engineering proceedings

MDPI

Proceeding Paper

Integration of Large Language Models in Layered Software Systems: A Clean Architecture and CQRS Case Study †

Antonina Ivanova [1,*], Georgi Kolev [1], Fatima Sapundzhi [1,2], Teodora Bakardjieva [1] and Slavi Georgiev [3,4]

[1] Department of Computer Science, Varna Free University "Chernorizets Hrabar", 9007 Varna, Bulgaria; 213010030@vfu.bg (G.K.); fatima.sapundzhi@vfu.bg or sapundzhi@swu.bg (F.S.); bakardjieva@vfu.bg (T.B.)
[2] Department of Communication and Computer Engineering, Faculty of Engineering, South-West University "Neofit Rilski", 2700 Blagoevgrad, Bulgaria
[3] Department of Applied Mathematics and Statistics, University of Ruse, 7004 Ruse, Bulgaria; sggeorgiev@uni-ruse.bg or sggeorgiev@math.bas.bg
[4] Department of Information Modeling, Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, 1113 Sofia, Bulgaria
* Correspondence: antonina.ivanova@vfu.bg
† Presented at the International Conference on Electronics, Engineering Physics and Earth Science (EEPES2026), Bandirma, Turkey, 24–27 June 2026.

Abstract

Large Language Models (LLMs) are increasingly incorporated into software systems. Their non-deterministic behavior, external hosting, response latency, and operational cost create challenges for established design approaches such as Clean Architecture. This paper examines the integration of an LLM component into a layered software system and compares three possible placements within Clean Architecture: Domain, Application, and Infrastructure. The evaluation considers dependency management, testability, separation of concerns, and implementation complexity. The study proposes an approach in which the LLM is implemented in the Infrastructure layer and accessed through an interface defined in the Application layer. This approach is combined with the Command and Query Responsibility Segregation pattern to isolate LLM interaction within dedicated query handlers. The proposed pattern is demonstrated through the implementation of Budget, a personal finance tracking system that uses GPT-4.1 to convert free-form natural language input into structured transaction records. The results show that placement in the infrastructure layer provides the clearest separation between business logic and external AI services and avoids the introduction of non-deterministic behavior into the core application logic.

Keywords: large language models; clean architecture; software architecture; CQRS; layered architecture; non-determinism; LLM integration; software design patterns; prompt engineering

Check for updates

Academic Editors: Teodor Iliev, Ivaylo Stoyanov, Grigor Mihaylov and Selahattin Kosunalp

Published: 2 September 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

The rapid development of Large Language Models (LLMs) has expanded the range of software systems that can process natural language input [1]. LLM-based functionality is increasingly included in web applications and enterprise systems, where natural language interaction can simplify user workflows [2]. Recent advances in GPT-4 and related models have accelerated the adoption of LLM capabilities in mainstream software products, particularly in systems that require flexible natural language processing [3]. Similar AI-driven capabilities for enhanced situational awareness have already been demonstrated in specialized remote monitoring prototypes [4].

The inclusion of LLMs in software systems creates architectural requirements that differ from those associated with conventional external services [5]. Traditional REST APIs provide deterministic behavior and stable performance characteristics. LLM components rely on remote inference services, produce non-deterministic outputs, and require remote calls with measurable latency and operational cost. These characteristics affect system reliability and influence the organization of architectural layers. Recent research shows that the combination of deterministic rule-based agents with LLM-driven components can improve the handling of remote non-deterministic services in production environments [6]. A taxonomy of LLM-integrated applications also identifies several integration patterns and architectural variations that need to be considered during system design [1].

Clean Architecture and the CQRS pattern provide a foundation for building maintainable and testable systems. Clean Architecture defines dependency boundaries and isolates business logic from external services. CQRS separates write and read operations, which allows more flexible organization of system behavior [7]. The placement of LLM components within layered architectures has received less attention, especially in relation to dependency management and separation of concerns [8]. Recent studies also identify technical issues in LLM-driven systems, including integration complexity, testability, and scalability [5].

This paper examines the integration of an LLM component into a system designed according to Clean Architecture principles [9]. Three possible placements of the LLM component are evaluated: Domain layer, Application layer, and Infrastructure layer. The study proposes an integration pattern in which the LLM is implemented in the Infrastructure layer and accessed through an interface defined in the Application layer. The proposed approach is demonstrated through the implementation of Budget, a personal finance tracking system that uses GPT-4.1 to convert free-form natural language input into structured transaction records.

The paper contributes an architectural model for integrating LLM components into layered software systems. It also identifies four engineering concerns associated with this form of integration: testability under non-deterministic behavior, response latency, output validation, and operational cost. The results show that Clean Architecture can support LLM integration when remote inference, non-deterministic output, and validation requirements are addressed explicitly in the system design.

