AI Tools introduced in Software Development. Analysis of Code quality, Security and Productivity Implications

Ana-Maria Dincă
Faculty of Electronics, Telecommunications and Information Technology, University POLITEHNICA of Bucharest
Bucharest, Romania
ana_maria.dinca@stud.etti.upb.ro

Sabina-Daniela Axinte
Faculty of Electronics, Telecommunications and Information Technology, University POLITEHNICA of Bucharest
Bucharest, Romania
axinte_sabina@yahoo.com

Gabriela Tod-Raileanu
Faculty of Electronics, Telecommunications and Information Technology, University POLITEHNICA of Bucharest
Bucharest, Romania
gabriela.tod@stud.etti.upb.ro

Ioan C. Bacivarov
Faculty of Electronics, Telecommunications and Information Technology, University POLITEHNICA of Bucharest
Bucharest, Romania
ibacivarov@yahoo.com

2024 IEEE 30th International Symposium for Design and Technology in Electronic Packaging (SIITME) | 979-8-3315-3951-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/SIITME63973.2024.10814830

***Abstract*—Chatbots have recently been integrated into a diverse range of applications, from office tools to code compilers. It is undeniable that AI assistants possess features that simplify and automate repetitive tasks, however, it is yet unclear how well they can handle creative tasks such as writing code. This paper presents an in-depth analysis of the quality of the solutions generated by AI tools, that have been specifically trained for software development tasks, including code writing, testing, reviewing and documenting features. Individual use cases are rigorously examined, while comparing the results for two similar intelligent assistants per each scenario, to identify the strengths and weaknesses of their proposed solutions. The impact on productivity is studied, as it is dependent on the quality of the generated responses. The implementation of an incomplete, outdated or insecure solution will impose additional fixes, and, if the majority of answers have to be revised, this counterproductive behavior will actually increase the development time rather than facilitate its reduction.**

Keywords—AI-assisted Programming, AI Code Safety, code quality, development efficiency

I. INTRODUCTION

The software development domain is constantly seeking the latest tools that can shorten the process of implementing new features, while simultaneously maintaining the standards of code quality and security posture. OpenAI released the demo for ChatGPT 3.5 in late 2022 and its popularity skyrocketed, attracting well over one million users in the first five days [1]. Since then, the market has been flooded with new AI chatbots and existing companies have been pushed to integrate and provide an AI-based assistant for their users.

This paper proposes an analysis of a series AI tools that can be utilized for a range of tasks commonly encountered in software development, including code writing, review and testing. The analysis consists of comparative assessments of two tools for each use case, reviewing the features they offer, the prices and to the actual hands-on user experience.

The first comparison is between two AI tools specialized in code writing, Claude 3 Opus and GitHub Copilot. The same prompts are entered into the chatbots and the results are implemented, analyzed and compared in order to evaluate the performance of the intelligent assistants, and generate a graph displaying the strengths and weaknesses of each AI tool.

The second comparison is performed for two different use cases of test writing: the Test-Driven Development methodology and the unit test writing. The first use case compares once again Claude and Copilot, while the second one compares Diffblue and Sapient.

The final use case is the code review, which compares Snyk and Tab Nine. The results are then weighted against those obtained using a prominent and a standard tool in this field, namely SonarQube.

The paper proceeds to examine the security concerns associated with the use of AI tools in software development from various perspectives, including the protection of the confidential data and the dangers imposed by an inexperienced user.

II. ANALYSIS OF AI TOOLS TRAINED FOR SOFTWARE DEVELOPMENT

Chatbots have been integrated into a diverse suite of applications, ranging from office tools to code compilers and creation platforms. Examples of such software products include Adobe Acrobat Reader, Microsoft Office 365, Google Workspace, Adobe Photoshop, and Intellij IDEA. It is anticipated that the list of software products that include an AI assistant will increase, due to the high public demand and the undeniable desire to increase productivity and cost effectiveness. A survey conducted by GitHub on the impact of GitHub Copilot on productivity revealed a 55% decrease in the total time spent completing a development task [2].

The software development field is on a relentless pursuit of the latest technological advancements in order to improve the quality of its deliverables. Therefore, it is quite common to use software tools that significantly reduce the time spent on writing implementations. These products have various applications, such as code auto-completion, or shortcuts that transform keywords into complete lines of code (e.g. writing "*sout*" and pressing the Tab key automatically writes "*System.out.println("")*", a command that prints text to the console). Other instruments perform static analysis of code fragments and report on potential issues that might arise from specific implementations, including an out-of-date library, or a dependency that contains a publicly disclosed vulnerability, or even lines of code that may be nesting errors. Once the intelligent assistants were launched, they encompassed all these previously mentioned features in one very easily accessible tool. Thus, people working in this industry were some of the earliest adopters of the new AI tools, as the generative text and the search result aggregation features simplified their daily responsibilities [3]. During those early days, there were incidents where developers inputted private source code into Chat GPT while searching for a bug fix or to improve algorithm performance. This resulted in the model being trained on the code, and to possibly leak the confidential information to the general public in future responses [4].

As the number of users of AI-based tools increased, it became evident that training a learning model to become an all-in-one assistant would require substantial quantity of input data and extensive system resources, which would be financially burdensome. This has resulted in the release of AI chatbots trained to provide support on specific subjects, such as software development, photo-video editing, creative generation, travel planning, budgeting. In certain fields, the AI models have begun to replace a significant percentage of the workforce, customer support jobs being the most prominent example. As a result, it is becoming increasingly difficult to reach a live agent, given that the majority of customer service organizations have adopted AI chatbots, with an additional percentage planning to transition in the future.

