TECHNOLOGIES AND ENGINEERING

Journal homepage: https://technologies-engineering.com.ua/en
Vol. 26, No. 1, 2025

Received: 29.08.2024
Revised: 20.01.2025
Accepted: 26.02.2025

UDC 004.021
Doi: 10.30857/2786-5371.2025.1.3

Optimising productivity and automating software development: Innovative memory system approaches in large language models

Olena Sokol*
Master of Science
Taras Shevchenko National University of Kyiv
01601, 60 Volodymyrska Str., Kyiv, Ukraine
https://orcid.org/0009-0005-5160-460X

Abstract. This study explored innovative approaches to enhancing memory systems in large language models to improve efficiency and automate software development. The primary focus was on optimising memory systems that enable long-term context storage and facilitate model adaptation to evolving interaction conditions. The research analysed contemporary methods of data storage and processing that enhance the ability of models to handle large volumes of information efficiently. This included the utilisation of specialised algorithms and memory mechanisms that improve the accuracy and adaptability of large language models in executing complex tasks. A secondary focus of the study examined the capabilities of large language models in automating software development. It assessed how these models can generate code, optimise it, and perform error detection. Particular attention was given to analysing the impact of automation on software quality and development time. In this context, the study investigated the use of large language models for automating repetitive tasks, generating tests, and implementing best programming practices. The findings indicated that enhancing the memory systems of large language models significantly improves their efficiency in tasks requiring long-term interaction. Integrating such models into software development processes has been shown to reduce both time and resource expenditures while enhancing product quality. The practical significance of this study lies in the formulation of recommendations for the optimal utilisation of large language models in the field of information technology

Keywords: contextual processing; algorithm optimisation; code generation; intelligent agents; big data processing; automation; interaction modelling

Introduction

The modern development of artificial intelligence (AI) is opening up new opportunities across various domains, including software development. Large language models (LLMs) occupy a central role in this process due to their ability to analyse, generate, and adapt to changing conditions. A crucial factor influencing the performance of these models is the memory system. Through memory optimisation, LLMs can interact effectively with users, preserve context, and adapt to new data. This study aims to explore innovative approaches to the utilisation of memory systems in LLMs, as well as to analyse the potential for automating software development using these models.

One of the key challenges in the operation of large language models is the preservation of long-term context when processing vast amounts of data. Traditional approaches to information processing often face limitations in speed and accuracy, which impact the adaptability of these models. W. Zhong *et al.* (2024) proposed the MemoryBank system, which enhances the long-term memory of language models by enabling more efficient context storage and retrieval. This significantly improved the accuracy and stability of models when handling lengthy texts. Innovative methods such as hierarchical memory structures, dynamic context windows, and compression algorithms have facilitated more efficient data storage. Q. Wu *et al.* (2020) introduced Memformer, an extended memory transformer that allows models to manage complex dependencies while optimising memory usage.

Suggested Citation:
Sokol, O. (2025). Optimising productivity and automating software development: Innovative memory system approaches in large language models. *Technologies and Engineering,* 26(1), 36-44. doi: 10.30857/2786-5371.2025.1.3.

*Corresponding author

Another important aspect is the integration of memory with big data storage and processing. The use of distributed memory systems, cloud computing, and specialised hardware solutions can significantly enhance model performance. For example, W. Kwon *et al.* (2023) proposed PagedAttention, a method for efficient memory management, which reduced latency and increased the speed of real-time data processing. S. Sagi (2024) explored optimisation techniques for graphics processing units (GPUs) aimed at improving the performance of LLMs, enabling the modelling of more complex problems while reducing computational costs. Innovations in memory technology also create new opportunities for the development of intelligent agents. With improved memory systems, such agents can not only analyse data in real time but also make informed decisions based on accumulated experience.

The automation of software development is another promising application of large language models. By leveraging automation capabilities such as code generation, optimisation, and verification, developers gain powerful tools to reduce development time and enhance software quality. M. Schäfer *et al.* (2023) investigated the use of LLMs for automated testing, demonstrating how these models generate test scripts and verify component compatibility. Beyond code generation, LLMs are also capable of optimising code. For instance, B. Liu *et al.* (2024) examined the potential of automatic software refactoring using LLMs, highlighting their contribution to improving development efficiency and quality. Moreover, ensuring high-quality software solutions throughout their lifecycle is essential for enhancing the efficiency of LLMs. As A. Shantyr (2024) states, the use of combined quality models not only facilitates the assessment of software system efficiency but also ensures compliance with established criteria throughout the entire development process.

LLMs are capable of automatically generating code based on specifications provided by developers. For example, X. Jiang *et al.* (2024) proposed a self-scheduling approach to code generation, enabling models to solve complex problems with minimal user intervention. Another significant area is automated code testing. F.F. Xu *et al.* (2022) examined the effectiveness of LLMs in generating test scenarios, highlighting their ability to detect errors at an early stage. Additionally, Z. Zheng *et al.* (2025) investigated the role of LLMs in software engineering tasks, focusing on their capacity to automate complex processes such as design and testing. Their findings underscored the importance of advancing memory systems in LLMs to enhance performance and adaptability. By integrating modern memory technologies and data processing algorithms, these models become more effective in tasks requiring long-term context retention and flexible interaction with users. This has created significant opportunities for software development automation, allowing for substantial reductions in the time and resources required to produce high-quality software products.

