IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 22 October 2025, accepted 17 November 2025, date of publication 20 November 2025,
date of current version 26 November 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3635168

RESEARCH ARTICLE

Large Language Models Versus Static Code Analysis Tools: A Systematic Benchmark for Vulnerability Detection

DAMIAN GNIECIAK AND TOMASZ SZANDALA
Faculty of Information and Communication Technology, Wroclaw University of Science and Technology, 50-370 Wroclaw, Poland
Corresponding author: Tomasz Szandala (Tomasz.Szandala@pwr.edu.pl)

ABSTRACT Modern software relies on a multitude of automated testing and quality assurance tools to prevent errors, bugs and potential vulnerabilities. This study sets out to provide a head-to-head, quantitative and qualitative evaluation of six automated approaches: three industry-standard rule-based static code-analysis tools (SonarQube, CodeQL and SnykCode) and three state-of-the-art large language models hosted on the GitHub Models platform (GPT-4.1, Mistral Large and DeepSeek V3). Using a curated suite of ten real-world C# projects that embed 63 vulnerabilities across common categories such as SQL injection, hard-coded secrets and outdated dependencies, we measure classical detection accuracy (precision, recall, F-score), analysis latency, and the developer effort required to vet true positives. The language-based scanners achieve higher mean F-1 scores,0.797, 0.753 and 0.750, than their static counterparts, which score 0.260, 0.386 and 0.546, respectively. LLMs' advantage originates from superior recall, confirming an ability to reason across broader code contexts. However, this benefit comes with substantial trade-offs: DeepSeek V3 exhibits the highest false-positive ratio, all language models mislocate issues at line-or-column granularity due to tokenisation artefacts. Overall, language models successfully rival traditional static analysers in finding real vulnerabilities. Still, their noisier output and imprecise localisation limit their standalone use in safety-critical audits. We therefore recommend a hybrid pipeline: employ language models early in development for broad, context-aware triage, while reserving deterministic rule-based scanners for high-assurance verification. The open benchmark and JSON-based result harness released with this paper lay a foundation for reproducible, practitioner-centric research into next-generation automated code security. ProjectAnalyzer Code and dataset are available on GitHub: https://github.com/Damian0401/MasterThesis

INDEX TERMS Large language models, software engineering, CI/CD, software quality.

I. INTRODUCTION

Rapidly developed software has become vital for the modern industry. Yet, its growing complexity and attack surface have outpaced the human capacity to reason about every line of code [1]. Static Application Security Testing (SAST) has long been the developer's first line of defence: tools such as SonarQube, CodeQL, and SnykCode integrate seamlessly into continuous-integration pipelines, flagging defects before they ever reach production, and are trusted by more than seven million developers across 400,000 organisations worldwide.

Recent empirical studies nevertheless reveal a persistent gap, while a single state-of-the-art analyser can highlight vulnerabilities in roughly half of real-world, vulnerability-contributing commits, false-positive noise, and rule coverage limitations continue to hamper day-to-day adoption. Concurrently, the remarkable rise of large language models (LLMs) has opened a new branch of code intelligence. Industry surveys chart a rapid uptick in LLM pilots across engineering teams. The research community has begun to fine-tune these models for vulnerability detection with promising results:

The associate editor coordinating the review of this manuscript and approving it for publication was Dominik Strzalka.

early experiments report competitive F-scores against graph-based and traditional sequence models, while comparative analyses on emerging model: GPT-4-class systems, Mistral-Large, DeepSeek V3, and others—suggest that LLMs can reason across larger contexts and identify subtle data-flow flaws that evade conventional pattern-based rules.

Yet, enthusiasm must be tempered with caution. The same generative power that allows an LLM to ''understand'' source code also enables failure modes alien to deterministic analysers. For instance, a recent industry report warns of ''slopsquatting'': hallucinated package names inserted by LLMs that attackers later register in public repositories, creating an entirely new supply-chain vector. Such findings underline the need for a principled, head-to-head assessment rather than anecdotes and marketing claims.TechRadar

This paper undertakes the first systematic comparison between three mature static analysers, SonarQube, CodeQL, and SnykCode, and three cutting-edge LLMs, available on GitHub: GPT-4.1, Mistral-Large, and DeepSeek V3, evaluated on a standard benchmark suite spanning multiple projects and vulnerability classes. We measure classical accuracy metrics (precision, recall, F1), as well as developer-centric costs, such as inspection effort per true positive, time-to-signal in CI, and the qualitative value of the explanations generated by each approach.

Our contributions consist of:

- Unified benchmark and protocol. We form a diverse dataset of real-world vulnerable functions, pair each with ground-truth labels, and release an open evaluation harness suitable for rule-based and generative systems.
- Comprehensive quantitative evaluation. We show where LLMs already rival, and occasionally surpass, specialist analysers in recall, while highlighting scenarios in which deterministic rules remain indispensable for suppressing false alarms.
- Qualitative insight. Through thematic analysis of model rationales and tool reports, we surface strengths (e.g., data-flow reasoning across files) and weaknesses (e.g., inconsistent CWE mapping) unique to each paradigm.
- Guidance for practitioners and researchers. We distil our findings into recommendations on tool choice, prompt engineering, and future benchmarks that capture emerging threat vectors.

II. STATE OF THE ART

Static code analysis represents an essential component within modern software development processes. Early detection of software defects, which, according to various studies, can lead to a significant reduction in the cost and impact of subsequent corrections [2], [3]. Furthermore, it simplifies the code review process by helping developers identify potential issues early, thereby reducing the time and effort required for manual code reviews [4]. It is worth noting that the utility of static analysis is not limited to professional and commercial use. Academic research highlights its educational value. A large-scale study analysing more than 500 student-developed software projects illustrates that regular static code analysis tools can significantly improve code quality and student learning outcomes [5]. With the increasing integration of artificial intelligence into the software development lifecycle, the role and application of static code analysis are also poised to evolve. AI-powered development tools are a way of reinventing traditional programming practices and introducing new possibilities. This approach will enhance the effectiveness of static analysis techniques and redefine their function, enabling more contextualised, personalised, and proactive code evaluation processes.

A. HOW STATIC CODE ANALYSIS WORKS

Static source code analysis involves examining the source code without executing it to detect potential bugs and weaknesses. The code is analysed for defined patterns or rules that indicate undesirable behaviour. Typical tools use control, data flow analysis, and pattern matching to known error signatures [6]. Static scanners can detect, for example, SQL Injection or code injection vulnerabilities through taint analysis, but typically only if such patterns have been predefined in their rules. In industry practice, rule sets are constantly expanded to include new bugs. Due to the multiplicity of use cases, they have been divided into categories [7], such as:

- **Naming** – Issues related to inconsistent, misleading, or nonstandard naming conventions for identifiers.
- **Style** – Violations of code formatting or stylistic guidelines can affect readability.
- **Concurrency** – Problems arising from incorrect handling of parallel execution or shared resources.
- **Exceptions** – Improper use or handling of exceptions and error conditions.
- **Performance** – Inefficiencies or bottlenecks that could degrade the system's performance.
- **Interoperability** – Compatibility problems between different systems, platforms, or APIs.
- **Security** – Vulnerabilities that could be exploited to compromise the system's integrity or data.
- **Maintainability** – Code complexity or structure issues that make future modification or understanding more difficult.
- **General** – Broad or uncategorised issues that do not fit neatly into other specific groups.