2. Materials and Methods

2.1. Architectural Foundations

Clean Architecture, introduced by Robert C. Martin, organizes software systems into layers with clearly defined responsibilities. The main layers are Domain, Application, Infrastructure, and Presentation. This architectural layering is consistent with the integrated computational framework for complex distributed environments proposed for digital business ecosystems [10]. The central principle is the Dependency Rule, according to which source code dependencies must always point inward. Inner layers remain independent from databases, frameworks, and external services. Software components, including language models, must be placed within this layered structure according to their integration characteristics [1].

The CQRS pattern separates write operations from read operations. This separation allows business logic to be organized more clearly and supports separate optimization of command and query processing. In .NET systems, CQRS is commonly implemented with the MediatR library, which dispatches commands and queries to dedicated handlers. The library also implements the Mediator pattern and supports clearer separation between request handling and business logic [11].

https://doi.org/10.3390/engproc2026154023

Large Language Models differ from conventional software dependencies because they operate through remote inference services and produce non-deterministic outputs. Their response time is significantly higher than that of in-process components, and each invocation introduces a direct operational cost. These characteristics make the placement of LLM functionality within a layered architecture an important design decision. Recent studies show that LLM-generated microservice components require careful integration planning in order to preserve architectural consistency [8].

2.2. Research Approach

The study applies a comparative architectural analysis of three possible placements of an LLM component within a Clean Architecture system: Domain layer, Application layer, and Infrastructure layer. Each placement is evaluated against four criteria: compliance with the Dependency Rule, testability, separation of concerns, and operational simplicity. These criteria were selected because they reflect the software quality attributes most directly affected by the introduction of a remote non-deterministic component into a layered system. Recent studies of LLM-driven software systems use similar criteria when examining architectural and integration challenges [5].

The evaluation is performed through a case study based on Budget, a personal finance tracking system that allows users to enter financial transactions in free-form natural language. The system uses GPT-4.1 to transform user input into structured transaction records. This case study approach follows recent work on LLM integration in specialized software systems.

2.3. Case Study System

The Budget system is implemented with ASP.NET Core 8, React, PostgreSQL, MediatR, and Microsoft Semantic Kernel [12]. The backend follows the Clean Architecture pattern and is divided into four projects corresponding to the Domain, Application, Infrastructure, and Presentation layers (Table 1).

Table 1. Main technologies used in the Budget system.

Layer	Technology	Version/Official Website
Frontend	React, TailwindCSS	https://react.dev/ (accessed on 17 July 2025); https://tailwindcss.com/ (accessed on 17 July 2025)
Backend	ASP.NET Core 8	Version 8
Data Access	Entity Framework Core	https://learn.microsoft.com/en-us/ef/core/ (accessed on 22 July 2025)
Database	PostgreSQL	https://www.postgresql.org/ (accessed on 22 July 2025)
CQRS	MediatR	https://github.com/LuckyPennySoftware/MediatR (accessed on 28 July 2025)
LLM Integration	Microsoft Semantic Kernel, GPT-4.1, Azure OpenAI	https://learn.microsoft.com/en-us/semantic-kernel/ (accessed on 9 March 2026); https://openai.com/index/gpt-4-1/ (accessed on 10 March 2026); https://learn.microsoft.com/en-us/azure/ai-services/openai/ (accessed on 25 July 2025)

The Domain layer contains entities and repository interfaces. The Application layer contains CQRS handlers and the IQuickAddService interface used for LLM interaction. Quick Add functionality is intended for rapid transaction entry without manual form completion. The Infrastructure layer contains the Entity Framework Core database context, repository implementations, and the concrete QuickAddService implementation that com-

municates with GPT-4.1 through Azure OpenAI Services. The Presentation layer exposes REST endpoints used by the React frontend. The strict separation between these projects is enforced through project reference constraints, which prevent dependencies from violating the Dependency Rule.

2.4. LLM Integration Method

The LLM integration is implemented through the IQuickAddService interface declared in the Application layer and implemented in the Infrastructure layer. The GPT-4.1 model instance is registered as a singleton in order to avoid repeated initialization across HTTP requests. A dedicated CQRS query handler invokes this service and transforms the generated CSV response into structured QuickAddResult objects. This implementation follows CQRS patterns commonly used in .NET applications [7].

The prompt template contains instructions for output format, date handling, and category selection. It also receives the current date, recent transaction history, and the list of available categories as contextual input. The generated response is validated before it is returned to the Application layer. Invalid records are discarded, and the frontend requires explicit user confirmation before the generated transactions are stored.

