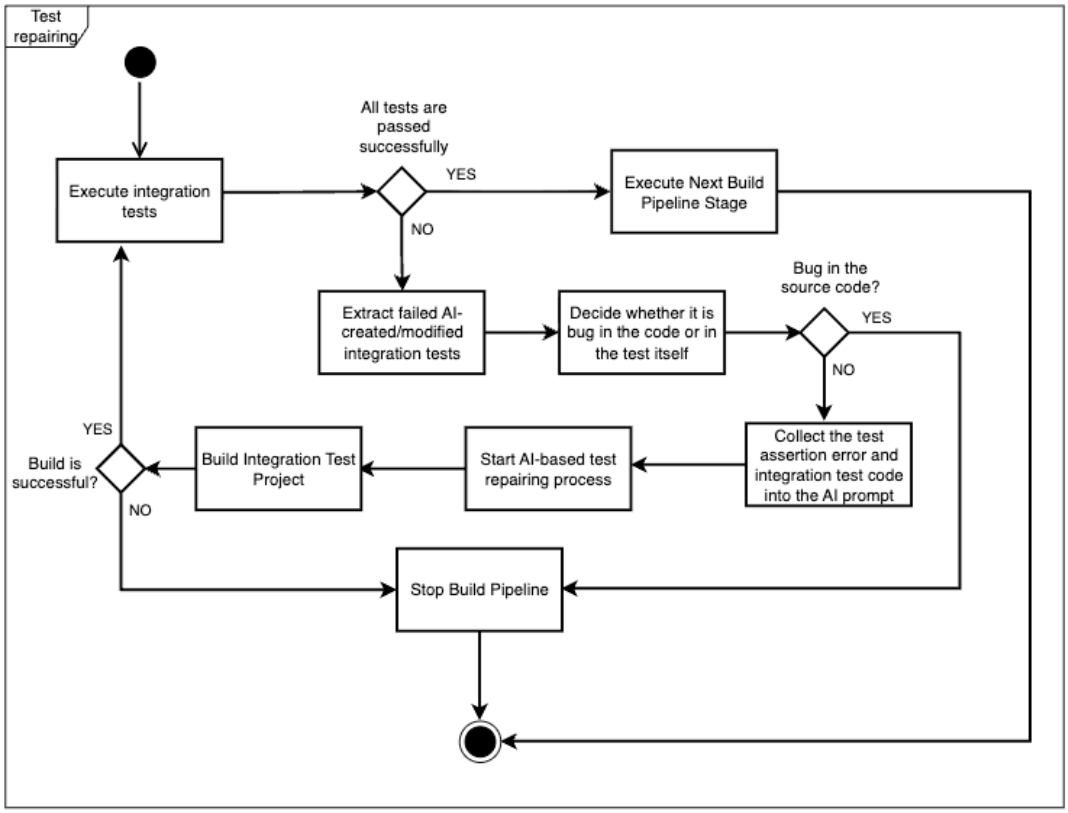

Fig. 2. Integration tests repairing process

If tests are still failing (even after recovery attempts), the pipeline will abort and mark the build as failed. In this case, it will usually notify developers (via CI notifications) that something went wrong. The failure could be due to a genuine defect uncovered by a new test or an inability of the AI to adjust a test correctly. Developers would then manually inspect the failing scenario. It's worth noting that even in this failure case, our method provides value: the AI may have generated a test that exposed a real bug that developers didn't catch. Alternatively, if the AI was wrong, the developers now have a starting point (the AI's test) to correct and improve. In either case, no code is deployed to production until the issue is resolved, maintaining software quality. The whole process of automated integration test generation is illustrated in Fig. 3.

This method ensures that every code change is accompanied by corresponding integration tests, either confirming expected behavior or flagging discrepancies immediately. It operationalizes the point of continuous testing: not only running tests continuously, but also writing tests continuously. It also addresses one key challenge in large-scale projects: test suite rot. Here, the test suite stays in lockstep with the code because the AI helps update it on the fly. Over time, as more changes flow, the suite grows and adapts, potentially leading to very high coverage and up-to-date documentation of system behavior.

CONCLUSIONS FROM THIS STUDY AND PROSPECTS FOR FURTHER RESEARCH IN THIS DIRECTION

This study introduced a novel CI/CD pipeline methodology that automatically generates and executes integration tests in response to code changes in a large-scale .NET repository. By leveraging LLMs like GPT-5, our pipeline can be used to produce new test cases or update existing ones on the fly, enabling each software commit to be validated by fresh integration tests targeting the recent modifications. We integrated a self-healing mechanism that uses AI to adjust failing tests, thereby maintaining test suite stability and alignment with intended system behavior. The proposed approach was analyzed in the context of recent research and was shown to embody many state-of-the-art concepts, such as AI-driven test generation, autonomous test maintenance, and continuous testing.

