

DOI: https://doi.org/10.48009/4_iis_2026_109

Lightweight AI code review process for cybersecurity enhancements

Thomas Lombardi, *Eckerd College*, lombardite@eckerd.edu

N. Brice Orange, *OrangeWave Innovative Science, LLC/Etelman Observatory Research Center*,
University of the Virgin Islands, orangewaveno@gmail.com

Bruce Gendre, *College of Science and Math, University of the Virgin Islands*, bruce.gendre@uvi.edu

Abstract

This paper documents a lightweight AI-assisted code review process for enhancing cybersecurity that is suitable for small and medium enterprises with significant financial and operating constraints. The case study outlines the technical and process details of our near zero-cost, virtualized solution for incorporating AI into the software development process for a small application in Python. The AI code review tool depends on Codellama running in a Docker container small enough to fit on a virtual machine. The AI-assisted code review tool was integrated into a development workflow with git precommit hooks to enhance code quality and security. The evolutionary approach permitted experimentation in small incremental steps that respect the constraints of the working environment while creating data points to support decision making. The study demonstrates the important role that prompt design and task decomposition play in generating high quality code reviews with AI especially when highly constrained. In particular, prompting by goal and prompting per file proved most effective in generating quality feedback.

Keywords: artificial intelligence, LLM, cybersecurity, code review, SME

Introduction

Large Language Models (LLM) pose a unique set of opportunities, threats and challenges to institutions throughout the world. The threats and challenges are particularly pronounced in the field of cybersecurity where new AI tools increase risks at every institution. At the time of writing, Anthropic's model, Mythos, is causing panic among large institutions from banks to intelligence agencies. In response to the widespread concern, Anthropic (2026) has delayed the public distribution of its model until the largest infrastructural institutions have had time to review the model, its capabilities, and the vulnerabilities exposed in their software and systems. If the largest institutions in the world cannot keep up with the cybersecurity risk inherent in LLM developments, where does this leave small and medium enterprises (SME) with fewer resources and personnel?

This study outlines one research facility's answer to this question. From the cybersecurity perspective, research facilities often share similar vulnerabilities to SMEs in the education and public health sectors. Threat agents attack educational, medical, and research facilities recognizing that such soft targets often lack the resources and personnel to prevent attacks or respond effectively. High profile security breaches at NSF-funded research facilities (Zhao & Gupta, 2023) highlight the stakes for research communities. In the

words of Luis Welbanks, a researcher at Arizona State University: “We’re lucky enough to make it through a regular night ... But now we need to consider the cybersecurity implications” (Zhao & Gupta, 2023). In this paper, we share the experiences of such a research facility working to incorporate AI into its development and cybersecurity processes as a case study in how SMEs can participate in the expected benefits of LLM technology without being overwhelmed by its challenges and threats.

Located in the U.S. Virgin Islands, the University of the Virgin Islands' (UVI) Etelman Observatory (EO) is the home of the Virgin Islands Robotic Telescope (VIRT), a 0.5 meter research-grade telescope that fills an observational void between mainland U.S. and European observatories (Morris, 2020). The EO and VIRT were designed in the early 2000s to support the science return of Swift (Gehrels et al., 2004) detected Gamma-ray bursts (GRB) (Giblin et al., 2003), some of the most powerful explosions in the universe (Klebesadel et al., 1973). During the past decade, EO has transformed from a dormant facility with a non-operational telescope and near-zero research output, into a vibrant astrophysics research program at UVI (Orange et al., 2021). EO now regularly reports GRB follow-up results from VIRT to the General Coordinates Network (GCN) (Gokuldass et al., 2021), publishes scientific research papers (Gendre et al., 2022, Giblin et al., 2023, Moore et al., 2024) as well as research papers in information systems (Lombardi et al., 2021), and provides rich opportunities for UVI students and faculty to pursue research (Orange & Morris, 2021). The cybersecurity challenges EO faces threaten the benefits the research facility offers to its scientific and local communities.

The research facility delivers its services in a context that is crucial for understanding the proposed strategy for incorporating AI into the software development process. First, the facility seeks to minimize cost as measured in dollar investment and personnel time. Since public funding provides the overwhelming bulk of the facility’s operating budget, the staff attempt to reduce cost as much as possible. Second, the solution had to respond to the needs of a small, part-time staff working remotely. Third, the facility is currently working toward the goal of increased automation. Fourth, the solution had to meet rigorous demands of an environment at odds with stable operation including challenges like frequent power outages and procurement problems. These constraints and requirements shaped the AI-component developed to enhance our code review and security audit processes.

The study’s approach addresses an important gap in the literature regarding how SMEs can effectively adopt AI technologies. Oldemeyer, Jede, and Teuteberg (2025a) highlight the large enterprise focus of current AI research in their systematic review of artificial intelligence in SMEs. Their work identifies 27 distinct challenges SMEs encounter in implementing AI solutions with lack of knowledge and high cost serving as the number one and two challenges (Oldemeyer, Jede, & Teuteberg, 2025a). In a related study, the same research team identified the evaluation of the cost/benefit ratio of AI adoption as a key challenge for SMEs (Oldemeyer, Jede, & Teuteberg, 2025b). In their discussion, the authors explicitly recommend “further case studies in SMEs on low-cost AI applications” (Oldemeyer, Jede, & Teuteberg, 2025b).

This paper documents one such case study, a lightweight AI-assisted code review process for enhancing cybersecurity, that is suitable for SMEs with significant financial and operating constraints. The study outlines the details of a near zero-cost, virtualized solution for incorporating AI into the software development process. After outlining the main contributions of this paper, we provide a review of the literature that guided the decision-making process, its research questions, and case study methodology. As will be explored below, the facility currently has no data suitable for making the complex decisions about how to incorporate AI into the development workflow. An evolutionary approach permits experimentation in incremental steps that respect the constraints of our working environment while creating data points to support decision making. After presenting the results, we discuss the degree to which the results apply to other institutions in the Discussion, Limitations, and Conclusion sections.

Main Contributions

This study contributes several types of knowledge to the research community in the area of AI-assisted code review. First, the case study integrates research from several areas of LLM research including AI adoption in SMEs, code quality, security audits, code reviews, and cost optimization. Second, the study establishes the feasibility of integrating LLMs into a code review and security audit process at near zero cost. Since many institutions grappling with AI integration must respond to financial and operating constraints, this case study responds to the practical needs of smaller enterprises. Third, the study applies AI-assisted code review to a real-world project in production rather than code snippets from a curated database. Our findings indicate that task decomposition improved the quantity and quality of feedback. Furthermore, because it monitors something public and non-proprietary, the facility can share more than most projects could share. We offer this study as a kind of tactical guide to other SMEs seeking to navigate the terrain beyond the dragons on the LLM map.

Background and Literature Review

While developers have quickly adopted AI for coding, they have adopted AI for security audits and code reviews at a much slower pace. According to the 2023 CodeSignal Trends Report (2023), 76% of developers use AI to learn technical skills and knowledge and 53% use AI to generate boilerplate code while only 15% of developers use AI to check for vulnerabilities. More recently, the 2025 Developer Survey (Stackoverflow, 2025), highlights similar trends with 58.7% of developers showing resistance to using AI for committing and reviewing code. The observed reticence of developers to apply AI to vulnerability assessment and code review generally motivate this study.

