

КОМП'ЮТЕРНА ІНЖЕНЕРІЯ

DOI: <https://doi.org/10.20535/kpissn.2026.1.350992>

UDC 614.2+574/578+004.38

D.L. Hruzin^{*}, <https://orcid.org/0009-0004-8534-2559>
O.A. Lytvynov¹, <https://orcid.org/0000-0001-7660-1353>

¹Oles Honchar Dnipro National University,
Dnipro, Ukraine, <https://ror.org/00qk1f078>

*Corresponding author: hruzin_d@365.dnu.edu.ua

ENGINEERING OF SOFTWARE SYSTEMS BASED ON SNAPSHOT-CENTRIC CQRS WITH EVENT SOURCING ARCHITECTURE

Background. Command Query Responsibility Segregation (CQRS) with Event Sourcing (ES) is a widely adopted solution for designing scalable and high-performance information systems. However, classical CQRS with ES implementations are often associated with increased complexity in development and maintenance.

Objective. The goal of this study is to optimise the development and maintenance of software systems built on CQRS with ES by introducing an alternative variation of the architecture.

Methods. The classical architectural variation was analysed, and the components that increase the complexity of system development and maintenance were identified. Based on this analysis, an alternative architectural variation (mCQRS) is proposed, that uses a lower-complexity component set. The solution is based on a relational database in which aggregate state snapshots are treated as the source of truth, thereby reducing implementation and maintenance complexity and facilitating potential migration to other architectural variations.

Results. Representative test projects were developed for both the classical and the proposed CQRS with ES variations. Cyclomatic complexity values for a typical command execution workflow (120 for Classical CQRS and 82 for mCQRS) indicate a 31.67 % decrease in complexity. Performance measurements show that server response time for queries is identical for both variations (44 ms), whereas the end-to-end time to reach system consistency for commands is 268 ms for Classical CQRS and 347 ms for mCQRS, corresponding to a 22.76 % decrease in performance. Despite this degradation, write-operation throughput remains high in the context of established industry practices.

Conclusions. The proposed approach improves the efficiency of development and maintenance and reduces the required level of developer expertise; it is suitable for systems in which write-operation performance is not critical.

Keywords: Software Architecture; optimisation models; CQRS; Event Sourcing comparative analysis.

Introduction

The complexity of software systems (SS) is continuously increasing, while the demands for project timelines and implementation quality are becoming stricter. To handle the situation, software developers are forced to seek new approaches, architectural patterns, and technologies. Renowned books [1, 2, 3] suggest a lot of patterns and solutions, focused on how to manage the complexity of modern business-oriented software systems, making them more flexible, scalable and maintainable.

One of the effective approaches used to build SS with high-performance requirements, like E-Commerce, Banking, or Financial Systems is an event-

driven architecture (EDA) [4], which states that the system can be seen as a simulator of the real business domain in which the interaction between the components is driven by events. Thus, the event raised by a certain component causes the reaction of other components and even integrated third-party systems. This approach is also used for service-oriented systems, i.e. the systems built as a composition of autonomous, heterogeneous components-services [5]. The event-driven action may include the invocation of a service, the triggering of a business process, and/or further information publication/syndication.

Command and Query Responsibility Segregation in combination with ES architecture (hereafter

Пропозиція для цитування цієї статті: Д.Л. Грузін, О.А. Литвинов, «Інженерія програмних систем на основі архітектури CQRS з ЕС, що ґрунтується на знімку стану системи», *Наукові вісті КНУ*, № 1, с. 60–74, 2026. doi: <https://doi.org/10.20535/kpissn.2026.1.350992>

Offer a citation for this article: D.L. Hruzin, O.A. Lytvynov, “Engineering of software systems based on snapshot-centric CQRS with event sourcing architecture”, *KPI Science News*, No. 1, pp. 60–74, 2026. doi: <https://doi.org/10.20535/kpissn.2026.1.350992>

referred to as the canonical CQRS approach) [6], which was proposed in [7, 8] can be seen as a variation of a more general EDA paradigm [9] with a concentration on increasing the speed of processing the requests divided into two categories: write and read operations.

The discussion on the use cases where CQRS with ES architecture is the most applicable is represented in [10]. The authors discuss that this sort of architecture solution is a good choice for systems that are based on events on a business level, e.g. trip systems, financial systems, etc.

The advantages of the CQRS with ES architecture compared to DDD [11, 12] include improved performance for read and write requests, as well as better flexibility and scalability due to asynchronous event processing and reduced risk of conflicts when making changes. This is because commands that modify data and queries that read data operate independently of each other. Another significant advantage is the instant storage of all events, enabling the system's state to be restored to any point in time from its creation to the present.

This study analyses the conventional CQRS with ES architecture, highlighting its limitations and reviewing existing mitigation strategies. An alternative architectural approach is introduced to address these limitations. A comparative evaluation of the classical and proposed approaches is conducted, focusing on performance and implementation complexity. While a thorough architectural comparison would necessitate a broader set of evaluation metrics, such an extensive analysis is considered beyond the scope of this work.

The object of study is the development and maintenance process of software systems based on CQRS with ES.

The subject of study is methods and architectural mechanisms for reducing development and maintenance complexity in CQRS with ES-based systems.

The purpose of the work is to optimise the development and maintenance of CQRS with ES-based software systems by introducing an alternative variation of CQRS with ES architecture.

Problem Statement

As highlighted in the earlier research [13], the primary challenges associated with CQRS with ES can be categorised into three distinct groups: technical development issues, development and maintenance complexity, changing the architectural solution modifications of SS during the later stages of development.

Technical development issues include event replay performance problems during aggregate assembly and projection rebuilds, the complexity of handling event versioning, managing General Data Protection Regulation (GDPR) compliance [14], given the immutability of events, and the lack of a guarantee that events will be delivered in the order they were published.

The development and maintenance complexity leads to additional difficulties. The complexity challenge is discussed in [15], noting that while the CQRS pattern itself is relatively simple, its combination with other approaches like DDD and ES, drastically increases complexity. As demonstrated in practice [16], when transitioning to a CQRS with ES architecture, the amount of code does not necessarily increase, but the number of layers and modules does. Each of these layers requires updates whenever a new feature is added or updated.

For example, when there is a business requirement to quickly implement a new test feature for demonstration purposes or to provide temporary functionality with minimal resource expenditure, this architecture necessitates the creation and testing of all layers, from command handlers to query handlers. The greater complexity of the system means that development and maintenance take more time and require a higher level of skill from the development team, leading to longer timelines and increased costs, which significantly reduces its attractiveness to investors.

The operation of changing the architectural solution of an SS at later stages of development is another challenging aspect. E.g. the migration of the SS from DDD to CQRS with ES architecture is far from trivial task [16]. The primary difficulty in changing the architecture of a live SS lies not in rewriting system components or redistributing layers, but in data migration. DDD architecture typically relies on a relational database [17], which is absent in CQRS with ES architecture. As a result, implementing an architectural transition strategy like Chicken Little [18] is difficult due to the need to maintain two sources of truth (the relational database and the event store) during the architectural shift, which can be a prolonged process. Other migration strategies face challenges in performing complex data migrations, including transformations and event log derivations [19]. Additionally, it is often necessary to maintain some synchronisation processes between the legacy database and the event store for a period of time after the release of the SS with the updated architecture to allow for fallback options.

