Generative AI in Software Development

eu-LISA Technology Monitoring Report

eu-LISA

Contents

1. Introduction

Since the launch of ChatGPT in 2022, generative AI, specifically in the form of large language models (LLMs), has spread in a way that no technology has ever done before. The pace of adoption has been truly astounding: by February 2026, the number of weekly active ChatGPT users was estimated at around 900 million [1]. Since 2022, other companies around the world have launched LLMs, some of which have remained closed (ChatGPT from OpenAI, Gemini from Google DeepMind, etc.), while others have been shared as open-source models under different types of licence (Llama and its variants from Meta AI, Mistral models from Mistral AI, the Gemma series from Google DeepMind, etc.).

Generative AI techniques have found widespread application across different business domains, in particular in knowledge work.

- LLMs have been effectively used for customer-facing applications (e.g. chatbots), for knowledge management (e.g. enterprise search and summarisation), for content creation (e.g. copywriting and social media content generation) and for productivity and automation (e.g. report generation, meeting summarisation).
- Diffusion models have been successfully used for image, audio and video generation. In fact, such models in combination with LLMs make it possible to produce a podcast from scratch with realistic voices without engaging humans beyond the generation and refining of prompts [2].
- Generative adversarial networks have been widely used for image synthesis, including deepfake generation. These models have also been effectively deployed by criminals (e.g. for impersonation or document fraud).

Considering the success of LLMs in effectively producing coherent text as output in response to a prompt, it is not surprising that one of the key areas where LLMs have found traction is generation of computer code or software development. A study [3] on the use of AI coding tools among software engineers has identified the following most common uses of AI coding assistants:

- 85 % of respondents use AI coding assistants to write new code;
- 73 % of respondents use AI coding tools for refactoring existing code;
- about 60 % of respondents use AI tools for testing and documentation;
- increasingly, AI coding assistants are being used to support code review.

As we explain in this report, AI coding assistants have evolved from performing simple code completion to acting as autonomous agents capable of carrying out complex sequences of actions and understanding context and organisational conventions, based on tasks given in natural language. Therefore, they have the potential to transform how software is developed.

Because of the ongoing transformation at the European Union Agency for the Operational Management of Large-Scale IT Systems in the Area of Freedom, Security and Justice (eu-LISA), and the adoption of a software factory approach, this report focuses on the application of generative AI in the context of software development, and specifically on AI coding assistants (see Table 1 for a high-level overview of generative AI use cases throughout the software development life cycle (SDLC)).

---

1 Malik, A., 'ChatGPT reaches 900M weekly active users', TechCrunch website, 27 February 2026, https://techcrunchom/2026/02/27/chatgpt-reaches-900m-weekly-active-users/.

2 A prompt is an instruction, consisting of a combination of words, numbers or symbols, given to a generative AI model to produce an output.

3 Cortex, *Engineering in the Age of AI: 2026 benchmark report*, San Francisco, CA, 2026, https://go.cortex.io/rs/563-WJM-722/ages/2026-Benchmark-Report.pdf?version=0.

Table 1: AI and generative AI use-cases throughout the SDLC

Stage of SDLC	Use-cases description
Gathering & analysis of requirements	**Translating stakeholder input into structured requirements.** Speech-to-text tools can help transcribe meetings, interviews and workshops. LLMs can then be used to edit transcripts, summarise and extract functional and non-functional requirements. As a next step, named-entity recognition techniques can be used for identifying system components, constraints and domain-specific terminology. **Generating user stories and acceptance criteria.** Generative AI tools can turn plain-language descriptions into agile user stories and automatically add given–when–then acceptance criteria. **Summarising and extracting insights from regulations, standards and policies.** AI tools can help process large volumes of legal text and extract information relevant to requirements. **Identifying gaps, conflicts and ambiguities.** AI tools can be used to compare new requirements with those of older projects, flag missing standard requirements and identify contradictory documentation on requirements.
System architecture & design	**Generating initial architecture proposals.** Dedicated LLMs trained on software architecture knowledge bases can be used to suggest high-level system architecture diagrams based on business and technical requirements, and to propose whether to choose a microservices or a monolith approach, depending on the scalability and maintainability needs. **Application programming interface (API) and interface design assistance.** LLMs can also be used to transform natural language into a structured format (e.g. JSON or XML) and support with the generation of API specifications from textual descriptions of service capabilities. **Security and design compliance recommendations.** AI can propose security strategies based on architecture, including authentication, authorisation and encryption, and it can simulate attack surfaces and propose mitigation measures.
System implementation & coding	**Coding assistance.** AI tools can be used as a partner in 'pair programming', offering suggestions, explanations and alternatives for implementation strategies. **Code generation based on specifications.** LLM-based tools can be used to generate working code from natural language descriptions, and they can support software engineers with routine tasks (application scaffolding, configuration files, folder structures). **Code modernisation.** Generative AI tools can analyse code and suggest cleaner, more maintainable code structures, and they can update deprecated APIs and libraries to newer versions. **Code translation.** Generative AI tools can translate code from one programming language into another, to help in migrating applications to modern frameworks while maintaining functionality.
System testing	**Test design, test case and test data generation.** Generative AI tools can generate tests based on code or specifications; similarly, they can be used to generate realistic synthetic test data where the use of production data is not possible. **Security testing.** Generative AI tools can effectively facilitate security testing by identifying vulnerabilities and generating inputs intended to break a system
System deployment & maintainance	AI tools can be effectively used for **system monitoring** to identify potentially problematic changes early, helping reduce system downtime during roll-out of changes, and for continuously monitoring system performance and identifying anomalies, performing root-cause analysis and suggesting fixes.

2. Evolution of generative AI for coding

Code completion systems have been part of modern integrated development environments (IDEs) for at least 30 years, since Microsoft introduced IntelliSense, the first context-aware autocomplete software based on the syntax or grammar of the programming language. In addition to improving coder productivity, early versions of code completion systems helped developers produce more readable and understandable code [4]. The so-called intelligent code completion systems, introduced in the late 2000s and based on statistical techniques, further enhanced coding assistants by filtering code proposals that were irrelevant to a particular context and assessing the relevance of every proposal based on a certain algorithm [5]. However, these code completion approaches were primarily limited to the completion of methods and APIs.