Among the possible explanations for the slow adoption of AI-assisted code review is the gap between the perception of these tools and the reality of using them. Amro & Alalfi (2025) expose the gaps between the perceptions of AI-assisted code review with Copilot and its actual performance in securing code. They ultimately conclude that manual code audits are necessary for ensuring software security. Pearce et al. (2025) came to a similar conclusion regarding the security of Copilot's generated code concluding that "...while Copilot can rapidly generate prodigious amounts of code, our conclusions reveal that developers should remain vigilant ('awake') when using Copilot as a co-pilot. Ideally, Copilot should be paired with appropriate security-aware tooling during both training and generation to minimize the risk of introducing security vulnerabilities." In his study of the perception of AI-driven code reviews, Cataldi (2025) summarizes his engagement with developers in the following terms: "this research provides evidence supporting a hybrid model, emphasizing the complementary role of AI-driven code review tools and human expertise." Broadly speaking, several studies suggest that developers are justified in their hesitancy to remove human judgement from the code review process.

To help bridge the gap between LLM perception and performance, this research integrates ideas from several topics in LLM research including code reviews, code quality, code security, and cost considerations. Both reason and empirical evidence support the notion that code reviews promote code quality. In their recent survey of the literature, Badampudi, Unterkalmsteiner & Britto (2023) documented many studies on the impact of code review on software quality. Their findings suggest a broad consensus on the effectiveness of code review in defect detection, fixing security issues, and improving software design. In one such study, Thompson and Wagner (2017) developed linear regression models to investigate the relationship between code review and code security and demonstrate that, overall, code review decreases the number of bugs. Given this broad consensus and the empirical evidence supporting the efficacy of code review, it is no surprise that researchers have addressed the role AI can play in the code review process.

The research regarding AI-assisted code review addresses a diverse set of issues including prompt engineering, small language models, fine-tuning models, and source code summarization. First, researchers have explored the role of prompting style in the quality of AI-assisted code reviews. Salih and Sarhan (2025) rely on structural (Liu, Yang & Liao, 2024), chain-of-thought (Li et al, 2023), and role-based (Hajipour et al., 2024) prompting strategies in their evaluation of Python code vulnerabilities across several models. Their study concludes that structural prompting produced the most reliable outcomes across all models. Many researchers also differentiate prompts based on the number of examples provided to the LLM: zero-shot, one-shot, few-shot (Sasaki et al., 2025). Second, several studies suggest that small language models (SLM), those with 1 billion parameters or fewer (Bappy et al., 2025), have a part to play in AI-assisted code reviews especially since these models reduce cost because they run well on less powerful hardware. One team demonstrated that some SLMs like Mistral are capable of reliably detecting some code smells in Python without fine tuning (de Oliveira et al., 2025). Third, researchers have addressed the role of tuning in quality feedback. For example, Bappy et al. (2025) report a significant improvement in the codegen-350M-mono model's ability to perform Common Weakness Enumeration detection in Python after fine tuning. Fourth, researchers are studying how best to augment source code summarization with LLM. Sun et al. (2025) for example performed a systematic study on code summarization across multiple languages and models. Among their findings was the counterintuitive observation that advanced prompting techniques do not necessarily perform better than zero-shot prompting in code summarization tasks.

The effective, long-term implementation of AI-assisted code reviews requires that such solutions are financially sustainable. Bender et al. (2021) recommend that researchers weigh the environmental and financial costs first when designing and developing LLMs. In particular, they highlight how many researchers focus on improvements in accuracy as primary contributions, to the detriment of other important goals like efficiency, low cost, and reduced energy consumption. More recent studies have pursued the suggestion of Bender et al. (2021). For example, Pan et al. (2025) demonstrate that on-premise deployments of LLMs are economically viable with break-even periods within 3 months for SLMs bringing language models within the reach of smaller organizations. Howell et al. (2023) present a framework for evaluating an NLP model's utility, expected net cost savings, comparing prompt engineering, fine-tuning, and knowledge distillation as strategies for optimizing an LLM. They conclude that smaller models are often more efficient albeit lower quality because their inference costs are much lower. Aryan et al. (2023) argue that enterprises should assess LLMs based on the degree to which those models are generalizable, evaluable, and cost-optimal prior to significant investment. In the explanation of their proposed framework, they highlight the challenges of estimating cost in LLMs: "The full cost profile of LLMs is currently unknown and requires future work and data to determine."

Research Questions

The study addresses the following research questions related to AI-assisted code review for cybersecurity.

RQ1: What cost-free tools and methods exist for implementing and improving AI-assisted code review?

Several research studies summarized in the Background and Literature Review section identified cost-effective strategies for building sustainable LLM solutions.

RQ2: What is the minimal implementation and cost required for an AI-assisted code review?

Taking the advice of Aryan et al. (2023) to heart, we develop a minimal and flexible implementation of an AI-assisted code review at near zero cost.

RQ3: How does prompt engineering affect AI-assisted code review quality and security?

Since prompt engineering is broadly studied and costs nothing, the case study attempts to isolate prompting approaches that enhance code review quality and security in the codebase.

RQ4: How does the AI-assisted output contribute to code quality and security?

This research evaluates the responses of the AI-assisted code review to determine where and when the feedback can improve code quality and security.

RQ5: How do the AI-assisted code review contributions compare to that of static security tools?

Static analysis tools in precommit hooks provide an important benchmark against which to measure our AI-assisted code review responses.

Methodology

To pursue these research questions, we adopt a single case study methodology for both description and exploratory analysis (Ebneyamini & Sadeghi Moghadam, 2018) based primarily on the prompts to and responses from the Codellama LLM. The approach outlined below is suited to cases of technology management where the goal is to evaluate decisions such as when and how to adopt AI into business processes. As a part of the software development process, we generate several types of data serving as source material for our study including AI-generated code review reports from several LLMs, static code analysis tool output, and the evolution of the source code itself. These primary data sources are augmented with secondary sources including the National Vulnerability Database (NIST, n.d.), public records of coding projects in Github (<https://github.com/>) and Pypi (<https://pypi.org/>), and publicly-available technical documentation. Practically, these sources served to direct the software development process as we restructured, refactored, and improved the code. Theoretically, each software commit generated data constituting a consistent framework for experimentation with AI-assisted code review.

The implementation of the case study methodology required a re-engineering of the software development process. First, we created a VirtualBox virtual machine (VM) running the Fedora 43 (64-bit) operating system with 8 GB RAM, 100GB Disk, and no GPU. The virtual machine runs on a modest Dell Laptop (Host) running Windows 11 Enterprise version 25H2 with an 11th Gen Intel(R) Core(TM) i5-11500H @ 2.90GHz, 32 GB RAM, and 1 TB of storage. The team prepared the VM with the necessary git configuration, software including Python 3, and required libraries. We opted to run codellama:latest with 7 billion parameters in a Docker container with Ollama version 0.19.0-rc0 (Ollama, n.d.). Finally, the source code was modified to include git precommit hooks for code formatters (Black, isort), linters, and code quality tools (Flake8, Pylint, Ruff, Mypy, Vulture), security audits (pip-audit, Bandit, Semgrep), and the AI-code-review tool. Table 1 in the appendix summarizes the precommit hooks and associated tools implemented for this study. In the case of commercial products like Semgrep, the project relies on community, free, or demo versions. The technology supporting these changes to the software development process incurred no costs beyond those required for a modest Host laptop.