Thus, this paper focuses on proposing an architectural approach that:

- Optimises the resolution of technical development issues associated with CQRS with ES architecture.
- Offers improved flexibility for migrating from one architectural solution to another, depending on project dynamics, compared to migrations between DDD and canonical CQRS architectures.

Review of the literature

In this section, we will describe technical development issues associated with CQRS with ES architecture and their known solutions, providing the minimum necessary contextual information required to understand the problem and solution.

As outlined in [13], write requests can be classified into two categories: creation-oriented and update-oriented commands. For update-oriented commands, a cache miss may occur during the aggregate retrieval stage, when the Repository attempts

to load the aggregate from the Cache but does not find it (Fig. 1). In this case, the Repository loads from the Event Store all events associated with the corresponding aggregate identifier, creates a new aggregate instance, and applies the events sequentially to reconstruct the aggregate state. This reconstruction procedure is referred to as event replay (i.e., replaying events).

The time of acquiring an aggregate by replaying events is proportional to the number of events and depends on its lifetime and the frequency of changes applied to the aggregate. In some cases, the replaying process per aggregate can take seconds, which affects the overall system performance. Thus, in the case of a large number of events and a large number of aggregates **replaying events, the process starts to cause performance issues.**

To address this issue, it is suggested to use an additional data store that retains snapshots of an aggregate's state at predefined intervals, either time-based (e.g., every 6 hours) or event-count-based (e.g., after every N events, such as 100 or 1,000)

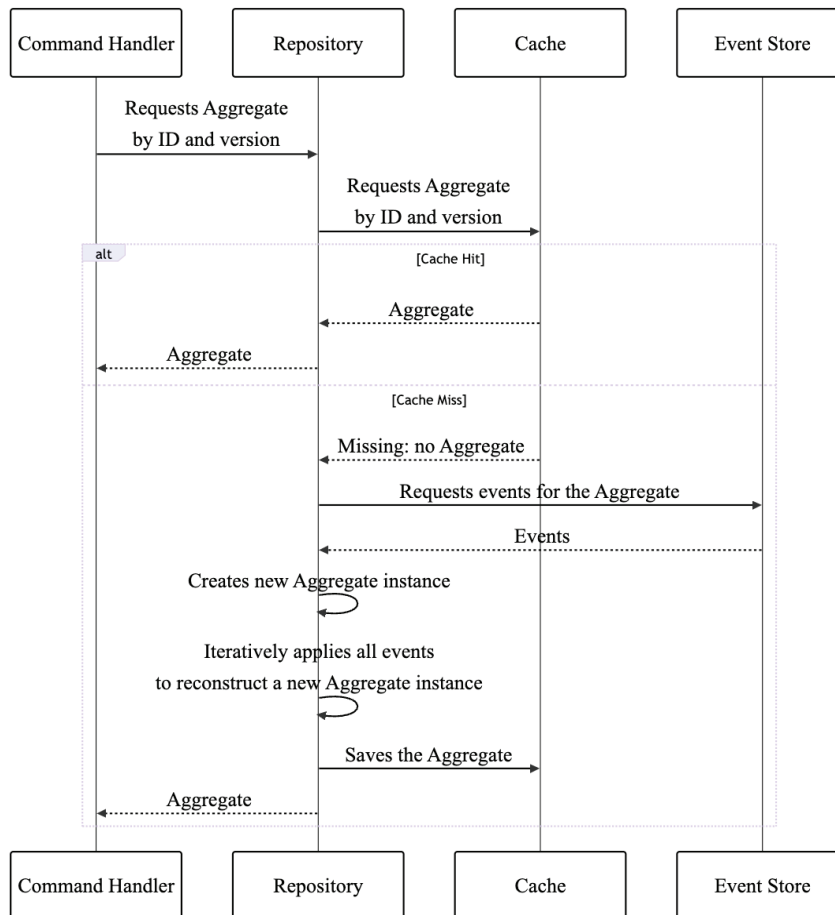


Fig. 1. Aggregate retrieval workflow in classical CQRS

[20]. A snapshot of an aggregate is a database record that contains the identifier and version of the stored aggregate, as well as the aggregate's state, often in a serialised form [21].

Another aspect of this problem is the rebuilding of projections. Rebuilding a projection is an operation where all projection data is deleted, and then all events are replayed one by one, starting from the very first event. As a result of this operation, the projection updates its data and aligns with the Source of Truth of the current version.

Similarly to the problem of aggregate assembling, the **rebuilding of projections can take a significant amount of time**: hours or even days in some cases. To solve this issue, developers use the snapshots of projections [22].

This approach solves the performance problem but adds complexity to the system. The Read Model in complex software systems is typically quite sophisticated. In accordance with DBB company [23] experience some of the projections may be subscribed to 90 percent of events. Applying the snapshot solution to some Read Model each event it is subscribed to should be modified to support both replay and snapshot driven modes of its reconstruction. This causes an increase in development effort during both development and maintenance phases.

Also, study [13] discusses several characteristics of the classical CQRS with ES approach, such as the need to handle event type versioning [24] and the immutability of events [25]. These characteristics complicate system modification, particularly the procedure for removing or anonymising all user data from the system in response to a request under the GDPR [14].

Another issue discussed in [13] is related to the event delivery subsystem based on an event bus, which does not guarantee that events will be delivered in the same sequence as they were originally published (Fig. 2) [26]. To avoid the corruption of data the projection stores the version of the aggregate state according to which it was updated [24,

27]. When the handler attempts to update the projection with version $(n - 1)$ to version $(n + 1)$ the version mismatch error is raised. The version mismatch error is typically resolved with a retry operation. The handler waits for a certain amount of time τ and then attempts to update the projection again. If, during the time τ , the projection was updated to version n according to another event, the next retry will be successful. There are cases when, after a certain number of attempts, the projection still cannot be updated. This issue can be resolved in several ways:

- The handler can request all events up to the current one from the Event Store, recreate the projection using the event replay algorithm, apply the changes related to the current event, and then update the projection.
- Log the version mismatch error for further investigation and action by a developer.

Since these situations are relatively rare, the first option is complex to implement and resource-intensive to execute, in practice, the second option is usually preferred.

Beyond technical development issues that already complicate system maintenance, CQRS with ES adds risky complexity to the system [7]. In practice, this complexity is typically mitigated through organisational and process-oriented practices, including focused team upskilling, higher-quality project documentation, expanded automated test coverage, and stricter code review and delivery pipelines. In addition, it is advised to apply CQRS with ES selectively (only to those subsystems where the expected benefits outweigh the added operational and conceptual overhead) rather than adopting it as a system-wide default. For example, in a microservices-based system [28], CQRS with ES can be limited to services that clearly benefit from event-driven separation and auditable state evolution, while other services may rely on alternative architectural approaches that better match their functional requirements and complexity constraints.