B. FORMAT OF RESULTS

Static code analysis tools are often integrated into other software development processes. The analysis results are typically a crucial element before the final implementation of the system under development.

SARIF (Static Analysis Results Interchange Format) was introduced in 2020 [8] as a standardised and extensible format for exchanging results produced by static analysis tools, enabling consistent interpretation, comparison, and

```
{
"$schema": "https://json.schemastore.org/sarif-2.1.0.json",
"version": "2.1.0",
"runs": [{
"tool": {
  "driver": {
    "name": "SonarQube",
    "semanticVersion": "1.0.0",
    "version": "1.0.0",
    "rules": [{
      "id": "SQL_INJECTION",
      "shortDescription": { "text": "SQL Injection vulnerability" }
    }]
  }
},
"results": [{
  "ruleId": "SQL_INJECTION",
  "ruleIndex": 0,
  "level": "error",
  "message": { "text": "User input is used directly in a SQL query. This can lead to
      ↪ SQL injection." },
  "locations": [{
    "physicalLocation": {
      "artifactLocation": { "uri": "Controllers/UserController.cs" },
      "region": {
        "startLine": 27,
        "endLine": 27,
        "startColumn": 17,
        "endColumn": 66
      }
    }
  }]
}]
}]
}
```

Listing 1. Example SARIF JSON output.

integration across different platforms. Since then, it has been adopted and integrated by many static analysis tools [9]. The structure shown in listing 1 represents a simplified example of a SARIF JSON output. The list includes only the fields necessary to generate a minimal report.

$schema: Specifies the URL of the JSON schema used to validate the structure, referencing the official SARIF 2.1.0 specification.
version: Defines the version of the SARIF format that is being used (here, *"2.1.0"*).
runs: An array containing analysis runs.
tool: Provides metadata on the analysis tool.
driver: Describes the tool driver (e.g., SonarQube).
name, **semanticVersion**, **version**: Identify the tool and its version.
rules: An array of rules-objects applied during analysis.
id: Unique identifier for the rule (e.g., *"SQL_INJECTION"*).
shortDescription: A brief text describing the rule.
results: An array of analysis results.
ruleId, **ruleIndex**: Reference the rule responsible for the finding.
level: Indicates the severity level of the result (e.g., *"error"*).
message: A descriptive message on the issue was found.
locations: An array indicating the location of the issue in the code.
physicalLocation: Provides detailed location data.
artifactLocation: The file where the issue occurs.
region: Describes the exact range in the file (line and column numbers).

This structure does not represent all available SARIF fields comprehensively. According to the official SARIF

schema [10], many additional fields and features are supported. The example includes only the fields required to produce a minimal report.

C. LIMITATIONS OF TRADITIONAL CODE ANALYSIS

Despite their widespread use, static approaches face significant challenges. First, designing a good analyser is time-consuming and difficult; each new type of defect requires the development of an appropriate rule. As a result, typical tools are limited to predefined defect patterns and may fail to recognise new or unusual problems [11]. Another limitation may be the lack of ability to fully understand the context of the analysed code, as shown by the authors of the paper [12]. This shortcoming underscores the need for more adaptive and intelligent approaches and methods to fill the gap between recognising a known pattern and deep semantic understanding.

D. USE OF LARGE LANGUAGE MODEL

In recent years, numerous works have examined large language models in the context of industry code quality [13], [14]. Models trained on massive collections of code and text demonstrate the ability to understand the context of a program and suggest corrections. This raises the question of whether they can take over the role of classic static analysers in bug detection. An essential advantage of LLMs is the lack of rigidly defined rules - the model can potentially catch an error based on its ''understanding'' of the context, even if the pattern is not explicitly programmed. Research shows that LLMs can detect certain defects without the need for a complete set of tests or rules, due to their ability to infer from the context of the code. The comparative potential of large language models and traditional static analysis tools has been explored in studies such as [12], which analysed the performance of GPT-3.5 Turbo and GPT-4o in relation to SonarQube. This work is extended by additional static analysers, including CodeQL and SnykCode, as well as a broader set of large language models such as GPT-4.1, DeepSeek V3, and Mistral Large, enabling a more comprehensive evaluation of their capabilities in the detection of defects in the source code.

III. METHODS

Ensuring code quality and reliability is crucial in modern IT systems. Static analysis tools typically identify vulnerabilities based on predefined patterns or rule sets. The effectiveness of such tools in improving software quality and their efficiency in minimising code defects has been demonstrated in the existing literature [15].

A. OVERVIEW OF EVALUATED TOOLS

For this work, three widely recognised static analysis tools have been selected for evaluation: SonarQube, SnykCode and CodeQL.

SonarQube is an open-source tool used for static code analysis, created in 2006 [16]. It is designed to automatically detect errors, security vulnerabilities, and issues related to code quality and technology debt. Over the course of almost 20 years on the market, it has evolved to support over 30 programming languages [17] and features a built-in interface with comprehensive dashboards and detailed reports. This tool is used by more than 400,000 organisations [18]. When choosing SonarQube, you can select one of two options to utilise this solution.

The user can self-host the Community version, which is open-source and can be used initially at no cost. In that case, it is in the consumer's interests to ensure the safety of processed data. The second option is to choose one of the two paid versions provided by SonarSource, which are Team and Enterprise [19]. The first version costs 65$ per month (as of 2025), and the cost of the higher tier is determined individually after contacting the sales department. SonarSource guarantees compliance with the EU General Data Protection Regulation (GDPR) and the California Consumer Privacy Act to ensure the privacy of its users' data [20]. Due to its long existence on the market, SonarQube is a very mature and proven tool.

CodeQL is an engine created and developed by GitHub. Its main goal is to discover security vulnerabilities and bugs in the analysed source code. The distinguishing feature of this tool over others is that it allows users to write custom queries using a specialised domain-specific language named QL. CodeQL is part of the GitHub Advanced Security platform and can be used with the most popular programming languages, such as C++, Java, and Python [21]. CodeQL, as part of GitHub Code Security, also offers two pricing levels: Team for the price of 4$ per month for each user and Enterprise starting at 21$ per month for each user [22]. GitHub processed personal data according to the declaration of respect for the GDPR and other applicable laws [23].