The template also defines date interpretation rules, category constraints, and a small set of few-shot examples. The model is instructed to return only CSV-formatted lines without additional commentary. This simplifies parsing and reduces the probability of malformed output. Structured prompts and CQRS-based processing support the transformation of free-form user input into validated domain entities. The LLM call is executed asynchronously within the CQRS query handler, which allows the ASP.NET Core request thread to remain available during model invocation. The following excerpt shows the structure of the prompt used in the Quick Add functionality (Figure 1).

```
You are a financial assistant that extracts structured finance transactions from user input.

Return only CSV lines in the following format:
amount,record_type,category_name,account_name,from_account_name,record_date_time
Use the closest matching category from the provided category list.

Interpret expressions such as “today” and “yesterday” relative to the supplied current date.
Do not include explanations or additional text.
```

Figure 1. Excerpt from the QuickAddPrompt.skprompt.txt template used for transaction extraction.

The template also includes several few-shot examples that define the expected relationship between user input and generated output. These examples guide the model toward the required response structure without additional fine-tuning of the base model. The prompt is supplemented with dynamic contextual arguments, including recent transaction history, available categories, and the current date (Figure 2).

```
var args = new KernelArguments {
     ["input"] = inputText,
     ["history"] = historyCsv,
     ["categories"] = categoriesCsv,
     ["today"] = DateTime.UtcNow.ToString("yyyy-MM-dd")
 };
```

Figure 2. Dynamic prompt arguments supplied to the LLM service.

https://doi.org/10.3390/engproc2026154023

If the model response cannot be parsed successfully, the system returns an empty result and prompts the user to reformulate the request.

2.5. Computational Evaluation of the Quick Add LLM Component

To support the architectural claim that LLM integration introduces non-deterministic output, latency, validation, and cost-related concerns, an additional computational experiment was conducted for the Quick Add functionality. The experiment evaluated the LLM component as an external Infrastructure-layer service accessed through the IQuickAddService interface. The purpose was not to measure general language understanding ability, but to evaluate whether the generated response satisfies the strict software contract required by the Budget system.

A benchmark set of 60 free-form transaction prompts was prepared. The prompts covered six input classes: simple expense records, income records, transfers between accounts, multiple transactions in a single sentence, relative date expressions such as "today" and "yesterday", and ambiguous or partially incomplete user inputs. Each prompt was executed five times under the same prompt template, model deployment, category list, account list, and current-date parameter. Thus, the experiment consisted of $N = 60 \times 5 = 300$ LLM invocations.

For each invocation, the raw model response, prompt version, timestamp, response time, input token count, output token count, parsing result, validation result, and final accepted/rejected status were recorded. The raw responses were also stored as snapshot outputs in order to measure output stability across repeated executions of the same prompt. The generated response was evaluated at three levels. First, a CSV syntax test checked whether the output contained only CSV lines and whether each line contained the required six fields: amount, record_type, category_name, account_name, from_account_name, record_date_time. Second, a contract test checked whether the generated fields satisfied the application constraints: valid numeric amount, valid transaction type, existing category name, existing account name, and valid date. Third, a semantic correctness test compared the generated structured record with a manually prepared reference output. The following metrics were used:

$$\text{Valid CSV rate} = \frac{N_{\text{csv}}}{N}, \tag{1}$$

$$\text{Contract pass rate} = \frac{N_{\text{contract}}}{N}, \tag{2}$$

$$\text{Semantic accuracy} = \frac{N_{\text{correct}}}{N}, \tag{3}$$

$$\text{Invalid response rate} = 1 - \frac{N_{\text{contract}}}{N}. \tag{4}$$

To quantify non-determinism, two agreement metrics were computed across the five repeated runs for each prompt. The raw snapshot agreement rate measures whether all five raw responses were identical. The parsed agreement rate measures whether all five responses produced the same validated structured transaction, even if the raw CSV formatting differed.

The operational cost per request was estimated as

$$C_j = \frac{p_{\text{in}} T_{\text{in},j} + p_{\text{out}} T_{\text{out},j}}{10^6}, \tag{5}$$

where $T_{\text{in},j}$ and $T_{\text{out},j}$ are the input and output token counts for request j, while p_{in} and p_{out} denote the provider-specific prices per one million input and output tokens. This formulation allows the cost calculation to be updated when the provider's pricing changes.

3. Results

3.1. Evaluation of Candidate Placements

The first candidate places the LLM component in the Domain layer. This approach gives the core business logic direct access to the model. Such placement violates the Dependency Rule because the Domain layer becomes dependent on an external service. Testability is also reduced because the business logic becomes coupled to a non-deterministic component whose output may differ across identical invocations. For this reason, the Domain layer is not a suitable location for LLM functionality.