The aim of this study was to develop and evaluate innovative approaches to the utilisation of memory systems in LLMs, with the objective of improving their performance and adaptability to evolving interaction conditions. Additionally, the study explored the potential of these models for automating software development.

Materials and Methods

To achieve the research objective, a comprehensive experimental approach was developed, incorporating innovative memory mechanisms in LLMs to optimise their performance. The primary focus was on testing and comparing different memory management methods, including MemoryBank, Memformer, and PagedAttention, to enhance the efficiency and accuracy of models in software development automation scenarios. The experiments were conducted on a high-performance computing infrastructure comprising clusters equipped with NVIDIA A100 graphics processors, enabling efficient data processing through parallel computing. The Amazon Web Services (AWS) cloud platform was utilised to scale resources, integrated with local servers. This setup facilitated the creation of a flexible environment for handling large datasets and various model configurations.

The three selected memory mechanisms were integrated into the foundational architecture of GPT-like models:

- MemoryBank, designed for long-term context storage through a hierarchical data structure, allowing for efficient information retention and retrieval when performing complex tasks. The memory access operation can be formalised as:

$$m_t = \alpha \times m_{t-1} + (1-\alpha) \times h_t + \textstyle\sum_{l=1}^{L} \beta_l \times c_{t-l}, \qquad (1)$$

where $\alpha \in 0.1$ – memory decay rate, β_l – layer-specific weights ($\sum_{l=1}^{L}$), c_{t-l} – context vectors from l-th layer, h_t – current hidden state.

- Memformer, a transformer with extended memory, capable of managing complex dependencies between tokens, thereby improving generation accuracy and analytical capabilities. The Memformer extends standard attention with (2):

$$\begin{aligned} &Attention(Q, K, V, M_t) = \\ &= softmax((QK^T + QM_t^T)/\sqrt{\ } d_k) \times [V\,; M_t\,], \end{aligned} \qquad (2)$$

where M_t is the persistent memory at time t, updated according to (3):

$$M_{t+1} = f(M_t, X_t, A_t), \qquad (3)$$

where f being a learnable update function, X_t the current input, and A_t the attention matrix.

- PagedAttention, a dynamic memory management approach that loads only relevant contextual data, optimising processing speed. The paged attention mechanism computes attention scores as:

$$A_{ij} = (e^{(q_i \times k_j)/\sqrt{\ } d)})/(\textstyle\sum_{\mathrm{l}} \in P(i)\, e^{(q_i \times k_l/\sqrt{\ } d)}), if j \in P(i) 0, otherwise, \qquad (4)$$

where $P(i)$ represents the set of token indices in pages relevant to token i, determined by a relevance function such that (5):

$$P(i) = j \vee relevance(i, j) > \theta, \tag{5}$$

where θ being a configurable relevance threshold.

To evaluate effectiveness, a test dataset comprising 10,000 diverse queries was created. This dataset included: generating text responses to short queries; creating large code segments in Python and C++; long-term interaction tasks where the query context exceeded 10,000 tokens. The performance and efficiency of the models were assessed using the following metrics: latency, measured in milliseconds to determine model response speed; tokens per Second (TPS), which assessed the number of tokens processed per second; accuracy, measuring the proportion of correct responses or generated code that met predefined requirements; memory Usage, indicating the amount of memory consumed by the model during task execution.

The experimental procedure comprised three main stages. At the first stage, the models were provided with a technical specification detailing functional and non-functional requirements. The task was to generate code that fully met these specifications from the outset. At the second stage, automated testing was conducted, where the models generated test scripts to verify the code, assessing their ability to detect logical and syntactic errors. At the third stage, existing code bases were refactored to identify duplication, detect complex logical conditions, and optimise the overall structure (Fig. 1).

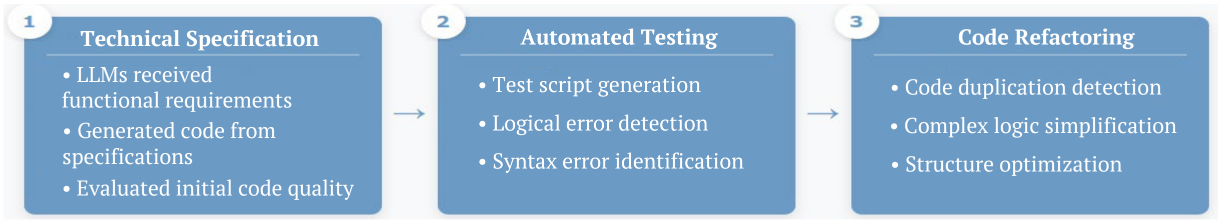
1
Technical Specification
• LLMs received functional requirements
• Generated code from specifications
• Evaluated initial code quality
2
Automated Testing
• Test script generation
• Logical error detection
• Syntax error identification
3
Code Refactoring
• Code duplication detection
• Complex logic simplification
• Structure optimization