SnykCode is a static application security testing tool created by the Snyk company. Unlike traditional SAST tools, SnykCode uses machine learning algorithms to detect issues in the source code [24]. This tool focuses mainly on security vulnerabilities such as SQL injection, cross-site scripting (XSS), and authentication-related issues. The first version was officially released in 2020 [25]. SnykCode is also available in two pricing plans, Team for 25$ per month for each contributing developer and Enterprise, where the price is determined individually after contacting the sales department, similar to SonarQube [26]. To ensure data privacy, Snyk leads a global program designed to align with the requirements of the GDPR and other privacy laws [27].

Before using each static analysis tool, it is crucial to be familiar with the license and data privacy arrangements. Another important aspect is to conduct a cost analysis. Key elements were visualised with Table 1 to facilitate a better comparison of each tool. All three solutions comply with data privacy regulations and offer multiple pricing tiers, but differ

TABLE 1. Comparison of code analysis tools.

	SonarQube	CodeQL	SnykCode
Hosting model	Self-hosted (Community) or cloud-based (Team, Enterprise)	Cloud-based via GitHub Code Security	Cloud-based (Team, Enterprise)
Cost of license	Free (Community), $65/month (Team), Enterprise (custom pricing)	$4/user/month (Team), from $21/user/month (Enterprise)	$25/user/month (Team), Enterprise (custom pricing)
Data privacy	Compliant with GDPR and CCPA	Compliant with GDPR and other applicable laws	Global program aligned with GDPR and other privacy laws

in hosting models and cost structures, which may influence the selection based on organisational needs.

B. LARGE LANGUAGE MODELS IN CODE ANALYSIS

The use of large language models can significantly improve code quality assessments, as they can analyse predefined rules, vulnerabilities, and contextual aspects of the code under review [12]. For this work, three large language models from leading companies have been selected for evaluation: GPT 4.1, DeepSeek V3, and Mistral Large.

GPT 4.1, developed by OpenAI, is known for its extensive context window of 1,047,576 tokens, significantly enhancing its ability to handle complex tasks, such as analysing large codebases or extensive documents. The price is set at $2 per million input tokens and $8 per million output tokens [28]. Regarding data privacy, user interactions with GPT 4.1 in consumer products can be used to train OpenAI models, but users can opt out through settings. For business products, data is not used for training by default [29]. The declared knowledge cut-off date is June 2024 [30].

DeepSeek V3 offers a cost-effective alternative with a context window of 64,000 tokens, attractively priced at $0.27 per million input tokens and $1.10 per million output tokens [31]. DeepSeek's privacy policy specifies that the model may collect user inputs, uploaded files, and network metadata such as IP addresses and device identifiers. This reflects a broader trend among commercial LLM providers, where the scope of data collection and retention policies varies depending on the vendor [32]. The declared knowledge cut-off date is July 2024 [33]. It was created by the Chinese company DeepSeek, founded in 2023. The model has gained popularity due to its competitive capabilities in relation to the cost of its exploitation.

Mistral Large, from Mistral AI, features a balanced context window of 32,000 tokens and costs $2 per million input tokens and $6 per million output tokens [34]. Its data privacy policy states that user data is not utilised for training except in specific cases, such as free-tier usage without explicit opt-out, feedback provision, or content moderation purposes [35].

Before integrating LLM into practical workflows, it is essential to understand its licensing terms, data privacy policies, and associated costs. These factors are critical in selecting a model that aligns with organisational requirements. Key aspects were visualised with a Table 2 to compare each LLM better. Each of the selected models has distinct advantages and limitations. GPT 4.1 excels with its unparalleled context capacity, which is essential for complex tasks, such as analysing a large codebase, but it has higher pricing. DeepSeek V3 offers significant cost benefits, but at the expense of increased data privacy risks. Mistral Large presents a similar pricing and approach to data privacy to GPT 4.1, but is slightly more cost-effective.

The effective use of large language models for static analysis requires a structured approach to data preparation and result handling. This section covers the process of selecting and preparing software projects, describes the SARIF format used to represent results, and presents the tool created to perform the analysis. For projects exceeding the model's context window, the source code was automatically segmented into coherent, dependency-aware units and processed in parallel. This strategy follows recent best practices in large-scale LLM evaluation [36], [37], ensuring that long repositories can be analysed without truncation or loss of semantic context.

C. DATASET

For the purpose of analysis, 10 projects were prepared in the popular programming language, which is C#. According to Table 3, the size of the prepared projects is between 3500 and 5500 characters and contains up to 10 files. The number of files was selected based on the analysis, which shows that it is the most common change based on the source control history [38].

Each project has a random number of vulnerabilities, such as:

- SQL injection,
- cross-site scripting,
- hardcoded secrets,
- command injection,
- weak encryption algorithms,
- deprecated dependencies.

The distribution is generally balanced across categories, with the exception of deprecated dependencies, which count 13 cases.

D. PROJECTANALYZER

Working with large language models requires intensive computational power, which presents a notable challenge for consistent evaluation. Furthermore, GPT 4.1, one of the three models originally intended for study, is not publicly accessible. As a result, to ensure equivalent conditions and

TABLE 2. Comparison of large language models.

	GPT 4.1	DeepSeek V3	Mistral Large
Knowledge cutoff	June 1, 2024	July 1, 2024	October 26, 2023
Context window (in tokens)	1,047,576	64,000	32,000
Cost of usage (per 1M tokens)	2$ input, 8$ output	0.27$ input, 1.1$ output	2$ input, 6$ output
Data privacy	User data may be used for training (opt-out available for consumers; not used by default for business products).	Extensive data collection, including inputs, chat history, and automated network data collection.	Data used for training only under specific conditions (e.g., free tier without opt-out, moderation purposes).

TABLE 3. Number of files, characters and vulnerabilities in each project.

Project	Number of files	Number of characters	Number of vuln.
S01	5	5317	8
S02	3	3145	3
S03	3	3431	6
S04	4	4016	1
S05	3	5117	6
S06	7	3791	8
S07	7	5036	13
S08	8	3878	7
S09	7	3709	8
S10	10	4938	3

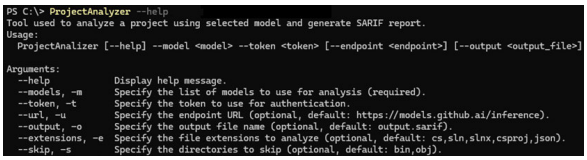

FIGURE 2. ProjectAnalyzer - help command.

FIGURE 3. ProjectAnalyzer - usage.

a fair basis for comparison, the GitHub Models platform was chosen as the hosting environment. GitHub Models is a publicly accessible repository developed through a collaboration between GitHub and Microsoft [39], providing a standardised and open place to prototype and build AI-powered solutions.

For the purpose of generating reports in SARIF format, the tool named *ProjectAnalyzer* was developed. It is designed to automate the analysis of software projects using large language models. The tool facilitates static analysis by scanning the current working directory for source code files with specified extensions (e.g., *cs*, *.csproj*, *.sln*). Once the relevant files are identified, the tool constructs a single aggregated prompt that represents the entire project, suitable for processing. The prompt is then sent by API to each of the selected models. Each model processes the prompt independently and returns an analysis in JSON format. This response is subsequently parsed and transformed into the SARIF format. The final report is saved to the specified output location, providing a machine-readable summary of the model's findings. C# was chosen as the programming language to develop the analyser due to its native and officially supported capabilities for interacting with the API used in this project [40].