The AI tools that have been specifically trained to facilitate software development possess a range of capabilities, including code autocompletion, generation, refactoring, review, documentation, bug detection and resolution, or test generation. Chatbot functionalities vary considerably, some incorporate a greater number of features, while others excel in very specific tasks.

Three of the most common tasks in software development have been selected for this research: code generation, review and testing. For each use case, a comparison will be conducted between two AI tools, that have been designed and trained specifically for that objective. They will be given the same input data and the same scenarios, and their performance will be evaluated based on the final result, how difficult they were to use, and whether they helped reduce the time it would have taken to complete the task. *Claude 3 Opus* and *GitHub Copilot* will be used to compare the code generation feature. A new feature will be implemented using the Test-Driven Development (TDD) methodology, and afterwards Unit tests will be written using the *DiffBlue* and *Sapient* frameworks. The final case study will be the code review, which will be performed using *Snyk* (a tool powered by DeepCode AI) and *Tab Nine*. The results will also be compared with those of *SonarQube*, a traditional static code analyzer.

III. Case study – Code generation – Claude 3 Opus vs. GitHub Copilot

Claude 3 Opus is the most performant and most expensive AI instrument in the Claude 3 model family, which is the latest iteration developed by *Anthropic* and released on 4th of March, 2024 [5]. It was chosen for the code generation test case because it has the highest score on AI evaluation benchmarks such as Code and Math problem-solving, characteristics that are essential for writing quality code.

GitHub Copilot is an intelligent code-writing assistant powered by *Codex*, an AI model built through a partnership between *OpenAI* and *GitHub*, and trained specifically for software development. It is capable of translating natural language commands into code, providing real-time guidance, and, through to its integration with various IDEs, it can understand the context in which it is being used to assist in implementing new features and debugging existing code.

TABLE II. Benchmark results of Claude vs GPT [6]

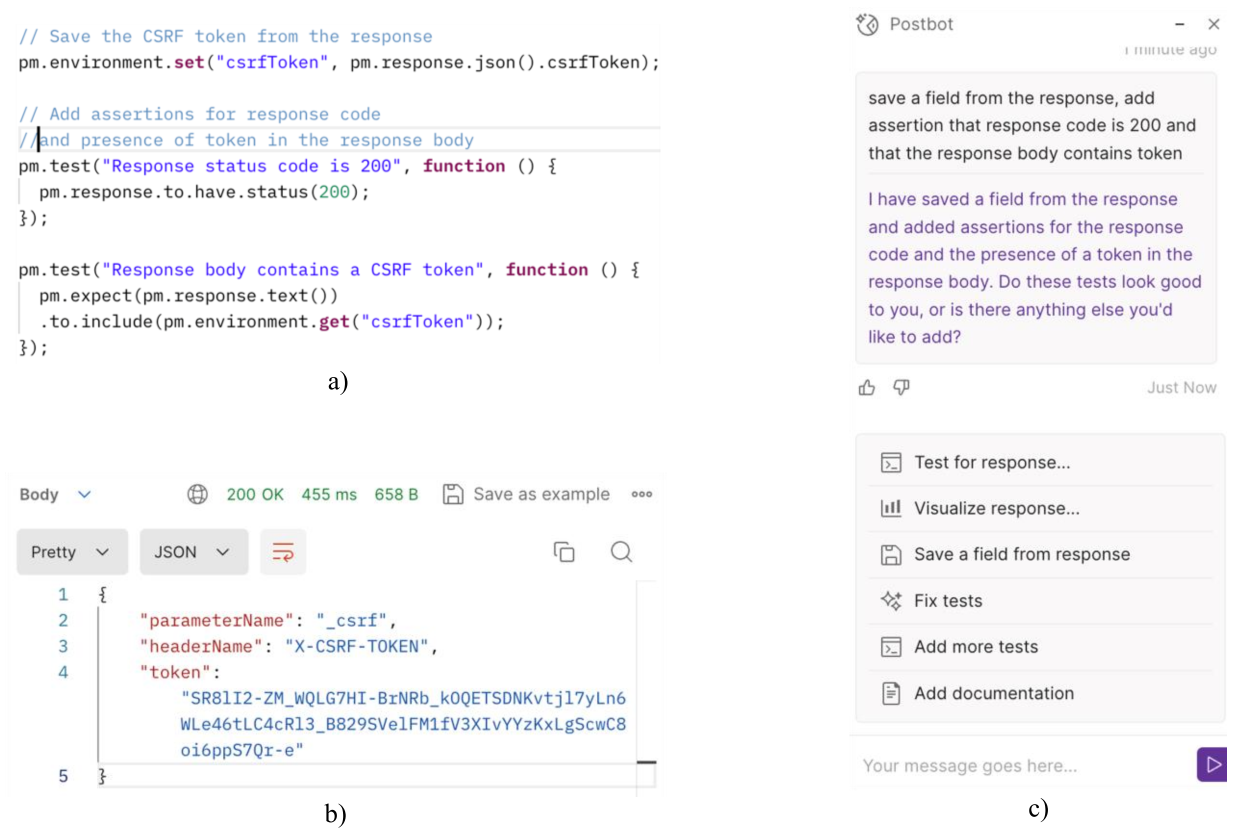
Benchmark	**AI Tool**				
	Claude 3 Opus	***Claude 3 Sonnet***	***Claude 3 Haiku***	***GPT 4***	***GPT 3.5***
Math problem-solving *MATH*	60.1% 0-shot CoT	43.1% 0-shot CoT	38.9% 0-shot CoT	52.9% 4-shot	34.1% 4-shot
Code *HumanEval*	84.9% 0-shot	73.0% 0-shot	75.9% 0-shot	67.0% 0-shot	48.1% 0-shot

In this case study, Claude 3 Opus and GitHub Copilot were chosen based on their promising results obtained on benchmark tests relevant for coding abilities, presented in Table I. They were used to write a new Spring Boot application from scratch, starting from the exact same input prompts, and the final results were compared against each other, evaluated in accordance with the coding best practices, and assessed in terms of the overall user experience.