The change of paradigm in intelligent code assistants happened in 2020 with the publication of a paper describing IntelliCode Compose – a general-purpose multilingual code completion tool, based on a generative transformer model trained on 1.2 billion lines of source code in multiple programming languages [6]. The transformer architecture upon which the modern LLMs are based [7] proved very effective for code generation, because of the large volume of code in multiple programming languages available online, which could be used for training purposes. Initially, LLM-based coding assistants trained on large-scale codebases could be used only for the completion of relatively simple programming tasks or for translating instructions into code. However, the introduction of DeepMind's AlphaCode in 2022 kicked off a new phase in their development, namely the emergence of code-generation systems capable of deeper reasoning and problem-solving at the level of international programming competitions [8].

With the development of LLMs, coding assistants incorporated new functionalities, including chat or voice-based interfaces. The move from autocomplete to conversation-based interaction, with either dedicated coding assistants (e.g. GitHub Copilot [9] or Cursor) or general-purpose LLM-based AI tools such as ChatGPT or Claude, has resulted in a fundamental transformation in how software engineers work today. Since the introduction of a chat interface in programming assistants, computer engineers have used code-generating models in two modes: acceleration (when the engineer knows what to do and uses generative AI to get there faster while staying in the flow state; this is primarily done using autocompletion) and exploration (when the engineer uses generative AI to explore options and understand how to proceed further). When using coding assistants such as Copilot in exploration mode, programmers effectively replace the website Stack Overflow, which they would normally refer to when facing a challenge.

---

4 Bruch, M., Monperrus, M. and Mezini, M., 'Learning from examples to improve code completion systems', *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM Sigsoft symposium on the foundations of software engineering*, 2009, pp. 213–222, https://doi.org/10.1145/1595696.1595728.

5 ibid.

6 Svyatkovskiy, A., Deng, S. K., Fu, S. and Sundaresan, N., 'IntelliCode Compose: Code generation using transformer', *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 1433–1443, https://doi.org/10.1145/3368089.3417058.

7 Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L. et al., 'Attention is all you need', in: Guyon, I., von Luxburg, U., Bengio, S., Wallach, H., Fergus, R. et al. (eds), *Advances in Neural Information Processing Systems 30*, La Jolla, CA, 2017, pp. 5998–6008.

8 Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J. et al., 'Competition-level code generation with AlphaCode', *Science*, Vol. 378, Issue 6624, 2022, pp. 1092–1097, https://doi.org/10.1126/science.abq1158.

9 Friedman, N., 'Introducing GitHub Copilot: Your AI pair programmer', GitHub Blog website, 23 February 2022 (created 29 June 2021), https://github.blog/news-insights/product-news/introducing-github-copilot-ai-pair-programmer/.

The move from autocomplete to conversation-based interaction, with either dedicated coding assistants or general-purpose LLM-based AI tools such as ChatGPT or Claude, has resulted in a fundamental transformation in how software engineers work

The next major advancement was the ability of coding assistants to work with a larger context window, moving from a single-file context to the awareness of a full code repository, which substantially improved the accuracy of these tools and their ability to solve more complex issues, thus also increasing code acceptance [10]. One of the early solutions to the problem of code assistants utilising information scattered across different files within a code repository is described in the RepoCoder paper [11]. RepoCoder proposes a framework to streamline repository-level code completion by incorporating a similarity-based retriever and an LLM pre-trained on code through an iterative retrieval-augmented generation pipeline, which allows it to utilise repository-level information for code completion. In addition to the RepoCoder, the paper also proposed a benchmark for evaluating coding assistants operating at the repository level (evaluation frameworks are described in the next section).

The latest stage (as of early 2026) in the evolution of coding assistants is the move towards agentic engineering and vibe coding. The prime example of an agentic engineering system, which has attracted a lot of attention since its release in early 2025, is Claude Code [12], although other similar coding agents, such as Devin [13], Amp [14] and one within Cursor [15], have been released since. What distinguishes an agent from a more traditional AI coding assistant is that it not only responds to a single prompt, but also can autonomously carry out complex, multi-step tasks requiring higher levels of reasoning and interaction with other software tools. This ability of AI coding assistants to carry out complex tasks autonomously has led to what has been termed 'vibe coding' [16], or the ability to create software through natural language without the need to interact with code.

The quality of code produced by AI coding agents will depend on the documentation that is provided to them in the codebase; documentation that is easy for LLMs to read (e.g. in Markdown) can significantly improve code quality [17]. Some authors also argue that simpler approaches that do not require autonomous decision-making, but instead rely on a three-phase process (issue localisation,

10 ZenML, 'Evolution of LLM integration in GitHub Copilot development', ZenML website, 2023, https://www.zenml.io/llmops-database/evolution-of-llm-integration-in-github-copilot-development.

11 Zhang, F., Chen, B., Zhang, Y., Keung, J., Liu, J. et al., 'RepoCoder: Repository-level code completion through iterative retrieval and generation', arXiv preprint paper, 2023, https://doi.org/10.48550/arXiv.2303.12570.

12 https://claude.com/product/claude-code.

13 https://devin.ai/.

14 https://ampcode.com.

15 https://cursor.com.

16 Andrej Karpathy, one of the co-founders of OpenAI, coined the term: https://x.com/karpathy/status/1886192184808149383.

17 Loftesness, K., 'Five best practices for using AI coding assistants', Google Cloud website, 7 October 2025, https://cloud.google.com/blog/topics/developers-practitioners/five-best-practices-for-using-ai-coding-assistants.

repair and patch validation), perform better when evaluated using a standard benchmark [18]. Finally, over-reliance on the output of LLMs, which is natural in the case of agentic engineering or vibe coding, is likely to lead to error-prone software with potentially severe gaps in security unless the code produced by agents is subject to stringent review.

18 See, for example, Xia C. S., Deng, Y., Dunn, S. and Zhang, L., 'Agentless: Demystifying LLM-based software engineering agents', arXiv preprint paper, 2024, https://doi.org/10.48550/arXiv.2407.01489.

3. Benchmarking and evaluation

LLMs are being actively deployed in commercial services, such as AI coding assistants. On the surface, especially to a non-expert observer, the scope of problems that LLMs cannot solve is shrinking by the hour. LLMs seem to be able to produce high-quality text and code. However, how do we know this is truly the case? Benchmarking and evaluation are the main tools used to assess how well AI-driven coding applications perform standardised sets of tasks, such as solving software development problems in the case of AI coding tools based on LLMs. Evaluation of frontier language models requires challenging benchmarks that aim to reflect the real-world applications of these models. Such evaluations also help advance the development of LLMs and their practical application [19].