A separate study was conducted on the issue of changing the architecture solution of SS at later stages of development [16]. Other works on this topic describe that migrating an SS to another architecture is not a simple task. There are various strategies for system replacement, such as Cold Turkey and Chicken Little [18]. Cold Turkey involves rewriting a legacy system from scratch, while the Chicken Little approach assumes small incremental steps

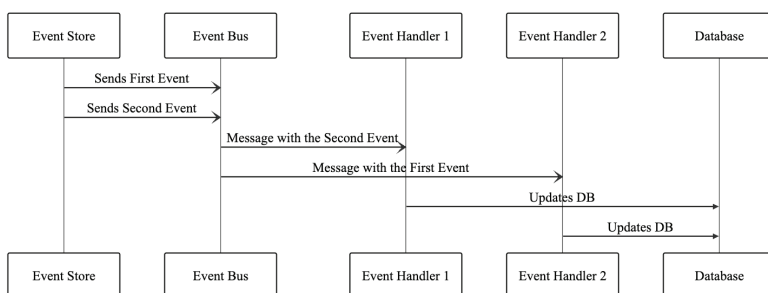


Fig. 2. Asynchronous events order

until the desired long-term objective is reached. Another approach is Butterfly [29], which focuses on the migration of legacy data in mission-critical environments. The data migration process is one of the most important and complex steps in system migration. One method to address this challenge migrating SS based on DDD to CQRS with ES is the derivation of event logs [30], which involves analysing the system and implementing an event logging module. These events are subsequently used to synchronise data between the old and new systems.

In addition to system replacement strategies, Salvatierra G. [31] considers a number of direct migration approaches, such as Screen Scraping, Sneed, Canfora, COB2WEB, and many others. However, all of these solutions are applied at a more abstract level, providing strategies for system migration, which are instrumental in successfully carrying out the migration but do not simplify the transition process itself.

The proposed methods for addressing technical development issues are applied in practice and effectively solve existing problems. However, the use of such solutions often increases the complexity of IS. The methods aimed at reducing system complexity are generally focused not on simplifying the CQRS with ES architecture itself, but on stabilising and simplifying other architectural decisions, system components, and the development process. Given this situation, the most appropriate solution for reducing complexity is to propose an alternative version of the CQRS with ES approach.

Materials and methods

To address the task of reducing the complexity of development and maintenance of the SS based on the CQRS with ES architecture, the following strategy is proposed:

1. Conduct an in-depth analysis of existing approaches consolidated into a single architectural solution (classical CQRS architectural variation).
2. Propose and describe a set of alternative solutions, grouped into a variation of the CQRS with ES architecture (mCQRS).
3. Apply both approaches in practice, measure the characteristics of typical test systems, and compare the methods based on their features and obtained metrics.

In the previous study [13], a technology for selecting an optimal CQRS with ES architectural variation for a specific project was proposed. In that study, Classical CQRS and mCQRS were considered as candidate variations. The Classical CQRS

variation was already analysed, whereas the mCQRS variation was only briefly outlined in terms of its key characteristics.

This paper provides a detailed description of the mCQRS variation. It also expands the description of Classical CQRS presented in [13] to support a transparent comparison and to enable consistent analogies between Classical CQRS and mCQRS.

Classical CQRS architectural variation

The component view of the SS built using the CQRS with ES approach is presented in Fig. 3. The analysis is based on the documentation in [8] and the example repository [32]. The SS is organised into three primary subsystems. The Write Model processes commands. The Read Model handles query requests. The Notification Subsystem provides an event-based communication layer between the Write Model and the Read Model.

The core workflow of the approach is summarised in Fig. 4. When a client submits a command, it is processed by the Write Model. Command handling produces a set of events. These events are published to the Notification Subsystem, which routes them to subscribers, including the Read Model event handlers. When a client issues a data retrieval request (query), the Read Model returns a Data Transfer Object (DTO) from a denormalised data store.

The Write Model subsystem is responsible for command execution. It first verifies that the requested command is supported by the system and that the initiating user has sufficient permissions. The command payload is then validated before further processing.

In case of successful validation, the command is routed to the Command Bus, which can be thought of as the gateway to the Command Processing Unit. The Command Bus is connected to a set of command handlers. A command handler is a component responsible for managing task execution. Typical actions of the command handler include the following.

The command handler requests an aggregate by its identifier and version from the repository. If the required version of the aggregate is available in the repository cache, it is immediately returned. If not, a new aggregate instance is created and populated with data from the latest snapshot, if one exists, or with default values otherwise.

After obtaining the aggregate, the command handler calls an appropriate method of the aggregate to change its state. If an error occurs, an acknowledgement response with error details is sent to the client, and the workflow gets terminated. If the ag-

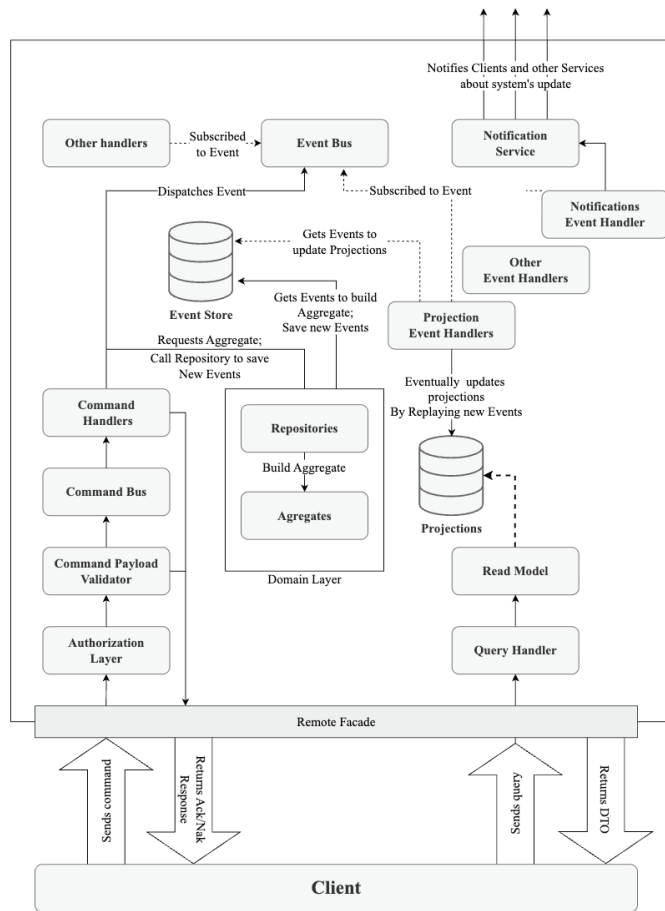


Fig. 3. Component diagram of Classical CQRS-based system

gregate is successfully updated, the method returns a set of events.

The command handler then calls the repository again, which updates the cache with the new version of the aggregate, saves the aggregate snapshot (if a predefined condition is met, e.g. every 100 events), and stores the set of events in the Event Store. After this, the events are posted to the Event Bus, i.e., passed to the Notification Subsystem.

The detailed view of the Command Processing Unit workflow is shown in Fig. 5.

The Notification Subsystem delivers events produced by Write Model command handlers to other internal components and, where required, to external environments (e.g., third-party services). Similar to the command-processing part, it includes a gateway referred to as the Event Bus and a set of event handlers. These handlers process events emitted as outcomes of command handling.