The AI-assisted code review runs in one of two different methods. When code is updated and committed, the precommit hooks including the AI-code review execute automatically generating reports for review. If one of the checks fails, the code commit will fail. Currently, the AI-code review only generates data and does not trigger a commit failure. Alternatively, when the team wishes to review the entire code base, a separate script executes all precommit hooks and generates an AI-code review for each file. Primarily, we examined the reports generated from the second method that reviews every file in the code base. The security reports provide the following information related to security:

- The specific prompt submitted to Codellama
- Bandit security scanner results
- Semgrep security scanner results
- pip-audit security scanner results
- The Codellama response to the prompt for each Python file in the codebase
- The total execution time in seconds

The reporting framework sketched above facilitated the rapid collection of data to support our case study and investigate our research questions. Over the course of the development cycle, we adjusted our prompt strategies to test several aspects of the AI-assisted code review process. We focused on zero-shot prompts employing structural, role-based, and chain-of-thought prompts to identify the relationship between prompt input and code review output. The study also examined the effects of bundling the code into a single request (prompt per project) as opposed to submitting a prompt per file. The team experimented with a prompt per goal strategy as well. Finally, we compared the output from Codellama to the static reports produced by our precommit toolset. The Results section documents our findings for each research question in detail.

Results

The software development process outlined in the Methodology section produced 279 independent code and security reviews exploring a variety of scenarios and prompting strategies. Additionally, the team ran code reviews at the beginning and end of the cycle with freely available models in OpenRouter as a kind of external validation of our results.

RQ1: What cost-free tools and methods exist for implementing and improving AI-assisted code review?

To support the development of a working prototype, the study had to identify the relevant variables for implementing and improving an AI-assisted code review tool. These variables represent decisions teams must make regarding how to incorporate AI into their software development processes. In a sense, this study focuses on the available decisions that carry no or extremely low cost. The literature identifies many no-cost tools and methods for implementing prototype systems like that developed for this study.

The choice of a no-cost AI model is the first significant variable to consider. Openrouter.ai provides a long list of free AI models with a wide variety of properties (OpenRouter, 2026). Many other repositories provide similar options including Docker Hub and Ollama. This study adopts a strategy (Lopez, 2024) based on codellama:latest a 3.8 GB model with 7 billion parameters due to its relative simplicity, small size, flexibility, and clear documentation. Closely related to the choice of AI model is the method of accessing the model. We opted for a local containerized model with Codellama due to its stability. For our project, the availability of the Codellama container outweighed the benefits of other options. Of particular interest was the ease with which the process could be captured in a git precommit hook. The AI-assisted code review could simply be added to the code formatters, quality tools, and security scanners in our software development process.

After selecting an AI model, a team will need to decide what and how to communicate with the model. Naturally, the team must turn to prompt engineering decisions like prompt size, the number of examples to provide in the prompt, and the overall prompt strategy. Prompt engineering is a vibrant area of research filled with taxonomies (Sasaki et al., 2025) and prompt engineering libraries (<https://www.promptingguide.ai/>). Here we catalog a few decisions teams must make while adapting these tools to their development workflow. Longer prompt sizes will cost more than shorter prompts in general and, therefore, the length of the prompt becomes a significant issue with no cost and yet great consequence for effectiveness (Chen, Zaharia & Zou, 2023). Just as the size of the prompt matters, so does the number of examples provided in the prompt with zero-shot, one-shot, and few-shot prompting strategies indicating the number of examples provided in the prompt. Yuen et al. (2025) demonstrated that regarding the task of generating class code few-shot outperformed zero-shot prompting. Other researchers focus on the design of the prompt (Salih & Sarhan, 2025) referring to structural, chain-of-thought, and role-based prompting. Researchers have demonstrated that the prompt design matters to the performance of AI-assisted code review (Guo et al., 2023; Salih & Sarhan, 2025).

While prompt engineering is a crucial cost-free technique for improving AI-assisted code reviews, other choices also matter such as task decomposition. At what grain of detail should we prompt in the first place? Our team identified the following scopes of interest: prompt per project, prompt per goal (security, code quality), prompt per group of files, prompt per file, prompt per method, and prompt per snippet or statement. Research studies have confirmed that the scope of the request matters for the results. For example, Mhatre et al. (2026) conclude that LLMs performed well on code snippets containing beginner level bugs, but struggled with complex security vulnerabilities in production-scale codebases. Similarly, Salih & Sarhan (2025) found that LLMs identified coding violations better when code snippets were processed individually rather than in batches. In other words, teams must decide on the appropriate resolution or scope of the request ranging from a single prompt per project to a prompt per statement or snippet. Overall, the research suggests that task decomposition is an important cost-free way to improve the results of AI-assisted code reviews.

In our research, we also identified areas deemed too expensive given our operational constraints. First, we identified training an LLM from scratch as beyond the reach of our financial and operational constraints. Cost estimates for running LLMs are extremely high. Chen, Zaharia & Zou (2023) review these costs and summarize their findings in these terms: “The substantial cost of employing LLMs in real-world scenarios presents a considerable barrier to their widespread usage.” Given these costs, practitioners and researchers are searching for cost-efficient techniques like fine-tuning smaller models. For example, Bappy et al. (2025) demonstrate significant performance increases in fine-tuning a small model for CWE detection in Python code. Other tuning options exist as well, but the concern with these approaches revolves around the time required for experimentation and maintenance. If a team needs to fine-tune a model frequently to keep up with new versions, then the solution will not be sustainable.

In summary, our team adopted the following practices to keep our AI-assisted code review as close to zero cost as possible. The prototype relies on an AI model (codellama:current) that runs in a Docker container with no required fine tuning. Furthermore, the prototype submits prompts programmatically while processing precommit hooks per file. The code review is further decomposed into two goal-specific queries: a code quality prompt and a cybersecurity prompt. That is, for each file two prompts are submitted, one pertaining to code quality generally and another specifically designed to elicit cybersecurity analysis. The model permitted flexible ways to experiment with prompt strategies including structural, chain-of-thought, and role-based. Given these choices, the team sought to verify that such a lightweight approach was feasible practically.

RQ2: What is the minimal implementation and cost required for an AI-assisted code review?

The study addressed these cost questions with two measurements: execution efficiency and cost in U.S. Dollars. Our approach demonstrates that an AI-assisted code review is possible in terms of cost. Basically, any low-cost commercial computer can run the free and open source software required for the implementation. The currency cost, however, does not provide a practical answer to our minimal implementation question: what happens if the required software cannot complete a meaningful analysis within a useful timeframe? The study demonstrates that our no-cost solution produces results in a relatively short time that did not burden our software development process. Table 2 (appendix) captures the descriptive statistics of the code review execution times for each Python file in the codebase. Bold entries in the Max (sec) column mark statistical outliers three standard deviations above the mean.