***Figure 1**. Experimental procedure for model evaluation*

***Source:** compiled by the author*

The results of the experiments were analysed using statistical tests, including the t-test to compare the mean values of performance metrics across different memory configurations. The t-test statistic was calculated using (6):

$$t = (\bar{x}_1 - \bar{x}_2) / \sqrt{[(s_1^2/n_1) + (s_2^2/n_2)]}, \tag{6}$$

where $\bar{x}_1$ and $\bar{x}_2$ are the means of the compared memory configurations, s_1^2, s_2^2 are their variances, and n_1, n_2 are the sample sizes. Results were considered statistically significant at $p < 0.05$, representing a 95% confidence level. Additionally, a qualitative analysis of long-term interactions was carried out, involving manual error coding and an evaluation of the contextual relevance of responses. The proposed methodology enabled a comprehensive assessment of the impact of innovative memory mechanisms on the efficiency of large language models, providing a foundation for their further optimisation and practical application in software development automation.

Results and Discussion

The impact of advanced memory systems on LLM performance. The findings of this study indicate that innovative memory mechanisms significantly enhance key performance metrics of LLMs. During the experiments, three approaches to memory organisation – MemoryBank, Memformer, and PagedAttention – were tested against the baseline model, which did not incorporate additional optimisations. The evaluation focused on four key parameters: response time (Latency) in seconds, throughput (Tokens per Second, TPS), where TPS = NumberOfTokens/Time, accuracy (Accuracy) percentage, which shows the correct answers, and total memory consumption (Memory Usage). The results demonstrated a clear advantage for models equipped with advanced memory mechanisms. Table 1 presents the average performance metrics for the baseline LLM configuration alongside each of the three approaches.

***Table 1**. Comparative performance indicators of LLM with different memory systems*

LLM configuration	Latency (Mc)	TPS	Accuracy (%)	Memory usage (GB)
Basic	145	1,800	82	12
MemoryBank	100	2,200	88	10
Memformer	90	2,400	90	11
PagedAttention	85	2,500	91	9

***Source:** created by the author*

As can be seen from Table 1, PagedAttention showed the best results: the model with this mechanism had the lowest response time (Latency = 85 ms) and the highest accuracy (91%). Memformer was a close competitor, showing a slightly higher response time (90 ms) and TPS at 2,400. At the same time, MemoryBank showed an improvement compared to the baseline version, but was still slightly inferior in speed to the other two options.

The next step was to study long-term interaction, when the query context exceeded 10,000 tokens. This scenario simulates situations in which the model has to "remember" a large amount of information from previous messages or code fragments. The results showed an advantage for all three improved memory systems. PagedAttention again showed the highest efficiency, reducing the time to search for relevant context by 35% compared to the baseline method. Memformer and MemoryBank also proved to be significantly more efficient, providing 20-30% faster response times. At the same time, PagedAttention was distinguished by dynamically "loading" only those pages of context that are needed for current processing, which gave it an advantage over hierarchical or segmented methods. In addition to quantitative indicators, the quality of the models' responses in the long-term mode was evaluated. PagedAttention demonstrated the highest rate of correct reference to the previous context (92%), as well as error-free consideration of changes made to the original data. Memformer had about 89% of correct responses, while in certain scenarios losing small details implicitly mentioned in previous queries. MemoryBank (85%) showed a good ability to combine different data segments in complex questions, but sometimes mistakenly ignored secondary information if it was located outside the main "window" of memory.

Thus, these new methods not only accelerated data processing but also enhanced the ability of models to maintain and update long-term context – an essential factor in tasks involving a large number of interconnected steps. Contemporary research indicates that the integration of extended memory mechanisms significantly improves the performance of LLMs, influencing both inference speed and response quality. By preserving intermediate states during data processing, models can rapidly revisit previously obtained results and refine them without the need to recompute all steps from the beginning.

U. Antero *et al*. (2024) highlighted that LLMs equipped with sufficient memory capacity can perform automatic code generation and verification more effectively, as they retain logical connections and can reuse previously generated fragments. This capability is particularly valuable when tasks require an analysis of a long history of changes or the broader context of a project. Similarly, B. Kim *et al.* (2024) noted that reducing memory access latency is a critical factor in determining overall model performance. Optimising memory architecture enables reduced processing time, even in scenarios involving a high degree of parallel operations. Furthermore, Z.R.K. Rostam *et al.* (2024) demonstrated that a well-configured memory system can significantly accelerate both training and inference, which is especially crucial for resource-intensive models.