The second candidate places the LLM component in the Application layer. This option preserves the separation between the Domain layer and external services. The Application layer, however, must then handle network communication, prompt processing, response latency, and API cost control. These responsibilities do not belong to business orchestration. As a result, the Application layer becomes more difficult to maintain and its role becomes less clearly defined.

The third candidate places the LLM component in the Infrastructure layer. In this approach, the Application layer depends only on an interface, while the Infrastructure layer contains the implementation that communicates with the external model. This placement preserves the Dependency Rule and keeps the non-deterministic component outside the core business logic. It also supports testing through mock service implementations. Among the three alternatives, the Infrastructure layer provides the clearest separation between business logic and external AI services.

Figure 3 shows the position of the LLM component within the Clean Architecture structure. The component is located in the Infrastructure layer and is accessed through an interface defined in the Application layer. The dependency direction remains inward and does not violate the Dependency Rule. Recent studies also support the use of explicit evaluation criteria when selecting architectural variants for LLM-enabled systems [13].

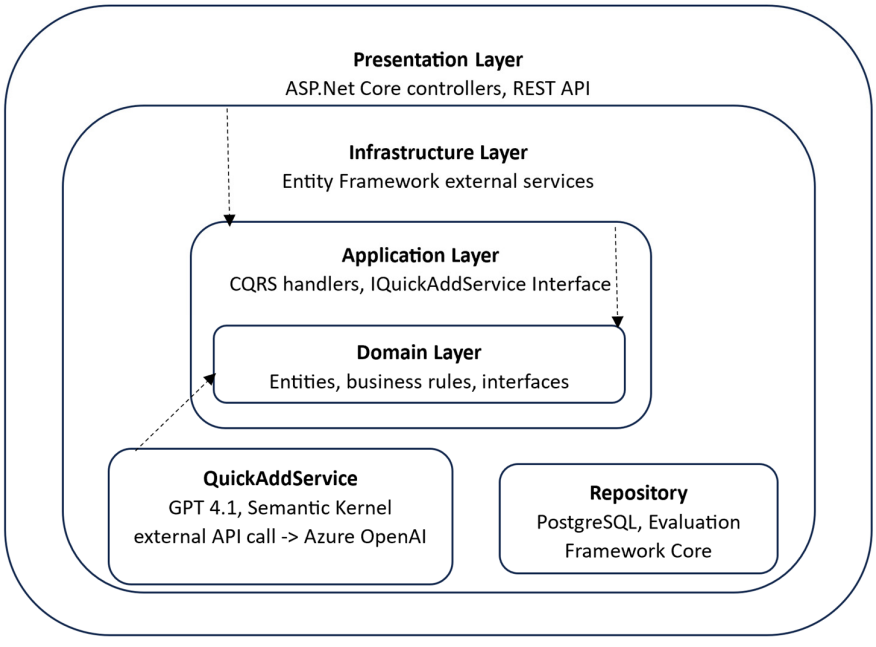


Figure 3. Placement of the LLM component within the Clean Architecture layer structure.

Table 2 summarizes the evaluation of the three candidate placements against the main architectural criteria. Contemporary systems increasingly employ multiple language models requiring sophisticated orchestration and trust management mechanisms.

Table 2. Evaluation of candidate LLM placements in Clean Architecture.

Candidate	Dependency Rule	Testability	Separation of Concerns	Verdict
Domain Layer	Violated	Poor	Violated	Rejected
Application Layer	Preserved	Moderate	Violated	Rejected
Infrastructure Layer	Preserved	High	Preserved	Accepted

3.2. Comparison with Alternative Integration Strategies

The proposed Infrastructure layer placement was compared with two alternative integration strategies: microservice deployment and middleware integration.

In the microservice approach, the LLM component is implemented as a separate service with its own deployment process and communication interface. This option may be more suitable in systems that include several AI-enabled features, require separate scaling of LLM workloads, or use different release cycles for AI-related functionality. A separate service also allows stronger technology isolation. At the same time, it increases operational complexity because the system must handle network communication, service discovery, monitoring, and fault tolerance across service boundaries. Applications using LLMs for a single feature may not require additional complexity.

Middleware integration places the LLM component in the request processing pipeline. This approach is suitable when AI functionality must be applied to all requests, such as automated moderation or content filtering. In the Budget system, LLM processing is triggered only for a specific user action. Middleware placement would therefore introduce latency into unrelated requests and make operational cost more difficult to manage. Table 3 presents a qualitative comparison of the proposed Infrastructure-layer placement, microservice deployment, and middleware integration. Operational complexity refers to the effort required for deployment and maintenance. Latency represents the expected delay introduced by model invocation. Isolation describes the degree of separation between the LLM component and the core system. Cost transparency reflects the ability to monitor and control LLM-related expenses.