The descriptive statistics reveal that even on modest hardware the AI-assisted code review is not unreasonably demanding in terms of efficiency. The maximum execution time for the entire codebase run as a single bundle was 4,743 seconds, roughly 1 hour and 20 minutes. According to Pylint's most recent report, the code base includes 536 lines including 352 lines of code, 77 lines of docstring, 6 comments, and 101 empty lines. The most complex of the files, `gcncirculars.py`, required from 481 to 3,364 seconds or from just over 8 minutes to just under an hour, and a typical time for this file is about 20 minutes. In other words, the amount of time required for this process fit nicely into the software development process. Waiting 20 minutes for a file during the commit process was considered acceptable. Moreover, the longer execution time for running the entire codebase fits comfortably into a nightly process run after hours. In other words, running an inexpensive solution to AI-assisted code review was practically feasible as a function of time.

RQ3: How does prompt engineering affect AI-assisted code review quality and security?

The findings indicate that task decomposition had an important effect on the specificity and quality of Codellama's responses. Consider for example the question of whether to submit all files in a single request (project-level prompting) or a prompt per file (file-level prompting). For this case, the quantity and quality of feedback was better for file-level prompting. In terms of quantity, Codellama produced a single page of feedback for all files when submitted as a single request while producing the same amount of feedback per file with file-level prompting. Similarly, the quality of the feedback was more specific to the code and more useful with file-level prompting. The evidence of this case suggests that file-level prompting is likely to produce results more easily incorporated into a code review process.

The review type (prompt by goal) also demonstrates the importance of task decomposition. Table 3 (appendix) presents the data for the top references across all prompts broken down by review type. Code reviews request generic code quality feedback without specific mention of security. Security reviews explicitly ask for security feedback only. Combined reviews include requests for code quality and security assessments. Even in the list of most commonly reviewed tools and libraries, the evidence suggests that creating separate prompts for code quality and security elicits different feedback. For example, it is not surprising that Pylint and RegEx are more often mentioned in a generic code quality review than in a specific security review.

The effect of decomposing prompts by goal is more pronounced when examining security protocols and tools specifically. Table 4 (appendix) highlights some of the most common tools referenced in the LLM reviews. HTTPS and OAuth are heavily weighted toward security reviews while the Python library secrets and the Bandit security scanner are only mentioned in security reviews. The most important takeaway from this evidence is that the scope (project-level vs file-level) and goal (prompt by goal) of a prompt matter a great deal to the type of feedback received and provide an effective way to improve AI-assisted code reviews within a low-cost or no-cost framework.

The specific content of the prompt also matters when considering what not to request. Nowhere was this more evident in the case study than in specific requests for Common Vulnerabilities and Exposures (CVE). Table 5 (appendix) presents the evidence related to four prompts that explicitly requested CVE information in the response. Three of the four prompts produced many references to what looked like properly formatted CVEs. In all 50 cases, however, the CVE reference did not exist, carried the rejected status in the NVD database, or simply was not aligned to the issue at hand. Based on this case study, we recommend that those wishing to incorporate LLMs into a low-cost AI-assisted code review process spend significant time focused on task decomposition as a low-cost way to boost feedback quality.

RQ4: How does the AI-assisted output contribute to code quality and security?

The most important contribution the AI-assisted code review provided to enhance our security was its knowledge of tools. In general, Codellama demonstrated clear knowledge of common tools, libraries, and concepts related to Python code quality and security. The case study identified 125 terms examined programmatically and 164 terms reviewed manually. Of these references, we found little evidence of tool or library confusion. In fact, Codellama provided useful information regarding libraries to improve our code security. For example, the LLM suggested a number of libraries for encryption including argon2 and bcrypt. Moreover, since the codebase relies on regular expressions to parse messages of interest, Codellama suggested re2 as a library that prevents regular expression denial-of-service attacks. The second contribution of note is the LLM's ability to summarize and document the code. In many instances the LLM produced code summaries that the team adopted into the documentation. The third contribution is the awareness that even when the suggestions are generic such as "improve documentation", they are still valuable in the same way that a calendar reminder is valuable. If nothing else, the reviews remind us to check the code against style guides like PEP-8 and review common security practices. Therefore, the case demonstrates that AI-code review assistants provide opportunities for learning about tool and library options, summarizing code and improving documentation, and enforcing code standards at very low cost.

Despite these benefits, the AI-assisted output exhibited several well-known problems. The responses often demonstrated a bias toward web and cloud applications rather than standalone applications. In many responses, Codellama assumed that users would interact with our software via a website. For example, Codellama suggested the following tools and libraries in its review: flask, django-csp, and mod_security2. Similarly, the output assumed that we deployed our software in the cloud rather than on premises suggesting tools like S3 bucket, Google Cloud Key Management Service, HashiCorp Vault, Azure Key Vault and AWS Secrets Manager. The team found a similar problem with the coding language itself: Codellama recommended libraries from languages other than Python. For example, FindSecBugs, Apache commons-logging, log4j, and logback serve Java code bases. These biases toward web-applications deployed in the cloud suggest that the feedback provided by any AI-assisted code review tool must be scrutinized and critically assessed prior to widespread automated use. In fact, the team suggests that human review is crucial to successfully incorporating this technology into a code review process.

The documented biases result from Codellama's training input and our prompting strategy. With time, experience, and experimentation, the prompting strategy can reduce some of the irrelevant output found in these reports. In other cases, however, Codellama is clearly hallucinating. For example, Codellama suggested the non-existent secure_logging library to address a logging vulnerability in our listener.py code. Despite the fact that it previously recommended loguru, logbook, and structlog for improving logging, the LLM still produced a reference to a non-existent library. In the same file review, Codellama recommended the secure_coding_practices library to address a lack of secure coding practices. In the worst examples, Codellama suggested coding solutions that did not respond to our code at all. For example, in a review of append_sheet.py, code designed to log data to a Google sheet, the suggested code included references to a

database not used (sqlite3), a library not used (requests), and an authentication method not used (HTTP Basic Auth), not to mention 3 CVEs not related to the real code or the hallucinated code in the response.

RQ5: How do the AI-assisted code review contributions compare to that of static security tools?

Based on this case study, LLMs can play a part in a robust security review provided that human oversight guides the review process and other tools help to construct guardrails for the AI. The previous research questions make clear that AI is hardly fool proof and since AI has difficulty incorporating context effectively, the team learned that static security tools like those outlined in Table 1 (appendix) are crucial for ensuring that an AI-assisted code review stays within reasonable guardrails. No example makes this clearer than the case of software supply-chain problems. pip-audit protected the code from supply-chain problems while the team could not find a single useful recommendation for this in any AI-generated report. This case highlights the importance of using an AI process as an addition rather than a substitution in a security review.

Static tools also provide important checks from a system perspective. Given current practices, the team could not expect the AI-assisted code review to identify problems in our containerized deployment. For example, prior to the security review, we launched the Docker container for the application as a root user. It is not realistic for the LLM to identify such problems. Rather it was Semgrep that identified the problem quickly and easily. The static tools also provided ways to cross check the output of the AI. For instance, if the AI reports an unusually long line, we know we can disregard this because the Black formatter simply won't allow it. In other words, certain errors and failures are easier to detect with static code review tools and the case study demonstrates the importance of their use.

Discussion

Based on our results, SMEs seeking to incorporate AI into their business should consider focusing on software development processes like code reviews and cybersecurity reviews. First, the case study demonstrates a highly customizable strategy for a near zero cost solution. Second, the proposed solution provides rich sources of data for improving decision making about the future direction of the deployment. Third, once implemented, the solution can be scaled and adapted in new ways as the SME develops.