Communication between Event Bus and event handlers commonly follows the publisher-subscriber pattern [33], where handlers subscribe to specific types of events. But it should also be considered that the use of the publisher-subscriber pattern is not only one way of how the Notification System could be realised. For example, instead of an active server variant based on notifying the subscribed clients about certain events, a variant of the passive

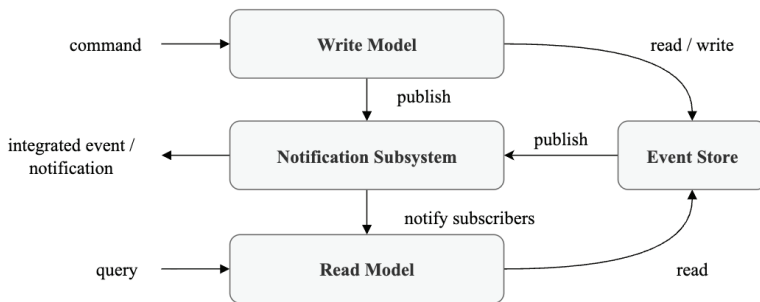


Fig. 4. Three basic subsystems and their interaction

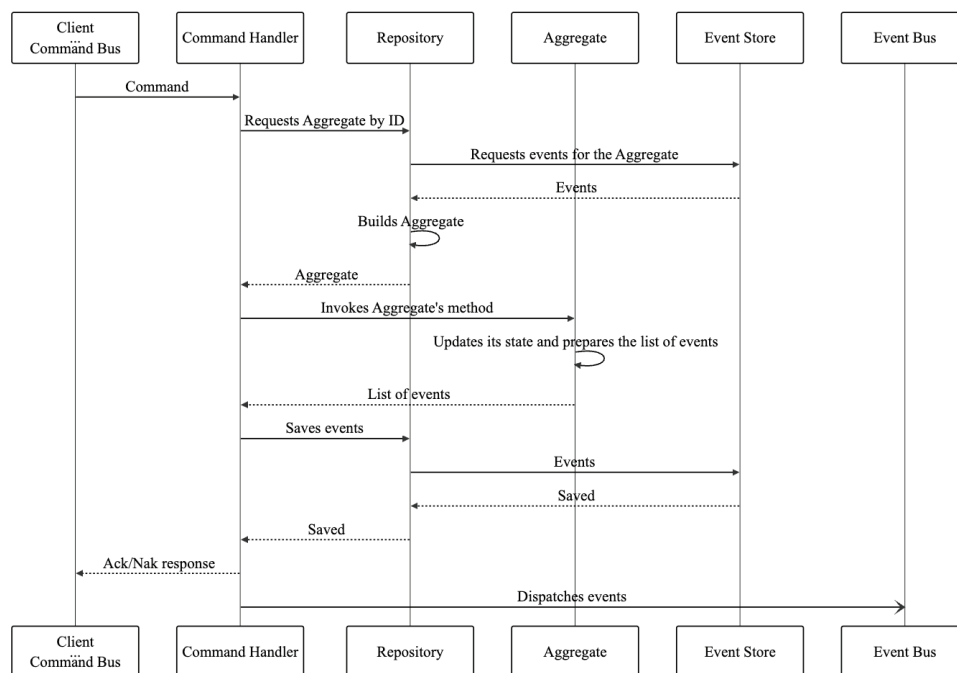


Fig. 5. Sequence diagram of the Command Processing Unit workflow

server pattern with a variation of polling the Event Store could be provided [34]. According to this variant event handlers continually request the Event Store whether the events they are interested in have occurred and, if so, pick them up and process. Of course, more sophisticated models of passive server patterns could also be used (e.g. different types of the Event Store replication) [35]. The advantages and disadvantages of the Passive server variants of the solution are well known and their detailed explanations are not the purpose of this work. We can only admit that the use of Event Bus can be regarded as the most common approach and that is the reason why we are focusing on its implementation and problems connected to this approach.

Some event handlers implement isolated business functions. For example, they may send notifications to external subscribers or perform post-processing required for internal validation and error detection. Another major class of handlers is dedicated to maintaining system consistency. In addition to the source-of-truth data store (typically the Event Store), the system includes a secondary storage layer that maintains projections, which must be updated for the system to eventually become consistent.

Projections (derived views, persistent read models) are denormalised, precomputed representations created to optimise read-side workloads. They may be customised for particular read scenarios, which

improves system responsiveness and overall performance. The storage and access mechanisms for projection data are implementation-dependent and are determined by architectural decisions and runtime constraints. A common strategy persists projections in SQL/NoSQL databases or cloud storage services. In some cases, projections are also stored in file-based repositories as documents or images. An alternative strategy keeps projections in memory and reconstructs them from the local event stream whenever server is rebooted. Consequently, projections are often treated as read-optimised services rather than as conventional persistent storage entities (e.g., database tables).

The transformation of an event stream into a projection is referred to as projecting [36]. Projection update handlers subscribe to events on the Event Bus and wait for relevant messages. When the Event Bus delivers a relevant event, the projection update handler advances the corresponding projection to the next version in accordance with the defined business logic and the event payload. The process is described in Fig. 6.

During projection updates, the Notification Subsystem also informs connected clients about changes in the system state.

Once all event handlers have successfully executed, the write request (command) processing is considered complete, and the system has reached a consistent state.

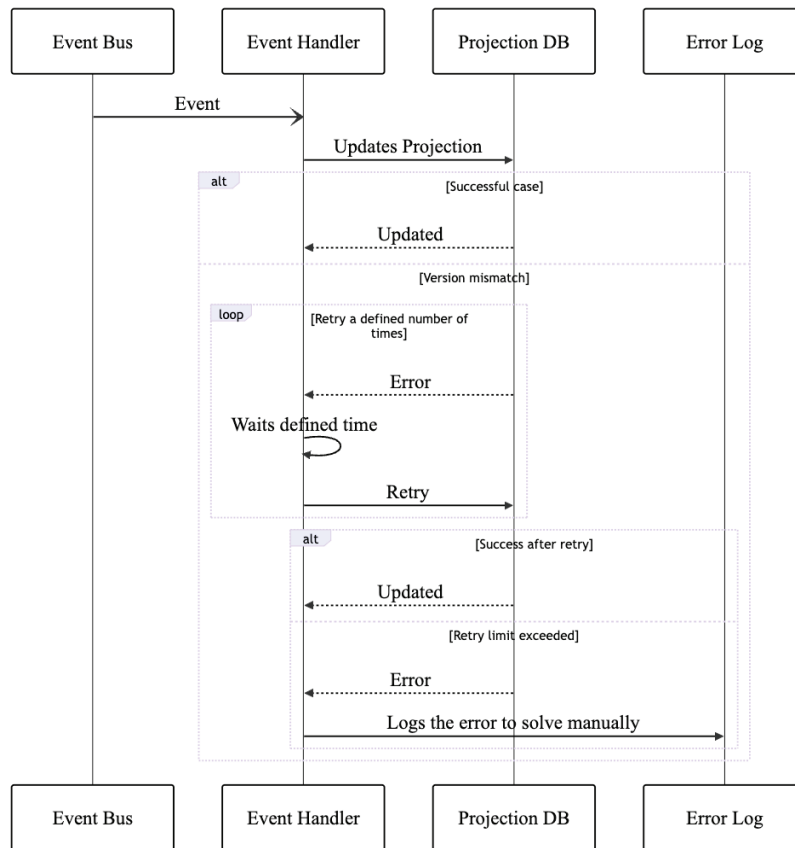


Fig. 6. Sequence diagram of the Update Projections workflow

The read request (query) process is fully handled by the Read Model subsystem. This unit includes Query Handlers, repositories for managing Projections and Projections.