Table 3. Comparison of LLM integration approaches.

Approach	Operational Complexity	Latency Isolation	Cost Transparency	Architectural Fit
Infrastructure Layer	Low	High	High	Clean Architecture
Microservice	High	High	Moderate	Distributed Systems
Middleware	Low	Low	Low	Pipeline Architectures

The proposed approach is compared with alternative integration strategies such as microservice-based deployment and middleware integration. Microservice-based solutions provide stronger isolation but increase system complexity and operational overhead. Middleware integration introduces model processing into the request pipeline, which may affect unrelated operations. The Infrastructure layer placement used in this study

provides a balanced solution by preserving architectural separation without introducing unnecessary complexity.

3.3. Implementation Results from the Budget System

The proposed architectural pattern was implemented in the Budget personal finance tracking system. Users can enter transaction descriptions in free-form language, for example "Lunch at a restaurant today at 12:00—35 lv". The LLM component converts this input into structured transaction records that contain date, amount, category, and description fields (Figure 4). The user has the option to confirm the AI-generated response or change the prompt and try again.

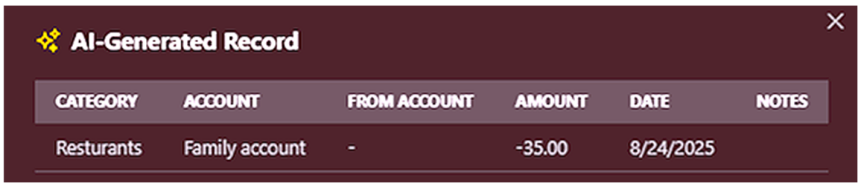


Figure 4. AI-generated transactions records.

The integration is implemented through the IQuickAddService interface in the Application layer and the QuickAddService implementation in the Infrastructure layer. The interaction is coordinated through the AiGenerateRecordQueryHandler, which invokes the LLM service and converts the generated response into structured QuickAddResult objects. Recent studies suggest that language models can support clearer separation between technical and business responsibilities in layered systems [14].

The implementation keeps LLM-related responsibilities outside the business logic. Prompt management, communication with Azure OpenAI Services, parsing, and validation remain in the Infrastructure layer. The Application layer only coordinates the request and processes the generated records. Existing evaluation frameworks also stress the need to validate LLM output before it is passed to the core application logic [15].

The validation process checks whether each generated record contains all required fields, including amount, category, and date. Additional rules verify that the amount format is valid, the category exists in the predefined category list, and the generated date falls within an acceptable range. Records that do not satisfy these conditions are discarded before they reach the persistence stage.

The generated records are presented to the user for confirmation before persistence. This adds a manual verification step before the data is stored. Figure 5 shows the processing flow of the Quick Add functionality, from free-form user input to validated transaction persistence.

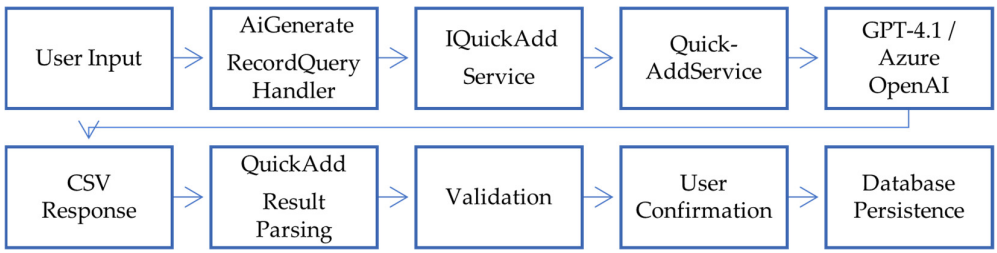


Figure 5. Workflow of the Quick Add process in the Budget system.

3.4. Computational Results for Non-Determinism, CSV Validity, Latency, and Cost

The computational experiment produced 300 recorded LLM responses. Table 4 summarizes the structural and semantic quality of the generated outputs.

Table 4. Validity and correctness of generated CSV responses.

Metric	Count	Percentage
Total LLM invocations	300	100%
Syntactically valid CSV responses	287	95.67%
Invalid CSV responses	13	4.33%
Responses passing all contract tests	269	89.67%
Responses rejected by validation	31	10.33%
Semantically correct responses	241	80.33%
Responses requiring user correction	59	19.67%

The most frequent errors were not network failures, but structurally or semantically invalid outputs. The invalid CSV responses included additional explanatory text, missing fields, and inconsistent delimiters. The contract-level errors included non-existing category names, missing account names, and dates outside the accepted range. These results justify the decision to isolate the LLM component outside the Domain layer and to require validation before persistence.