SMEs often express concern about the costs of AI implementations. Teams will require time to learn the technology, its strengths and weaknesses, and how to adapt it for their code review and cybersecurity practices. The benefit of starting with AI-assisted code review is that it requires little effort or cost to start the learning process. The required technology can be acquired for free and implemented fairly quickly. The startup cost is extremely low for the ability to incorporate the power of AI into a business process.

The data provided by this process will enhance a SME's decision-making abilities. Once implemented and integrated into the code review process, the AI-assisted code reviews produce data to support decision making about the technology and the codebase. In this case study, Codellama was much more reliable regarding tools and libraries than it was for CVEs. Moreover, through careful task decomposition the team learned that prompt per file and prompt per goal proved more effective than other approaches available at the time. In future work, the team will curate unique prompts per file and goal to increase context awareness, response quality, and specificity. The team expects to curate prompts related to files in much the same way that developers curate test scripts.

SMEs can grow into their implementation as their experience with AI-assisted code review grows. For example, studies demonstrate that the techniques outlined here should translate effectively to other

programming languages. Mhatre et al. (2026) demonstrated that ChatGPT-4, Claude 3, and LLaMa 4 can reliably identify syntactic and semantic issues in well-scoped code in Python and C++. Furthermore, AI code reviews can extend to continuous integration/continuous deployment (CI/CD) pipelines for a price.

Overall, our team believes that the solution outlined here will benefit many SMEs as they adapt their business processes to the growing capabilities of AI. In future work, we plan to mine the reports more effectively and extend the approach to other programming languages like C++. Practically this means the reports need to be structured to enhance data mining value. In particular, we plan to output the report in a structured way that facilitates faster data processing. Furthermore, we expect to build on the prompt by goal strategy to isolate code documentation and code testing goals.

Limitations

The case study focuses on the details of a project at a specific SME, a research facility focused on astronomical observations. While we believe that our approach can benefit other SMEs, it is unclear how generalizable the approach is. In particular, larger institutions with significant budgets will largely not benefit from this study since their constraints will be vastly different from those of SMEs. The study focused on a small codebase in Python that runs in a Docker container as a standalone application. Therefore, the team believes that the results presented in this case apply most directly to other SMEs relying on similar technologies and working within similar financial and operating constraints.

In addition to the issues outlined above, the case study suffers from several threats to validity. First, following Korn et al. (2026), we acknowledge prompting itself as a threat to validity. Despite reviewing results from 20 different prompts comprising different wording, styles, and approaches, we cannot claim to capture the variety of knowledge stored in the Codellama model. Second, since assessing bias in LLMs is a robust area of research, we cannot be sure perhaps ever if our results are representative in any meaningful sense. Institutions must simply ask a cost/benefit question: Is the time and money we put into the AI-assisted code review process worth the feedback and actionable items we derive from that process?

Conclusions

Many SMEs face cybersecurity risks due to emerging AI threats without the resources to address those challenges. The case study presented here suggests a framework for using an AI-assisted code review process to enhance cybersecurity and perhaps get ahead of the expected challenges of modern AI. The team demonstrated the following through this case study:

- Task decomposition is one of the most important cost-free tools for enhancing effectiveness of an AI-assisted code review for cybersecurity.
 - File-level prompting generally worked better than project-level prompting;
 - Goal specific prompting (quality, security) worked better than combined prompting;
- Bias analysis is required for teams looking to incorporate LLM into their code review process.
 - The practical effects of model bias cannot be predicted for any particular project.
 - Testing for bias and specific use cases improves effectiveness.
- Human oversight is crucial for extracting value from an AI-assisted code review process.
 - Manual review of reports is crucial for an effective process.
 - Static tools serve an important role in checking for the biases and errors in LLMs.

We hope to avoid the problems currently predicted in the cybersecurity world due to AI. This study documents an early step toward incorporating AI in a way allowing SMEs to fight fire with fire in this new cybersecurity environment.

Acknowledgements

We would like to thank the reviewers for their helpful comments and guidance. We acknowledge the use of AI in prompt generation and the code review process examined in the study. The authors acknowledge support in part from NASA EPSCoR 80NNSC22M0063, NSF AST 2319415, and NASA EPSCoR 80NNSC24M0112.

References

- Amro, A., & Alalfi, M. H. (2025). Github's Copilot Code Review: Can AI Spot Security Flaws Before You Commit? *arXiv preprint arXiv:2509.13650*. <https://doi.org/10.48550/arXiv.2509.13650>
- Anthropic. (2026). "Claude Mythos Preview \ Red.anthropic.com." *Anthropic.com*, <https://red.anthropic.com/2026/mythos-preview/>
- Aryan, A., Nain, A. K., McMahon, A., Meyer, L. A., & Sahota, H. S. (2023). The costly dilemma: generalization, evaluation and cost-optimal deployment of large language models. *arXiv preprint arXiv:2308.08061*. <https://doi.org/10.48550/arXiv.2308.08061>
- Badampudi, D., Unterkalmsteiner, M., & Britto, R. (2023). Modern Code Reviews—Survey of Literature and Practice. *ACM Trans. Softw. Eng. Methodol.*, 32(4). <https://doi.org/10.1145/3585004>
- Bappy, M. A. H., Mustafa, H. A., Saha, P., & Salehat, R. (2025). Case Study: Fine-tuning Small Language Models for Accurate and Private CWE Detection in Python Code. *arXiv preprint arXiv:2504.16584*. <https://doi.org/10.48550/arXiv.2504.16584>
- Bender, E. M., Gebru, T., McMillan-Major, A., & Shmitchell, S. (2021, March). On the dangers of stochastic parrots: Can language models be too big? *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency* (pp. 610-623).
- Cataldi, S. (2025). Perceptions and challenges of AI-driven code reviews: A qualitative exploration of develop experiences. *Issues in Information Systems*, 26(2). https://doi.org/10.48009/2_iis_127
- Chen, L., Zaharia, M., & Zou, J. (2023). Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*. <https://doi.org/10.48550/arXiv.2305.05176>.
- CodeSignal. (2023). *Developers and AI Coding Assistant Trends*. 2023 Trends Report. Retrieved on 4/25/2026 from <https://codesignal.com/report-developers-and-ai-coding-assistant-trends/#how>