> *Benchmarking and evaluation are the main tools used to assess how well AI-driven coding applications perform standardised sets of tasks, such as solving software development problems in the case of AI coding tools based on LLMs*

One of the early benchmarks, HumanEval [20], introduced alongside Codex, an LLM fine-tuned on publicly available code, relied on hand-written programming problems in Python, and included tasks that aimed to assess language comprehension, reasoning, algorithms and simple mathematics. Initial evaluations showed that Codex could effectively solve 28.8 % of problems [21]. A similar evaluation of LLMs' programming capabilities based on the Mostly Basic Programming Problems dataset [22] showed that large non-fine-tuned models could solve ca 60 % of problems, whereas the largest fine-tuned models could solve close to 84 %. Since then, LLMs trained on code have advanced significantly; therefore, these benchmarks can be used for evaluating smaller models only, not frontier models, which require more complex problems.

Later benchmarks included evaluations based on problem sets from programming competitions. A prime example is the evaluation of AlphaCode in 2022, which demonstrated that transformer-based models can reach the performance level of a median human competitor; it also showed that success in solving problems in programming competitions is not automatically transferable to success in solving real-world engineering problems, which require additional work [23].

The next generation of benchmarks focused on real-world software engineering problems and are currently considered the gold standard. One of the most popular benchmarking tools for software engineering is software engineering benchmark

19 Srivastava A., Rastogi, A., Rao, A., Shoeb, A. A. M., Abid, A. et al., 'Beyond the imitation game: Quantifying and extrapolating the capabilities of language models', arXiv preprint paper, 2023 (created 2022), https://doi.org/10.48550/arXiv.2206.04615.

20 Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde de Oliveira Pinto, H. et al., 'Evaluating large language models trained on code', arXiv preprint paper, 2021, https://doi.org/10.48550/arXiv.2107.03374.

21 ibid.

22 Austin, J. Odena, A., Nye, M., Bosma, M., Michalewski, H. et al., 'Program synthesis with large language models', arXiv preprint paper, 2021, https://doi.org/10.48550/arXiv.2108.07732.

23 Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J. et al., 'Competition-level code generation with AlphaCode', *Science*, Vol. 378, Issue 6624, 2022, pp. 1092–1097, https://doi.org/10.1126/science.abq1158.

(SWE-bench)[24]. LLM-based coding assistants are evaluated by providing models with inputs defined as a code repository and issue descriptions. The code repository and issue descriptions are based on real-life problems from GitHub. The agent is expected to generate a patch that resolves the described problem. When the SWE-bench was set up in 2024, most models exhibited low levels of performance, resolving ca 2 % of issues. Since then, the benchmark has been improved, including in collaboration with OpenAI[25], which has resulted in SWE-bench Verified, the state-of-the-art benchmarking tools for AI coding assistants. Currently, leading models (e.g. Claude Opus 4.5, Gemini 3 Flash) can solve ca 77 % of issues[26]. State-of-the-art benchmarks today include multilingual benchmarks, which extend beyond a single language (primarily Python, due to availability of code) to multiple programming languages[27].

As the ability of coding assistants to work across code repositories improved, the need to develop adequate evaluation benchmarks increased, such as the RepoBench[28], RepoEval[29] and CrossCodeEval[30]. The main objective of these benchmarks is to test the ability of code completion models to perform complex tasks that require in-depth contextual understanding of the code within a codebase, to resolve a task or to complete code accurately.

With the rise of agentic engineering using LLMs, such as Claude Code, the emerging frontier of evaluation is focusing on automated software engineering agents and AI coding assistants with very complex long-horizon problems requiring complex reasoning chains. Such problems would require hours to days for a professional software engineer to complete. One example of such a benchmark is SWE-bench Pro[31], introduced by Scale AI, which showed that the most advanced LLMs at the time, such as Claude Sonnet 4.5 and OpenAI's GPT-5 (high) achieved a resolve rate of over 40 %.

Several methodological challenges are continuously being addressed by the developers of benchmarks. The most important is contamination. The pace of new model development is very high, and models are being trained on code available on GitHub, which may be included in the evaluation dataset and therefore affect the results of the evaluation. This issue is well documented in SWE-bench+[32].

24 https://www.swebench.com/.

25 Chowdhury, N., Aung, J., Shern, C. J., Jaffe, O., Sherburn, D. et al., 'Introducing SWE-bench Verified', OpenAI website, 13 August 2024, https://openai.com/index/introducing-swe-bench-verified/.

26 https://www.swebench.com/index.html.

27 For example, SWE-bench Multilingual (https://www.swebench.com/multilingual-leaderboard.html) covers C, C++, Go, Java, JavaScript/TypeScript, PHP, Ruby and Rust.

28 Liu T., Xu, C. and McAuley, J., 'RepoBench: Benchmarking repository-level code auto-completion systems', arXiv preprint paper, 2023, https://doi.org/10.48550/arXiv.2306.03091.

29 Zhang F., Chen, B., Zhang, Y., Keung, J., Liu, J. et al., 'RepoCoder: Repository-level code completion through iterative retrieval and generation', arXiv preprint paper, 2023, https://doi.org/10.48550/arXiv.2303.12570.

30 Ding Y., Wang, Z., Ahmad, W. U., Ding, H., Tan, M. et al., 'CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion', arXiv preprint paper, 2023, https://doi.org/10.48550/arXiv.2310.11248.

31 Deng X., Da, J., Pan, E., Yiming He, Y., Ide, C. et al., 'SWE-bench Pro: Can AI agents solve long-horizon software engineering tasks?', arXiv preprint paper, 2025, https://doi.org/10.48550/arXiv.2509.16941.

32 Aleithan R., Xue, H., Mohajer, M. M., Nnorom, E., Uddin, G. et al., 'SWE-bench+: Enhanced coding benchmark for LLMs', arXiv preprint paper, 2024, https://doi.org/10.48550/arXiv.2410.06992.

4. Current landscape of tools

Not all AI-supported coding assistants are the same. The tools available on the market can be divided into three categories: tier 1, tier 2 and tier 3.

Tier 1 includes coding assistants integrated in the IDE (e.g. Visual Studio Code or JetBrains). Here, AI is embedded into the existing development environment with the aim to enhance, but not replace, the developer's workflow. Some of the most popular AI coding assistants in this category are as follows.

- GitHub Copilot [33], which is the market leader and a tool that defined this category. GitHub Copilot followed the general trajectory of evolution of AI coding assistants, starting with autocomplete and most recently adding Copilot agent mode. It is also a multi-model tool, allowing the user to choose from a selection of models (e.g. ChatGPT, Claude, Gemini).
- Amazon Q Developer [34] is an Amazon Web Services (AWS)-native tool with a strong focus on the AWS toolchain and enterprise security features. It includes security scanning alongside code suggestions.
- Tabnine [35] is one of the earliest tools available with a strong focus on privacy and enterprise deployments, including on-premises models, allowing enterprises to ensure that source code stays in a confined environment and is not shared with third-party APIs.
- JetBrains AI Assistant [36] is deeply integrated in the JetBrains IDE and broader ecosystem (e.g. IntelliJ IDEA and PyCharm).