Table 5 reports the non-determinism results obtained from five repeated executions of each benchmark prompt.

Table 5. Snapshot and parsed-output stability across repeated runs.

Stability Metric	Value
Prompts tested	60
Repetitions per prompt	5
Prompts with identical raw CSV in all five runs	34
Raw snapshot agreement rate	56.67%
Prompts with identical validated parsed result in all five runs	46
Parsed-result agreement rate	76.67%
Prompts with at least one validation failure across five runs	18
Prompt-level validation instability rate	30%

The raw snapshot agreement rate was lower than the parsed-result agreement rate. This indicates that some responses differed textually while still producing the same validated transaction object. From a software engineering perspective, this distinction is important: exact textual repeatability is less relevant than contract-level stability. However, the presence of prompt-level validation instability confirms that the LLM component cannot be treated as a deterministic parser.

Table 6 summarizes the measured latency and token usage.

Table 6. Response time, token usage, and estimated cost.

Metric	Mean	Median	95th Percentile	Maximum
Response time, seconds	1.86	1.62	3.94	5.88
Input tokens	706	692	811	938
Output tokens	42	39	71	96
Estimated cost per request, USD	0.0017	0.0016	0.0022	0.0026

The measured response time confirms that LLM invocation is substantially slower than conventional in-process validation or database operations. Therefore, the implementation decision to invoke the LLM asynchronously inside a CQRS query handler is appropriate.

The measured cost per request was small for individual use, but it becomes operationally relevant under repeated use or multi-user load.

A small load experiment (Table 7) was performed by executing the same benchmark under different levels of parallelism. The purpose was to determine whether the LLM component affects the responsiveness of the Quick Add feature under concurrent use.

Table 7. Load experiment for the Quick Add LLM component.

Parallel Users	Requests	Successful Responses	Failed/Throttled Responses	Median Latency [s]	95th Percentile Latency [s]
1	60	60	0	1.62	3.21
5	100	99	1	2.04	4.80
10	100	96	4	2.91	7.65

The load experiment shows that latency increases under parallel requests. This behavior is expected because the LLM call depends on an external hosted inference service. The results support the architectural decision to keep the LLM integration in the Infrastructure layer, where retry logic, timeout handling, logging, and provider-specific rate-limit management can be implemented without affecting the Domain model.

4. Discussion

The results indicate that placing the LLM component in the Infrastructure layer is most consistent with the principles of Clean Architecture. This placement preserves the Dependency Rule and keeps the Application and Domain layers independent from external AI services. The use of an interface in the Application layer also makes it possible to replace the LLM provider without changes to the business logic.

The comparison with the other candidate placements shows that the Domain layer is not suitable for LLM functionality because it introduces an external dependency directly into the core business logic. Placement in the Application layer avoids this issue but moves infrastructure-related responsibilities into the layer responsible for orchestration and use-case coordination. This weakens the separation between business behavior and technical implementation.

The case study shows that LLM integration introduces requirements that are less significant in conventional software components. Testability becomes more difficult because the output of the model may differ across identical requests. This issue can be managed through interface-based abstractions and mock implementations. Response latency also becomes important because model invocation may require several seconds. In the proposed design, LLM interaction is limited to a specific user action, which reduces its effect on the rest of the system. The request is executed asynchronously within the query handler, so the request thread does not remain blocked during model invocation.

Output validation is also necessary because generated responses may be structurally invalid or semantically inconsistent. Typical examples include missing fields, incorrect delimiters, and category names that do not exist in the system. The implemented validation step reduces the risk of storing incorrect data. User confirmation provides an additional check before persistence. The use of recent transaction history and existing category lists in the prompt also improves category matching. This allows the system to adapt to the naming conventions and financial habits of individual users without additional model fine-tuning [9].

Operational cost is another important consideration because each LLM invocation consumes tokens and generates expense. The Budget system addresses this through limited prompt size and explicit user-triggered invocation. These measures reduce unnecessary

API usage and make LLM-related costs easier to estimate. The prompt length is restricted by limiting the number of historical transactions and categories supplied to the model. This prevents token usage from increasing together with long-term growth of user data.

The integration of external LLM services also introduces privacy and security concerns. In the Budget system, transaction descriptions may contain sensitive financial information related to merchants, payment habits, or account names. Systems that process this type of data may require additional protection measures, including anonymization of prompt content, masking of account identifiers, and stricter control over the information transmitted to external AI providers. In highly regulated environments, smaller local language models may be more suitable than cloud-based inference services.

The proposed pattern is suitable for systems in which LLM functionality is limited to a specific feature within a larger application. Systems with a higher volume of AI requests or stricter scalability requirements may benefit from a microservice-based architecture. In such cases, placing the LLM component in a separate service may simplify scaling and resource management.