When the system receives a request to read data, the Query Handler requests data from the corresponding repository, which retrieves pre-prepared data from Projections.

CQRS with ES approach with Snapshot database as a source of truth (mCQRS)

For certain systems, immediate response for write operations may not hold significant value. Usually, these are systems with a low percentage of write operations, like 20 % or less. Therefore, it is advisable to consider the following modified system architecture, which simplifies the development and maintenance of the system, as well as the potential transition of the architecture either towards classical CQRS or DDD approach.

The essence of the modification lies in shifting the focus from the event store to the database snapshot (Fig. 7). In contrast to the classical method, the

snapshot database in this approach closely resembles a relational database. The aggregate state is not serialised and may even be distributed across multiple tables. The snapshot is considered the source of truth where all the relevant information is stored. This modification makes event replay operation unnecessary for the majority of cases, replacing it with fetching data from the latest snapshot version with optional, on-the-fly, transformation.

When a command to add/modify data is received (Fig. 8) Command Handler in the same manner as in the classical solution requests aggregate by identifier and version. But instead of building aggregate from scratch (or snapshot) by replaying events, it just simply takes an up-to-date snapshot of it from the DB. Then, just as in classical CQRS, the corresponding aggregate method is called, which returns a set containing one or more events.

To update the snapshot, events are immediately applied to the aggregate. This procedure also helps ensure there are no errors during subsequent event replay operations (if they are needed). If applying an event leads to unexpected behaviour, this issue is detected before the event and the aggregate's new state

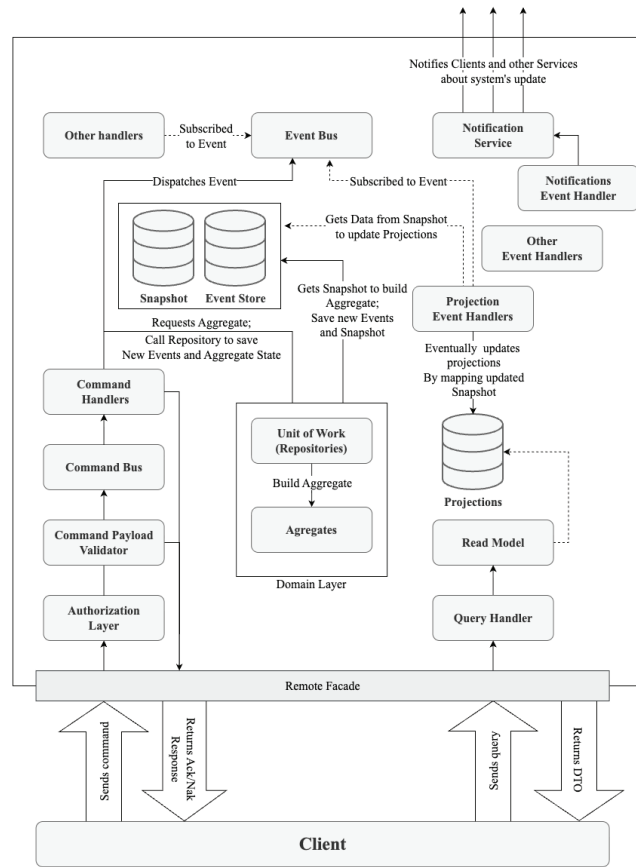


Fig. 7. Component diagram of mQRS-based system

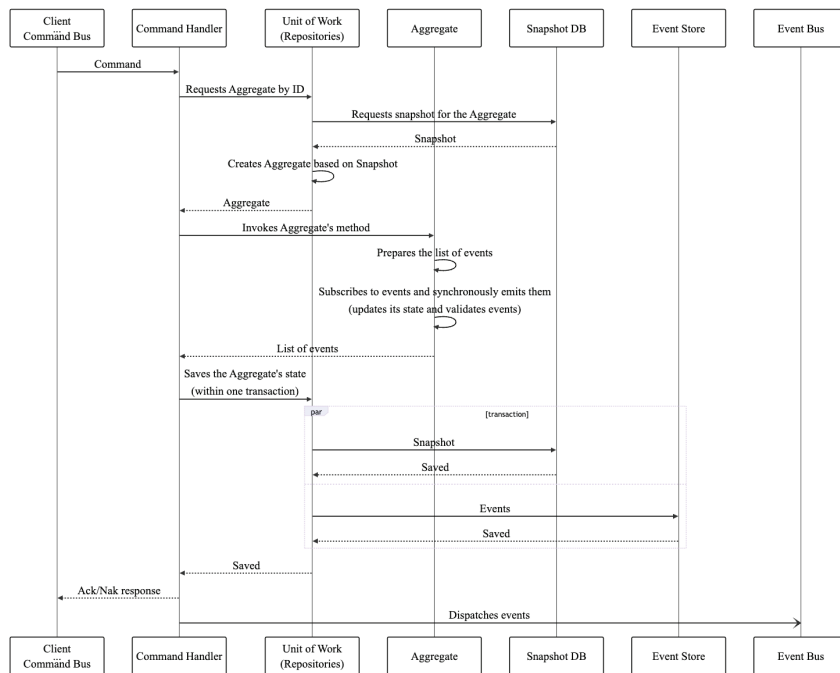


Fig. 8. The write operation flow in the proposed approach

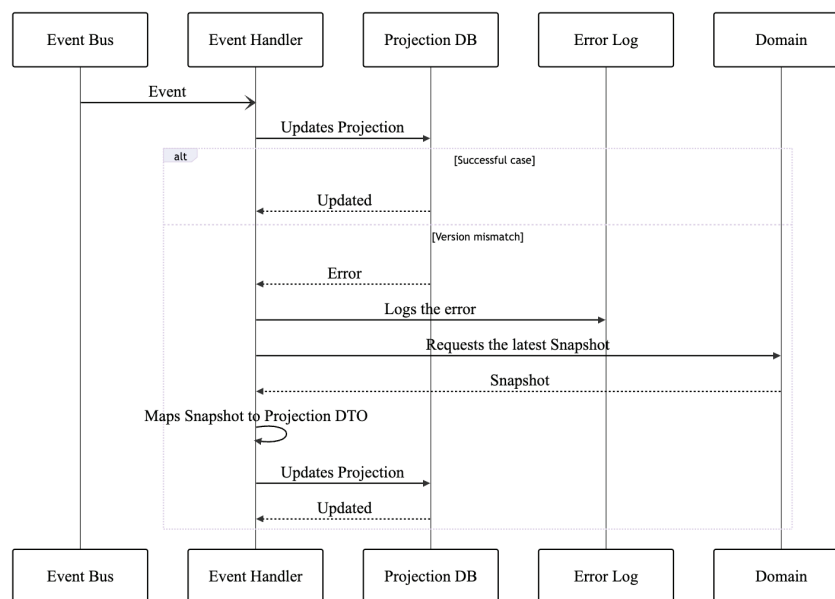


Fig. 9. Projections update in the proposed approach

are saved, allowing the operation to be cancelled. If the aggregate's state is modified successfully, it confirms that the events are valid and can be safely stored.

After the state is successfully updated, event store repository stores events and snapshot repositories update within one transaction that guarantees consistency of the Snapshot and Event Store. Despite the fact that events replay operation is not-required anymore, it is still possible when needed. Subsequently, Command Handler dispatches events to the Event Bus, to which multiple event handlers are subscribed.