The first prompt consisted of the instructions to create a new Spring Boot application, along with a couple of APIs that could be used in order to create, retrieve, update and delete PDF files (CRUD operations). The prompt introduced was "*Write me a Spring Boot application using Java 11 that manages PDF files and implements all CRUD operations*".

The second feature to be implemented was to create a new API that would send a specific document to an institution, both elements identified in the backend by id, on behalf of the authenticated customer making the REST call. The prompt sent to both chatbots was longer, in order to provide more details for the implementation: "*I want a new API for this Spring Boot application (Spring version 3.3.0-RC1), this API should send a file identified by an id to a public institution, identified also by an id. When called, the application will compose an email with the subject "Document", adding the name of the file, "sent on behalf of", and the name of the user who sent the request. The user should be identified by the cookie or the session. The table that identifies the institution should have at least one column for unique identification, one column for the name of the institution, one column for the email address*".

The implementations were tested using Postman, a modern API testing tool that allows automatic script validation before or after a request. Therefore, for each REST call, assertions were set up to check that the response code and body were the expected ones, and to store information that could be used in subsequent calls in environment variables, such as generated tokens, or file ids generated after save requests, that could be used in specific file retrieval operations. The entire set of API calls, each with its own assertions, was moved into a runnable folder, creating a semi-automated test scenario, where a CSRF token was requested from the server, then a new file was saved to the server through a POST request, the file was retrieved by executing a GET call, then the document was deleted and a final retrieve



```
// Save the CSRF token from the response
pm.environment.set("csrfToken", pm.response.json().csrfToken);

// Add assertions for response code
//and presence of token in the response body
pm.test("Response status code is 200", function () {
  pm.response.to.have.status(200);
});

pm.test("Response body contains a CSRF token", function () {
  pm.expect(pm.response.text())
  .to.include(pm.environment.get("csrfToken"));
});
```

a)

```
{
    "parameterName": "_csrf",
    "headerName": "X-CSRF-TOKEN",
    "token":
        "SR81I2-ZM_WQLG7HI-BrNRb_kOQETSDNKvtjl7yLn6WLe46tLC4cR13_B829SVelFM1fV3XIvYYzKxLgScwC8oi6ppS7Qr-e"
}
```

b)

save a field from the response, add assertion that response code is 200 and that the response body contains token

I have saved a field from the response and added assertions for the response code and the presence of a token in the response body. Do these tests look good to you, or is there anything else you'd like to add?

c)

Fig. 1. Postman AI assistant

request was made, in order to check that the file was no longer on the server, returning a file not found response. All of these post-request scripts were effortlessly written using Postman's AI companion, *Postbot*, shown in Fig. 1, which is able to create scripts based on the information provided in the chat box, facilitating the automation process, especially for people unfamiliar with the specific syntax used in Postman scripts.

A. Claude 3 Opus

During this research, Claude 3 Opus was accessed via TextCortex, therefore each response had to be manually moved from the chatbot interaction window to the IDE. The response to the first prompt was very robust, as it included the required project dependencies in the *pom.xml* file, the necessary configuration properties for the *application.properties* file, and generated separate *Java classes* for the model entity to be persisted to the database, the repository service, the processing service, the controller and the Spring Boot Application class. The response ended with the API paths defined by the generated implementation that should be accessible to a user once the application is deployed.

The generated code was manually copied into an IDE, and the JVM was able to compile it successfully on the first attempt. The application was then launched on the local environment in order to test the functionalities implemented with Claude 3 Opus. The Spring Boot Starter version used in the dependency configuration file was the latest available (3.3.0-RC1), which by default adds a layer of security that blocks all REST calls that use unknown identity tokens or do not include any authentication headers whatsoever. The application logs contained an auto-generated security token, that could be used in order to access the exposed APIs in the absence of a separate mechanism to generate unique keys for user authentication.

The implementation consisted of four operations: retrieving a file from the server, adding a new file, editing an existing file and deleting a file. When testing the new functionalities using Postman, which authenticates the REST calls using the auto-generated authentication token, it was revealed that the only operation that could reach the server was the GET operation. All other operations were rejected by the application due to the Cross-Site Request Forgery (CSRF) filter, that is automatically enabled in Spring Boot Starter since version 3.2.0. The problem was solved by asking for a solution to the CSRF filter that was blocking the requests. The first solution provided was to disable the filter, which would have been counterproductive from a security perspective. Eventually, Claude 3 Opus generated a SecurityConfiguration file and a new API that would generate a CSRF token, that could be used in subsequent POST, PUT and DELETE calls.