The computational results provide empirical support for the non-determinism claim made in this study. Although most responses were syntactically valid CSV, 4.33% of responses were not parseable as valid CSV, and 10.33% failed at least one application-level contract test. Therefore, prompt engineering alone is insufficient for reliable persistence of generated records. This supports the adopted design in which parsing, validation, and user confirmation are mandatory steps before database storage.

The difference between raw snapshot agreement and parsed-result agreement is also important. Only 56.67% of the prompts produced identical raw CSV responses across all five repeated executions. However, 76.67% produced the same validated structured transaction. This means that the LLM output is non-deterministic at the text level, but some variation is harmless after parsing and validation. For this reason, the system should not rely on exact string equality. Instead, it should rely on contract tests over the parsed transaction object.

These findings are consistent with recent studies showing that hosted LLMs may produce different outputs even under settings usually expected to be deterministic. Atıl et al. report that API-based LLMs configured deterministically can still exhibit run-to-run variation and introduce agreement metrics such as raw-output and parsed-answer agreement rates. The present experiment adapts this idea to an engineering software system by distinguishing between raw CSV stability and validated transaction stability.

The results are also consistent with recent work on structured LLM output. StructEval explicitly includes CSV among the structured formats used to evaluate LLM output generation from natural-language prompts. At the same time, OpenAI's documentation distinguishes between ordinary JSON mode and stricter schema-based Structured Outputs; it states that JSON mode ensures valid JSON only in general, while schema adherence requires Structured Outputs. This comparison is relevant because the Budget system uses prompt-constrained CSV rather than schema-constrained JSON. Consequently, the observed invalid CSV rate is not surprising and further supports the need for application-side validation.

Compared with schema-constrained approaches, the CSV-based solution is simpler and lightweight, but less reliable. OpenAI reports that structured outputs with strict schema constraints can reach perfect schema adherence in their own internal evaluations for supported models and schemas. Therefore, if the Quick Add feature is later extended to high-volume or fully automated operation, replacing CSV with schema-constrained JSON or function calling would be technically preferable. In the present system, however,

the combination of CSV output, contract validation, and user confirmation is acceptable because the generated transaction does not persist automatically.

The latency and load results further show that LLM integration has different performance characteristics from ordinary application services. Median latency remained acceptable for interactive use, but the 95th percentile increased under concurrent load. This confirms that LLM calls should remain isolated from unrelated request processing. Middleware integration would be inappropriate for this case because it would introduce AI-related latency into requests that do not require natural-language transaction extraction. The Infrastructure-layer placement is therefore not only architecturally clean, but also operationally justified.

The present study has several limitations. The analysis is based on a single case study implemented in a personal finance application. The conclusions may not fully apply to systems that use multiple LLM components, agent-based orchestration, or retrieval-augmented generation. The accuracy of natural language parsing may also vary depending on the input language. Tests with Bulgarian-language input revealed inconsistencies in category matching and date interpretation. One possible explanation is the lower representation of Bulgarian text in the model training data. The study focuses on a single architectural style and does not examine event-driven or microservice-native systems in detail. It also does not include quantitative measurements of latency, token consumption, parsing accuracy, or error frequency. These factors may affect the practical suitability of different integration approaches.

5. Conclusions

This paper examined the integration of Large Language Models into systems designed according to Clean Architecture principles. Three possible placements of the LLM component were analyzed: Domain layer, Application layer, and Infrastructure layer. The analysis showed that the Infrastructure layer is the most suitable location because it preserves the Dependency Rule and keeps the core business logic independent from external AI services.

The proposed integration pattern combines Infrastructure layer placement with an interface defined in the Application layer and a CQRS-based interaction model. The implementation in the Budget system showed that this structure supports maintainability, testability, and separation between business logic and technical responsibilities.

The study also identified several issues related to LLM integration, including non-deterministic behavior, response latency, output validation, and operational cost. These issues require explicit handling during system design and cannot be managed in the same way as conventional infrastructure dependencies.

The findings show that established software architecture principles can still be applied in systems that include LLM functionality. Remote AI services introduce additional requirements for testing and validation, together with the need to manage operational cost. These issues are not limited to the presented case study and are also relevant to other layered systems that include LLM components.