During analysis, a system prompt, shown in Figure 2, was included as a system message. It was intended to instruct the model on how it should behave and in what format the results should be returned. Figure 2 illustrates the help command of the *ProjectAnalyzer* tool, displaying available arguments, flags, and usage options, while Figure 3 illustrates the usage of the tool, showing the summary of the completed analyses for each selected model, and reporting the time required to generate the full report. The token used to access models from the GitHub Models platform was hidden.

IV. RESULTS

A set of standardised ML model performance metrics was utilised for each static analysis tool and the large language model under consideration to evaluate all the projects prepared. Specifically, the evaluation measured the execution time, the number of detected vulnerabilities, and the number of true positive identifications. These indicators form the basis for a more in-depth and comparative analysis.

A. DEFINITION OF EVALUATION METRICS

The total number of identified vulnerabilities was determined by counting the number of entries present in the Results section of the SARIF reports generated by the analysis tools [10]. Each entry in this section represents a distinct issue identified during the analysis process, and the aggregate count serves as a quantitative indicator of detection capabilities.

Each reported vulnerability was individually reviewed and compared with a reference dataset of known vulnerabilities for the given project to determine the number of true positive results. Only findings that accurately matched a real problem were counted as true positives. The only exception to this rule was the location region of the results produced by the large language models. The rationale behind this exception is explained in detail in the Discussion section.

```
You are a static code analysis engine. Your task is to review the provided source
    ↪ code and identify security vulnerabilities. Focus on detecting vulnerabilities
    ↪  such as (but not limited to):
- SQL Injection
- Cross-Site Scripting (XSS)
- Command Injection
- Insecure Deserialisation
- Insecure or missing authentication/authorisation mechanisms
- Hardcoded credentials or secrets
- Improper input validation or lack of sanitisation
- Use of outdated or vulnerable libraries
- Insecure use of cryptography (e.g., weak algorithms, hardcoded keys)
- Insecure file handling (e.g., path traversal, unrestricted uploads)

Only analyse and report issues that pose a security risk. Do not report code smells,
    ↪  general bugs, or non-security-related issues.
Your output must be a JSON array, enclosed between triple backticks (```json and
    ↪ ```), with each finding represented as a JSON object in the following format:
[{"RuleId":"string","RuleDescription":"string","Level":"Error"|"Warning"|"Note"|"
    ↪ None", "Message":"string","Path":"string","Category":"string","StartLine":
    ↪ integer, "EndLine":integer,"StartColumn":integer,"EndColumn":integer}]

Field description:
- RuleId: A short identifier for the rule or issue.
- RuleDescription: A brief description of the rule being violated.
- Level: Severity of the issue (Error, Warning, Note, or None).
- Message: A concise explanation of the specific issue found.
- Path: The file path where the issue occurs.
- Category: The general category.
- StartLine, EndLine: Line range of the issue.
- StartColumn, EndColumn: Column range of the issue.

Ensure the JSON is well-formed and strictly adheres to this JSON structure. All
    ↪ fields are required.
```

Listing 2. **Prompt sent used for analysis.**

The number of false positive results was determined by subtracting the number of confirmed true positives from the total number of vulnerabilities identified by each tool or model. A false positive is defined as a reported vulnerability that does not correspond to any real or known problem in the reference data set for the analysed project.

The F1 score is a standard evaluation metric used to assess the overall effectiveness of a classification model by combining precision and recall into a single harmonic mean. In the context of vulnerability detection, it provides a balanced view of how well a tool or model identifies true vulnerabilities while minimising incorrect detections. To fully understand and calculate the F1 score, it is essential to introduce the individual metrics on which it depends, including false negatives, precision, and recall, which together determine the overall value of the score.

False negatives are calculated by subtracting the number of true positive results from the total number of known vulnerabilities in the reference data set for each project.

$$\begin{aligned}&\text{False Negatives}\\&\quad = \text{Total Known Vulnerabilities} - \text{True Positives}.\end{aligned}$$

Precision is defined as the proportion of correctly identified vulnerabilities (True Positives) among all reported vulnerabilities and is calculated as:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}.$$

Recall, on the other hand, measures the proportion of actual vulnerabilities that were correctly identified by the tool and is given by:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}.$$

The **F1 score** is then calculated as the harmonic mean of precision and recall, providing a single measure that balances both:

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}.$$

This metric is handy when comparing tools with varying balances of precision and recall, as it offers a comprehensive view of detection performance. All F1 scores reported in this work were calculated based on manually verified true positive results.

Finally, the execution time was systematically measured for each run of the static analysis tools and the implemented analyser. These measurements were recorded in seconds, maintaining a precision of three floating-point decimal places, to support accurate performance comparisons and detailed evaluation.

B. EXPERIMENTS

The results presented in the following tables illustrate the raw performance data gathered during these experiments. This data is then further analysed to provide insight into each method's strengths, limitations, and overall reliability. The experiments were run in a uniform execution environment to eliminate variability due to system performance. Furthermore, each static analysis tool and large language model was executed using their default configuration settings, unless explicitly stated otherwise, to maintain fairness and reproducibility in all approaches tested.

Table 4 presents the total number of vulnerabilities found and the number of true positive results for the static analysis tools. In contrast, Table 5 provides the corresponding data for the large language models. It should be noted that *SnykCode* consistently reports a significantly higher number of vulnerabilities compared to the other tools evaluated. This trend may indicate a higher detection sensitivity, suggesting that *SnykCode* employs a more aggressive strategy to identify potential problems.

Table 6 shows the total execution time measured for both the large language models and the static analysis tools. These measurements were obtained following the procedure described in the previous section. It should be noted that the execution time recorded for *CodeQL* is significantly longer than that of the other tools, which may reflect its deeper or more comprehensive analysis process.

TABLE 4. Number of total found and true positive (TP) findings for each project and tool.

Project	SonaQube Total	SonaQube TP	CodeQL Total	CodeQL TP	Snyk Total	Snyk TP
S01	6	4	6	3	10	7
S02	1	1	1	1	1	1
S03	2	2	3	3	3	3
S04	0	0	0	0	0	0
S05	0	0	3	3	1	1
S06	1	1	3	3	7	7
S07	1	1	5	5	14	10
S08	0	0	1	0	7	5
S09	4	4	4	3	11	8
S10	0	0	0	0	3	0

TABLE 5. Number of total found and true positive (TP) per project and model.