When the second prompt was introduced, Claude 3 Opus suggested a simple but efficient implementation by generating another controller file for the new REST call and the FileShareService class. At first glance, it was obvious that the new code would not compile when pasted into the IDE, because one of the newly generated components used EmailService, a file that was not present in the proposed solution or in the existing code. Therefore, when an implementation for this service was requested, the chatbot provided the necessary code, along with a new dependency configuration file, as new libraries were required. A new application configuration file was also generated, in order to customize the mechanism responsible for sending emails.

The email service implementation used Gmail credentials, although no email service provider preference was specified, and when used, this solution proven to be obsolete for two years, as Google revoked access to applications that did not use OAuth for authentication [7]. When the error message was presented to Claude, it revised the solution, elaborating on the additional steps required in order to generate a unique passkey to authenticate the application to the SMTP server. Once the changes were made, and the final result was validated against the expected behaviour, which it successfully passed, as the automatically sent email was received with the correct subject, body and file attachment.

Implementing a new application from scratch can be time consuming, requiring decisions on technologies to be used, security configurations, feature design, project structure and code implementation. Using Claude 3 Opus significantly reduced the time consumed by design and code implementation, as it created a functional project that could pass the expected behavioral test scenario, but the resulting application would not be usable in a real environment. The generated code was a good starting point, that was easily modified, either by asking the chatbot additional questions and configurations, or by using the developer's experience.

The first remark is that a longer prompt automatically results in a better response, but this implies a higher level of technical understanding of programming, specific to a developer, and an investment of time in the design of the application. Although the prompt specifically asked for an application that manages PDF files, Claude 3 Opus proposed an implementation using *H2* and the *Java Persistence API (JPA)* for database communication, even though *H2* is a *relational database*. The proposed implementation could be used with a *NoSQL* data source, but there would be clear performance and functionality limitations. Also, the database model was chosen "*for simplicity*", as the chatbot replied when asked about this choice of architecture.

There was no suggestion for the project structure, even though Claude had generated all the files needed to implement the desired functionalities in the new application. The directory organization was requested in the chat box, but the solution provided was obsolete, citing an old practice of separating the classes by type (e.g. Controller classes in a controllers directory, etc.), instead of proposing a solution closer to today's best practices in software development of grouping the classes by the functionality they are used in (e.g. a directory containing the controller, services, repositories and models used to implement CRUD operations for PDF files).

The naming conventions used by the AI assistant were short and simple, clearly describing the functions of the software components in the application. The API verbs proposed for managing PDF files followed the naming conventions, but the verb proposed for the file sharing feature with an institution did not. The proposed endpoint was "*/api/share/{fileId}/{institutionId}*", and it appended two ids, and it would be confusing for a user to see a REST request towards "*/api/share/52/6*", as there would be no representation of what each id represented. Thus, the API was modified to include the entity id in the URL, transforming to "*/api/share/file/{fileId}/institution/{institutionId}*".

It was clear that the code implementation generated by Claude was not thought "*security first*", as the security configuration file had to be specifically requested, and all the safety features protecting the application were enforced by default by the Spring Boot Parent version chosen prior to inputting the first prompt in the chatbot.

Claude 3 Opus is not currently integrated with any IDEs, thus it does not have access to the development context, but it compensates with its advanced conversation recollection. Although there was the inconvenience of moving the implementations and the errors back and forth from the console to the chat box, the AI compensated with its ability to debug the code and find the root cause of the issues. It took into account recommendations from the user that would help to speed up the resolution, such as suggesting that a problem could originate from a particular library. Another suggestion was that the security configuration should generate CRFS tokens that could be used by users, rather than disabling the CRFS protection altogether. There were instances where the implementation omitted a file mentioned in the code, but when asked for the missing component, it generated a fully functional class that fit well within the project.

Claude had a strong understanding of coding standards and generally came up with good and complete solutions to the problems he was presented with. The tool was user-friendly, and as long as it is used by an experienced developer, it can be a valuable tool for improving productivity and code quality.

B. GitHub Copilot

GitHub Copilot is a versatile tool that can be used from various locations, including the terminal, the GitHub mobile app, and directly from the IDE, as it can be integrated with some of the most popular code editors, such as Visual Studio, Visual Studio Code, all JetBrains IDEs, Vim, Neovim, and Azure Data Studio. For this research, the Copilot plugin was installed and configured in IntelliJ, providing convenient access to the AI, while also ensuring its real-time access to the development context.

The code generated by GitHub Copilot as a response to the first prompt differed slightly from the one produced by Claude. It provided the basic the configuration files, with the exception of the pom.xml file, which contained the library dependencies required by the project. Other than this omission, the code implementation resembled the one generated by Claude, with identical Controllers, Services, and database Entities. Notable differences were observed in the Java class names and API endpoint definitions. The data source configuration was not compatible with its intended use case, because a software application handling file objects should utilize *NoSQL* models. Copilot suggested the use of a *relational H2 in-memory database*, with entities linked to the database schema using JPA. This persistence context configuration is identical to the one generated by Claude.

The integration of GitHub Copilot as a plugin in the IDE used for development facilitated the seamless insertion of the generated responses into the project. However, when the code was compiled, there were thrown errors that prevented the application from starting, because the Spring context was missing multiple classes from imported libraries. Copilot has access to the console errors thrown during the code execution, thus a solution was simply requested in the interactive chat.