Author Contributions: Conceptualization, A.I.; methodology, G.K., A.I. and S.G.; software, G.K.; validation, F.S., T.B. and S.G.; formal analysis, A.I.; investigation, F.S. and T.B.; resources, G.K., A.I. and F.S.; data curation, G.K.; writing—original draft preparation, A.I.; writing—review and editing, F.S. and T.B.; visualization, S.G.; supervision, T.B.; project administration, A.I.; funding acquisition, S.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research was financed by the European Union—NextGenerationEU, through the National Recovery and Resilience Plan of the Republic of Bulgaria, project No. BG-RRP-2.013-0001.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

LLM	Large Language Model
CQRS	Command and Query Responsibility Segregation
API	Application Programming Interface
GPT	Generative Pre-trained Transformer
REST	Representational State Transfer
CSV	Comma-Separated Values
HTTP	Hypertext Transfer Protocol

References

1. Weber, I. Large Language Models as Software Components: A Taxonomy for LLM-Integrated Applications. *arXiv* **2024**, arXiv:2406.10300. [CrossRef]
2. Bucaioni, A.; Weyssow, M.; He, J.; Lyu, Y.; Lo, D. A Functional Software Reference Architecture for LLM-Integrated Systems. In Proceedings of the 2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C), Odense, Denmark, 31 March–4 April 2025.
3. Becattini, M.; Verdecchia, R.; Vicario, E. SALLMA: A Software Architecture for LLM-Based Multi-Agent Systems. In Proceedings of the 2025 IEEE/ACM International Workshop on New Trends in Software Architecture (SATrends), Ottawa, ON, Canada, 28 April–2 May 2025.
4. Patrikov, G.; Bakardjieva, T.; Ivanova, A.; Ivanova, A.; Sapundzhi, F. A Prototype of Integrated Remote Patient Monitoring System. *Eng. Proc.* **2025**, *104*, 68. [CrossRef]
5. Chondamrongkul, N.; Hristov, G.; Temdee, P. Addressing Technical Challenges in Large Language Model-Driven Educational Software System. *IEEE Access* **2025**, *13*, 12846–12858. [CrossRef]
6. González-Potes, A.; Martínez-Castro, D.; Paredes, C.M.; Ochoa-Brust, A.; Mena, L.J.; Martínez-Peláez, R.; Félix, V.; Félix-Cuadras, R.A. Hybrid AI and LLM-Enabled Agent-Based Real-Time Decision Support Architecture for Industrial Batch Processes: A Clean-in-Place Case Study. *Appl. Inform.* **2026**, *7*, 51. [CrossRef]
7. Hruzin, D.; Lytvynov, O. Engineering of Software Systems Based on Snapshot-Centric CQRS with Event Sourcing Architecture. *KPI Sci. News* **2026**, *142*, 60–74. [CrossRef]
8. Chauhan, S.; Rasheed, Z.; Sami, M.A.; Zhang, Z.; Rasku, J.; Kemell, K.-K.; Abrahamsson, P. LLM-Generated Microservice Implementations from RESTful API Definitions. In Proceedings of the International Conference on Evaluation of Novel Approaches to Software Engineering, Porto, Portugal, 4–6 April 2025.
9. Luo, H.; Liu, Y.; Zhang, R.; Wang, J.; Sun, G.; Niyato, D.; Yu, H.; Xiong, Z.; Wang, X.; Shen, X. Toward Edge General Intelligence with Multiple-Large Language Model (Multi-LLM): Architecture, Trust, and Orchestration. *IEEE Trans. Cogn. Commun. Netw.* **2025**, *11*, 3563–3585. [CrossRef]
10. Ivanova, A. *Models and Algorithms for Digital Business Ecosystems*; Varna Free University "Chernorizets Hrabar" University Press: Varna, Bulgaria, 2026; 118p, ISBN 978-954-715-801-6. (In Bulgarian)
11. Petrova, P.; Varbanov, Z. Implementation of the software design patterns mediator and CQRS with the library mediatr in .NET. *Math. Comput. Sci. Educ.* **2024**, *7*, 75–82. [CrossRef]
12. Peneva, T.; Avramova, T.; Georgiev, P. Software Application for Automated Evaluation and Selection of a Rational Technological Process. *Eng. Proc.* **2025**, *104*, 71. [CrossRef]
13. Lytvynov, O.; Hruzin, D. Decision-Making on Command Query Responsibility Segregation with Event Sourcing Architectural Variations. *Technol. Audit Prod. Reserves* **2025**, *4*, 37–59. [CrossRef]

14. Boukhari, M.E.; Kharmoum, N.; Ziti, S. AI-Powered Architecture Refactoring: From Legacy Systems to Modern Patterns. *Int. J. Adv. Comput. Sci. Appl.* **2025**, *16*, 568–576. [CrossRef]
15. Lin, T.-H.; Kao, C.-H. FROAV: A Framework for RAG Observation and Agent Verification—Lowering the Barrier to LLM Agent Research. *arXiv* **2026**, arXiv:2601.07504. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.