Project	GPT-4.1 Total	GPT-4.1 TP	Mistral Total	Mistral TP	DSeek Total	DSeek TP
S01	13	8	11	7	11	7
S02	3	3	3	2	3	3
S03	8	6	8	6	8	6
S04	2	1	3	1	5	1
S05	4	4	5	4	6	5
S06	10	7	8	6	8	6
S07	15	11	9	8	10	8
S08	7	5	5	5	7	6
S09	9	8	7	7	7	7
S10	3	2	2	2	5	3

V. ANALYSIS OF RESULTS

As part of the effort to perform a comprehensive comparative analysis using key evaluation metrics, including False Positives, False Negatives, Precision, Recall, and the F1 Score. These metrics were computed for all large language models and static analysis tools to facilitate an objective and balanced comparison of their detection capabilities and overall effectiveness.

The detailed results of this evaluation are presented in Tables 7-12. The detailed results of this evaluation are presented in Tables 7–12. Tool-based methods include *SonarQube* (Table 7), *CodeQL* (Table 8), and *SnykCode* (Table 9), while model-based approaches include *GPT-4.1* (Table 10), *Mistral Large* (Table 11), and *DeepSeek V3* (Table 12). These tables compile the computed metrics for each tool across all target projects, allowing for a direct, side-by-side comparison of their effectiveness. The tabulated data form the basis for the comparative analysis discussed in the following sections.

The static analysis tools - *SonarQube*, *CodeQL*, and *SnykCode* - achieved average F1 scores of 0.260, 0.386, and 0.546, respectively. In comparison, the large language models evaluated, *GPT-4.1*, *Mistral Large* and *DeepSeek V3*, obtained average F1 scores of 0.797, 0.753, and 0.750, respectively. Detailed average results are presented in Table 13.

A. COMPARISON

Figure 3 illustrates the analysis runtime as a function of project size, measured by the number of characters. As shown, for a typical change size as reported by [38], the number of characters does not substantially impact the analysis time. However, a clear outlier is *CodeQL*, consistently demonstrating significantly longer runtimes within this range. This behaviour can probably be attributed to its architecture and operational design. Unlike lighter tools tailored for rapid feedback, CodeQL is primarily

TABLE 6. Execution time for each tool and model.

Project	SonarQube	CodeQL	SnykCode	GPT-4.1	Mistral	DeepSeekV3
S01	39.928s	195.102s	27.370s	12.741s	35.149s	17.835s
S02	32.984s	186.703s	20.887s	4.407s	11.178s	5.815s
S03	33.544s	188.552s	27.071s	8.140s	24.019s	11.607s
S04	35.511s	207.137s	13.921s	3.526s	10.181s	7.345s
S05	34.426s	207.640s	16.886s	6.715s	14.790s	7.472s
S06	30.202s	197.323s	18.113s	9.536s	22.919s	29.860s
S07	35.921s	199.423s	21.590s	17.802s	26.623s	13.581s
S08	32.335s	185.923s	16.547s	7.633s	15.669s	12.701s
S09	38.701s	191.757s	19.025s	8.196s	22.696s	9.931s
S10	63.905s	213.088s	29.266s	4.769s	67.273s	9.083s

TABLE 7. Calculated metrics using SonarQube.

SonarQube					
Project	False Positive	False Negative	Precision	Recall	F1 Score
S01	2	4	0.667	0.500	0.571
S02	0	2	1.000	0.333	0.500
S03	0	4	1.000	0.333	0.500
S04	0	1	0.000	0.000	0.000
S05	0	6	0.000	0.000	0.000
S06	0	7	1.000	0.125	0.222
S07	0	12	1.000	0.077	0.143
S08	0	7	0.000	0.000	0.000
S09	0	4	1.000	0.500	0.667
S10	0	3	0.000	0.000	0.000

TABLE 8. Calculated metrics using CodeQL.

CodeQL					
Project	False Positive	False Negative	Precision	Recall	F1 Score
S01	3	5	0.500	0.375	0.429
S02	0	2	1.000	0.333	0.500
S03	0	3	1.000	0.500	0.667
S04	0	1	0.000	0.000	0.000
S05	0	3	1.000	0.500	0.667
S06	0	5	1.000	0.375	0.545
S07	0	8	1.000	0.385	0.556
S08	1	7	0.000	0.000	0.000
S09	1	5	0.750	0.375	0.500
S10	0	3	0.000	0.000	0.000

TABLE 9. Calculated metrics using SnykCode.

SnykCode					
Project	False Positive	False Negative	Precision	Recall	F1 Score
S01	3	1	0.700	0.875	0.778
S02	0	2	1.000	0.333	0.500
S03	0	3	1.000	0.500	0.667
S04	0	1	0.000	0.000	0.000
S05	0	5	1.000	0.167	0.286
S06	0	1	1.000	0.875	0.933
S07	4	3	0.714	0.769	0.741
S08	2	2	0.714	0.714	0.714
S09	3	0	0.727	1.000	0.842
S10	3	3	0.000	0.000	0.000

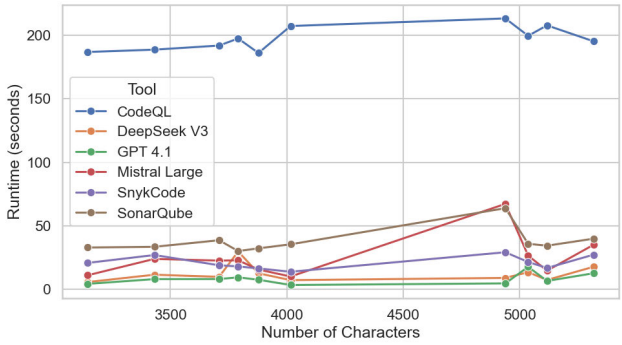

FIGURE 3. Tool Runtime vs. Number of characters.

intended for continuous integration and deployment (CI/CD) environments, where more extensive computational resources are allocated, and longer processing times are acceptable in exchange for deeper and more thorough analysis.

B. NUMBER OF FALSE POSITIVES

Figure 4 illustrates the juxtaposition of the average false positive rate (FP) with the total number of vulnerabilities found by each tool and the large language model. A lower ratio indicates higher precision in vulnerability detection, reflecting a lower percentage of incorrect alerts than the total number of reported vulnerabilities.

As shown in the figure, itSonarQube and *CodeQL* have the lowest FP ratios, suggesting that these traditional tools are more accurate and targeted in their analysis, generating fewer false alerts and highlighting vulnerabilities with greater reliability. Their performance indicates a level of reliability that can be especially appreciated in software development environments where accuracy and confidence in results are essential.

On the other hand, *DeepSeek V3* shows the highest FP ratio in the investigated solutions. This behaviour may result in the need to increase the effort in verification on the part of developers, who must spend more time manually reviewing and validating the analysis results. This suggests that while the tool may be accurate in its assessments, it may also

TABLE 10. Calculated metrics for each project using GPT-4.1.