The first proposed solution was to downgrade the Spring Boot Parent version from 3.3.0-RC1, the most recent release version, to 3.2.0, the latest stable version. However, based on accumulated experience, it was determined that the libraries could not be absent due to the use of a different iteration of the same major Spring release (3.x.x). When this observation was presented to the conversation with the AI, it rejected the suggestion altogether. As the problem remained unsolved, an alternative solution was explored. The error message and the Spring Boot Parent version were pasted in the Claude 3 Opus chat box in order to request assistance. The AI indicated that the library used in the implementations was deprecated, and another dependency was used starting with Spring Boot versions 3.0.0+. The solution was to replace the "*javax.**" library in the implementation, with "*jakarta.**". This fixed the code and demonstrated that downgrading the versions in the same major Spring release was not a viable solution. This insight has already been deduced through the perspective of an experienced software engineer, which is not possessed by a junior developer, who may become stuck in a repetitive cycle of incorrect suggestions from Copilot.

The APIs were tested using the previously generated Postman test suites, and due to the 3.x.x Spring Boot version, the application was again protected by default against CSRF calls. Therefore, a security configuration file was requested, to generate the CSRF tokens needed to allow users to send PDF files for persistence. Copilot provided the requested Java classes and configurations, yet the code did not compile. Upon requesting a solution, the AI returned an error message indicating that the selected parent version was "*not yet released and it's not available in the Spring Milestones repository*". Additionally, it incorrectly stated that "*The latest stable version of Spring Boot at the time of writing is 2.6.1*". Further verifications revealed that the most recent version released was 3.2.5, therefore GitHub Copilot was operating on outdated information, a limitation not exclusive to the AI built on top of OpenAI's Codex. Anthropic's Claude also indicated that the latest stable release version of Spring was 3.0.6. This situation highlights the importance of verifying the responses received from AI assistants, and how an inexperienced user might blindly follow the proposed suggestions, resulting in erroneous implementations and an increase in the development time. After implementing the CSRF token generation functionality, when trying to save a PDF file, the application returned an HTTP error code 415, for *unsupported media type*. The issue was debugged and resolved through the built-in chat, which was able to interpret the console errors.

The second prompt was introduced in the chat box and Copilot generated all the required Java classes for the implementation of the email sending feature. These included the API definition, the new Controller, the email service, and the additional configurations. A previous problem appeared yet again when the library imports were generated with the *javax* library instead of *jakarta*. A solution was previously proposed to Copilot, but it was unsuccessful in training the model and preventing it from repeating its mistakes. This was manually fixed, the implementation was tested, and it was determined that it functioned correctly. The implementation utilized a better naming convention than Claude, with the endpoint designed to illustrate the function of the ids: "*/send/{fileId}/to/institution/{institutionId}*". The principle of functionality segregation was not respected in the code, as the endpoint was added in the same Controller file as the CRUD operations. This approach does not align with the clean code principles, which are essential in order to build reliable, scalable and efficient functionalities.

Using GitHub Copilot to build an application from scratch reduced the time typically required for the manual development of software components, including Controllers and Services. However, a considerable amount of time was spent on debugging issues introduced due to faulty implementation and disregarding user suggestions. Copilot did not demonstrate the capacity to learn from past mistakes, repeating the same errors within a couple of hours. Consequently, is not recommended for inexperienced users who would blindly trust its recommendations.

TABLE II. EVALUATION METRICS OF CLAUDE AND COPILOT

Metrics		Claude 3 Opus	GitHub Copilot
Accessibility	Ease of use	✓	✓
	Integration with IDE	✗	✓
	High Availability	✓	✓
	Total score	2	3
Feature Implementation	Data source fit for application	✗	✗
	Provided all required files	✗	✓
	Provided all required configuration files	✓	✓
	Validated input file type	✗	✗
	Suggested project structure	✗	✗
	Total score	1	2
Security	Provided security configuration from the start	✗	✗
	Filter out malicious input	✗	✗
	Up to date on libraries information	✗	✗
	Total score	0	0
Collaboration	Understood development context	✓	✓
	Accepted suggestions from user	✓	✗
	Understand and fix issue in two or less prompts	✓	✗
	Can interpret application log files	✓	✓
	Can offer useful support in debugging	✓	✗
	Total score	5	2
Code quality	Respected naming standards for Java classes	✓	✓
	Respected naming standards for API endpoints	✗	✓
	Separated functionalities in different classes	✗	✓
	Generated a modern project structure	✗	✗
	Code compiled on first run	✓	✗
	Total score	2	3
	Final score	10	10

C. Rating the final results and the overall user experience

This chapter has thus far presented the use of AI chatbots to generate code starting from simple prompts, documented the responses, then compared them to the best practices in software development, or to previous experiences using AI tools. However, it might not be clear yet what are the advantages and disadvantages of using intelligent assistants in software development.

This section presents a comparative analysis of the results obtained using Claude and Copilot. The metrics are organized into five categories: accessibility, feature implementation, security, collaboration and code quality. Each section contains multiple statements, which have been rated as either true or false for each AI tool under test.