In this step, the synchronous response is sent to the client. The system returns an acknowledgement response indicating that the Event Store has been updated and that the corresponding changes will be propagated across the system eventually. It should be noted that if the updated entity in the Snapshot Database contains all data required by the client application, it is technically possible to return it immediately instead of an acknowledgement response.

Projection update event handlers are subscribed to appropriate events and eventually update projections (Fig. 9). Version mismatch issue can be solved by taking the latest data from the Snapshot DB, followed by mapping if necessary. This operation is much simpler than the equivalent one in the classical CQRS approach, as it does not require event replay operation. After updating the projections, notifications about the data

update are sent to clients (e.g. mobile or web applications).

Querying data process works without any modifications and keeps the response time as short as possible in the system based on CQRS with ES architecture.

Experiments and results

The experiment aims to compare the classical CQRS and mCQRS approaches with respect to system performance and complexity.

The experiment involves developing two Representative Test Projects (RTPs) with identical functionality: one based on the classical CQRS architectural variation and the other on mCQRS. For each system, complexity and performance metrics were measured. The source code for both RTPs is available at [37]. The systems were deployed on an AWS cloud server and connected to a remote database (Amazon RDS).

For complexity measurement the McCabe's cyclomatic complexity [38] method is used. According to this method, the measurement is based on the amount and level of functions, methods, and procedures (e.g. loops and conditions). The higher this amount, the more difficult it will be for the developer to build, understand, and modify the code. For quantitative complexity assessment, the program code is divided into blocks and represented as a directed graph. Cyclomatic complexity (CC) is calculated using the following formula:

$$CC = E - N + 2P,$$

where E is the number of edges in the graph, N is the number of nodes in the graph, P is the number of connected components.

Cyclomatic complexity measurement was automated with SonarCloud tool [39]. Complexity was calculated for each RTP both at the system level and for its individual components to identify modifications that simplify development. Additionally, the complexity of adding a new simple aggregate root was measured to assess the modification complexity of the system [40].

The resulting metrics are summarised in Table 1. In these calculations, specification (test) files were excluded because they substantially inflate the metric yet do not reflect the intrinsic complexity of the system.

Table 1. Cyclomatic complexity metrics

	Classical CQRS	mCQRS
Common part	42	20
Snapshot repository	9	0
Event Store	12	7
Base projection repository	8	0
Aggregate	78	62
Aggregate repository	14	8
Aggregate	9	6
Projections repository	24	17
Total RTP (With 1 aggregate)	120	82
Total (With N aggregates)	$42 + 78 \cdot N$	$62 + 20 \cdot N$

Fig. 10 describes how complexity is distributed across the components of systems based on Classical CQRS and mCQRS variations. The purpose of this diagram is to illustrate the component hierarchy and the portion of the overall system complexity each component contributes.

To evaluate performance, average response times for write and read operations in both RTPs were measured. Averages were computed over one thousand requests per operation. The measurement script is available at [41].

Another time-based metric relevant to eventually consistent systems is the end-to-end propagation latency – the time from receiving a modification request to the corresponding projection update. This metric was also measured [41]. For half of the requests, entities were served from cache (warm), whereas for the other half, the cache was empty

(cold). The resulting average propagation latencies are summarised in Table 2.

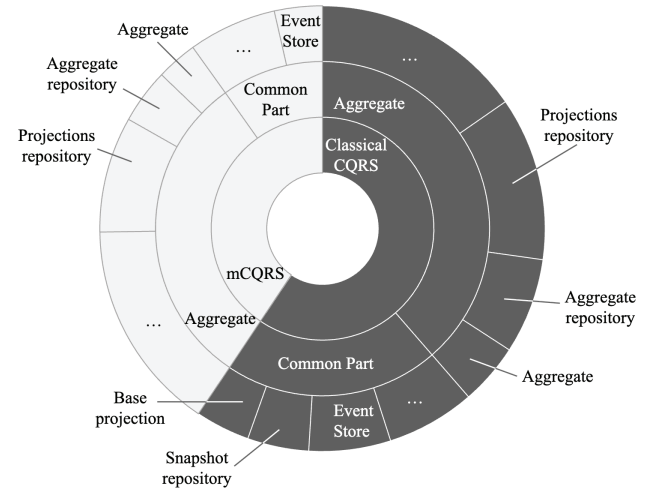


Fig. 10. Complexity distribution between components

Table 2. Performance metrics

	Classical CQRS (ms)	mCQRS (ms)
Query response time	44	44
Command response time	107	186
System eventual update time	268	347

Discussion

Having a Snapshot DB as the source of truth instead of Event Store simplifies the development and maintenance of the system. It allows the replacement of the event replay operation with retrieving the record from the database and mapping it to a new aggregate. This is supported by cyclomatic complexity metrics for the “Aggregate repository” component: 14 for classical CQRS versus 8 for mCQRS, “Projection repository” component: 24 versus 17, as well as the “Base projection” component, which exists only in the classical CQRS variation and has a complexity value of 8.

The projection update operation is simpler for machine processing than event replay. The complexity of the operation to populate an aggregate using data from a database record depends only on the number of properties in the aggregate (m). In contrast, the complexity of event replay also de-

depends on the number of events that occurred with the aggregate instance (n). Therefore, if the number of aggregate properties remains unchanged, the complexity of the populate operation using Big O notation method [42] is $O(m)$, while the event replay complexity is $O(n \cdot m)$. However, if we consider a dynamic system where the number of aggregate properties increases over time, the complexity of the populate operation would be expressed as $O(f(m))$, and event replay would be $O(n \cdot f(m))$. Thus, it can be stated that unlike the event replay algorithm, the performance of the populate operation does not degrade as more events are stored in the database.

Due to the fact that in the mCQRS approach events are not strictly immutable, rapid incorporation of changes to stored data becomes possible. This significantly simplifies tasks such as deleting user data in compliance with GDPR.

On the other hand, the immutability of events allows for a higher level of security for the Event Store. Given the absence of a need to modify events, the Event Store can be protected against changes at the configuration level. Additionally, encryption methods can be applied, and, for example, reusing hashes of previous events when adding new ones (by analogy with blockchain technology [43]) enhances data integrity.

As mentioned earlier, there are situations when the version mismatch issue cannot be resolved with multiple retries. Using the mCQRS approach, the issue is resolved by simply updating a projection from the latest snapshot (Fig. 9).

Optimising the performance of the classical CQRS approach typically involves using snapshots for both aggregates and read models. The mCQRS approach addresses these issues by merging the snapshots of the aggregate and projections, thereby simplifying the management of snapshots and mitigating the complexity. This is partially supported by the cyclomatic complexity metrics for “Snapshot repository” and “Base projection repository” component, which exists only in the classical CQRS variation and has a complexity value of 9 and 8 respectively.

The primary drawback of a system based on the mCQRS approach in comparison with classical CQRS one is the high response time for write operations, especially when adding new entities. This is a logical consequence of waiting for data writes or updates in both relational database and Event Store. Saving a series of events to the Event Store undoubtedly occurs much faster (107 ms vs 186 ms based on the experiment data). However, it is worth noting that since event processing in both variations occurs

in a similar manner, the difference in response time and the time required for the system eventual update remains the same ($186 - 107 = 79$ and $347 - 268 = 79$). The longer the eventual update logic takes, the less impact this difference has on overall system performance.