GPT-4.1					
Project	**False Positive**	**False Negative**	**Precision**	**Recall**	**F1 Score**
S01	5	0	0.615	1.000	0.762
S02	0	0	1.000	1.000	1.000
S03	2	0	0.750	1.000	0.857
S04	1	0	0.500	1.000	0.667
S05	0	2	1.000	0.667	0.800
S06	3	1	0.700	0.875	0.778
S07	4	2	0.733	0.846	0.786
S08	2	2	0.714	0.714	0.714
S09	1	0	0.889	1.000	0.941
S10	1	1	0.667	0.667	0.667

TABLE 11. Calculated metrics for each project using Mistral Large.

Mistral Large					
Project	**False Positive**	**False Negative**	**Precision**	**Recall**	**F1 Score**
S01	4	1	0.636	0.875	0.737
S02	1	1	0.667	0.667	0.667
S03	2	0	0.750	1.000	0.857
S04	2	0	0.333	1.000	0.500
S05	1	2	0.800	0.667	0.727
S06	2	2	0.750	0.750	0.750
S07	1	5	0.889	0.615	0.727
S08	0	2	1.000	0.714	0.834
S09	0	1	1.000	0.875	0.934
S10	0	1	1.000	0.667	0.800

TABLE 12. Calculated metrics for each project using DeepSeek V3.

DeepSeek V3					
Project	**False Positive**	**False Negative**	**Precision**	**Recall**	**F1 Score**
S01	4	1	0.636	0.875	0.737
S02	0	0	1.000	1.000	1.000
S03	2	0	0.750	1.000	0.857
S04	4	0	0.200	1.000	0.334
S05	1	1	0.833	0.833	0.834
S06	2	2	0.750	0.750	0.750
S07	2	5	0.800	0.615	0.696
S08	1	1	0.857	0.857	0.857
S09	0	1	1.000	0.875	0.934
S10	3	1	0.400	0.667	0.500

TABLE 13. Average Precision, Recall and F1 Score for each tool and model.

Tool	Avg. Precision	Avg. Recall	Avg. F1 Score
SonarQube	0.567	0.187	0.260
CodeQL	0.625	0.284	0.386
SnykCode	0.686	0.523	0.546
GPT-4.1	0.757	0.877	0.797
Mistral Large	0.783	0.783	0.753
DeepSeek V3	0.723	0.847	0.750

overwhelm users with alerts that may not correspond to actual threats.

It is also important to note that the performance of *SnykCode* is closely aligned with that of LLM-based tools, with its FP ratio between those of traditional analysers and large language models. This similarity may be due to its hybrid analytical approach, which combines machine learning algorithms with techniques commonly employed in static analysis [24]. The fusion of these methods probably shapes its detection behaviour, resulting in a reporting pattern that reflects characteristics of both traditional and AI-enhanced approaches.

C. AVERAGE F1 SCORE

Figure 7 builds on the insights from Figure 5, Figure 6, and compares the effectiveness of each tool using the F1 score. The FP ratio highlights the effectiveness of a tool in isolation. Still, the F1 score provides a more comprehensive measure that balances both the precision of correct detections and the completeness of the tool in recognising issues.

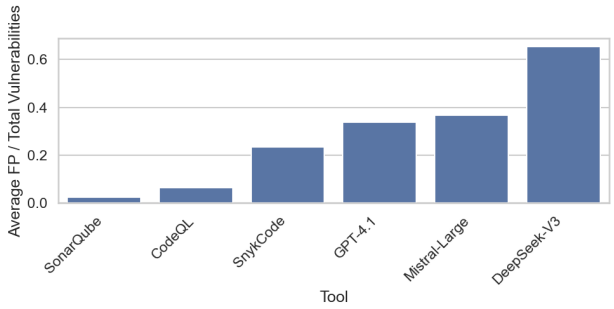

FIGURE 4. Average FP to total found vulnerabilities ratio by tool.

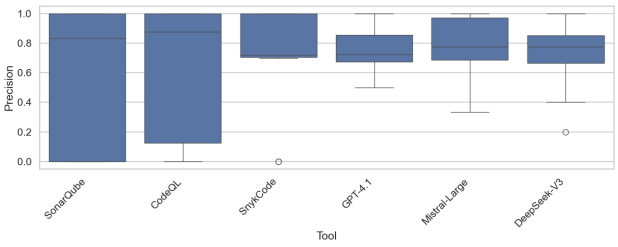

FIGURE 5. Comparison of effectiveness (Precision).

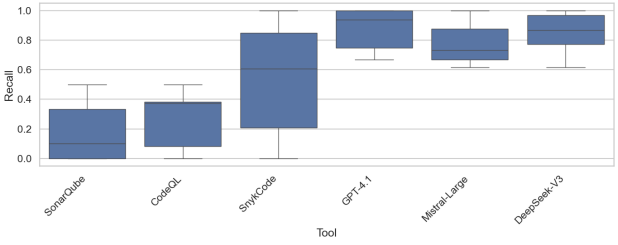

FIGURE 6. Comparison of Effectiveness (Recall).

Figure 5 and Figure 6 show that *SnykCode* performs better than other traditional static analysis tools in the F1 score and competes with systems based on large language models, further validating the effectiveness of its hybrid analysis strategy. The tool created by *Snyk Company* achieved the highest average F1 score of 0.55; however, it should be noted that it has two outliers for projects *S04* and *S10*, which can be easily seen earlier, in Figure 5. In contrast, although strong in precision as shown earlier, *SonarQube* and *CodeQL* have more limited recall, leading to greater variance and lower medians in their F1 score distributions.

GPT-4.1, *Mistral-Large* and *DeepSeek V3* achieve higher F1 scores than traditional tools such as *SonarQube* and *CodeQL*. Despite its high false-positive rate, *DeepSeek V3* performs well in terms of F1 score, suggesting that this compensates for its lower precision shown in Figure 3. It is worth mentioning that with the use of large language models, outliers can also be observed for *Mistral-Large* and *DeepSeek V3* in the *S04* project. These can be observed in Figure 7.

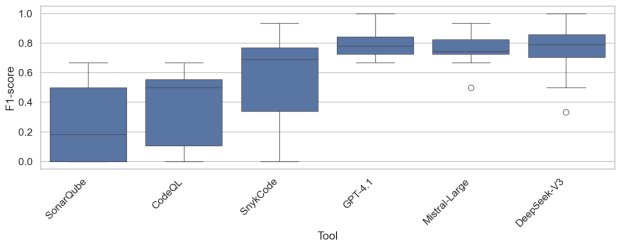

FIGURE 7. Comparison of Effectiveness (F1 score).

VI. DISCUSSION, LIMITATIONS, AND FUTURE WORK

One of the fields found in the SARIF schema is *region*, which reports the location of vulnerabilities found with line and column precision in a given file [10]. All of the large language models tested were unable to determine it correctly. All results correctly identified the file, but the incorrect location was always reported. Upon further analysis, no correlation was found in these errors. They are likely caused by one of the steps the text undergoes before processing: BPE tokenisation, which transformer-based models use [41]. This is a known and common problem. It can be a major limitation in building systems that require precisely determining error locations.