According to the data presented in Table II, the code generated by intelligent chatbots displays a lack of focus on ensuring security, as presented in Fig. 2. To enhance the

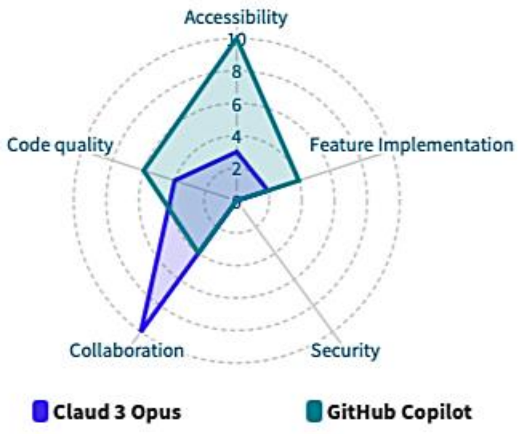


Fig. 2. Claude vs. Copilot feature plot chart

reliability of the code, it is necessary to implement measures such as user authentication, input filtering against malicious files, and other security enhancements. Furthermore, the collaboration section highlighted the capacity of Claude 3 Opus to retain context, despite not being integrated into the development environment. It exhibited an ability to recall information accurately and contextualize it adequately, providing tailored responses. Additionally, it demonstrated the capacity to accept user feedback and adapt the investigation trajectory in response to such input.

Both AI tools offer free plans, though the period and the benefits differ. The free plan provided by Anthropic allows the users to access the Claude chatbot from its official web site. However, the specific model that responds in the chat is chosen by the company and there is a limit to the number of responses that can be generated. For example, the most recent iteration of the models, Claude 3.5 Sonnet, is currently available for free interaction on the official web page. The cheapest plan costs $20 per month per person, and it allows users to choose which model to chat with. Furthermore, it provides priority bandwidth, availability and additional conversation tokens. GitHub Copilot offers a free 30-day trial, allowing potential users to determine whether the tool is suitable for their requirements. The cheapest plan costs $10 per month per person and provides access to fundamental AI chat functionalities. The more expensive plans are well-suited for businesses that require synchronization between developers and can benefit from supplementary assistance with code integration, merge requests evaluation, and other features that facilitate teamwork. It is important to highlight that for individual use, Copilot offers a single AI model, whereas Anthropic offers three AI models, for $20 per month.

IV. Case study – Test generation – DiffBlue vs. Sapient

There are two contexts that require writing tests: firstly, when a new feature is introduced, and the tests are written prior to implementation, as described by the test-driven development methodology (TDD). Secondly, when new test cases are identified for an existing feature, typically following the discovery of a bug.

In order to generate an implementation using AI assistants, a new feature had to be implemented according to the TDD methodology. A new empty method was manually added prior to requesting that Claude and Copilot write unit tests for that method, which is to be used by clients in order to upload images. Additionally, the method should only accept JPG or PNG files. The two assistants generated virtually identical tests, which were validated by JUnit 5 assertions. The test cases were short and precise, calling the endpoint with different scenarios, utilizing both valid and invalid input files, in multiple combinations. For example, there were the two "happy path" scenarios, where the files sent were JPG or PNG files, with the correct media type specified in the REST calls. There were test cases where the media type specified differed from the file type extension, or the file was not attached.

The tests were run, and they failed, as anticipated, because the feature was not implemented yet. Once the code had been generated and inserted into the previously empty method, the tests were run again, and all of them passed successfully. The only difference between the code generated by Claude or by Copilot was URL path, which is not of significant importance, as they both respected the naming conventions. The TDD approach ensured that the initial requirements were correctly validated, preventing the tendency to adapt the tests after the feature was developed and potentially overlooking technical specifications. AIs were useful in defining corner cases for input validations, reducing the time spent on tasks such as brainstorming testing scenarios or writing numerous unit tests that were similar, but required minor modifications.

A second context for the generation of tests was the writing of additional testing cases for already implemented features [8]. Therefore, *Diffblue* and *Sapient* were used in order to generate tests for the endpoint utilized for document upload. Diffblue generated 12 short but concise test methods, using *JUnit 5* and *MockMvc*, in order to simulate the REST calls initiated by users. Despite the numerous input validations, there was no "happy path test" that utilized the correct values when making an API call. Sapient provided the option of implementing both unit tests and integration tests, and it injected all the necessary dependencies into the test files. The test implementation was overly complex and difficult to comprehend. The generated code was dense and lacked spacing, making it challenging to read. Additionally, the solution did not utilize SpringBoot to initiate the application context, and the property placeholders in the Java classes were unable to retrieve their values from the configuration files. Consequently, the values had to be manually injected into the context, which is an anti-pattern that increases the probability of manual errors and requires significant time for debugging in the event of future malfunctions.

It is notable that neither Diffblue nor Sapient are interactive, thus they do not accept user prompts and the outputs generated cannot be customized. Additionally, none of the tests generated by either of the four AI assistants suggested any security checks on the inputs. Such checks might have included validating that the files were not corrupted, that there were no embedded scripts, or that the file names did not exploit any traversal path vulnerabilities.

V. Case study – Code review – SNYK vs. Tab Nine

Static code analyzers have been on the market for a long time and have become a reliable tool for writing qualitative code, ever-present in the software development environment. New AI-powered assistants have now emerged that are specifically designed to analyse code, with notable examples including Snyk and Tab Nine.

Snyk is a complex product, powered by *DeepCode AI*, to provide intelligent analysis of the code and its vulnerabilities, as outlined in the official product documentation. It can be accessed directly from the Snyk website and requires access to the Git repository in order to perform an analysis. The resulting output is a comprehensive report that details the total number of critical, high, medium, and low severity vulnerabilities identified within the source code and in the project dependencies. The first part of the analysis identified a high vulnerability due to a file uploading endpoint that was vulnerable to an XSS attack, because the input fields lacked the requisite filters and encoding algorithms. The second part scanned the project's dependencies and identified one high vulnerability, associated with the H2 in-memory database dependency, and six medium vulnerabilities related to other libraries. No recommendations were provided by Snyk regarding the remediation of the identified vulnerabilities, as shown in Fig. 3.