- de Oliveira, I. S., Carneiro, J., Ribas, J., & Pereira, J. A. (2025, September). Code Smell Classification in Python: Are Small Language Models Up to the Task?. *Simpósio Brasileiro de Engenharia de Software (SBES)* (pp. 699-705). SBC.
- Ebneyamini, S., & Sadeghi Moghadam, M. R. (2018). Toward Developing a Framework for Conducting Case Study Research. *International Journal of Qualitative Methods*, 17. <https://doi.org/10.1177/1609406918817954>
- Gehrels, N., Chincarini, G., Giommi, P. E., Mason, K. O., Nousek, J. A., Wells, A. A., ... & Zhang, W. W. (2004). The Swift gamma-ray burst mission. *The Astrophysical Journal*, 611(2), 1005-1020.
- Gendre, B., Orange, N. B., Moore, E., Klotz, A., Coward, D. M., Giblin, T., ... & Morris, D. (2022). Modeling the GRB 170202A Fireball from Continuous Observations with the Zadko and the Virgin Island Robotic Telescopes. *The Astrophysical Journal*, 929(1), 16.
- Giblin, T. W., Novotny, S. J., Meehan, M., Jackson, D., Orange, N. B., & Christman, D. (2023, June). Integrating Machine Learning into the International Space Station Antenna Management Software. In *2023 IEEE Cognitive Communications for Aerospace Applications Workshop (CCAAW)* (pp. 1-6). IEEE.
- Giblin, T.W., Neff, J.E., Hakkila, J., Drost, D.M., & Andreasian-Thomas, N. (2003). Development of a Gamma-Ray Burst Automated Response and Remote 0.5m Telescope at Etelman Observatory at the University of the Virgin Islands. HEAD meeting #7 *American Astronomical Society Meeting*, 35, 624.
- Gokuldass, P., Morris, D., Orange, N., Cucchiara, A., & Strausbaugh, R. (2021, January). A Comprehensive Catalog of Gamma-Ray Bursts Observed by the Virgin Islands Robotic Telescope. In *American Astronomical Society Meeting Abstracts* (Vol. 237, pp. 135-01).
- Guo, Q., Cao, J., Xie, X., Liu, S., Li, X., Chen, B., & Peng, X. (2024, February). Exploring the potential of chatgpt in automated code refinement: An empirical study. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering* (pp. 1-13).
- Hajipour, H., Hassler, K., Holz, T., Schönherr, L., & Fritz, M. (2024, April). Codelmsec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models. In *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)* (pp. 684-709). IEEE.
- Howell, K., Christian, G., Fomitchov, P., Kehat, G., Marzulla, J., Rolston, L., ... & Bradley, J. (2023, July). The economic trade-offs of large language models: A case study. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track)* (pp. 248-267).
- Klebesadel, R. W., Strong, I. B., & Olson, R. A. (1973). Observations of gamma-ray bursts of cosmic origin. *Astrophysical Journal*, vol. 182, p. L85, 182, L85.

- Korn, A., Zaruchas, L., Arora, C., Metzger, A., Smolka, S., Wang, F., & Vogelsang, A. (2026). Reporting LLM Prompting in Automated Software Engineering: A Guideline Based on Current Practices and Expectations. *arXiv preprint arXiv:2601.01954*. <https://doi.org/10.48550/arXiv.2601.01954>
- Li, H., Hao, Y., Zhai, Y., & Qian, Z. (2023, November). Assisting static analysis with large language models: A chatgpt experiment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 2107-2111).
- Liu, Z., Yang, Z., & Liao, Q. (2024, July). Exploration on prompting LLM with code-specific information for vulnerability detection. In *2024 IEEE International Conference on Software Services Engineering (SSE)* (pp. 273-281). IEEE.
- Lombardi, T., North, M., Orange, N. B., Coulanges, K., & Johnson, J. (2021). VIRTBot: exploring chatbot design for promoting scientific initiatives. *Issues in Information Systems*, 22(4).
- Lopez, G. (2024). *codellama-python*. Github. <https://github.com/falconcr/codellama-python>.
- Mhatre, A., Nader, N., Diehl, P., & Gupta, D. (2026, February). Can LLMs Find Bugs in Code? An Evaluation from Beginner Errors to Security Vulnerabilities in Python and C++. In *SoutheastCon 2026* (pp. 1-8). IEEE.
- Moore, E., Gendre, B., Orange, N. B., & Panther, F. H. (2024). Constraints on the ultra-high energy cosmic ray output of gamma-ray bursts. *Monthly Notices of the Royal Astronomical Society*, 530(1), 555-559.
- Morris, D.C. (2020). The University of the Virgin Islands B.S. with a Concentration in Astronomy: Developing Careers in STEM through Astrophysics. *2020 235th American Astronomical Society Meeting*, 52, 117.03.
- NIST. (n.d.). National Vulnerability Database. Retrieved on 5/9/2026 from <https://nvd.nist.gov/>.
- Oldemeyer, L., Jede, A., & Teuteberg, F. (2025a). Investigation of artificial intelligence in SMEs: a systematic review of the state of the art and the main implementation challenges. *Management Review Quarterly*, 75(2), 1185-1227.
- Oldemeyer, L., Jede, A., & Teuteberg, F. (2025b). Too Expensive or Worthwhile? Approaches to AI Assessment in SMEs. *PACIS 2025 Proceedings*. 2. <https://aisel.aisnet.org/pacis2025/transform/transform/2>
- Ollama. (n.d.). Ollama Release v0.19.0. Github. <https://github.com/ollama/ollama/releases/tag/v0.19.0>.
- OpenRouter. (2026, May). Free AI Models on OpenRouter. Retrieved on 5/9/2026 from <https://openrouter.ai/collections/free-models>.
- Orange, N.B., Gokuldass, P., Lombardi, T., & Parsons, M. (2021). The Etelman Observatory and the Virgin Islands Robotic Telescope: A Pivotal and Resilient Educational Center and Research Instrument. *2021 237th American Astronomical Society Meeting*, 53, 139.02.

- Orange, N.B., & Morris, D.C., (2021), "Observing the Need to Foster Inclusion and Embrace Diversity," *Futurum*, <https://doi.org/10.33424/FUTURUM149>.
- Pan, G., Chodnekar, V., Roy, A., & Wang, H. (2025). A Cost-Benefit Analysis of On-Premise Large Language Model Deployment: Breaking Even with Commercial LLM Services. *arXiv preprint arXiv:2509.18101*. <https://doi.org/10.48550/arXiv.2509.18101>.
- Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2025). Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *Communications of the ACM*, 68(2) (February 2025), 96–105. <https://doi.org/10.1145/3610721>.
- Salih, H. A. M., & Sarhan, Q. I. (2025). A Study of Large Language Models in Detecting Python Code Violations. *ARO-THE SCIENTIFIC JOURNAL OF KOYA UNIVERSITY*, 13(2), 215-225.
- Sasaki, Y., Washizaki, H., Li, J., Yoshioka, N., Ubayashi, N., & Fukazawa, Y. (2025). Landscape and taxonomy of prompt engineering patterns in software engineering. *IT Professional*, 27(1), 41-49.
- Stackoverflow. (2025). *AI. 2025 Developer Survey*. Retrieved on 4/25/2026 from <https://survey.stackoverflow.co/2025/ai/>.
- Sun, W., Miao, Y., Li, Y., Zhang, H., Fang, C., Liu, Y., ... & Chen, Z. (2025, April). Source code summarization in the era of large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)* (pp. 1882-1894). IEEE.
- Thompson, C.& Wagner, D.(2017). A Large-Scale Study of Modern Code Review and Security in Open Source Projects. *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. Association for Computing Machinery, New York, NY, USA, 83–92. <https://doi.org/10.1145/3127005.3127014>.
- Yuen, A., Pangas, J., Polash, M. M. H., & Abdellatif, A. (2025, September). Prompting Matters: Assessing the Effect of Prompting Techniques on LLM-Generated Class Code. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 803-808). IEEE.
- Zhao, C., & Gupta, T. D. (2023). Cyberattack shuts major telescopes. *Science (New York, NY)*, 381(6660), 824-824. <https://doi.org/10.1126/science.adk4482>.