Additionally, D. Nguyen *et al.* (2024) proposed an approach in which the use of larger mini-batches, combined with optimised GPU allocation procedures, enhances training efficiency. They emphasised that memory mechanisms play a crucial role in preventing the duplication of effort when processing similar fragments. Another strong argument in favour of extended memory is its ability to accurately track logical dependencies within code. R. He *et al.* (2024) highlighted that such an approach is particularly important in the development of complex software solutions, where each module depends on data or logic from preceding components. Similarly, G. Marvin *et al*. (2024) noted that effective prompt engineering is directly influenced by the model's capacity to retain context. The greater the relevance and accuracy of recalled information, the more precise and coherent the responses.

However, I. Ozkaya (2023) warned that if a model's memory becomes cluttered with unnecessary or irrelevant information, accuracy may decline. Therefore, implementing a mechanism for filtering or selectively "forgetting" redundant data is essential. Many contemporary approaches incorporate memory-cleaning algorithms or recursive refinement techniques to ensure that only the most relevant details remain in focus. Finally, G. Dolcetti *et al.* (2024) emphasised the importance of feedback from testing and static analysis. They argued that if a model retains the outcomes of previous attempts and learns from them, it can iteratively improve code generation accuracy and accelerate error correction. This further underscores the critical role of well-structured memory mechanisms in enabling LLMs to operate effectively in complex scenarios.

Software development automation: Results of innovative approaches. The second phase of the study focused on analysing the impact of advanced memory systems on the ability of LLMs to automate software development. A series of experiments was conducted in which the models generated code in Python and C++, automatically created test scripts, and refactored existing modules in accordance with best practices.

Initially, each model was provided with a technical specification detailing both functional and non-functional requirements. The task was to generate fully functional code based on these specifications. To evaluate the results, two key criteria were considered. The time (in seconds) required to generate a working prototype. The percentage of generated code that met all requirements without further corrections To facilitate comparison, a graph was constructed in which the bars represent the proportion of correctly generated code, while the line indicates the generation time (Fig. 2).

In the field of software engineering, advanced LLM memory enables a significant increase in automation across all stages of development – from the initial formulation of requirements to the refactoring of large code bases. To effectively interact with code or technical documentation, a model must be capable of "understanding" and maintaining focus on the context of multiple diverse files, sometimes even across different programming languages.

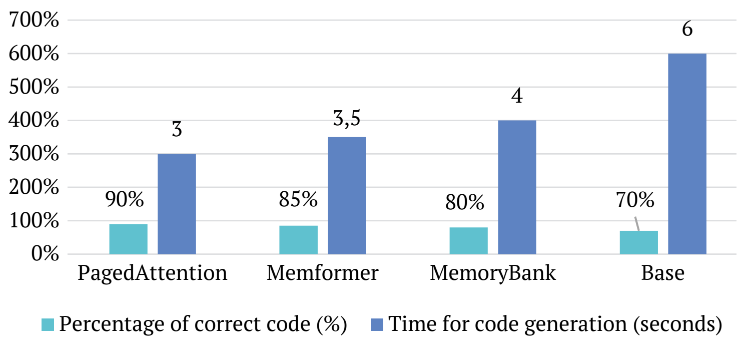

Figure 2. *Comparing the efficiency of code generation with different memory systems*
Source: *created by the author*

According to the findings of M.K. Görmez *et al.* (2024), LLMs equipped with advanced memory systems can rapidly generate prototypes while simultaneously producing test scenarios that cover the most critical aspects of functionality. This capability allows for the early detection and resolution of vulnerabilities during development. Additionally, R. He *et al*. (2024) observed that memory support enhances the model's ability to resolve conflicts between different project components, particularly in cases where modifications to one module may introduce issues in another. If an LLM retains an understanding of the logic and structure of all modules, it becomes significantly more effective in identifying compromise solutions. At the same time, G. Dolcetti *et al.* (2024) emphasised the effectiveness of feedback mechanisms: when a detected error is recorded in the model's memory, it is accounted for in subsequent code generation attempts, thereby substantially reducing the recurrence of similar shortcomings. Another crucial aspect is the ability of LLMs to generate unit tests and integration tests. As part of the experiment, the models were tasked with independently developing test scripts to verify the compliance of the source code with its specifications. In addition to evaluating the speed of test generation, the experiment measured the percentage of detected logical and syntactic errors that had been intentionally introduced into the source code. The results showed that PagedAttention successfully generated adequate tests for 94% of functions, detecting 87% of errors. Memformer achieved comparable results, with 90% of tests deemed adequate and 85% of defects detected. Meanwhile, MemoryBank demonstrated slightly lower performance, with 87% of adequate tests and an 82% error detection rate. The baseline LLM configuration, in contrast, identified only 75% of artificial errors and required significantly more time to generate test scripts. Beyond test generation, the models also analysed existing projects for code duplication, complex logical checks, and anti-patterns. Table 2 presents a summary of the refactoring results.