Tab nine is a more convenient AI tool that can be integrated into an IDE. It provides a chat box for user interaction, but it does not automatically run an analysis of the source code. It is an additional step that can be easily overlooked, particularly in stressful situations, such as deadlines or periods of intense work. Tab nine is not solely dedicated to code analysis; it is also capable of generating implementations, documentation, or explaining code blocks. When utilised, it offers suggestions for code improvements and refactoring, athough it did not identify any vulnerabilities in the code. Additionally, the user must possess a certain degree of familiarity with the project files to be able to request that Tab Nine generate a report.

SonarQube is one of the most popular and reliable software solutions, certified to the latest version of the ISO27001:2022 [9]. It is capable of detecting potential vulnerabilities due to the implementation of new features and scans the project dependencies for known Common Vulnerabilities and Exposures (CVEs). Additionally, it performs automatic scans and has customisable rules, allowing each developer or team to add and prioritise their own standards and restrictions.

A comparison of the costs associated with the licenses for the aforementioned tools reveals that Snyk offers a team subscription at a price of $25, Tab Nine provides a pro subscription at a price of $12, and Sonar offers a subscription at a price of $13. Consequently, the AI tools that perform code analysis do not offer any superior functionality compared to the comprehensive package available from SonarQube, which encompasses both their functionalities at a lower price than paying for the two AIs individually.

VI. Security concerns raised by using AI

The process of developing qualitative software components consists of implementing solutions by adapting one or more design patterns to the specific requirements. While the patterns are generic, the resulting source code is considered intellectual property and is protected by the law. The EU *Software Directive* (Directive 2009/24/EC) explicitly protects the computer programs as literary works [10].

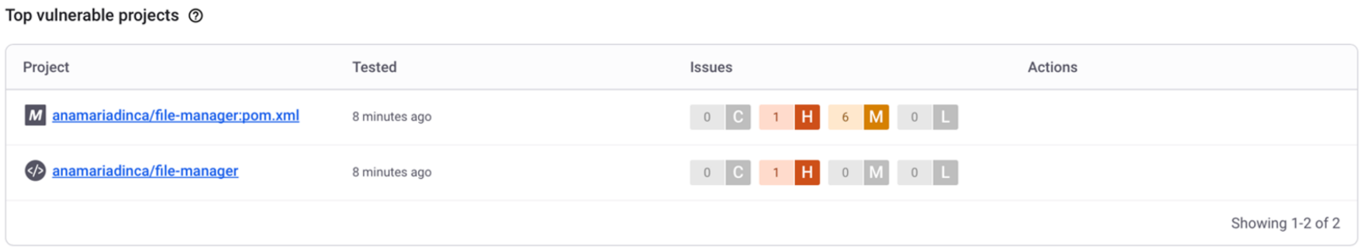
Generally, AI tools do not process information locally, they send it to their cloud environments for processing and response generation. Depending on the terms and conditions agreed upon by the user, the input can be used for training the artificial intelligence model. One of the most notorious incidents involved a couple of Samsung employees feeding the chatbots with pieces of classified information, in order to simplify their tasks, resulting in the same information being reproduced in responses sent to other users [4]. This has prompted some IT companies to ban the use of AI tools for task completion altogether, or to run private instances of various AI tools. This is accomplished by purchasing the licenses and deploying the technology in their private environments, a strategy designed to prevent the leakage of confidential information.

In addition to the release of classified information into the public domain, which represents the most obvious security concern associated with the use of AI chatbots in software development, there is another risk that is worth mentioning. Namely, the potential decrease in the quality of the product as a result of an inexperienced developer using and implementing solutions based solely on the responses generated by the AI. Section III presents a case study demonstration how blindly implementing a solution can lead to using vulnerable or deprecated dependencies, or the omission of the security filter configuration step for a software application. It is therefore recommended to use AI tools solely as a helper, as it is strongly discouraged to copy and paste solutions. Furthermore, users should carefully consider the inputs they feed into the free and public versions of AI tools, to avoid disclosing any confidential information.

VII. Conclusions

This paper proposed an investigation into the influence of AI tools in software development, given that their adoption rates have increased significantly since 2018, when the technology reached a certain level of maturity, and the responses generated by these tools became reliable and coherent. The research has been developed through a process of comparisons between different intelligent assistants, using them for practical applications and analysis of the final results. The second section presented a brief overview of the AI tools that have been developed and trained with a specific focus on software development. It also assesses the diverse categories of these tools, and their potential to enhance productivity.

The first case study involved a comparative analysis of two AI-powered *code generation* tools, *Claude 3 Opus* and *GitHub Copilot*. They are powered by different AI models, Claude is independent, whereas Copilot is powered by *OpenAI Codex*, a descendant of GPT-3 that was specifically



Top vulnerable projects

Project	Tested	Issues	Actions
anamariadinca/file-manager:pom.xml	8 minutes ago	0 C 1 H 6 M 0 L	
anamariadinca/file-manager	8 minutes ago	0 C 1 H 0 M 0 L	

Showing 1-2 of 2

Fig. 3. The vulnerabilities found and reported by Snyk

trained for software development [11]. The two AI tools were provided with the same input prompts, and they produced similar code, with minor discrepancies resulting from following different naming conventions. In essence, the implementation was nearly identical and could be considered overall complete. The two dependency configuration files that had been generated required an older version of a mandatory library, therefore the most recent versions had to be identified and modified manually. Despite having direct access to the project context, Copilot provided a significantly inferior debugging experience compared to Claude, which was accessed through a web interface. Furthermore, Claude performed better at identifying code improvements and accepting user feedback, resulting in a more collaborative and enjoyable experience. After using both AIs for practical applications, a final impression can be that they are useful for repetitive snippets of code and for offering assistance in finding small solutions. However, it should be noted that every response must be filtered through a personal evaluation, therefore it would be detrimental to allow junior developers to utilize such tools given that they lack the experience required to conduct the necessary analysis.