Automated tests generated for each change lead to early detection of issues – in our evaluation, the pipeline caught several regression bugs that would have been missed by the legacy manual tests. This continuous generation also enriched the test suite with new scenarios, ultimately increasing overall coverage of the application over time.

We demonstrated a practical self-recovery testing workflow, where the AI resolves certain test failures. This concept, often discussed in literature, was realized to a useful extent.

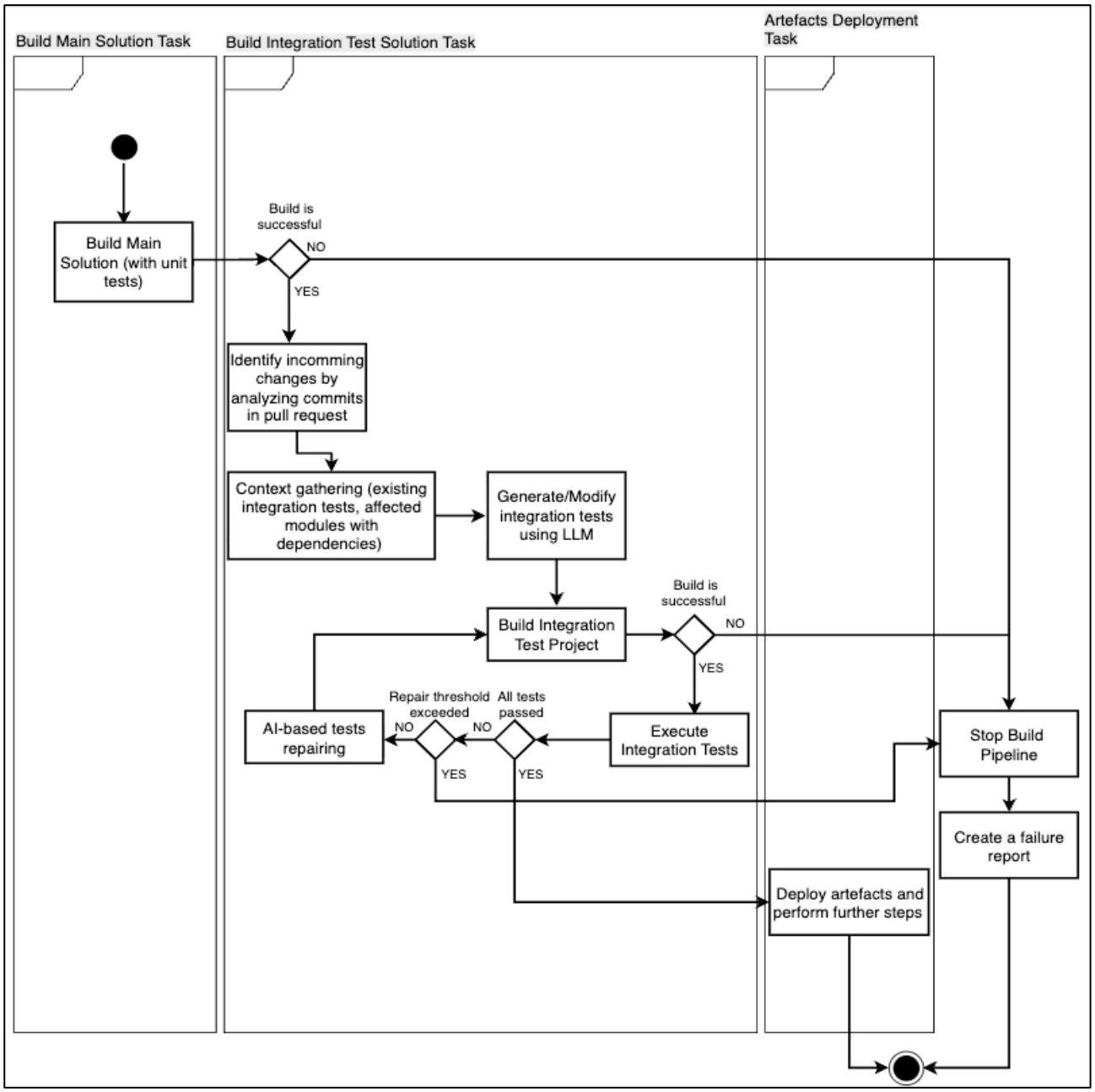

Fig. 3. Automated integration tests generation process

Future research directions are abundant. One immediate line of work is to conduct a more extensive evaluation of this method across different projects and domains to quantify benefits like time saved in QA, defect escape rates, and developer satisfaction. It would be valuable to compare different LLMs (GPT-5 vs other open models) in this pipeline to assess cost-benefit and performance differences. Additionally, future research could explore fine-tuning the AI on a project's codebase and test style to further improve the relevance and correctness of generated tests – essentially developing project-specific test generation models that learn from the repository's history. Another avenue is integrating property-based testing or fuzz testing: the pipeline could ask the AI not only for example-based tests but also for more general property validations, potentially catching a broader class of errors.