Table 2. *Comparing code refactoring results*

LLM configuration	**Duplication reduction (%)**	**Complexity optimisation (number of reduced conditions)**	**Recommended patterns (pcs.)**
Basic	15	8	2
MemoryBank	30	13	5
Memformer	35	15	6
PagedAttention	40	18	7

Source: *created by the author*

PagedAttention proved to be the most effective in reducing code duplication (by 40%) and in suggesting design patterns, while Memformer and MemoryBank also significantly outperformed the baseline model. This result can be attributed to the extended memory capabilities, which allow the model to retain a larger portion of the code in focus, quickly analyse its structure, and propose improvements without losing context. Refactoring often requires consideration of global dependencies between modules, and in this regard, PagedAttention demonstrated the greatest flexibility, as it dynamically loaded only the relevant "pages" of data.

Contextual extension also plays a crucial role. A.N.T. Dieu *et al.* (2024) demonstrated how a model can dynamically access the history of interactions, including past testing and code reviews. This capability is particularly useful for quickly adapting to changes in requirements or resolving logical inconsistencies. Similar approaches have been proposed by K. Wu (2024), and J. Park & H. Sung (2023), who focused on runtime optimisation. Their research highlights that during GPU-based code generation, the model can load only the relevant portion of the context, thereby improving processing speed and reducing memory overhead. Meanwhile, Y. Bajaj & M.K. Samal (2023) suggested using LLMs for semi-automatic test generation, where the model, by retaining an understanding of the code structure, can immediately identify common logical pitfalls.

This approach is particularly relevant for Continuous Integration/Continuous Delivery (CI/CD) pipelines. According to X. Hou *et al*. (2024), an LLM integrated into CI/CD workflows can validate key functional components with each new commit, referencing the history of successful and failed results to enhance software reliability. In turn, S. Jalil (2023) highlighted that such automated scenarios are becoming increasingly important in complex projects characterised by a high frequency of changes. The integration of memory enables LLMs to retain key contextual information and adapt more rapidly to evolving conditions. This approach, as emphasised by N. van Viet & N. Vinh (2024), forms the foundation for intelligent systems capable of independently determining development or testing priorities.

Finally, R. Hoda *et al*. (2023) introduced the concept of "Augmented Agile", in which an LLM with an advanced memory mechanism effectively becomes part of the development team, assisting in operational decision-making related to task distribution and code structure. This marks a significant shift in software development, reducing the burden on engineers by automating routine error detection and resolution. Overall, the findings from the second phase of the study confirmed that optimised memory systems substantially enhance the capabilities of LLMs in automating software development – from code generation and testing to comprehensive refactoring. This opens up the prospect of integrating such models into Continuous Integration (CI) and Continuous Delivery (CD) pipelines, where they could autonomously handle a significant portion of routine tasks, thereby accelerating release cycles and reducing the risk of errors.

Comparative analysis and prospects for use. Based on the data obtained, several general conclusions can be drawn regarding the effectiveness of each of the examined approaches and their potential applications in real-world conditions. First, all three memory systems – MemoryBank, Memformer, and PagedAttention – demonstrated superior performance compared to the baseline LLM. According to the metrics presented, these systems achieve 30-40% faster data processing and 6-9% higher accuracy, which is particularly critical for scenarios requiring stable handling of large contexts or long-term interaction with users.

Among these systems, PagedAttention exhibited the best balance between speed and accuracy. Due to its ability to dynamically "load" context pages, it achieved the lowest latency while maintaining high accuracy (91%). It also outperformed other approaches in refactoring tasks, as it efficiently analysed complex project structures and suggested design patterns. Memformer proved particularly effective in complex, multi-layered scenarios, where projects contain multiple levels of logical dependencies and large datasets. Its hierarchical model scales effectively while maintaining high accuracy (90%), though its implementation is more complex and may be excessive for simpler applications. Conversely, MemoryBank is most suitable for medium-sized or segmented projects. While it lags slightly behind the other two approaches in overall performance, it stands out for its ease of implementation and flexibility in managing individual data blocks. This makes it an ideal choice for small teams or projects requiring rapid deployment without unnecessary complexity.

Comparing MemoryBank, Memformer, and PagedAttention with the baseline LLM, all three memory mechanisms demonstrate significant improvements in performance and accuracy, particularly when handling large contexts or highly interconnected tasks. However, each approach has distinct advantages, making it more suitable for specific scenarios. PagedAttention dynamically "loads" only the necessary pages of information, optimising efficiency and reducing processing overhead. Memformer scales effectively in complex scenarios involving multiple logical dependencies. MemoryBank is easier to implement and well-suited for small to medium-sized projects that require flexible memory management.

It is worth noting that X. Hou *et al.* (2024) highlighted the versatility of long-term memory mechanisms, noting that they are beneficial not only for code generation but also for adaptive documentation analysis and project history management. According to S. Jalil (2023), LLMs that account for previous changes and specifications serve as integrators between different development team departments, facilitating better cross-team collaboration. Meanwhile, N. van Viet & N. Vinh (2024) observed that the scaling and advancement of memory architectures has paved the way for self-optimising systems capable of monitoring design changes and autonomously suggesting improvements. F. Chalumeau *et al.* (2024) further described how innovative memory methods are particularly valuable for addressing combinatorial explosion problems, where the model must track and manage multiple intermediate states simultaneously.