The second case study was an investigation into the *test writing* processes in two different scenarios: the *TDD* methodology and the implementation of *unit tests*. The first scenario was tested by implementing a new feature, starting from an empty method and requesting that Claude and Copilot write tests for the new feature, based on a behavioural description. The tests generated covered a significant number of cases and the coverage could be considered sufficient. The first run of the tests was unsuccessful, as expected, due to the absence of the implementation in the new method. After generating the code, all tests passed successfully. The implementations produced by Claude and Copilot were largely identical, as the new feature was relatively straightforward, and the prompts used were sufficiently detailed. In the second usage scenario, the unit tests were generated using *Diffblue* and *Sapient*. These AI-powered tools are not interactive, making it impossible to influence or regenerate implementations. Diffblue used JUnit 5 and MockMvc, two widely used and contemporary tools for unit testing, in order to develop 12 test cases that were sufficiently diverse, and which did not include a "happy path test". Sapient also implemented a multitude of tests that covered a considerable number of scenarios. However, the implementations were overly complex and difficult to follow, consequently making it more challenging to modify or correct the tests in the future.

The final case study was the *code review*, a stage of the software development process that is highly susceptible to the human error. The reviews are typically performed in two stages. The initial stage, the static review, is generally conducted using *SonarQube*, a tool widely utilized in the industry, and a dynamic review, performed by other developers within the team, which provides the final approval before integrating the code changes. AI tools are attempting to fulfil both of these roles, but their performance was rather modest. *Snyk* was able to detect a couple of libraries that had declared transient vulnerabilities, whereas *Tab Nine* was incapable of doing so, but it did make minor code refactoring suggestions. These features are both provided by SonarQube, a cheaper tool than using two AI tools for code review.

The last section addresses the *security concerns* associated with the use of AI in software development. It is emphasized that the use of free AI tools should be firmly discouraged when working with confidential information, as there may be clauses that authorize the model to train on the data that has been inputted, as an instance of this occurring has been documented previously [4]. There are alternative solutions that may be more expensive, such as purchasing private use licenses of popular AI tools. Nevertheless, it is important to consider the potential increase in productivity or reduced risk of secret information, as well as the impact on profit margins. Another notable security concern is the lack of experience in the utilization of such tools, including the integration of new implementations without a thorough review. Furthermore, the absence of security configurations within the solutions generated by AI tools represents a significant vulnerability.

In conclusion, the use of specialized AI tools in software development has the potential to increase the productivity when utilized by experienced users, particularly for repetitive coding tasks. The implementations, however, must be subjected to meticulous scrutiny. It is strongly advised that code snippets not be pasted into free AI tools for any reason, in order to prevent the models to train on confidential information. When using intelligent assistants to generate code, it is important to consider the security implications and implement changes in accordance with a security-first methodology. This approach eliminates the additional step of redoing the feature in order to accommodate security mechanisms that have been overlooked, which could result in a doubling of the implementation time and costs.

References

[1] Statista, "Adoption rate for major milestone internet-of-things services and technology in 2022, in days", December 2022. https://www.statista.com/statistics/1360613/adoption-rate-of-major-iot-tech/. [Accessed 29 May 2024].

[2] E. Kalliamvakou, "Research: quantifying GitHub Copilot's impact on developer productivity and happiness", September 2022. https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/. [Accessed 29 May 2024].

[3] V. Carmine, P. Sebastiano, P. Fabio, P. Sebastian, C. G. Harald and Z. Andy, "How developers engage with static analysis tools", Empirical Software Engineering, vol. 25, no. 2, pp. 1419–1457, March 2020.

[4] M. Gurman, "Samsung Bans ChatGPT, Google Bard, Other Generative AI Use by Staff After Leak", Bloomberg, May 2023. https://www.bloomberg.com/news/articles/2023-05-02/samsung-bans-chatgpt-and-other-generative-ai-use-by-staff-after-leak. [Accessed 02 August 2024].

[5] Anthropic, "Introducing the next generation of Claude", March 2024. https://www.anthropic.com/news/claude-3-family. [Accessed 20 May 2024].

[6] "Reflection AI: Breakthrough 70B and 405B Models", 2024. https://reflectionai.ai. 2024. [Accessed 09 September 2024]

[7] Google, "Control access to less secure apps," 2022. https://support.google.com/a/answer/6260879?hl=en. [Accessed 02 June 2024].

[8] M. A. Job, "Automating and Optimizing Software Testing using Artificial Intelligence Techniques", International Journal of Advanced Computer Science and Applications, vol. 12, no. 5, January

[9] ISO/IEC 27001:2022, "Information security, cybersecurity and privacy protection—Information security management systems".

[10] "Directive 2009/24/EC of the European Parliament and of the Council of 23 April 2009 on the legal protection of computer programs", Publications Office of the European Union.

[11] OpenAI, "OpenAI Codex", 10 August 2021. https://openai.com/index/openai-codex/. [Accessed 16 08 2024].