Tier 2 includes AI-forward code editors. Here, the IDE is built around AI models, which are at the core of the software development workflow, rather than being an add-on. Compared with tier 1, this is a relatively recent development, emerging in 2023. A key actor in this space is Cursor [37], which is the original AI-forward IDE based on open-source Visual Studio Code, but with deep integration of AI, including codebase indexing, multi-file edit proposals and chat interface for natural language interaction.

Tier 3 includes autonomous software development agents. Here, the developer takes the role of an architect, orchestrator and reviewer; the agents interpret, plan and execute multi-step tasks independently. The main distinction here is that the tools do not require the engineer to write code; instead, they accept high-level task descriptions in natural language and execute those tasks autonomously, with the developer reviewing the output produced at each step. New model capabilities enabled by agentic systems have spurred the development of vibe-coding practices. Some of the most popular agentic AI coding tools include Devin AI, which is the category-defining tool; Claude Code, which operates through the command-line interface (CLI) and has gained popularity among the general public beyond software engineers; OpenAI's Codex CLI, a direct competitor to Claude Code; GitHub Copilot agent mode, which is Microsoft's integration of agentic capabilities in the existing Copilot; and OpenHands, which is the leading open-source agentic coding platform and has been widely adopted by software developers.

Tool adoption depends on organisation- or industry-specific factors. Tier choice should follow workflow fit and align with operational, regulatory and security risk tolerance.

33 https://github.com/copilot.

34 https://aws.amazon.com/q/developer/.

35 https://www.tabnine.com/.

36 https://www.jetbrains.com/ai/.

37 https://cursor.com/.

5. AI coding assistants and code quality

One of the clear effects of the widespread adoption of AI coding assistants is that it has led to a significant increase in the volume of code produced. In addition, in 2025, it was claimed that, within a year, 90 % of all new code may be written by AI [38]. The large-scale uptake of AI coding tools, their intensive use throughout the developer's workflow and the growing trend towards autonomous agentic engineering raise concerns regarding the quality and reliability of code generated by AI. Because AI-generated code tends to be repetitive, uniform and pattern-based, it reshapes the stylistic, structural and quality norms of software engineering [39]. This means that the approaches used to evaluate the code quality of human-generated code may not be suitable for the assessment of code generated by AI. Another issue is that software engineering teams often rely on a patchwork of AI tools, each of which is based on a different model and a different understanding of the existing codebase. This may lead to inconsistent code quality and additional security issues [40].

A recent large-scale study [41], comparing human-written and AI-generated code with a focus on software quality, provides a few interesting insights.

- There are systematic differences between the defects contained in AI-generated code and those in human-written code. AI produces code that is less structurally complex, but more repetitive and prone to a specific set of defects (e.g. variable assignment errors). Human-authored code tends to exhibit more algorithmic flaws and inappropriate exception handling, which are associated with the low maintainability and suboptimal code design choices in real-world codebases.
- AI models tend to generate better-quality Python code, while the quality of Java code is significantly worse (the study focused only on Python and Java as they are the two most widely used programming languages representing different programming paradigms and used for different purposes).
- AI-generated code is more likely to trigger vulnerabilities associated with high-risk common weakness enumeration (CWE) categories, such as command injection and hard-coded secrets. Code generated by AI is, on average, more vulnerable than human-generated code.

A smaller study[42], carried out by a vendor of software for code review, also suggests that code written with the support of AI contains ca 1.7 times more issues than code generated by human developers. Other findings of the study can be summarised as follows:

- AI-generated code contains more critical or major issues, and logic/correctness issues are 75 % more common in AI-generated code;
- code written with AI support also violates established structures or style, leading to code readability concerns;
- AI-generated code contains 1.5 times more security issues;
- AI-generated code introduces naming inconsistency and mismatched terminology.

38 Procopio, J., 'Anthropic's CEO said all code will be AI-generated in a year', Inc. website, 20 March 2025, https://www.inc.com/joe-procopio/anthropics-ceo-said-all-code-will-be-ai-generated-in-a-year/91163367.

39 Cotroneo D., Improta, C. and Liguori, P., 'Human-written vs. AI-generated code: A large-scale study of defects, vulnerabilities and complexity', arXiv preprint paper, 2025, https://doi.org/10.48550/arXiv.2508.21634.

40 Cortex, *Engineering in the Age of AI: 2026 benchmark report*, San Francisco, CA, 2026, https://go.cortex.io/rs/563-WJM-722/ages/2026-Benchmark-Report.pdf?version=0.

41 Cotroneo D., Improta, C. and Liguori, P., 'Human-written vs. AI-generated code: A large-scale study of defects, vulnerabilities and complexity', arXiv preprint paper, 2025, https://doi.org/10.48550/arXiv.2508.21634.

42 Cortex, *Engineering in the Age of AI: 2026 benchmark report*, San Francisco, CA, 2026, https://go.cortex.io/rs/563-WJM-722/ages/2026-Benchmark-Report.pdf?version=0.

The development of AI coding assistants over the past three years suggests that they are likely to close the quality gap in the near future

All this combined may lead to the overall degradation of code quality and necessitate additional attention during code review. Therefore, even if AI enhances developer productivity, the increase in the overall volume of code produced and the number of issues lead to a significant increase in time spent on code review. Similarly, resolution times for issues in AI-generated code tend to increase due to the abovementioned code readability issues.

However, not all studies show degradation of code quality with the adoption of AI coding assistants. The 2025 Google Cloud DevOps Research and Assessment (DORA) study on the impact of generative AI in software development [43] suggests that the adoption of AI tools leads to improvement in documentation quality, code quality and code review speed, although these results should be taken with a pinch of salt, considering that they are based on survey data and not a systematic evaluation of code.

The development of AI coding assistants over the past three years suggests that they are likely to close the quality gap in the near future. Meanwhile, there are means of improving the quality of output produced by AI coding assistants, such as providing them with access to high-quality context. Therefore, effective deployment of AI coding tools requires a high-quality and well-maintained codebase, with well-maintained documentation in machine-readable formats [44].

43 Google Cloud, *DORA – Impact of generative AI in software development*, Google, Mountain View, CA, 2025, https://services.google.om/fh/files/misc/dora-impact-of-generative-ai-in-software-development.pdf.

44 ibid.