CodeQL achieved the worst results among all the static code analysis tools tested, in terms of the time required for analysis. This may be due to the intended use of this tool, which was designed for professional applications that use many predefined rules that the user can define. The need to load them into memory and process them can be a potential reason for these results.

A well-known problem that occurs when using large language models is hallucinations. Is it a situation where the generated response is deviated, based neither on training data nor input from the user [42], [43]. Such an answer only appears to be correct, but it contains false information. Table 14 contains examples of such findings. In each case, the latest available versions of the libraries are used in the analysed projects, but the models consider them obsolete. This situation cannot occur with classical code analysis tools, as they typically maintain a list of deprecated dependencies that is updated regularly.

When analysing the results, false positives and false negatives were considered. From the end user's point of view, false negatives are more dangerous, as they result in overlooking real vulnerabilities that could impact the system's functioning. However, it is worth noting that false positive results also have negative effects. The primary one is that developers pay less attention to the results of the analysis. As a consequence, this can lead to the ignored real issues [44]. False positives not only affect tool precision but also influence developer experience. Frequent, inaccurate alerts may erode trust in automated analyses and increase cognitive load, causing developers to overlook genuine warnings, an effect particularly relevant when deploying LLMs in continuous feedback environments.

TABLE 14. Hallucinated findings by project and model.

Project	Model	Rule ID	Message
S02	Mistral Large	VULNERABLE_LIBRARY	The version '9.0.5' of the 'Microsoft.Data.Sqlite' package is outdated. Consider updating to a more recent version.
S03	Mistral Large	WEAK_ALGORITHM	Using an outdated version of Microsoft.Data.SQLite library.
S06	GPT 4.1	OUTDATED_DEPENDENCY	The Newtonsoft.Json package is used (version 13.0.3). Ensure to update it against security vulnerabilities.

Each LLM's output was manually reviewed to assess developer effort—the time and number of steps required to verify a true positive. GPT-4.1 and Mistral-Large typically produced concise, human-readable rationales (2–4 lines) that accelerated validation, while DeepSeek V3 often produced verbose or speculative messages, increasing review time. In contrast, CodeQL requires developers to manually trace rule identifiers through documentation, resulting in higher verification overhead.

Beyond metric-level performance, the practical deployment of LLM-based vulnerability detection requires careful integration into existing CI/CD environments. In practice, organisations can incorporate LLM scanners as an early triage stage, automatically generating SARIF-compatible reports and human-readable rationales, before traditional SAST tools perform deterministic verification. Such a setup reduces missed vulnerabilities while containing false-positive overhead. Cost-benefit considerations depend on token volume, model pricing, and review effort, and should be balanced against the latency and precision of conventional tools. In hybrid pipelines, the qualitative explanations generated by LLMs can further assist human reviewers by providing justifications that enhance trust calibration and decision-making.

A. FUTURE WORKS

Although this thesis provides a comparative analysis of traditional static analysis tools, such as SonarQube, CodeQL, and SnykCode, and the large language models GPT-4.1, Mistral Large, and DeepSeek V3, in the context of vulnerability detection, several directions remain open for further exploration and development. One aspect worth exploring is prompt engineering, i.e. manipulating the content of queries that are sent to the models and their impact on the results received.

Another direction could be an attempt to create a hybrid solution that uses both classical tools and language models. These approaches can be used to prioritise and verify vulnerabilities found. This could help mitigate the weaknesses of both methods.

VII. CONCLUSION

After analysing all projects using the selected static code analysis tools and large language models, conclusions can be drawn. Large language models perform well in detecting code vulnerabilities, as seen from comparing the F1 score values shown in Figure 7. The ability to analyse the entire context of the code, as opposed to traditional tools that rely on defined patterns, can be considered the reason. It is also worth noting that the increased probability of reporting false positives comes with the greater ability to detect vulnerabilities. This is shown in Figure 4. Another vital factor to consider when using large language models as code analysis tools is their inability to pinpoint the exact location of the vulnerabilities they identify. When developing more elaborate code analysis systems, the failure to provide location information when using the SARIF format may preclude the use of large language models.

In conclusion, classical tools for static code analysis are recommended in situations requiring reliability and precision, such as software audits, which are necessary for systems that must be reliable and used in critical sectors. In such situations, defining specific patterns is often required. Solutions based on large language models have a higher sensitivity, but generate more false-positive results. They can be used during the development process to ensure that developers working on the system are aware of defects.

REFERENCES

[1] D. Ajiga, P. A. Okeleke, S. O. Folorunsho, and C. Ezeigweneme, "The role of software automation in improving industrial operations and efficiency," *Int. J. Eng. Res. Updates*, vol. 7, no. 1, pp. 22–35, Aug. 2024.

[2] D. Kastner, L. Mauborgne, and C. Ferdinand, "On software safety, security, and abstract interpretation," in *Computer Safety, Reliability, and Security*. Cham, Switzerland: Springer, 2018, pp. 662–665.

[3] R. D. Venkatasubramanyam, S. Gupta, and U. Uppili, "Assessing the effectiveness of static analysis through defect correlation analysis," in *Proc. IEEE 10th Int. Conf. Global Softw. Eng.*, Jul. 2015, pp. 100–104.

[4] D. Singh, V. R. Sekar, K. T. Stolee, and B. Johnson, "Evaluating how static analysis tools can reduce code review effort," in *Proc. IEEE Symp. Vis. Lang. Hum.-Centric Comput. (VL/HCC)*, Oct. 2017, pp. 101–105.

[5] D. Nikolić, D. Stefanović, M. Nikolić, D. Dakić, M. Stefanović, and S. Koprivica, "Uncovering determinants of code quality in education via static code analysis," *IEEE Access*, vol. 12, pp. 168229–168244, 2024.

[6] A. G. Bardas, "Static code analysis," *J. Inf. Syst. Oper. Manage.*, vol. 4, no. 2, pp. 99–107, 2010. [Online]. Available: https://www.ittc.ku.edu/~alexbardas/

[7] J. Novak, A. Krajnc, and R. Žontar, "Taxonomy of static code analysis tools," in *Proc. 33rd Int. Conv. (MIPRO)*, 2010, pp. 418–422.

[8] M. C. Fanning and L. J. Golding, "Static analysis results interchange format (SARIF) version 2.1.0," OASIS Committee Specification 01, Jul. 2019. [Online]. Available: https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html

[9] S. Kummita and G. Piskachev, "Integration of the static analysis results interchange format in CogniCrypt," 2019, *arXiv:1907.02558*.

[10] SchemaStore Contributors. (2020). *SARIF JSON Schema (Version 2.1.0)*. Accessed: May 28, 2025. [Online]. Available: https://json.schemastore.org/sarif-2.1.0.json

[11] G. Liang and Q. Wang, "CODAS: An extensible static code defect analysis service," *Comput. Sci.*, vol. 39, no. 1, pp. 14–18, 2012.