Further efficiency gains are also linked to advancements in hardware solutions. L.A.S. Julien *et al.* (2023) highlighted the emergence of enhanced chips and memory devices, which have simultaneously reduced power consumption and accelerated data processing. J. Cheng *et al*. (2021) emphasised the importance of efficient memory allocation in multi-threaded code synthesis environments, where multiple program segments may need to access shared memory areas within the model. According to M. Alenezi & M. Akour (2025), these advancements offer not only technical but also economic benefits, enabling companies to release updates more quickly and reduce long-term maintenance costs. It is also worth noting how memory mechanisms facilitate applications beyond text-based tasks. M. Zhu *et al.* (2019) demonstrated that in generative image modelling (DM-GAN), dynamic memory improves the alignment between textual descriptions and visual outputs. By analogy, in software engineering, similar concepts enable the integration of diverse data types, including code fragments, UML diagrams, and technical documentation, thereby enhancing project coherence and facilitating multi-modal processing.

Additionally, J. Cruz-Benito *et al.* (2019) observed that extended memory mechanisms significantly enhance the accuracy of code autocompletion. When a model can reference the global structure of a project, the number of logical errors and incorrect recommendations is substantially reduced. Concluding the review, Z. Lyu *et al*. (2025) introduced the Top Pass technique, which combines ranking of the best results with a memory mechanism. This approach enables the model to rapidly filter potential answers, discarding the least relevant options and refining its outputs more effectively.

Thus, the prospects for integrating innovative memory systems into LLMs are extensive, ranging from enhancing productivity and accuracy to improving integration within development workflows. By retaining and adaptively managing context, these models become more flexible and versatile, enabling them to handle increasingly complex tasks in dynamic and continuously evolving environments. The findings of this study confirm that innovative memory systems significantly enhance both the performance and flexibility of large language models. With improved long-term context management, LLMs are evolving into powerful tools for automating software development, capable of independently generating code, testing it, and suggesting optimisations. Among the tested approaches, PagedAttention delivered the best overall performance, though the choice of a specific memory mechanism depends on factors such as project scale, complexity, speed requirements, memory constraints, and ease of implementation. Looking ahead, these technologies are poised to become industry standards, fostering more effective collaboration between humans and AI and laying the groundwork for further innovations in software engineering.

Conclusions

The findings of this study confirmed the high efficiency of innovative memory management approaches in LLMs, particularly MemoryBank, Memformer, and PagedAttention. Each of these mechanisms significantly enhanced model performance and accuracy, which is especially critical for tasks requiring long-term context retention and handling a large number of interconnected actions. The implementation of MemoryBank improved long-term context processing through its efficient hierarchical memory structure, leading to reduced latency and greater accuracy in complex tasks. Memformer demonstrated high effectiveness in scenarios involving complex dependencies between tokens, enabling deeper context analysis and more precise generation. PagedAttention, in turn, outperformed other approaches in key metrics, achieving the highest data processing speed and optimal memory usage by dynamically loading only relevant data.

In the domain of software development automation, the study highlighted the potential of LLMs in executing a broad spectrum of tasks, from code generation and test script creation to refactoring existing software modules. The integration of advanced memory mechanisms significantly reduced development time while improving code quality, minimising both logical and syntactic errors. The use of PagedAttention in CI/CD processes proved particularly promising, as its dynamic context management optimised the code testing and verification pipeline.

Overall, the study confirmed that enhancing memory systems not only improves the performance and accuracy of large language models but also expands their potential applications. Further research in this field could explore the integration of LLMs into multi-component systems that support decision-making across various sectors, from medicine to financial analytics. In particular, advancements in technologies similar to PagedAttention could lay the foundation for new solutions prioritising scalability, adaptability, and efficiency. Innovative memory management approaches have the potential to become the de facto industry standard, enabling fast, precise, and efficient problem-solving across a wide range of complexities. These developments could fundamentally reshape the way data is processed and software is developed, unlocking new opportunities for automation, personalisation, and seamless integration into existing workflows.

Acknowledgements

None.

Conflict of Interest

None.

References

[1] Alenezi, M., & Akour, M. (2025). AI-Driven innovations in software engineering: A review of current practices and future directions. *Applied Sciences*, 15(3), article number 1344. doi: 10.3390/app15031344.

[2] Antero, U., Blanco, F., Onativia, J., & Salle, D. (2024). Harnessing the power of large language models for automated code generation and verification. *Robotics*, 13(9), article number 137. doi: 10.3390/robotics13090137.

[3] Bajaj, Y., & Samal, M.K. (2023). Accelerating software quality: Unleashing the power of generative AI for automated test-case generation and bug identification. *International Journal for Research in Applied Science and Engineering Technology,* 11(7), 345-350. doi: 10.22214/ijraset.2023.54628.