6. Adoption of AI coding tools and their impact on productivity

Since their introduction, AI coding assistants have been widely adopted. A small-scale survey (of 609 developers) was carried out by Qodo; the results were published in a report in 2025 [45] and suggest that over 80 % of developers use AI coding tools on a regular basis (daily or weekly) [46]. Digging deeper into the specifics shows a more nuanced picture. For example, while close to 60 % of respondents believe that the quality of their code has improved with the use of AI coding assistants, over 20 % believe that the quality of their code has been degraded. Similarly, nearly half of the respondents to the survey indicated that AI coding tools had a minimal positive effect, no effect or a negative effect on developer productivity.

Similar results can be observed from the large-scale developer survey carried out by Stack Overflow, which involved close to 50 000 respondents. This survey also shows that over 80 % of engineers use AI tools in the development process; however, when it comes to trust in the accuracy of these tools, 46 % of respondents are distrusting. Consequently, developers are unlikely to trust the ability of AI coding tools to handle complex tasks (only 29 % of developers believe AI coding tools can handle these) and are unlikely to delegate high-responsibility tasks to them, such as deployment and monitoring (less than 25 % plan to do so). One of the main frustrations with AI tools that has been cited by the respondents is that code produced by AI is almost, but not entirely, correct, which results in significant time spent debugging the code [47]. An early study that looked at how programmers used AI coding tools argued that output produced by the coding assistant can also break programming flow: if the AI tool is offering a long suggestion (e.g. over 10 lines of code), the programmer is likely to dismiss it or spend time reviewing it before eventually dismissing it. This could negatively affect productivity and cause some programmers to give up on the tool entirely [48].

Some studies relying mainly on synthetic tasks [49] have showed a positive impact on developer productivity resulting from the introduction of AI coding assistants [50]. Other studies using randomised

45 For more details, see Qodo, *Qodo Report – 2025 state of AI code quality*, Tel Aviv, 2025, https://www.qodo.ai/wp-content/uploads/2025/06/2025-State-of-AI-Code-Quality.pdf.

46 Considering that the survey was carried out by the provider of an AI coding assistant, the results may be biased.

47 Similar findings were reported in Google Cloud, *DORA – State of AI-assisted software development*, Google, Mountain View, CA, 2025, https://services.google.com/fh/files/misc/2025_state_of_ai_assisted_software_development.pdf.

48 Barke, S., James, M. B. and Polikarpova, N., 'Grounded Copilot: How programmers interact with code-generating models', *Proceedings of the ACM on Programming Languages*, Vol. 7, Issue OOPSLA1, 2023, pp. 85–111, https://doi.org/10.1145/3586030.

49 A synthetic task refers to a programming problem that has been constructed specifically for benchmarking purposes, rather than drawn from real-world software development work.

50 See, for example, the following:

Peng, S., Kalliamvakou, E., Cihon, P. and Demirer, M. 'The impact of AI on developer productivity: Evidence from GitHub Copilot', arXiv preprint paper, 2023, https://doi.org/10.48550/arXiv.2302.06590;

Paradis, E., Grey, K., Madison, Q., Nam, D., Macvean, A. et al., 'How much does AI impact development speed? An enterprise-based randomized controlled trial', arXiv preprint paper, 2024, https://doi.org/10.48550/arXiv.2410.12944;

Weber, T., Brandmaier, M., Schmidt, A. and Mayer, S., 'Significant productivity gains through programming with large language models', *Proceedings of the ACM on Human–Computer Interaction*, Vol. 8, Issue EICS, 2024, pp. 1–29, https://doi.org/10.1145/3661145.

controlled trials and other techniques to evaluate the effects of AI coding tools on developer productivity carrying out tasks in real-world settings (e.g. open-source development tasks) produced more mixed results, with at least one study suggesting that the use of AI coding assistants made developer work less efficient by about 19 % [51]. An anecdotal account of a programmer attempting to replicate the study produced similar results, making him (an experienced programmer) less efficient by ca 21 % [52]. Furthermore, the author also analysed data on the volume of software output since 2022 [53] and did not identify a significant increase compared with the time before the introduction of AI coding tools.

The study referred to above suggests that the increase in productivity and the consequent increase in software output claimed by the providers of these tools have not yet materialised. This does not mean that the tools are not and will not be useful. AI in general, and LLMs more specifically, are rapidly evolving technologies. Therefore, it is important to continuously follow external research and carry out internal evaluations to have a better understanding of the impacts of these tools on developer productivity.

In addition to the mixed results regarding the effect of AI coding assistants on developer productivity, several studies also claim that the use of AI coding assistants is likely to result in security degradation. One paper analysed security degradation for AI-generated code in a controlled experiment, using iterative prompting strategies with the aim to improve the code produced. The results showed a 37.6 % increase in the number of vulnerabilities after five iterations, thus challenging the assumption that iterative refinement through prompting improves code security, and emphasising the need for human expertise in the loop [54]. As a result, the efficiency gained in development may be lost or significantly reduced due to the need for more extensive code review. Security aspects of the use of AI coding tools are covered in the next section.

51 Becker J., Rush, N., Barnes, E. and Rein, D., 'Measuring the impact of early-2025 AI on experienced open-source developer productivity', arXiv preprint paper, 2025, https://doi.org/10.48550/arXiv.2507.09089.

52 Judge, M., 'Where's the shovelware? Why AI coding claims don't add up', Substack website, 3 September 2025, https://mikelovesrobots.substack.com/p/wheres-the-shovelware-why-ai-coding.

53 Software output measured as the number of new iOS or Android app releases, new public GitHub repositories, etc.

54 Shukla S., Joshi, H. and Syed, R., 'Security degradation in iterative AI code generation: A systematic analysis of the paradox', *IEEE International Symposium on Technology and Society (ISTAS)*, 2025, pp. 1–8, https://doi.org/10.1109/ISTAS65609.2025.11269659.

7. Security implications of AI coding assistants

Although the security of software is associated with code quality, it deserves a dedicated section, as it is one of the better-studied aspects of software development supported by AI coding assistants. Security risks linked to the adoption of AI coding assistants can be roughly divided into three layers:

- **output layer –** models generating insecure code;
- **input or data layer –** models processing sensitive code and other data;
- **agent layer –** new attack surface created by autonomous AI agents.

Each of these categories or layers has a different threat model, and different mitigation strategies, all of which are addressed in this section.