[12] I. R. da Silva Simões and E. Venson, "Evaluating source code quality with large language models: A comparative study," 2024, *arXiv:2408.07082*.
[13] B. Ahmad, H. Pearce, R. Karri, and B. Tan, "LASHED: LLMs and static hardware analysis for early detection of RTL bugs," 2025, *arXiv:2504.21770*.
[14] T. Szandała, "ChatGPT vs human expertise in the context of IT recruitment," *Expert Syst. Appl.*, vol. 264, Mar. 2025, Art. no. 125868.
[15] W. R. Nichols, "The cost and benefits of static analysis during development," 2020, *arXiv:2003.03001*.
[16] M. Alqaradaghi and T. Kozsik, "Comprehensive evaluation of static analysis tools for their performance in finding vulnerabilities in Java code," *IEEE Access*, vol. 12, pp. 55824–55842, 2024.
[17] SonarSource. (2025). *Languages Overview—SonarQube Documentation*. Accessed: May 28, 2025. [Online]. Available: https://docs.sonarsource.com/sonarqube-server/10.8/analyzing-source-code/languages/overview/
[18] SonarSource. (2024). *Sonar Extends Code Security Coverage With SonarQube Advanced Security*. Accessed: May 28, 2025. [Online]. Available: https://www.sonarsource.com/company/press-releases/sonar-extends-code-security-coverage-with-sonarqube-advanced-security/s
[19] SonarSource. (2025). *Pricing*. Accessed: May 28, 2025. [Online]. Available: https://www.sonarsource.com/plans-and-pricing/
[20] SonarSource. (2025). *Data Privacy*. Accessed: May 28, 2025. [Online]. Available: https://www.sonarsource.com/company/privacy/
[21] GitHub. *About Code Scanning With CodeQL*. Accessed: May 28, 2025. [Online]. Available: https://docs.github.com/en/code-security/code-scanning/introduction-to-code-scanning/about-code-scanning-with-codeql
[22] GitHub. (2025). *About Code Scanning With CodeQL*. Accessed: May 28, 2025. [Online]. Available: https://github.com/pricing
[23] GitHub. (2025). *GitHub General Privacy Statement*. Accessed: May 28, 2025. [Online]. Available: https://docs.github.com/en/site-policy/privacy-policies/github-general-privacy-statement
[24] Snyk. (2024). *Snyk Code Documentation*. Accessed: May 28, 2025. [Online]. Available: https://docs.snyk.io/scan-with-snyk/snyk-code
[25] R. Maira. (2020). *Developer Fist SAST With Snyk Code*. Accessed: May 28, 2025. [Online]. Available: https://snyk.io/blog/developer-first-sast-with-snyk-code/
[26] Snyk. (2025). *Plans and Pricing*. Accessed: May 28, 2025. [Online]. Available: https://snyk.io/plans/
[27] Snyk. (2025). *How Snyk Handles Your Data*. Accessed: May 28, 2025. [Online]. Available: https://github.com/snyk/user-docs/blob/main/docs/working-with-snyk/how-snyk-handles-your-data.md
[28] OpenAI. (2025). *API Pricing*. Accessed: May 28, 2025. [Online]. Available: https://openai.com/api/pricing/
[29] OpenAI. (2025). *GPTs Data Privacy FAQs*. Accessed: May 28, 2025. [Online]. Available: https://help.openai.com/en/articles/8554402-gpts-data-privacy-faqs
[30] OpenAI. (2025). *Introducing GPT-4.1 in the API*. Accessed: May 28, 2025. [Online]. Available: https://openai.com/index/gpt-4-1/
[31] DeepSeek. (2025). *Models & Pricing*. Accessed: May 28, 2025. [Online]. Available: https://api-docs.deepseek.com/quick_start/pricing
[32] DeepSeek. (2025). *DeepSeek Privacy Policy*. Accessed: May 28, 2025. [Online]. Available: https://cdn.deepseek.com/policies/en-U.S./deepseek-privacy-policy.html
[33] DeepSeek. (2025). *DeepSeek V3 Model Card*. Accessed: May 28, 2025. [Online]. Available: https://www.prompthub.us/models/deepseek-v3
[34] Mistral AI. (2025). *Pricing*. Accessed: May 28, 2025. [Online]. Available: https://mistral.ai/pricing#api-pricing
[35] Mistral AI. (2025). *Terms of Service*. Accessed: May 28, 2025. [Online]. Available: https://mistral.ai/terms#terms-of-service
[36] M. Islam, J. Ali Khan, M. Abaker, A. Daud, and A. Irshad, "Unified large language models for misinformation detection in low-resource linguistic settings," 2025, *arXiv:2506.01587*.
[37] M. Islam, T. Huang, E. Ahn, and U. Naseem, "Multimodal generative AI with autoregressive LLMs for human motion understanding and generation: A way forward," 2025, *arXiv:2506.03191*.
[38] M. Ferreira, D. Gonçalves, M. Bigonha, and K. Ferreira, "Characterizing commits in open-source software," in *Proc. 21st Brazilian Symp. Softw. Qual.*, Nov. 2022, pp. 1–10.
[39] GitHub. (2025). *GitHub Models Marketplace*. Accessed: May 28, 2025. [Online]. Available: https://github.com/marketplace?type=models
[40] Microsoft. (May 2025). *Microsoft.extensions.ai Libraries*. Accessed: Jun. 5, 2025. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/ai/microsoft-extensions-ai
[41] Y. Chai, Y. Fang, Q. Peng, and X. Li, "Tokenization falling short: On subword robustness in large language models," 2024, *arXiv:2406.11687*.
[42] Y. Bang, Z. Ji, A. Schelten, A. Hartshorn, T. Fowler, C. Zhang, N. Cancedda, and P. Fung, "HalluLens: LLM hallucination benchmark," 2025, *arXiv:2504.17550*.
[43] T. Szandała, "Aiops for reliability: Evaluating large language models for automated root cause analysis in chaos engineering," in *Proc. Int. Conf. Comput. Sci.* Cham, Switzerland: Springer, 2025, pp. 323–336.
[44] F. Cheirdari and G. Karabatis, "Analyzing false positive source code vulnerabilities using static analysis tools," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2018, pp. 4782–4788.

DAMIAN GNIECIAK is currently pursuing the master's degree in computer science with WIT. He is a Software Engineer with a professional focus on cloud technologies. He is developing a specialization with Microsoft Azure. He is particularly interested in the design and implementation of scalable, secure, and highly available solutions.

TOMASZ SZANDALA received the Ph.D. degree in computer science, in 2022. He is currently a Postdoctoral Researcher with the Scuola Universitaria Professionale della Svizzera italiana, Lugano, supported by an ESKAS scholarship. His academic work centers on applied computer science and DevOps. In addition to academic pursuits, he is a Certified DevOps Engineer with expertise in Kubernetes and Google Cloud Platform, frequently publishing industrial articles that bridge theoretical and practical insights across his academic and professional experience.