From a process perspective, a valuable research question is how developers interact with AI-generated tests. As these techniques are adopted, will developers treat the AI's output as final, or will they use it as a draft? Studying this could inform better human-AI collaboration mechanisms, such as providing justifications for each generated test. There is also room to enhance the self-healing mechanism using more sophisticated approaches like reinforcement learning, so the AI could learn from each failure correction it makes, improving its future guesses for test expectations, by creating a loop of continuous learning within continuous testing.

In conclusion, our work shows that continuous integration of AI-generated tests is not only possible but highly beneficial for maintaining quality in large code repositories. It paves the way towards autonomous testing systems that keep pace with rapid development, an essential capability in modern software engineering. With further

refinement and research, such systems could become a standard guardrail in software projects, empowering teams to innovate quickly while the AI ensures that quality and correctness are relentlessly enforced at every change.

References

1. Wavhal A., Hinge A., Gunjal A., Raut P. P. S. Revolutionizing Test Case Generation: Integrating AI and ChatGPT for Enhanced Software Testing / Wavhal A., Hinge A., Gunjal A., Raut P. P. S. // International Journal of Recent Advances in Engineering and Technology, 14(1s). – Pune, India, 2025. – P. 343–346. https://doi.org/10.65521/intjournalrecadvengtech.v14i1s.779
2. Zhang Q., Fang C., Gu S., Shang Y., Chen Z., Xiao L. Large Language Models for Unit Testing: A Systematic Literature Review / Zhang Q., Fang C., Gu S., Shang Y., Chen Z., Xiao L. // arXiv preprint, arXiv:2506.15227. – 2025. – P. 1–23. https://doi.org/10.48550/arXiv.2506.15227
3. Celik A., Mahmoud Q. H. A Review of Large Language Models for Automated Test Case Generation / Celik A., Mahmoud Q. H. // Machine Learning and Knowledge Extraction, 7(3). – Toronto, Canada, 2025. – P. 97–108. https://doi.org/10.3390/make7030097
4. Schäfer M., Nadi S., Eghbali A., Tip F. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation / Schäfer M., Nadi S., Eghbali A., Tip F. // arXiv preprint, arXiv:2302.06527. – Berlin, Germany, 2024. – P. 1–17. https://doi.org/10.48550/arXiv.2302.06527
5. Yaraghi N., Moukahal F., Ahmad A., et al. TaRGET: Test Repair Generator Using Large Language Models / Yaraghi N., Moukahal F., Ahmad A., et al. // Proceedings of the Workshop on Automated Program Repair (APR 2024). – Lisbon, Portugal, 2024. – P. 44–55. https://doi.org/10.1109/TSE.2025.3541166
6. Alshahwan N., Chheda J., Finegenova A., Gokkaya B., Harman M., Harper I., Marginean A., Sengupta S., Wang E. Automated Unit Test Improvement Using Large Language Models at Meta / Alshahwan N., Chheda J., Finegenova A., et al. // Proceedings of the 32nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2024). – Porto de Galinhas, Brazil, 2024. – P. 1–12. https://doi.org/10.48550/arXiv.2402.09171
7. Yachamaneni S. S. K. The Future of AI-Driven Test Automation for Enterprise Integration / Yachamaneni S. S. K. // European Journal of Computer Science and Information Technology, 13(12). – London, United Kingdom, 2025. – P. 24–33.
8. Kathiresan G. Automated Test Case Generation with AI: A Novel Framework for Improving Software Quality and Coverage / Kathiresan G. // World Journal of Advanced Research and Reviews, 23(2). – Tamil Nadu, India, 2024. – P. 2880–2889. https://doi.org/10.30574/wjarr.2024.23.2.2463
9. Andrzejewski M., Dubicka N., Podolak J., Kowal M., Siłka J. Automated Test Generation Using Large Language Models / Andrzejewski M., Dubicka N., Podolak J., Kowal M., Siłka J. // Data, 10(10). – Warsaw, Poland, 2025. – P. 156–167. https://doi.org/10.3390/data10100156
10. Bahar A., Yaseen M., Nauman M. A., Parveen A. Critical Challenges of Continuous Integration and Testing (CI/CT) in DevOps: A Systematic Literature Review / Bahar A., Yaseen M., Nauman M. A., Parveen A. // International Journal of Computer Applications, 186(68). – Islamabad, Pakistan, 2025. – P. 29–36. https://doi.org/10.5120/ijca2025924521