Appendix of Tables

Table 1. Tools Incorporated as Git Precommit Hooks in Software Development Process

Tool	Type	Purpose	Source
AI-code-review	AI tool	General review of code quality and security	https://github.com/ollama/ollama/releases/tag/v0.19.0
Bandit	security	Code security checks	https://pypi.org/project/bandit/
Black	code formatting	Enforce code formatting rules	https://pypi.org/project/black/

Flake8	code quality	Wrapper for multiple code quality tools	https://pypi.org/project/flake8/
isort	code formatting	Orders import statements	https://pypi.org/project/isort/
Mypy	code quality	Type checking	https://pypi.org/project/mypy/
pip-audit	security	Checks for supply chain vulnerabilities in libraries	https://pypi.org/project/pip-audit/
Pylint	code quality	Static code analyzer	https://pypi.org/project/pylint/
Ruff	code quality	Fast code linter	https://astral.sh/ruff
Semgrep	security	Checks for security threats	https://semgrep.dev/
Vulture	code quality	Find unused code	https://github.com/jendrikseipp/vulture

Table 2. Descriptive Statistics for AI-Assisted Code Review Execution Time

File	Review Count	Mean (sec)	Std Dev (sec)	Min (sec)	Max (sec)
__init__.py	8	578	180	332	862
append_sheet.py	24	535	195	205	1004
config.py	23	394	217	124	1004
coordinates.py	22	728	808	212	4256
db_iface.py	22	567	257	188	1264
file_processor.py	22	606	243	279	1241
file_writer.py	21	433	203	187	979
gen_circulars.py	22	1137	773	481	3364
gen_processor.py	22	680	333	293	1574
kafka_handler.py	22	513	181	229	819
listener.py	22	477	167	153	962
all_code.py	6	4329	377	3888	4743

Table 3. Top References by Review Type

Review Type	JSON (%)	TinyDB (%)	Regex (%)	pydoc (%)	Pylint (%)
Code Review	10 (36%)	11 (46%)	9 (69%)	5 (39%)	9 (75%)
Combined Review	6 (21%)	6 (25%)	1 (8%)	2 (15%)	0 (0%)
Security Review	12 (43%)	7 (29%)	3 (23%)	6 (46%)	3 (25%)
Total	28 (100%)	24 (100%)	13 (100%)	13 (100%)	12 (100%)

Table 4. Security References by Review Type

Review Type	HTTP (%)	HTTPS (%)	OAuth (%)	secrets (%)	Bandit (%)
Code Review	2 (20%)	1 (9%)	0 (0%)	0 (0%)	0 (0%)
Combined Review	1 (10%)	1 (9%)	1 (8%)	0 (0%)	0 (0%)
Security Review	7 (70%)	9 (82%)	11 (92%)	9 (100%)	8 (100%)
Total	10 (100%)	11 (100%)	12 (100%)	9 (100%)	8 (100%)

Table 5. Explicit Prompts for Common Vulnerabilities and Exposures (CVE)

Prompt ID	P 13	P 15	P 17	P 19
-----------	------	------	------	------

Total # of CVEs	4	28	0	18
% of Total Requests	50%	93%	0%	78%
Total # of Requests	8	30	11	23

Table 6. Prompts

Prompt ID	Type	Strategy	Prompt
1	Combined review	Role-based	You are a senior developer performing a code review. Review the attached Python code and suggest modifications to the code to improve its PEP-8 compliance, structure, efficiency, and security.
2	Combined review	Structural	\n Review this code, provide suggestions for improvement, coding best practices, improve readability, and maintainability. Remove any code smells and anti-patterns. Provide code examples for your suggestion. Respond in markdown format. If the file does not have any code or does not need any changes, say 'No changes needed' (Lopez, 2024).
3	Combined review	Role-based	You are a senior software developer performing a modern code review. Review this code to provide suggestions for improvement, coding best practices, improve readability, improve maintainability, and improve security. Identify code smells and anti-patterns and suggest improvements. Provide code examples for your suggestions. Respond in markdown format. If the file requires no changes or contains no code, say 'No changes required.'
4	Code review	Role-based	You are a senior software developer performing a modern code review. Review this python code to improve its readability, maintainability, and efficiency. Suggest improvements based on coding best practices and standards. Suggest improvements to documentation. Identify code smells and anti-patterns and suggest improvements. Provide code examples in your responses. Respond in markdown format. If the file requires no changes or contains no code, output 'No changes required.'
5	Security review	Role-based	You are a senior cybersecurity engineer performing a security audit. Review the python code for security vulnerabilities, insecure coding practices, and poor coding implementations. Suggest improvements to code security with code examples. Respond in markdown format. If the code requires no changes, output 'No security vulnerabilities detected.'
6	Code review	Role-based	You are a senior software developer performing a modern code review. Review this python code to improve its readability, maintainability, and efficiency. Suggest improvements based on coding best practices and standards. Suggest improvements to documentation. Identify code smells and anti-patterns and suggest improvements. Provide code examples in your responses. Respond in markdown format. If the file requires no changes or contains no code, output 'No changes required.'
7	Code review	Role-based	You are a senior software developer performing a modern code review. Review the Python code and suggest modifications to the code to improve its PEP-8 compliance, structure, efficiency, and documentation. Respond in markdown format.
8	Code review	Chain-of-thought	Analyze the provided Python code for PEP-8 violations, potential issues, or areas of improvement. Use a chain of thought approach to systematically evaluate the code. Reply in markdown format.
9	Security review	Chain-of-thought	Analyze the provided Python code for security issues and vulnerabilities. Use a chain of thought approach to systematically evaluate the code. Respond in markdown format.
10	Code review	Structural	Analyze the provided Python code for PEP-8 violations, potential issues, or areas of improvement. Reply in markdown format.

11	Security review	Structural	Analyze the provided Python code for security issues and vulnerabilities. Respond in markdown format.
12	Code review	Structural (AI-generated)	Hey CodeLlama! I'm working on a project that involves writing some Python scripts, and I was wondering if you could analyze the code I've written so far to provide feedback on PEP-8 compliance, efficiency, readability, and quality. Could you also provide some metrics such as the number of lines of code, function count, and complexity score? That would be really helpful in helping me improve my coding skills. Respond in markdown format. Thanks!
13	Security review	Hybrid (AI-generated)	Please provide a detailed Python code security analysis from codellama, including the identification of specific Common Vulnerabilities and Exposures (CVE), insecure code, suggestions for improvements based on secure coding practices, and step-by-step reasoning. Assume the perspective of a highly skilled cybersecurity professional.
14	Code review	Hybrid (AI-generated)	Title: Python Code Review - Standalone Application Introduction: Thank you for considering this request to conduct a Python code review for a standalone application. As an experienced Python developer, I am confident that your expertise in the language will be instrumental in ensuring the success of this project. Please provide your input on the following Python code and suggest any necessary changes or improvements based on your experience with Python. Role: The role of a Python code reviewer is to analyze and evaluate the quality of the provided code, and provide feedback on its structure, functionality, and overall performance. As a Python developer, you have extensive knowledge of the language and its various features, which will be invaluable in conducting this review. Chain of thought analysis: The following chain of thoughts will guide our discussion during the code review: 1. Initial Review: Start by reviewing the overall structure and organization of the code. Identify any redundancies or inefficiencies in the code and provide feedback on how to improve its efficiency and readability. 2. Functionality: Evaluate the functionality of each function in the code, including input parameters, output values, and potential errors that may arise. Provide suggestions for improving the robustness and reliability of the code. 3. Performance: Assess the performance of the code based on factors such as execution time, memory usage, and any bottlenecks that may be present. Provide feedback on how to optimize the code for better performance. 4. Security: Review the security features of the code, including authentication and authorization mechanisms, to ensure that it is secure and protected against unauthorized access. 5. Testing: Evaluate the testing strategy employed in the code, including any automated tests or manual testing procedures. Provide feedback on how to improve the testing coverage and effectiveness. 6. Dependencies: Review the dependencies of the code, including any external libraries or frameworks used, and provide feedback on their compatibility with the requirements of the project. 7. Documentation: Evaluate the documentation provided in the code, including its quality and completeness. Provide feedback on how to improve its usefulness for other developers who may need to maintain or contribute to the code. 8. Best Practices: Review the code against industry best practices and provide feedback on any areas where the code deviates from these guidelines. 9. Conclusion: Summarize your findings in markdown format and provide recommendations for further improvements, including any necessary changes or additions to the