The task of migrating a system between two different architecture approaches can be logically divided into two sub-tasks: functionality transformation and data migration.

The data migration can be considered as the most difficult part, especially in cases when data storage approaches are completely different. For instance, source architecture relies on a relational database, while the target one uses Event Store. When performing such a migration, it is necessary to create an event store along with a snapshot and projection database(s), whose structure will differ from the existing relational database. This task is particularly complex for live IS, since migration must proceed smoothly and without huge downtimes.

Conversely, migrating in the other direction requires designing a relational database for a system at later stages of development, taking into account all the nuances of data structure that were previously stored as events and a set of denormalised projections.

From a data-migration perspective, the mCQRS approach is more flexible than classical CQRS. The Snapshot DB essentially serves as a relational database after migration to such approach as DDD. And the Event Store, can be used as a base for migration to the approach that consider Event Store as a source of truth (e.g. classical CQRS).

Conclusions

The scientific novelty. For the first time, an alternative modification of the CQRS with ES architecture has been proposed. This variation is well-suited for a specific class of systems where the speed of write operations is not a critical metric. Based on performance measurements for the pilot IS, the average write operation response time is 107 ms for classical CQRS versus 186 ms for the proposed approach, with the overall command processing time being 268 ms and 347 ms, respectively.

The proposed approach simplifies the resolution of several challenges associated with the classical CQRS architectural variation. The total cyclomatic complexity of a SS using the proposed approach is $62 + 20 \cdot N$, compared to $42 + 78 \cdot N$ for classical CQRS (where N represents the number of aggregate roots). Additionally, it facilitates the migration

process, particularly in comparison to the transition between DDD and classical CQRS, making architectural changes at later stages of development more manageable.

The practical significance. The proposed modification of the CQRS with ES architecture can be applied to the development of real-world systems, where the classical CQRS approach is suitable, and where the speed of write operations is not a critically

important parameter. Employing this architectural strategy will make the development of SS more efficient and reduce the requirements to the developers' level of skill compared to using the classical CQRS approach.

An experiment was also conducted, demonstrating how the proposed model can be applied in practice to describe system processes, evaluate them, and compare their parameters.

References

- [1] M. Fowler, "Patterns of Enterprise Application Architecture", *Addison-Wesley Professional*, 2002, 560 p. ISBN 978-0321127426 Available: <https://raw.githubusercontent.com/ZoranLi/Books1/master/Patterns%20of%20Enterprise%20Application%20Architecture.pdf>
- [2] B. Woolf and G. Hohpe, "Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions", *Addison-Wesley Professional*, 2004, 736 p. ISBN 978-0321200686. Available: <https://github.com/ivanarandac/Books/blob/master/Enterprise%20Integration%20Patterns%20-%20Designing%20Building%20And%20Deploying%20Messaging.pdf>
- [3] E. Evans, "Domain-Driven Design: Tackling Complexity in the Heart of Software", *Addison-Wesley Professional*, 2004, 534 p. ISBN 978-0321125217. Available: <https://github.com/gmoral/Books/blob/master/Domain%20Driven%20Design%20Tackling%20Complexity%20in%20the%20Heart%20of%20Software%20-%20Eric%20Evans.pdf>
- [4] Neal Ford *et al.*, "Building evolutionary architectures (2nd ed.)", O'Reilly Media, Sebastopol, CA, 2022, 262 p. Available: https://www.thoughtworks.com/content/dam/thoughtworks/documents/books/bk_building_evolutionary_architectures_second_edition_free_chapter.pdf
- [5] Y. Zhong, "Using Event Sourcing and CQRS to Build a High Performance Point Trading System", *E-Business and Applications: 5th International Conference*, Bangkok, New York: Association for Computing Machinery, 25 February 2019, pp. 16–19. Available: <https://doi.org/10.1145/3317614.3317632>
- [6] D. Betts *et al.*, "Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure (1st ed.)", Microsoft patterns & practices, 2012. Available: https://download.microsoft.com/download/e/a/8/ea8c6e1f-01d8-43ba-992b-35cfcaa4fae3/cqrs_journey_guide.pdf
- [7] M. Fowler, "CQRS". Available: <https://martinfowler.com/bliki/CQRS.html>
- [8] G. Young, "CQRS Documents by Greg Young". Available: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf
- [9] H. Taylor *et al.*, "Event-Driven Architecture: How SOA Enables the RealTime Enterprise", *Addison-Wesley Professional*, 2009, 272 p. ISBN 978-0321591388. Available: <https://www.oreilly.com/library/view/event-driven-architecture-how/9780321591388/>
- [10] M. Overeem *et al.*, "An empirical characterization of event sourced systems and their schema evolution – Lessons from industry", *Journal of Systems and Software*, 2021, Vol. 178, Iss. 4. Available: <https://doi.org/10.1016/j.jss.2021.110970>
- [11] K. Truysers, "Introduction to Domain Driven Design, CQRS and Event Sourcing", 2013. Available: <https://www.kenneth-truysers.net/2013/12/05/introduction-to-domain-driven-design-cqrs-and-event-sourcing/>
- [12] V. Vernon, "Implementing Domain-Driven Design", *Addison Wesley*, 2013, 656 p. ISBN 978-0321834577. Available: <https://ptgmedia.pearsoncmg.com/images/9780321834577/samplepages/0321834577.pdf>
- [13] O. Lytvynov and D. Hruzin, "Decision-making on Command Query Responsibility Segregation with Event Sourcing architectural variations", *Technology audit and production reserves*, 2025, Vol. 4, Iss. 2 (84), pp. 37–59. Available: <https://doi.org/10.15587/2706-5448.2025.337168>
- [14] General Data Protection Regulation. Available: <https://gdpr-info.eu/>
- [15] N. Korkmaz and M. Nilsson, "Practitioners' view on command query responsibility segregation: thesis, MSc Information Systems", *Lund University: School of Economics and Management*, 2014, 53 p. Available: <https://lup.lub.lu.se/luur/download?func=downloadFile&recordId=4864802&fileId=4864803>
- [16] O. Lytvynov and D. Hruzin, "On the migration of Domain-driven Design to CQRS with Event Sourcing software architecture", *Information Technology Computer Science Software Engineering and Cyber Security*, 2024, Vol. 1, Iss. 1, pp. 50–60. Available: <https://doi.org/10.32782/IT/2024-1-7>
- [17] P. Revesz, "Relational Databases", *Introduction to Databases*, London: Springer, 2010, Chap. 4, 743 p. Available: https://doi.org/10.1007/978-1-84996-095-3_3
- [18] M. L. Brodie and M. Stonebraker, "Ai S. DARWIN: On the Incremental Migration of Legacy Information Systems", 1995. Available: <https://dsf.berkeley.edu/papers/S2K-93-25.pdf>