Let's look first at the **output layer.** Security of AI-generated code is the most studied category of security risk due to the relative simplicity of performing experiments with AI coding models. One of the earliest studies [55], completed in 2021, assessed the security of code produced by GitHub Copilot against well-defined security weaknesses in MITRE's CWE [56]. The study concluded that ca 40 % of code completions suggested by Copilot contained vulnerabilities. This is probably due to the code corpus on which Copilot was trained, open-source code from GitHub, which contains vulnerabilities. The effect of time is another plausible explanation for the prevalence of security issues: the state-of-the-art security practices become outdated due to the evolving cybersecurity landscape, while old coding practices may still be contained in the model, thus replicating these vulnerabilities in newly generated code. The authors therefore suggest that Copilot should be paired with appropriate security-aware tooling during both model training and code generation to minimise security vulnerabilities.

The authors of a more recent paper, presented at a conference in late 2025, carried out a large-scale analysis of public GitHub repositories [57]. The analysis identified 4 241 CWE instances across 77 vulnerability types; and while 87.9 % of AI-generated code did not contain identifiable CWE vulnerabilities, there were significant differences regarding vulnerability rates across languages, with Python being consistently more prone to security flaws, while JavaScript and TypeScript were consistently less prone to containing vulnerabilities.

A pervasive issue with the use of AI coding assistants is the trust developers put in these tools. A 2023 study on how developers use AI coding assistants to solve security-related tasks concluded that participants with access to an AI assistant were more likely to write incorrect and insecure solutions across a range of experimental tasks than a control group performing the same task without AI assistants. The AI assistants often used unsafe libraries, used libraries inappropriately, did not understand the edge cases of interacting with file systems or databases or did not correctly sanitise user input. This was further exacerbated by the widespread trust of participants in outputs produced by AI coding tools, which led to limited verification of these outputs and potentially undetected

55 Pearce H., Ahmad, B., Tan, B., Dolan-Gavitt, B. and Karri, R., 'Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions', arXiv preprint paper, 2021, https://doi.org/10.48550/arXiv.2108.09293.

56 https://cwe.mitre.org/.

57 Schreiber, M. and Tippe, P., 'Security vulnerabilities in AI-generated code: A large-scale analysis of public GitHub repositories', in: Han, J., Xiang, Y., Cheng, G., Susilo, W., Chen, L. (eds), Information and Communications Security, ICICS 2025, Lecture Notes in Computer Science, Vol. 16219, Springer, Singapore, 2026, https://doi.org/10.1007/978-981-95-3537-8_9

vulnerabilities in AI-generated code [58].

Hallucination is a known issue of LLMs; this is also a security concern in the context of AI coding assistants, which may generate non-existent software packages or libraries. As code produced by LLMs can often be trusted by developers, and therefore subject to limited review, these non-existent software libraries could be widely used across codebases and later exploited by malicious actors, by introducing malware, for example.

An issue related to code security is code maintainability. The use of AI assistants may lead to codebases that are more difficult to maintain in the long term. Poor code maintenance may also result in security vulnerabilities [59].

Looking at the **input layer**, an important concern is enterprise code privacy, which may arise when developers send parts of source code to cloud-based AI coding assistants as part of their prompts, or when providing AI coding assistants access to the code repository for context. Here, the risks will depend on the specific provisions in the terms of service on data retention and the use of proprietary data for training. Most mainstream providers of AI coding assistants include privacy safeguards and ensure the protection of proprietary information for enterprise users, by allowing them to opt out of the use of user data for training. However, some logged data or telemetry used for debugging, monitoring and quality assurance, which may include code snippets and prompts, may be stored and later used for model training.

Finally, let's turn to the **agent layer**. With the development of AI coding tools from simple autocomplete to far more sophisticated agentic assistants capable of performing complex workflows autonomously, the security risks gained a new qualitative dimension. This is because, in order to function, agentic tools are provided with access to the file system, shell execution and web browsing capabilities. The main feature of LLMs, namely that they operate using inputs expressed in natural language, makes them vulnerable to indirect prompt injection [60]. Agentic systems may interact with external tools, capabilities and data sources; without effective control, they can be attacked through all these channels [61], thus creating a data exfiltration risk.

> *By creating integrations between LLMs and enterprise systems using MCP servers, traditional security boundaries that rely on system isolation are effectively removed. Therefore, the use of one compromised MCP server may affect multiple systems*

The use of model context protocol (MCP) as an interface for connecting AI assistants with external tools introduces an additional vulnerability with a number of attack vectors, which arises due to the trust between AI agents and external tools. By creating integrations between LLMs and enterprise systems using MCP servers, traditional security boundaries

58 Perry N., Srivastava, M., Kumar, D. and Boneh, D., 'Do users write more insecure code with AI assistants?', *CCS '23: Proceedings of the 2023 ACM Sigsac conference on computer and communications security*, 2023, pp. 2785–2799, https://doi.org/10.1145/3576915.3623157.

59 Gent, E., 'AI coding is now everywhere. But not everyone is convinced.' MIT Technology Review website, 15 December 2025, https://www.technologyreview.com/2025/12/15/1128352/rise-of-ai-coding-developers-2026/.

60 OWASP GenAI Security Project, 'LLM01: 2025 – Prompt injection', OWASP GenAI Security Project website, https://genai.owasp.org/llmrisk/llm01-prompt-injection/.

61 A comprehensive taxonomy of attacks on agentic systems is provided in Maloyan N. and Namiot, D., 'Prompt injection attacks on agentic coding assistants: A systematic analysis of vulnerabilities in skills, tools and protocol ecosystems', *International Journal of Open Information Technologies*, Vol. 14, No 2, 2026, pp. 1–10, http://injoit.org/index.php/j1/article/view/2423.

that rely on system isolation are effectively removed. Therefore, the use of one compromised MCP server may affect multiple systems. MCP servers may be vulnerable to a range of attack vectors, including supply chain attacks (e.g. unvetted dependencies within MCP server packages), implementation errors in authorisation and authentication processes, tool and agent manipulation attacks (e.g. prompt injection and tool poisoning) and local execution risks [62].

This limited overview of the security challenges associated with the use of AI coding assistants suggests that the potential productivity gains in code delivery should be assessed against the effort necessary to ensure the security of software developed with the help of AI. This does not mean that securing AI-supported development is not feasible; different approaches have been developed to tackle the key security challenges associated with the use of LLMs, including in the context of AI coding assistants.

The most obvious way to reduce security risks when using LLMs as coding assistants is to deploy LLMs in a local isolated environment

The most obvious way to reduce security risks when using LLMs as coding assistants is to deploy LLMs in a local isolated environment. This will ensure that all data and code remain within the local environment. However, this also brings certain challenges. For example, ensuring a certain level of performance of local LLMs will require investment in dedicated hardware (graphics processing units (GPUs), storage). LLMs will be mostly limited to open-source LLMs, which come with the possibility of supply chain attacks. As open-source models are updated less frequently than commercial models, the likelihood that the code produced by open-source models will contain vulnerabilities may potentially be higher. Finally, the responsibility for establishing security operations for local LLMs, and for liability in the case of an incident, is with the organisation.