			codebase. By following this chain of thought analysis, we can ensure that the Python code review process is comprehensive and effective in identifying areas for improvement and providing valuable feedback for future development.
15	Security review	Hybrid (AI-generated)	Please perform a detailed security analysis of the following Python code using Codellama, and provide a comprehensive report with the following elements: 1. Identify specific Common Vulnerabilities and Exposures (CVE) related to this code. 2. Check for insecure coding practices that could lead to vulnerabilities. 3. Suggest improvements based on secure coding practices. 4. Document the findings with step-by-step reasoning. Assume the perspective of a highly skilled cybersecurity professional who is experienced in Python development and security testing. Provide a detailed report in markdown format with specific examples and reasoning to support your recommendations.
16	Code review	Hybrid (AI-generated)	I am reaching out to request a Python code review for a standalone application that I have developed. The application is a consumer of topics from publisher. It is written in Python and is currently hosted on my local machine. I would greatly appreciate it if you could provide your expertise and insights on the following aspects of the code: 1. Code structure and organization 2. Functionality and features 3. Performance optimization 4. Best practices and coding standards 5. Code readability and maintainability 6. Error handling and exception management 7. Testing and debugging 8. Security vulnerabilities 9. Scalability and reliability 10. User experience and usability. I would like to request that you review the code based on the following roles and chain of thought: 1. Technical reviewer - Provide a detailed analysis of the technical aspects of the code, including its structure, functionality, and performance optimization. 2. Business analyst - Evaluate the value proposition of the application from a business perspective, including its impact on the organization and the potential return on investment (ROI). 3. User experience reviewer - Assess the user experience of the application, including its usability, accessibility, and overall satisfaction. 4. Security expert - Identify any security vulnerabilities or best practices that may need to be addressed in the code. 5. Best practices advocate - Ensure that the code follows industry-standard coding conventions, such as PEP 8, and that it adheres to the principles of object-oriented programming and other software development methodologies. 6. Testing specialist - Review the testing strategy and ensure that all tests are effective and cover the required functionality. 7. Debugging expert - Assess the debugging process and identify any areas where the code could be more robust in handling errors and exceptions. 8. Scalability reviewer - Evaluate the scalability of the application and its ability to handle increasing user traffic and data volume over time. 9. Maintainability specialist - Analyze the code's maintainability and ensure that it is easy to understand, modify, and extend for future development. 10. User feedback reviewer - Review any user feedback or support requests related to the application and provide suggestions for improving the user experience. I would be grateful if you could provide your expertise and insights on each of these aspects of the code review task. Your input will help me make informed decisions about the future development and maintenance of this standalone application. Thank you in advance for considering my request. I look forward to receiving your feedback.

17	Security review	Hybrid (AI-generated)	Dear codellama, I am writing to request a detailed analysis of Python code that I have been tasked with reviewing for potential security vulnerabilities. The code in question is a complex system that relies on various APIs and libraries to function, and I believe that it would be beneficial to have a cybersecurity expert like yourself provide a comprehensive review of the code's secure coding practices, identifying specific Common Vulnerabilities and Exposures (CVE), checking for insecure code, and providing suggestions for improvements based on secure coding practices. The code is a Python application that interfaces with multiple APIs and libraries to collect data from various sources. The code has been developed by a team of developers who are experienced in Python development, but I believe that it would be beneficial to have an expert review the code to identify any potential security vulnerabilities. I am particularly interested in your feedback on the secure coding practices used by the developers in creating this code. Are there any best practices or guidelines that they could have followed to ensure the code's security? Additionally, I would appreciate any specific comments on identifying CVE and checking for insecure code. Finally, I would like to request your suggestions for improvements based on secure coding practices. Respond in markdown format.
18	Code review	Hybrid (AI-generated)	I am writing to request a code review for a standalone Python application that I have developed. I believe that my code has some potential issues that could be addressed through a thorough code review, and I would greatly appreciate any feedback or insights that you may have. In particular, I would like to request a code review with the following focus: 1. Design pattern implementation: I am interested in ensuring that the design patterns used in my code are appropriate for the problem domain and that they are implemented correctly. 2. Code organization: I would like to receive feedback on how I can improve the organization of my code, including suggestions for better file structure, naming conventions, and commenting practices. 3. Test coverage: I am interested in increasing the test coverage of my code and would appreciate any recommendations or best practices for writing effective unit tests. 4. Performance optimization: I would like to know if there are any areas where my code could be optimized for performance, such as reducing the number of database queries or improving the efficiency of loops. 5. Error handling: I am interested in learning how I can improve the error handling mechanisms in my code, including suggestions for better error handling practices and techniques. 6. Security: I would like to receive feedback on any security vulnerabilities or best practices that I may have overlooked in my code. I am confident that a thorough code review with an experienced Python developer will help me improve the quality and maintainability of my code, as well as provide valuable insights for future development. Respond in markdown format. Thank you in advance for considering my request.
19	Security review	Hybrid (AI-generated)	Prompt: Detailed Python Cybersecurity Analysis. As a highly skilled cybersecurity professional, I have been tasked to conduct a detailed analysis of a Python codebase to identify potential vulnerabilities and suggest improvements in secure coding practices. Please provide me with the following information for this analysis: 1. The Python code to be analyzed. 2. A list of Common Vulnerabilities and Exposures (CVE) that have been identified for the code. 3. Any insecure coding practices or security risks that have been identified. 4. Suggestions for improving secure coding practices within the codebase. 5. Step-by-step reasoning to

			support each suggestion. 6. Documentation of the findings, including any relevant context or background information By conducting a thorough analysis of this Python code, I will be able to identify potential vulnerabilities and suggest improvements in secure coding practices that will help protect against security threats and improve the overall security posture of the codebase.
20	Security review	Chain-of-thought	Analyze the provided Python code for security issues and vulnerabilities. Use a chain of thought approach to systematically evaluate the code. Respond in markdown format.