[4] Chalumeau, F., Shabe, R., de Nicola, N., Pretorius, A., Barrett, T.D., & Grinsztajn, N. (2024). *Memory-enhanced neural solvers for efficient adaptation in combinatorial optimization*. doi: 10.48550/arXiv.2406.16424.

[5] Cheng, J., Fleming, S., Chen, Y.T., Anderson, J., Wickerson, J., & Constantinides, G.A. (2021). Efficient memory arbitration in high-level synthesis from multi-threaded code. *IEEE Transactions on Computers,* 71(4), 933-946. doi: 10.1109/tc.2021.3066466.

[6] Cruz-Benito, J., Sánchez-Prieto, J.C., Therón, R., & García-Peñalvo, F.J. (2019). Measuring students' acceptance to AI-Driven assessment in eLearning: Proposing a first TAM-Based research model. In P. Zaphiris & A. Ioannou (Eds.), *6th international conference: Learning and collaboration technologies. Designing learning experiences* (pp. 15-25). Cham: Springer. doi: 10.1007/978-3-030-21814-0_2.

[7] Dieu, A.N.T., Nguyen, H.T., & Cong, C.T.D. (2024). The enhanced context for AI-generated learning advisors with Advanced RAG. In L.-S. Lê, M. Kappes & J. Küng (Eds.*), 18th international conference on advanced computing and analytics* (pp. 94-101). Ben Cat: Institute of Electrical and Electronics Engineers. doi: 10.1109/ACOMPA64883.2024.00021.

[8] Dolcetti, G., Arceri, V., Iotti, E., Maffeis, S., Cortesi, A., & Zaffanella, E. (2024). *Helping LLMs improve code generation using feedback from testing and static analysis*. doi: 10.48550/arxiv.2412.14841.

[9] Görmez, M.K., Yılmaz, M., & Clarke, P.M. (2024). Large language models for software engineering: A systematic mapping study. In M. Yilmaz, P. Clarke, A. Riel, R. Messnarz, C. Greiner & T. Peisl (Eds.), *31st European conference: Systems, software and services process improvement* (pp. 64-79). Cham: Springer. doi: 10.1007/978-3-031-71139-8_5.

[10] He, R., Ying, A., & Hu, X. (2024). Improving OpenDevin: Boosting code generation LLM through advanced memory management. *Applied and Computational Engineering*, 68(1), 320-327. doi: 10.54254/2755-2721/68/20241506.

[11] Hoda, R., Dam, H., Tantithamthavorn, C., Thongtanunam, P., & Storey, M. (2023). Augmented agile: Human-centered AI-assisted software management. *IEEE Software*, 40(4), 106-109. doi: 10.1109/MS.2023.3268725.

[12] Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., & Wang, H. (2024). Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology,* 33(8), article number 220. doi: 10.1145/3695988.

[13] Jalil, S. (2023). *The transformative influence of large language models on software development*. doi: 10.48550/arXiv.2311.16429.

[14] Jiang, X., Dong, Y., Wang, L., Fang, Z., Shang, Q., Li, G., Jin, Z., & Jiao, W. (2024). Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7), article number 182. doi: 10.1145/3672456.

[15] Julien, L.A.S., Tang, X., & Gaillardon, P. (2023). Innovative memory architectures using functionality enhanced devices. In M.M.S. Aly & A. Chattopadhyay (Eds.), *Emerging computing: From devices to systems: Looking beyond moore and von neumann* (pp. 47-83). Singapore: Springer. doi: 10.1007/978-981-16-7487-7_.

[16] Kim, B., *et al.* (2024). The breakthrough memory solutions for improved performance on LLM inference. *IEEE Micro*, 44(3), 40-48. doi: 10.1109/MM.2024.3375352.

[17] Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L. Yu, C.H., Gonzales, J., Zhang, H., & Stoica, I. (2023). Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles* (pp. 611-626). New York: Association for Computing Machinery. doi: 10.1145/3600006.3613165.

[18] Liu, B., Jiang, Y., Zhang, Y., Niu, N., Li, G., & Liu, H. (2024). *An empirical study on the potential of LLMs in automated software refactoring*. doi: 10.48550/arXiv.2411.04444.

[19] Lyu, Z., Li, X., Xie, Z., & Li, M. (2025). Top Pass: Improve code generation by pass@k-maximized code ranking. *Frontiers of Computer Science*, 19(8), article number 198341. doi: 10.1007/s11704-024-40415-9.

[20] Marvin, G., Hellen, N., Jjingo, D., & Nakatumba-Nabende, J. (2024). Prompt engineering in large language models. In I.J. Jacob, S. Piramuthu & P. Falkowski-Gilski (Eds.), *Proceedings of ICDICI 2023: Data intelligence and cognitive informatics* (pp. 387-402). Singapore: Springer. doi: 10.1007/978-981-99-7962-2_30.

[21] Nguyen, D., Yang, W., Anand, R., Yang, Y., & Mirzasoleiman, B. (2024). *Mini-batch coresets for memory-efficient training of large language models*. doi: 10.48550/arXiv.2407.19580.