The capabilities for machine learning (CaMeL) framework aims to protect LLMs by enveloping them in a protective layer, which extracts control and data flows from user queries and uses a Python interpreter to enforce security policies without any modifications to the LLMs [63]. There are other mitigation measures to address security challenges pertinent to the output layer. For example, AI-generated code should be subjected to mandatory code review using code quality tools and static code analysis tools (e.g. SonarQube), and to a security-focused review targeting common AI failure modes (e.g. hard-coded secrets). Software composition analysis tools (e.g. Open Worldwide Application Security Project (OWASP) dependency-check [64]) can help identify outdated or vulnerable dependencies (e.g. non-existing packages).

When it comes to the input layer, it is important to ensure that the terms of service are very clear, that no telemetry or logging data are collected and used for model training, to ensure high levels of security, and that the solution is compliant with the applicable law (e.g. the General Data Protection Regulation(GDPR)). For example, Microsoft ensures compliance, in particular regarding EU data residency, by establishing an EU data boundary, as does AWS with the European sovereign cloud. Additional organisational and technical measures (e.g. self-hosting or sovereign cloud) should also be considered to ensure

62 Ruiz, F., 'The model context protocol (MCP): Architecture, security risks, and best practices', Fluid Attacks website, 14 November 2025, https://fluidattacks.com/blog/model-context-protocol-mcp-security.

63 Debenedetti E., Shumailov, I., Fan, T., Hayes, J., Carlini, N. et al., 'Defeating prompt injections by design', arXiv preprint paper, 2025, https://doi.org/10.48550/arXiv.2503.18813.

64 The OWASP Foundation, 'OWASP Dependency-Check', OWASP website, https://owasp.org/www-project-dependency-check/.

higher levels of data security. Deploying secret scanning tools (e.g. TruffleHog or GitHub secret scanning) is also an important protective measure.

The secure operation of MCP servers can also be facilitated by employing a range of protective measures focusing on server vetting and supply chain security, by enforcing access controls, by implementing barriers to prevent model manipulation through input controls and by the hardening and monitoring of infrastructure on which MCP servers are run [65].

Mitigation measures to ensure the security of agentic workflows are also available. For example, to ensure that agents do not directly interfere with production software, agentic tools such as Devin and GitHub Copilot coding agent can run in a fully sandboxed cloud environment, executing code in a fully isolated cloud environment. For complete isolation, it is also possible to run agentic coding workflows on local infrastructure using open-source tools such as goose [66], which allow the user to run AI agents on a local machine, connect to a range of extensions (databases, APIs, browsers, etc.) via MCPs and use any LLM via the Agent Client Protocol [67]. Securing agentic workflows by running those on local infrastructure addresses the needs of organisations that cannot use cloud-based services due to high-level security requirements.

In addition, horizontal organisational practices, such as AI usage policies, robust governance systems, periodic security audits and developer training and awareness, play an important role in security risk mitigation. Finally, standard security practices, such as containerisation, permission systems, audit logging, etc., remain relevant as elements of the overall security framework.

65 For a more comprehensive overview, see, for example, the following:

Model Context Protocol, 'Security best practices', Model Context Protocol website, https://modelcontextprotocol.io/docs/tutorials/security/security_best_practices;

Ruiz, F., 'The model context protocol (MCP): Architecture, security risks, and best practices', Fluid Attacks website, 14 November 2025, https://fluidattacks.com/blog/model-context-protocol-mcp-security.

66 https://goose-docs.ai/.

67 https://agentclientprotocol.com/get-started/introduction.

Table 2: Summary of security risks and mitigation measures

Layer	Security risks	Mitigation measures
Output	• AI-generated code contains security vulnerabilities • Developers over-trust AI output and skip review, leaving vulnerabilities undetected • Use of unsafe or inappropriate libraries and failure to sanitise input • Hallucinated (non-existent) packages that can be hijacked by malicious actors • Poor code maintainability, leading to long-term security debt	• Mandatory code review with static analysis tools (e.g. SonarQube) • Security-focused review targeting AI failure modes (e.g. hard-coded secrets) • Software composition analysis to detect vulnerable or non-existent dependencies (e.g. OWASP dependency-check) • Pair AI coding tools with security tooling during model training and code generation • Using CaMeL (*)-style protective layers to enforce security policies around LLMs without modifying the model
Input	• Proprietary source code sent to cloud-based AI tools may be retained or used for model training • Logged telemetry data (including prompts) may be stored and used for training • Risk of non-compliance with data protection law (e.g. the GDPR)	• Deploy LLMs in a local isolated environment to keep all data and code on the premises • Verify that terms of service explicitly prohibit telemetry / logging data for use in model training • Use providers offering EU data residency or sovereign cloud
Agent	• Indirect prompt injection via external tools, data sources or web content that agents interact with • Data exfiltration through agentic access to file systems, shell execution, browsers • MCP supply chain attacks via unvetted dependencies in MCP server packages • Implementation errors in MCP server authorisation and authentication • Tool poisoning and agent manipulation attacks through compromised MCP servers • Removal of traditional system isolation boundaries when LLMs connect to enterprise systems via MCP	• Run agentic tools in fully sandboxed cloud environments • Run agentic workflows on local infrastructure for complete isolation • Vet MCP servers: enforce access controls, implement input controls to prevent model manipulation, harden and monitor server infrastructure
Cross-cutting measures	AI usage policies, governance frameworks, periodic security audits, developer training and awareness, containerisation, permission systems, audit logging	

* Debenedetti E., Shumailov, I., Fan, T., Hayes, J., Carlini, N. et al., 'Defeating prompt injections by design', arXiv preprint paper, 2025, https://doi.org/10.48550/arXiv.2503.18813.

8. Organisational and societal implications

There is no doubt that AI coding assistants are reshaping software development as a profession and, by extension, the software development industry. However, the effects are mixed and context-dependent.

It is true that the widespread use of AI coding assistants will probably lead to a significant surge in code output, pull requests and commits. More holistic productivity metrics that incorporate less straightforward performance metrics are likely to lead to a more nuanced picture (e.g. adding code instability metrics to pure throughput, as is done in the DORA framework [68]). If AI-produced code leads to more instability and therefore requires additional work to address failed changes, some efficiency gained in terms of throughput will be lost in addressing instability.