- [19] M. Breitmayer *et al.*, “Deriving Event Logs from Legacy Software Systems”, *Lecture Notes in Business Information Processing*, 2023, Vol. 468, pp. 409–421. Available: https://doi.org/10.1007/978-3-031-27815-0_30
- [20] D. Comartin, “Snapshots in Event Sourcing for Rehydrating Aggregates”. Available: <https://codeopinion.com/snapshots-in-event-sourcing-for-rehydrating-aggregates/>
- [21] O. Evsyukov, “The Bermuda Aggregate. And the Rescue of the Drowning”. Available: <https://youtu.be/Br4TL-486ZM?t=1500>
- [22] O. Dudycz, “Snapshots in Event Sourcing”. Available: <https://www.eventstore.com/blog/snapshots-in-event-sourcing>
- [23] DBB Software’s official company site. Available: <https://dbbsoftware.com/>
- [24] G. Young, “Versioning in an Event Sourced System”. Available: <https://leanpub.com/esversioning>
- [25] S. Kleanthous, “Event immutability and dealing with change”. Available: <https://www.eventstore.com/blog/event-immutability-and-dealing-with-change>
- [26] R. Laigner *et al.*, “An Empirical Study on Challenges of Event Management in Microservice Architectures”, in *An Empirical Study on Challenges of Event Management in Microservice Architectures*. Available: <https://doi.org/10.48550/arXiv.2408.00440>
- [27] P.R.G. Vasconcellos *et al.*, “Applying Event Sourcing in a ERP System: A Case Study”, *XLIV Latin American Computer Conference*, São Paulo, IEEE, 1–5 October 2018, pp. 80–89. Available: <https://doi.org/10.1109/CLEI.2018.00019>.
- [28] D.K. Pandiya and N.G. Charankar, “Optimizing Performance and Scalability in Micro Services with CQRS Design”, *International Journal of Engineering Research & Technology*, 2024, Vol. 13, Iss. 4. Available: <https://doi.org/10.17577/IJERT-V13IS040284>
- [29] B. Wu *et al.*, “The Butterfly Methodology: a gateway-free approach for migrating legacy information systems”, 1997. Available: <https://doi.org/10.1109/ICECCS.1997.622311>
- [30] M. Breitmayer *et al.*, “Deriving Event Logs from Legacy Software Systems”, 2023. Available: https://doi.org/10.1007/978-3-031-27815-0_30
- [31] G. Salvatierra *et al.*, “Legacy System Migration Approaches”, 2013. Available: <https://doi.org/10.1109/TLA.2013.6533975>
- [32] G. Young, “Link to GitHub repository”. Available: <https://github.com/gregoryyoung>
- [33] M. Driscoll, “The Publish-Subscribe Pattern”, *wxPython Recipes: A Problem – Solution Approach*, 2017, Chap. 4, 369 p. Available: https://link.springer.com/chapter/10.1007/978-1-4842-3237-8_4?utm_source=researchgate.net&utm_medium=article
- [34] P. Shaddel, “Understand and Implement Long-Polling and Short Polling in Node.js”. Available: <https://levelup.gitconnected.com/understand-and-implement-long-polling-and-short-polling-in-node-js-94334d2233f3>
- [35] L.G. Cretu, “Event-driven replication in distributed systems”, *India Software Engineering: the 4th Annual Conference, Thiruvananthapuram Kerala*, New York: Association for Computing Machinery, 24–27 February 2011, pp. 95–98. Available: <https://doi.org/10.1145/1953355.1953367>
- [36] “Practical and focused guide for survival in post-CQRS world. Projections”. Available: <http://cqrs.wikidot.com/doc/projection>
- [37] D. Hruzin, “Link to Zenodo repository”. Access mode: <https://zenodo.org/records/18263117>. Available: <https://doi.org/10.5281/zenodo.18263116>.
- [38] T.J. McCabe, “A Complexity Measure”, *IEEE Transactions on Software Engineering*, 1976, Vol. SE-2, Iss. 4, pp. 308–320. Available: <https://doi.org/10.1109/TSE.1976.233837>
- [39] SonarQube Cloud official site. Available: <https://www.sonarsource.com/products/sonarcloud/>
- [40] D. Hruzin, “Link to SonarQube scan report”. Available: https://sonarcloud.io/component_measures?metric=complexity&selected=dmitryhruzin_CQRS-variations-test%3Atypescript%2Fapps&id=dmitryhruzin_CQRS-variations-test
- [41] D. Hruzin, “Link to GitHub repository”. Available: <https://github.com/dmitryhruzin/CQRS-variations-test/blob/main/scripts/run-measurement.js>
- [42] S. Gayathri Devi and Dr K. Selvam, “An abstract to calculate big O factors of time and space complexity of machine code”, *Conference: International Conference on Sustainable Energy and Intelligent Systems (SEISCON 2011)*, Jan. 2021. Available: <https://doi.org/10.1049/cp.2011.0483>.
- [43] Z. Zheng *et al.*, “An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends”, *Conference: 6th IEEE International Congress on Big Data*. Available: <https://doi.org/10.1109/BigDataCongress.2017.85>

Д.Л. Грузін, О.А. Литвинов

ІНЖЕНЕРІЯ ПРОГРАМНИХ СИСТЕМ НА ОСНОВІ АРХИТЕКТУРИ CQRS З EVENT SOURCING ЩО ҐРУНТУЄТЬСЯ НА ЗНІМКУ СТАНУ СИСТЕМИ

Проблематика. Розмежування відповідальності команд і запитів (CQRS) у поєднанні з підходом фіксації та збереження подій (англ. Event Sourcing, ES) є поширеним рішенням для проектування масштабованих і високопродуктивних інформаційних систем. Утім, класичні реалізації CQRS з ES часто пов’язані з підвищеною складністю розроблення та супроводу.

Мета дослідження. Метою роботи є оптимізація розроблення та супроводу програмних систем, побудованих на CQRS з ES, шляхом запровадження альтернативної варіації цієї архітектури.

Методика реалізації. Проаналізовано класичну варіацію архітектури й виокремлено компоненти, що підвищують складність розроблення та супроводу системи. На основі проведеного аналізу запропоновано альтернативну варіацію архітектури (mCQRS), яка передбачає використання набору компонентів із нижчою складністю. Рішення ґрунтується на використанні реляційної бази даних, в якій знімки стану агрегатів розглядаються як джерело істини, що знижує складність реалізації та супроводу програмного забезпечення, а також спрощує потенційний перехід до інших архітектурних підходів.

Результати дослідження. Побудовано репрезентативні тестові проекти для класичної та mCQRS варіацій. Значення цикломатичної складності реалізації типового процесу виконання команди (120 для Classical CQRS та 82 для mCQRS) свідчать про спрощення на 31.67 %. Водночас час відповіді сервера на запити є однаковим для обох варіацій (44 мс), тоді як повний час досягнення узгодженості системи для команд становить 268 мс для Classical CQRS та 347 мс для mCQRS. Попри зниження продуктивності на 22,76 % швидкодія операцій запису залишається високою в контексті усталених галузевих практик.

Висновки. Запропонований підхід підвищує ефективність розроблення й супроводу та зменшує вимоги до рівня компетентностей розробників; він є доцільним для класу систем, в яких швидкодія операцій запису не є критичною.

Ключові слова: архітектура програмного забезпечення; оптимізаційні моделі; CQRS; Event Sourcing порівняльний аналіз.

Рекомендована Радою
факультету прикладної математики
КПІ ім. Ігоря Сікорського

Надійшла до редакції
19 січня 2026 року

Прийнята до публікації
09 березня 2026 року

Опубліковано
30 березня 2026 року