[22] Ozkaya, I. (2023). Application of large language models to software engineering tasks: Opportunities, risks, and implications. *IEEE Software,* 40(3), 4-8. doi: 10.1109/MS.2023.3248401.

[23] Park, J., & Sung, H. (2023). XLA-NDP: Efficient scheduling and code generation for Deep Learning model training on Near-Data Processing Memory. *IEEE Computer Architecture Letters*, 22(1), 61-64. doi: 10.1109/LCA.2023.3261136.

[24] Rostam, Z.R.K., Szénási, S., & Kertész, G. (2024). Achieving peak performance for large language models: A systematic review. *IEEE Access,* 12, 96017-96050. doi: 10.1109/access.2024.3424945.

[25] Sagi, S. (2024). Advancing AI: Enhancing large language model performance through GPU optimization techniques. *International Journal of Science and Research*, 13(3), 630-633. doi: 10.21275/sr24309100709.

[26] Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2023). An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1), 85-105. doi: 10.1109/tse.2023.3334955.

[27] Shantyr, A. (2024). Specifics of quality assessment models application at development and use stages of software systems. *Information Technologies and Computer Engineering*, 21(1), 127-138. doi: 10.31649/1999-9941-2024-59-1-127-138.

[28] van Viet, N., & Vinh, N. (2024). Large language models in software engineering. *Journal of Education for Sustainable Innovation,* 2(2), 146-156. doi: 10.56916/jesi.v2i2.968.

[29] Wu, K. (2024). *Code generation and runtime techniques for enabling data efficient deep learning training on GPUs.* Urbana: University of Illinois Urbana-Champaign. doi: 10.13140/RG.2.2.15485.47840.

[30] Wu, Q., Lan, Z., Gu, J., Geramifard, A., & Yu, Z. (2020). *Memformer: The memory-augmented transformer*. doi: 10.48550/arXiv.2010.06891.

[31] Xu, F.F., Alon, U., Neubig, G., & Hellendoorn, V.J. (2022). A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM sigplan international symposium on machine programming* (pp. 1-10). New York: Association for Computing Machinery. doi: 10.1145/3520312.3534862.

[32] Zheng, Z., Ning, K., Zhong, Q., Chen, J., Chen, W., Guo, L., Wang, W., & Wang, Y. (2025). Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering*, 30, article number 50. doi: 10.1007/s10664-024-10602-0.

[33] Zhong, W., Guo, L., Gao, Q., Ye, H., & Wang, Y. (2024). MemoryBank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI conference on artificial intelligence* (pp. 19724-19731). Washington: Association for the Advancement of Artificial Intelligence doi: 10.1609/aaai.v38i17.29946.

[34] Zhu, M., Pan, P., Chen, W., & Yang, Y. (2019). DM-GAN: Dynamic memory generative adversarial networks for text-to-image synthesis. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (pp. 5802-5810). Long Beach: Institute of Electrical and Electronics Engineers.

Оптимізація продуктивності та автоматизація розробки програмного забезпечення: інноваційні підходи до системи пам'яті у великих мовних моделях

Олена Сокол

Магістр

Київський національний університет імені Тараса Шевченка

01601, вул. Володимирська, 60, м. Київ, Україна

https://orcid.org/0009-0005-5160-460X

Анотація. Це дослідження розглядало інноваційні підходи до вдосконалення систем пам'яті у великих мовних моделях для підвищення ефективності та автоматизації розробки програмного забезпечення. Основна увага приділялася оптимізації систем пам'яті, які забезпечують довготривале зберігання контексту та полегшують адаптацію моделі до мінливих умов взаємодії. У дослідженні проаналізовано сучасні методи зберігання та обробки даних, які підвищують здатність моделей ефективно обробляти великі обсяги інформації. Це включало використання спеціалізованих алгоритмів і механізмів пам'яті, які підвищують точність і адаптивність великих мовних моделей при виконанні складних завдань. Другим напрямком дослідження було вивчення можливостей великих мовних моделей в автоматизації розробки програмного забезпечення. Оцінювалося, як ці моделі можуть генерувати код, оптимізувати його та виявляти помилки. Особливу увагу приділено аналізу впливу автоматизації на якість програмного забезпечення та час розробки. У цьому контексті дослідження вивчало використання моделей великих мов для автоматизації повторюваних завдань, генерації тестів та впровадження найкращих практик програмування. Отримані результати свідчили про те, що покращення систем пам'яті великих мовних моделей значно підвищує їхню ефективність у задачах, що потребують довготривалої взаємодії. Показано, що інтеграція таких моделей у процеси розробки програмного забезпечення дозволяє скоротити витрати часу та ресурсів при одночасному підвищенні якості продукту. Практична значущість цього дослідження полягає у формулюванні рекомендацій щодо оптимального використання моделей великих мов у сфері інформаційних технологій

Ключові слова: контекстна обробка; оптимізація алгоритмів; генерація коду; інтелектуальні агенти; обробка великих даних; автоматизація; моделювання взаємодії