As mentioned above, AI-produced code may also lead to poor code maintainability and to security vulnerabilities. This has repercussions at the organisational level: a developer team producing an initial codebase may be praised for its efficiency, but other teams responsible for code review, for code maintenance down the line or for system security may be negatively affected. Nevertheless, considering the pace at which these technologies are evolving, the situation may change and the concerns about the quality, maintainability and security of code produced by AI may become irrelevant in the not-so-distant future.

Another challenge for organisations is the potential loss of the talent pipeline if most of the work done by junior developers is delegated to AI-based tools.

Similarly, heavy use of AI-driven tools by software engineers may eventually lead to other unintended consequences: lower levels of work satisfaction among developers, if their main responsibilities are reduced to reviewing code produced by AI, and deskilling of software developers due to extensive use of AI coding assistants in the long term [69].

An additional unintended consequence of the widespread use of AI coding assistants and the increasing democratisation of software development through agentic systems and vibe coding may be societal issues, especially in countries where significant emphasis has been placed on education in computer science, software engineering and adjacent fields in the past two decades [70].

One of the positive outcomes of the accessibility of software development through vibe coding at the organisational level is the democratisation of experimentation. Allowing subject matter experts who are not software developers to build functioning software and experiment with it in sandboxed environments will significantly reduce the load on in-house software teams, at least in the initial stages, such as building prototypes (minimum viable products) to test ideas and new functionalities [71]. Similarly, domain experts can now build custom tools that are not mission-critical without significant financial and time commitments. However, this may also be a double-edged sword, in that reducing barriers to experimentation with software may also lead to increased demand for in-house software teams to maintain and troubleshoot AI-generated software.

---

68 Harvey, N., 'DORA's software delivery performance metrics', DORA website, 5 January 2026, https://dora.dev/guides/dora-metrics/.

69 See, for example, Becker J., Rush, N., Barnes, E. and Rein, D., 'Measuring the impact of early-2025 AI on experienced open-source developer productivity', arXiv preprint paper, 2025, https://doi.org/10.48550/arXiv.2507.09089.

70 For an overview of the employment effects of AI on early-career workers, see, for example, Brynjolfsson E., Chandar, B. and Chen, R., 'Canaries in the coal mine? Six facts about the recent employment effects of artificial intelligence', *Stanford Digital Economy Lab*, 2025, https://digitaleconomy.stanford.edu/wp-content/uploads/2025/11/CanariesintheCoalMine_Nov25.pdf.

71 Mehta, I., 'A quarter of startups in YC's current cohort have codebases that are almost entirely AI-generated', TechCrunch website, 6 March 2025, https://techcrunch.com/2025/03/06/a-quarter-of-startups-in-ycs-current-cohort-have-codebases-that-are-almost-entirely-ai-generated/.

Finally, there is also the risk of vendor concentration. Considering that these tools rely on foundation models provided by a handful of key players (Anthropic, Google, OpenAI), such concentration may result in adverse effects due to a number of risks (e.g. model outage, pricing changes, model deprecation, geopolitical events).

Conclusions

AI coding assistants are no longer an experiment. According to survey data, in 2025, at least 85 % of developers used these tools on a regular basis; the use of AI coding tools can be considered standard practice in organisations developing software. Therefore, the question is no longer whether to adopt such tools, but how to adopt them in a way that helps organisations deliver software while minimising the associated risks.

Therefore, the question is no longer whether to adopt such tools, but how to adopt them in a way that helps organisations deliver software while minimising the associated risks

Looking beyond the hype, especially regarding the possibilities opened up by vibe coding, productivity gains are real but not evenly distributed. Junior developers are likely to experience significant gains in productivity, but, given their limited understanding of the code produced, this is likely to have a negative effect on senior developers who are responsible for code review. Most of these gains, at least for the moment, will appear in certain categories of work that are relatively standardised, whereas developers working on more complex problems are, for the time being, unlikely to benefit from the use of AI coding tools.

Owing to the quality and security issues associated with AI-generated code, some of the productivity benefits for developer teams will be counterbalanced by the additional work necessary for code review and ensuring the security of code. Some of this work may be covered by dedicated AI tools developed specifically for code review. However, ultimately, human beings bearing responsibility for the security and performance of code will need to decide whether AI tools can be trusted with ensuring the security and stability of AI-generated code.

The use of AI coding assistants is also not cost-free. Operating AI coding assistants locally will require investments in hardware, software and staff resources to maintain the local solutions. The costs of using cloud-based AI coding assistants (e.g. Claude Code) are calculated primarily based on token usage [72]. Although Anthropic estimates suggest that, for 90 % of users of Claude Code, costs remain under EUR 30 per active day [73], the costs can vary depending on the size of the context used and can rise significantly if AI agent teams are deployed. It is also not clear whether current pricing will be maintained by the service providers in the long term. Therefore, a robust cost–benefit analysis needs to be performed to assess whether the productivity (and other) benefits outweigh the costs of services and other associated costs.

AI broadly, and generative AI specifically, is developing at such a rapid pace that the only meaningful conclusion one can make is that this paper will be outdated as soon as it is published. Therefore, it is extremely important to continuously monitor and keep abreast of developments in these technologies. Models that are not capable of something today will likely be capable of it within months. In fact, the recent announcement by Anthropic (responsible for the Claude family of models) of the launch of Project

72 Costs are calculated using the price per million tokens (MTok), which is different for input and output (ca EUR 3–5 per MTok for input tokens and EUR 15–25 per MTok for output tokens for Claude Code, for example, depending on the model used). A token is the smallest unit into which text data can be broken down for an AI model to process (e.g. word/character).

73 Claude Code Docs, 'Manage costs effectively', Claude Code Docs website, https://code.claude.com/docs/en/costs.

Glasswing suggests that LLMs, specifically the unreleased model Claude Mythos, have reached a level of coding capability that surpasses humans at finding and exploiting software vulnerabilities [74]. Whether this announcement is a piece of strategic marketing or reflects reality is impossible to verify; however, observing the pace of development in LLMs over the past few years, it seems like a plausible scenario, if not now, then certainly in the not-so-distant future.

If this is true, it has several implications for the future of software security and for the need for global cooperation, like that developed in the area of non-proliferation of nuclear weapons, considering the severe consequences such capabilities may have across all areas of life.

74 Anthropic, 'Project Glasswing', Anthropic website, 7 April 2026, https://www.anthropic.com/glasswing.

Manuscript completed in June 2026.

Luxembourg: Publications Office of the European Union, 2026

ISBN 978-92-95237-22-3 doi:10.2857/7440006 Catalogue number: EL-01-26-016-EN-N